UNIVERSIDAD
DE MÁLAGA

ESCUELA DE INGENIERÍAS INDUSTRIALES

Departamento de Ingeniería de Sistemas y Automática

Área de Conocimiento de Ingeniería de Sistemas y Automática

TRABAJO FIN DE GRADO

Detección de obstáculos estáticos y dinámicos y evitación de colisiones para un vehículo a escala

Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Autor: D. Jorge Ruiz Rico

Tutor: Dr. D. Jesús Morales Rodríguez

Málaga, Enero de 2023

Agradecimientos

A mis padres, por su apoyo incondicional y sus consejos que me han dado la fuerza necesaria para seguir adelante superando todos los retos y dificultades que se me han presentado.

A mi tutor, Jesús Morales, por su paciencia y su ayuda durante todo el proceso de desarrollo del trabajo sin el que nada de esto habría sido posible.

A mi hermana, por haber sido un referente desde el primer día y modelo a seguir de constancia, superación y responsabilidad. Un apoyo esencial para haber logrado todo lo que he conseguido hasta ahora.

A mis compañeros y amigos Carlos y Adrián, porque juntos hemos superado todos los baches que se nos han presentado, incluyendo el TFG. Con ellos, he finalizado esta etapa de mi vida y me llevo a dos amigos incondicionales. Hemos ido año por año apoyándonos los unos en los otros sin dejar que ninguno se quede atrás.

Resumen

El objetivo del presente trabajo es el desarrollo e implementación de un software en un vehículo a escala, que permita, mediante un control activo del mismo, realizar una evitación de obstáculos estáticos y/o dinámicos además de un control activo de colisiones.

Todo esto, se desarrollará en ROS (Robot Operating System), verificando el diseño en el simulador Gazebo. Antes de plantear la solución al trabajo, se realizará un estudio del estado del arte que permita conocer a fondo los tipos de conducción autónoma reconocidas actualmente, así como una breve descripción de las herramientas a utilizar y del propio vehículo con el que se realizará el ensayo.

Palabras clave: ROS, Gazebo, conducción autónoma, control de colisiones, evitación de obstáculos.

Abstract

The objective of the present work is to implement and develop a software for a scale vehicule, which allows, through an active control of it, to perform a static and/or dynamic obstacles avoidance as well as an active collision avoidance.

All this, will be developed in ROS (Robot Operating System), verifiying the design in the Gazebo simulator. Before proposing the solution of the work, a study of the state of the art will be carried out that allows an in-depth knowledge of the currently recognized types of autonomous driving, as well as, a brief description of the tools that will be used during the work and of the vehicle itself with which the test will be carried out.

Keywords: ROS, Gazebo, autonomous driving, collision control, obstacle avoidance.

Índice

Índice de figuras

Capítulo 1 . Introducción

1.1 Antecedentes

El presente trabajo se desarrolló con el objetivo de cubrir uno de los retos a realizar en el “Autonomous Driving Challenge”, una competición interuniversitaria patrocinada por SEAT y Volkswagen cuyo objetivo es el desarrollo de funciones de conducción completamente autónomas en un vehículo real a escala (ver *Figura 1*).

Figura 1. Vehículo a escala de la competición [2]

Con motivo de la situación actual, el final de la pandemia y el conflicto bélico entre Rusia y Ucrania, se han visto obligados a cancelar la competición de este año académico 2021-2022. Es por ello por lo que se plantea el presente trabajo, para cubrir uno de los retos de la competición y cuyo objetivo junto con otro grupo de trabajos, es el de abordar el problema de la conducción autónoma en su totalidad.

Es importante saber, que en función del diseño del vehículo y de su capacidad de cubrir tareas que son normalmente realizadas por el hombre, se le asigna un nivel de autonomía siguiendo la clasificación que se muestra en la *Figura 2*.

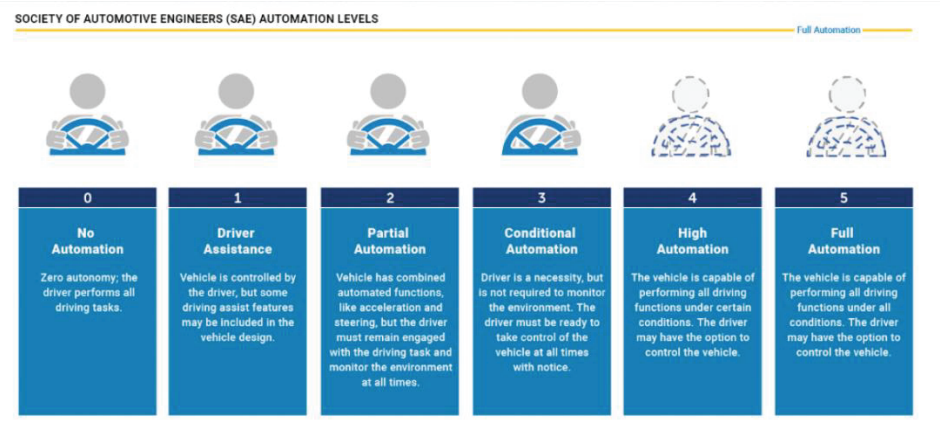


Figura 2. Niveles de conducción autónoma. [5]

En el caso de este proyecto, el coche diseñado alcanzaría un nivel de autonomía 5, todo ello, una vez juntados los diseños de los distintos objetivos de la competición.

1.2 Objetivos

El objetivo principal del trabajo es desarrollar un software capaz de realizar la detección activa de obstáculos estáticos y dinámicos, así como, la evitación de colisiones.

Se entiende que este trabajo es un fragmento de un programa superior que, si se realiza de manera óptima, se podrá sincronizar e implementar en un vehículo a escala para que este sea capaz de realizar todas las tareas que se le planteen de manera autónoma, no solo la evitación de obstáculos, sino también el seguimiento de carriles, la detección de aparcamiento y estacionamiento en línea y en batería, la detección y reacción ante señales de tráfico, etc.

Es por ello por lo que se busca que el código que aquí se genere sea lo más claro y genérico posible, para poder en un futuro ser integrado en otro programa sin problemas de sincronismo, pudiendo ser mejorado o adaptado, según las necesidades del proyecto.

Para realizar este trabajo, se empleará el sistema operativo Ubuntu, se desarrollará el software en ROS y se utilizará el simulador Gazebo para ir comprobando la validez del diseño.

1.3 Fases del trabajo

El trabajo se va a dividir en tres fases principales. Una primera fase se va a basar en la búsqueda de información, para la identificación y comprensión de las herramientas que se van a utilizar. Aquí se aprovechará para definir los elementos físicos que se van a utilizar para poder probar este programa y que forman parte del vehículo a escala que se dispone en el departamento de la universidad, como es el LIDAR.

Una vez definidos todos los elementos del vehículo, se comenzará con la fase de desarrollo del software, para lo que se empleará la herramienta Visual Code, que permite tener todo el código organizado de una manera clara e ir modificándolo según sea necesario.

Aunque se irán haciendo comprobaciones durante la parte de desarrollo, la última fase se basa en la verificación del programa en su conjunto en el simulador, intentando mantener la filosofía de la competición. Para poder realizar esta comprobación, se incorporarán códigos que no son objetivos de este trabajo, pero que se serán de diseño propio, como es el de seguimiento de carril mediante la técnica del Pure Pursuit.

Una vez comprobado el correcto funcionamiento del diseño en el simulador, se pasará a las pruebas experimentales, donde se cargará el diseño en el vehículo UMA racecar para ver si este es capaz de cumplir con los objetivos satisfactoriamente.

1.4 Estructura de la memoria

A continuación, se van a definir los puntos que van a ser desarrollados en el presente trabajo, de tal manera que, mediante la lectura en orden de los mismos, cualquier persona sea capaz de realizar las simulaciones con las herramientas adecuadas.

Capítulo 1: Introducción

En este capítulo se va a introducir el trabajo, definiendo cuáles son los objetivos del mismo y de qué punto se parte, para después, ir poco a poco avanzando hasta lograr el objetivo final. Se irán haciendo referencias a aquellos conceptos o temas que son necesarios conocer para la correcta comprensión de este trabajo.

Capítulo 2: Software y hardware empleado

En este segundo capítulo, se dará una introducción a las herramientas empleadas, tanto desde el punto de vista del software como del hardware

Capítulo 3: Detección y evitación de obstáculos

En el tercer capítulo, se dará una breve introducción al tema a tratar planteando diferentes alternativas para abordarlo y, finalmente, la solución seleccionada.

Capítulo 4: Desarrollo del software

A continuación, se desarrollará en este capítulo el software que hará que el vehículo realice las tareas encomendadas. En el caso de este proyecto, serán tres códigos los que trabajarán de manera simultánea, subscribiéndose y publicando en diferentes topics que más adelante se explicarán, para conseguir que el vehículo cumpla con los requerimientos especificados.

Capítulo 5: Resultados y simulación

En este capítulo, se explicarán los pasos a seguir para crear el entorno de trabajo y clonar el repositorio de Github. Se harán referencias a los anexos en

este punto, para facilitar al lector lanzar el entorno de simulación sin el que no sería posible realizar las distintas pruebas del código.

Aquí también se mostrarán los resultados obtenidos durante las simulaciones y pruebas experimentales, como comprobación del correcto funcionamiento del código desarrollado. Aun así, se proporcionará acceso a un repositorio de GitHub donde estarán colgadas todas las carpetas y códigos necesarios para realizar la simulación.

Capítulo 6: Conclusiones y líneas de mejora

Por último, se sacarán algunas conclusiones de la realización de este trabajo, que a su vez servirán para proponer posibles líneas de mejora que den lugar a futuros trabajos de fin de grado.

Capítulo 2 . Software y hardware empleado

En este punto del trabajo, se pretende dar a conocer todas las herramientas empleadas, tanto desde el punto de vista del software, como de hardware. De esta manera, se busca que el lector, a través de su lectura, tenga una idea de los posibles pasos a seguir para arrancar con un proyecto que siga esta línea.

2.1 Robot Operating System (ROS)

Desde el punto de vista del software, se tienen distintas herramientas útiles para abordar el proyecto. La primera de todas se considera esencial para trabajos de este estilo, es el Sistema Operativos Robótico o ROS.

Se entiende por ROS como un conjunto de framworks para el desarrollo de software para robots. Tiene su origen en 2007 en el laboratorio de inteligencia artificial de Standford. ROS proporciona servicios estándar para el control de dispositivos de bajo nivel, implementación de funcionalidad de uso común, paso de mensajes entre procesos, mantenimiento de paquetes, etc. (ver *Figura 3*)

Figura 3. Robot Operating System. [3]

El procesamiento de la información tiene lugar en los nodos, capaces de recibir, mandar y multiplexar mensajes de los sensores, actuadores y/o estados. Su librería está pensada para sistemas de tipo UNIX, aunque está en proceso de adaptación actualmente para ser compatible con otros sistemas operativos.

Podemos dividir ROS en dos partes esenciales, la parte del sistema operativo y la de ros-pkg. Esta última, es una suite de paquetes aportados por

contribuciones de distintos usuarios que van implementado diferentes funcionalidades como son la percepción, planificación, mapeado, etc.

Es de vital importancia comprender el funcionamiento de los nodos en ROS ya que son la base de este trabajo y permiten poder tener varios procesos trabajando al mismo tiempo de manera coordinada.

Un nodo puede publicar información en un topic, subscribirse a otro para procesarla o realizar ambas cosas de manera simultánea. Por ejemplo, puede estar subscrito al topic que almacena la información del LIDAR, tomar esa información y publicar en otro topic variables de acción para que otro nodo gestione el movimiento del vehículo en consecuencia. Esa es la filosofía de los nodos, además, esto permite compartimentar los diseños y que estos puedan ser reutilizados en otros programas sin problemas de compatibilidad.

2.2 Gazebo

Una vez entendida la base donde se realizará el proyecto, se va a definir uno de los simuladores principales y más utilizados en el trabajo con robots móviles, como es este caso. Gazebo, es un simulador libre para trabajos con robots hasta en 3D (ver *Figura 4*).

Figura 4. Simulador Gazebo. [4]

Este simulador tiene capacidad para trabajar con sensores y actuadores en tiempo real. Esto hace que el usuario sea capaz de recrear escenas realistas con iluminación, sombras y texturas. Permite recrear circuitos a escala e ir modificando objetos incluso durante la simulación, como el añadir un obstáculo

en un circuito para el vehículo o incluso desplazarlo. Esto la convierte en una herramienta realmente versátil que permitirá realizar comprobaciones periódicas del programa que se vaya desarrollando para le vehículo.

Como tarea inicial, se va a implementar un circuito a escala en Gazebo igual al que se empleó en la competición en otras ediciones de tal forma que, cada estudiante, tendrá que incorporar al circuito los elementos que afecten a su trabajo.

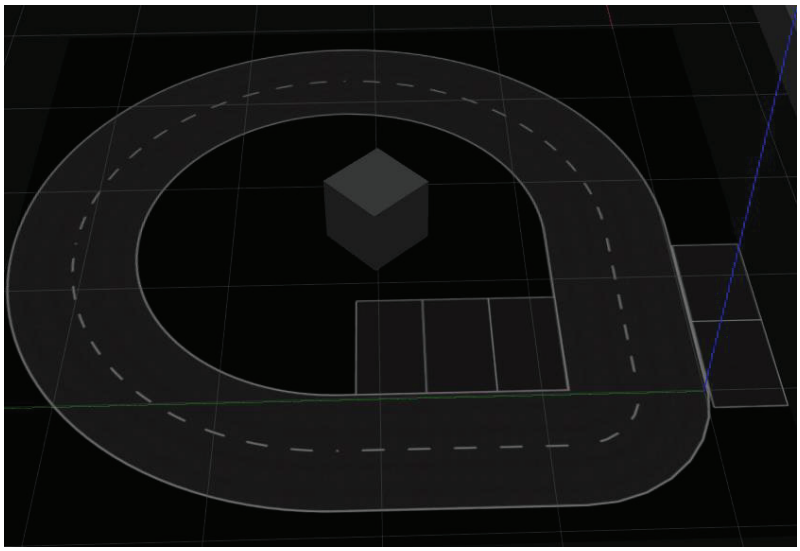


Figura 5. Circuito de entrenamiento. [Elaboración propia]

En la *Figura 5*, se puede ver un ejemplo de lo que sería un diseño de un circuito de entrenamiento en Gazebo con el que ir comprobando el comportamiento del vehículo, antes de llevarlo al circuito real.

En este trabajo, se estudiará la incorporación de obstáculos estáticos, que se pueden interpretar como vehículos en doble fila, peatones parados o bicicletas y obstáculos dinámicos, como puede ser realizar un adelantamiento de otros vehículos en marcha que vayan a una velocidad demasiado baja. Tal y como se mencionará más adelante, se dejarán una serie de propuestas de mejoras de este código para futuros trabajos que lo lleven a cubrir todas las exigencias de una conducción realista.

2.3 Visual Studio Code

Adicionalmente, cabe mencionar la herramienta de edición que se ha empleado para el desarrollo del software del vehículo. Esta es Visual Studio Code (ver *Figura 6*), un editor de código fuente desarrollado por Microsoft.

Figura 6. Visual Studio Code. [6]

Es compatible con varios lenguajes de programación, en el caso de este proyecto, se realizará toda la programación a través de esta herramienta y empleando el lenguaje Python ya que se considera más intuitivo que otros compatibles como el C++ y, además, es más utilizado para robótica móvil actualmente.

2.4 GitHub

Por otro lado, tenemos la plataforma GitHub (ver *Figura 7*). Es una herramienta muy versátil a la hora de realizar proyectos que se desean compartir de manera abierta o privada con otras personas para que el desarrollo sea colaborativo o como propuesta de soluciones ante distintos retos.

Figura 7. GitHub. [1]

Esta plataforma, se ha empleado para analizar proyectos de otros usuarios y aprender a trabajar con las herramientas empleadas en el trabajo, como ROS y Gazebo. Otra alternativa es GitLab (ver *Figura 8*), que es otra plataforma con la misma funcionalidad y de la que se ha obtenido igualmente información que facilitaron los organizadores de la competición *Autonomous Driving Challenge,* como base para iniciar este proyecto.

Figura 8. GitLab. [19]

2.5 Sensor LIDAR

Para realizar todas las tareas anteriormente mencionadas, habrá que apoyarse en los sensores de los que dispone el vehículo. En este caso, se necesitará trabajar con la información proporcionada por el sensor LIDAR.

LIDAR, es el acrónimo de “Laser Imaging Detection And Ranging”. Como ya se puede intuir, un sensor LIDAR emite haces de rayos láser infrarroja cuyo

objetivo es rebotar con objetos que se encuentren en su rango de trabajo y así poder determinar la distancia a la que se encuentran. Para ello, se calcula el tiempo en el que el rayo va, rebota y vuelve, conocido el tiempo de vuelo.

Con un sistema de referencia apropiado, también se podría emplear el LIDAR para determinar la velocidad a la que se mueven otros objetos, aunque para ello, necesitaría apoyarse en 2 mediciones consecutivas. El LIDAR consta de unas lentes para la emisión de los haces láser y otra lente para la captación de los haces que son reflejados.

Resulta una herramienta realmente versátil en la conducción autónoma porque, si realizamos una correcta compartimentación del rango de trabajo del LIDAR, podemos cubrir los obstáculos por zonas y saber si se encuentra a un lado o a otro del vehículo y a qué altura. Esto resulta de vital importancia debido a que, en función de donde se encuentre el objeto, el vehículo tendrá que reaccionar de una manera distinta.

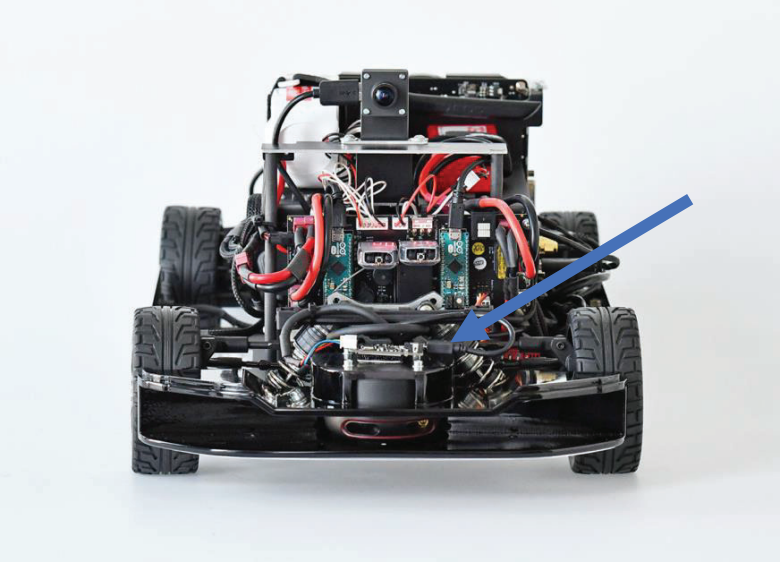


Figura 9. Modelo de vehículo de la competición. [2]

En la *Figura 9*, se puede observar el vehículo que se empleó en la última edición del Autonomous Driving Challenge. En la parte frontal del vehículo, se puede ver marcado con una flecha azul el sensor LIDAR. Normalmente, se suelen situar en una zona no muy cercana al suelo para darle un mayor campo de visión y libre de los demás elementos del vehículo. En el caso de este trabajo, se empleará un modelo de este vehículo desarrollado por la universidad con características similares tanto físicamente como en software.

El campo de visión del LIDAR es de aproximadamente 270º con un alcance de 6 metros y una frecuencia de actualización de entorno los 10 Hz. Esta información es de vital importancia y habrá que tenerla en cuenta a la hora de realizar el procesado de la información proporcionada por el LIDAR.

Capítulo 3 . Detección y evitación de obstáculos

3.1 Introducción

Actualmente, la detección de obstáculos y la evitación de colisiones es el principal reto dentro del campo de la conducción autónoma. Estos sistemas son críticos, ya que se ponen en juego las vidas de las personas si no se realiza un diseño óptimo. Es por ello por lo que se debe hacer hincapié en el desarrollo de nuevas y mejoradas técnicas de detección de obstáculos.

Entre las distintas soluciones que se pueden plantear para abordar esta tarea, una de ellas es el empleo de redes neuronales entrenadas para actuar en función de las imágenes que procesan procedentes de cámaras 360º (ver *Figura 10*).

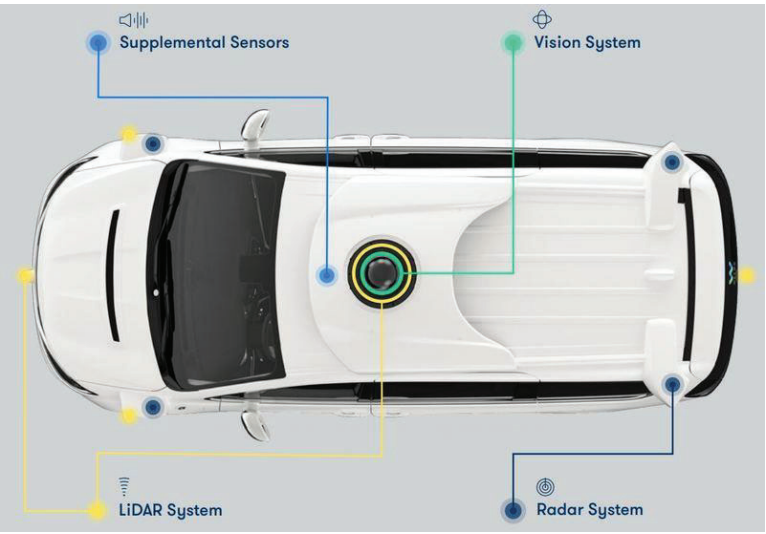


Figura 10. Cámara 360. [7]

Esta opción sería viable para este trabajo, el problema, es que el vehículo de la competición tiene una cámara en la parte superior con un rango de 130º y una trasera de 80º. Con esto se podría cubrir un gran rango de trabajo, pero no sería suficiente para una conducción segura. Además, las cámaras se pueden emplear para el seguimiento de carril, luego se ha optado por otra alternativa viable que libera de trabajo a las cámaras y requiere menos capacidad de procesamiento.

Otras alternativas a tener en cuenta son los sónares (equipados en el vehículo de la competición) o sensores de presión, que pueden servir de apoyo como seguridad pasiva para la evitación de colisiones o atropellos. Todos esos caminos pueden llevar a futuras líneas de trabajo que, si se realizan con las mismas herramientas, se pueden adaptar y funcionar de manera coordinada hasta cubrir todos los requerimientos de una conducción autónoma segura.

3.2 Detección de obstáculos con el LIDAR

En el capítulo 2.5, se explicó el funcionamiento del LIDAR desde el punto de vista del hardware. En este punto, se va a ver como procesar de manera correcta la información que este nos proporciona (ver *Figura 11*) para poder generar un programa capaz de mantener una seguridad de tipo activa en un vehículo, analizando lo que sucede en su entorno.

```
# Single scan from a planar laser range-finder
#
# If you have another ranging device with different behavior (e.g. a sonar
# array), please find or create a different message, since applications
# will make fairly laser-specific assumptions about this data

Header header            # timestamp in the header is the acquisition time of
                         # the first ray in the scan.
                         #
                         # in frame frame_id, angles are measured around
                         # the positive Z axis (counterclockwise, if Z is up)
                         # with zero angle being forward along the x axis

float32 angle_min        # start angle of the scan [rad]
float32 angle_max        # end angle of the scan [rad]
float32 angle_increment  # angular distance between measurements [rad]

float32 time_increment   # time between measurements [seconds] - if your scanner
                         # is moving, this will be used in interpolating position
                         # of 3d points
float32 scan_time        # time between scans [seconds]

float32 range_min        # minimum range value [m]
float32 range_max        # maximum range value [m]

float32[] ranges         # range data [m] (Note: values < range_min or > range_max should be discarded)
float32[] intensities    # intensity data [device-specific units].  If your
                         # device does not provide intensities, please leave
                         # the array empty.
```

Figura 11. Formato del mensaje del tipo Laser Scan. [Realización propia]

Lo primero de todo es acceder a la información proporcionada por el LIDAR, para ello, es necesario subscribirse al tipo de mensaje LaserScan que recoge todas las mediciones realizadas por el LIDAR en tiempo real y devuelve una serie de arrays y variables que pueden ser empleadas para detectar los obstáculos.

La información principal de este topic es la recogida en el array ranges. En este array, se almacena la distancia al objeto detectada por cada haz de luz emitido por el LIDAR. De esta manera, si condicionamos una distancia mínima a un objeto para realizar un cambio de carril, ya tendríamos un primer comienzo en la evitación de obstáculo. Es importante saber que, si el LIDAR no detecta nada en uno de sus haces de luz, devuelve *inf* (infinito) como respuesta.

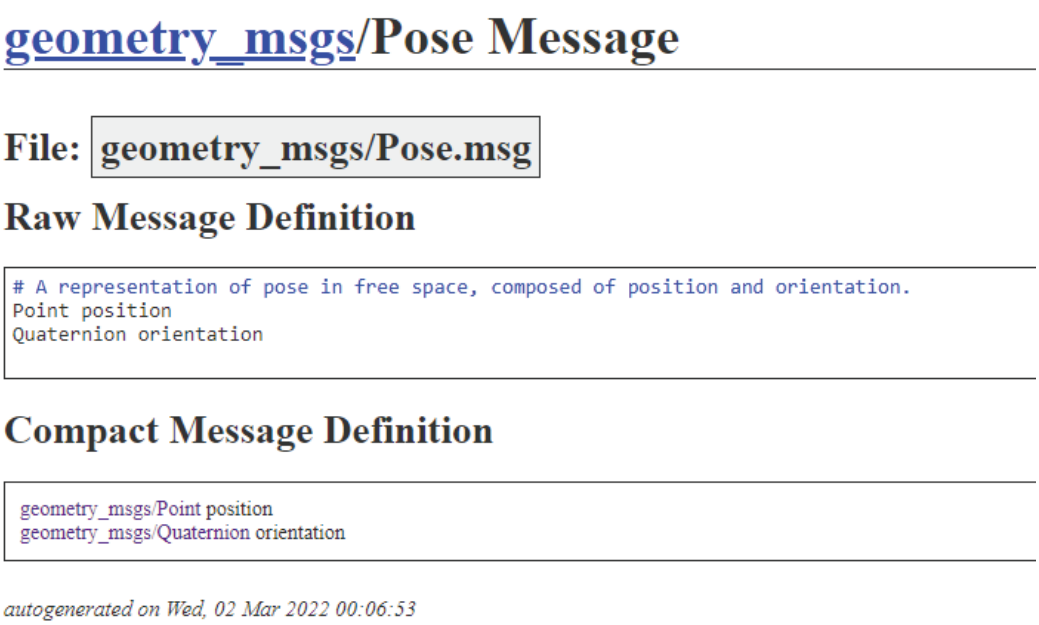
Como este trabajo debe adaptarse a las normas de la competición en la cual se iba a participar, el vehículo debe ser capaz de detectar un obstáculo detenido en cualquier punto de la carretera y evitarlo sin tocarlo.

Por otro lado, en cualquier momento se puede incorporar un segundo vehículo circulando a una velocidad demasiado baja, y deberá adelantarlo si originar situaciones de peligro.

Por último, si se detecta toda la vía bloqueada, que puede ser por peatones o bicicletas cruzando u otras circunstancias, el coche deberá detenerse por completo y esperar a poder reanudar la marcha en condiciones de seguridad.

3.3 Ubicación del vehículo mediante el topic ODOM

Un paso previo a la detección de obstáculos es ubicar el propio vehículo en el espacio. Esto resulta esencial porque si queremos que el vehículo sea capaz de dirigirse a un punto objetivo, lo primero que deberá hacer es localizarse a sí mismo en el espacio mediante un sistema de referencia fijo, para después, calcular los giros necesarios hasta llegar al punto objetivo, así como la distancia mínima la mismo.



geometry_msgs/Pose Message

File: geometry_msgs/Pose.msg

Raw Message Definition

```
# A representation of pose in free space, composed of position and orientation.
Point position
Quaternion orientation
```

Compact Message Definition

```
geometry_msgs/Point position
geometry_msgs/Quaternion orientation
```

autogenerated on Wed, 02 Mar 2022 00:06:53

Figura 12. Mensaje tipo Pose. [Realización propia]

Tal y como vemos en la *Figura 12*, este topic nos da información acerca de la posición y la orientación del vehículo, aunque no de la forma deseada. Con esto lo que se quiere decir es que, para un vehículo, lo ideal es tener información de ángulo de cabeceo, alabeo y guiñada, pero este topic nos da un cuaternio.

Para obtener esta información, tendremos que realizar una transformación del cuaternio a Euler, mediante la inclusión de una librería tf.

Con esta modificación, se obtienen los ángulos de Euler, siendo el de guiñada el que se necesita para el cálculo del giro necesario para orientar el vehículo correctamente.

3.4 Modelo simplificado de Ackermann

Otro punto a tener en cuenta a la hora de trabajar con un vehículo a escala es qué modelo se emplea para simular la conducción del mismo. Uno de los más comunes para iniciar el diseño es el modelo simplificado de Ackermann, tipo de mensaje que puede verse en la *Figura 13.*

ackermann_msgs/AckermannDrive Message

File: ackermann_msgs/AckermannDrive.msg

Raw Message Definition

```
## Driving command for a car-like vehicle using Ackermann steering.
#  $Id$

# Assumes Ackermann front-wheel steering. The left and right front
# wheels are generally at different angles. To simplify, the commanded
# angle corresponds to the yaw of a virtual wheel located at the
# center of the front axle, like on a tricycle.  Positive yaw is to
# the left. (This is *not* the angle of the steering wheel inside the
# passenger compartment.)
#
# Zero steering angle velocity means change the steering angle as
# quickly as possible. Positive velocity indicates a desired absolute
# rate of change either left or right. The controller tries not to
# exceed this limit in either direction, but sometimes it might.
#
float32 steering_angle          # desired virtual angle (radians)
float32 steering_angle_velocity # desired rate of change (radians/s)

# Drive at requested speed, acceleration and jerk (the 1st, 2nd and
# 3rd derivatives of position). All are measured at the vehicle's
# center of rotation, typically the center of the rear axle. The
# controller tries not to exceed these limits in either direction, but
# sometimes it might.
#
# Speed is the desired scalar magnitude of the velocity vector.
# Direction is forward unless the sign is negative, indicating reverse.
#
# Zero acceleration means change speed as quickly as
# possible. Positive acceleration indicates a desired absolute
# magnitude; that includes deceleration.
#
# Zero jerk means change acceleration as quickly as possible. Positive
# jerk indicates a desired absolute rate of acceleration change in
# either direction (increasing or decreasing).
#
float32 speed                   # desired forward speed (m/s)
float32 acceleration            # desired acceleration (m/s^2)
float32 jerk                    # desired jerk (m/s^3)
```

Compact Message Definition

```
float32 steering_angle
float32 steering_angle_velocity
float32 speed
float32 acceleration
float32 jerk
```

Figura 13. Mensaje tipo Ackermann. [Realización propia]

Este modelo se basa en suponer que el vehículo se comporta como un triciclo, es decir, sustituir las dos ruedas delanteras por una rueda situada en el centro del eje delantero, como puede verse en la *Figura 14*. De esta forma, controlando un solo ángulo de guiñada y la velocidad lineal, se tiene control sobre el vehículo.

Es cierto que, en la vida real, hay que tener en cuenta que las ruedas delanteras no tienen por qué girar a la misma velocidad, pero el error que se comete al asumir esta simplificación es lo suficientemente pequeño para que compense y se reduce considerablemente el problema.

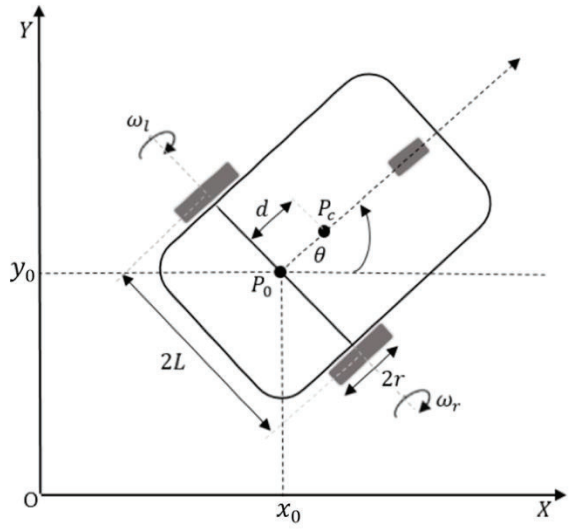


Figura 14. Modelo simplificado de un vehículo. [8]

Una vez comprendido esto, se puede comenzar con el diseño del software que calculará en tiempo real qué giro debe darle a ambas ruedas y su velocidad, para de esta manera, dirigir el vehículo de un punto objetivo al siguiente. Es imprescindible antes de meterse de lleno con el diseño, entender cómo funciona ROS, subscribirse y publicar información en los nodos y lanzar el entorno en gazebo donde se va a realizar la simulación.

Capítulo 4 . Desarrollo del software

Antes de continuar con el desarrollo del software, se va a explicar su filosofía, ya que es importante entender la programación de los distintos nodos para saber identificar las reacciones que tiene el vehículo en cada situación considerada.

Este proyecto, se ha basado en trabajar con ROS tal y como se explicó anteriormente. Es por ello por lo que, si se observa el software diseñado, el lector se podrá dar cuenta de que este ha sido compartimentado en lo que llamamos nodos. Esta forma de aislar el software puede parecer a simple vista perjudicial, sin embargo, tiene una serie de ventajas que se va a enumerar a continuación:

- ✓ En primer lugar, al tener el software compartimentado por tareas, facilitamos la comprensión del mismo, ya que no está todo el programa en un único gran archivo lo que dificultaría enormemente su comprensión.
- ✓ Por otro lado, si el usuario se asegura de separar bien las tareas, evita tener comandos contradictorios, por ejemplo, publicar en dos sitios diferentes velocidad a 1 y a 0 al mismo tiempo.
- ✓ Por último, el hecho de tener nodos que se encarguen de recopilar información y otros que se encarguen de la gestión de esta, los hace útiles para futuros proyectos en los que, por ejemplo, se podría reutilizar el nodo que procesa el LIDAR o el que desplaza un objeto siguiendo un circuito mediante el Pure Pursuit.

Una vez vista la estrategia seleccionada para el diseño, se puede pasar al análisis de los tres códigos desarrollados por separado. Aunque se puede acceder a estos códigos a través del enlace a GitHub proporcionado anteriormente, se dejaran en los anexos los tres programas completos ya que se consideran la base de este trabajo y se facilita así el acceder a los mismos.

Antes de comenzar con la explicación, se va a aportar un diagrama de los nodos principales que van a trabajar, para que el lector comprenda de una manera más sencilla la red de trabajo que se ha construido y como se comunican

los nodos entre sí. Una vez comprendido esto, resultará mucho más sencillo entender el código en sí.

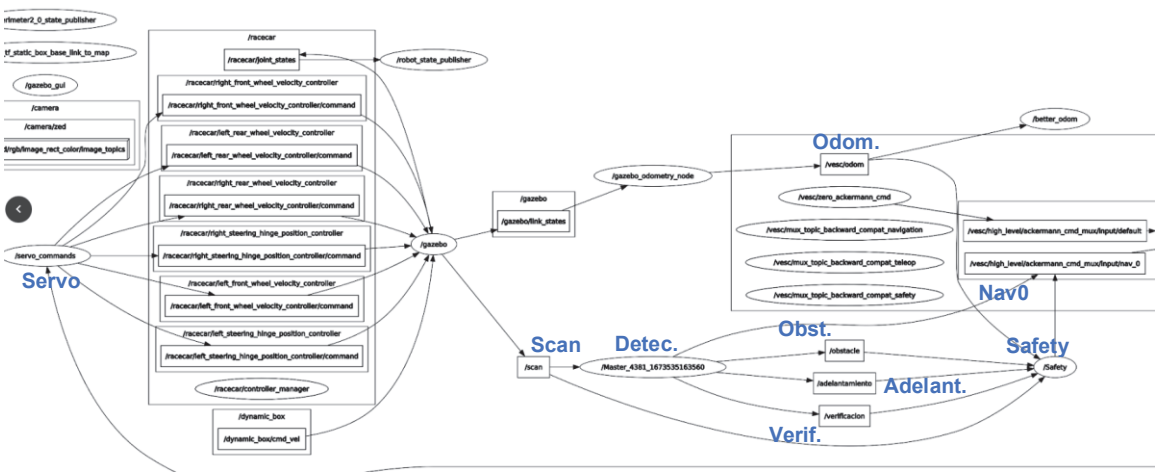


Figura 15. Diagrama obtenido con rqt [Realización propia]

Una vez se tienen lanzados los nodos, en una ventana de comandos, escribir "*rqt_graph*", de esta manera, se obtiene un diagrama como el de la *Figura 15*.

Se han resaltado en azul, los nodos y topics principales del trabajo. Como se puede observar la misma figura, se dispone de un nodo Servo que se comunica con todos los actuadores del coche. Además, estos mandan la información al nodo "*/gazebo"* que es el que permite realizar las simulaciones.

Como se puede ver también en la *Figura 15*, los topics marcados como Scan y Odom, se corresponden con el topic que recoge la información del LIDAR y de la odometría del vehículo, respectivamente. Esta información, es procesada por los nodos Safety y Detección (Detec.) que van a regular el comportamiento del vehículo en el circuito.

Esto es así porque, como se puede ver en la *Figura 15*, esos dos nodos son los únicos que publican información en el topic "*nav0*" que es el que regula el movimiento del vehículo.

Se puede ver fácilmente la comunicación entre los nodos en este diagrama. El nodo Detección (Detec.), publica la información que procesa del LIDAR (Scan) en tres topics (Obst., Adelant. y Verif,) a los que está subscrito el nodo Safety que es el que, en función de la información almacenada en estos topics, actuará sobre la velocidad y giro del vehículo para garantizar que se realice una conducción autónoma segura.

Como se puede observar, esta filosofía de los nodos resulta mucho más intuitiva que entrar de lleno a trabajar sobre los códigos, ya que proporciona una visión de tipo red neuronal que resulta más natural y fácil de asimilar que las líneas de código.

Una vez comprendida la comunicación interna entre los nodos, se puede pasar al desarrollo de los códigos diseñados para este trabajo, empezando por el del nodo Detección.

4.1 Detección

Aunque se puede acceder al código completo en los anexos del presente trabajo, se van a comentar las partes más relevantes del código de manera que se comprenda su comportamiento. El código aquí explicado, está adaptado para funcionar en la simulación, por ello en los anexos de esta memoria, se explicarán las modificaciones necesarias del código para que pueda funcionar en el vehículo UMA racecar.

```
import numpy as np
import rospy
from sensor_msgs.msg import LaserScan
from ackermann_msgs.msg import AckermannDriveStamped
from std_msgs.msg import Float32
lidar = 0
control = 0
mindist = 0
continua = 0
movil = 0
j = 0
# This code Take all the information of the sensors and publish variables in differents topics for the master code
# This function checks all the possible obstacles
```

Figura 16. Inicialización detección. [Realización propia]

Como paso inicial observable en la *Figura 15*, se importan las librerías necesarias para el programa, algunas significativas son:

- Import numpy as np: Numpy, es una librería que permite realizar análisis de datos y cálculos numéricos trabajando con grandes volúmenes de datos. Esto facilita el procesamiento y representación de la información recogida del LIDAR, como se verá más adelante.
- Import rospy: esta es una librería para ROS exclusiva de Python. Esta, permite al programador crear de manera rápida una interfaz con los topics, servicios y parámetros de ROS.
- From sensor_msgs.msg impor LaserScan: esto permitirá al programador acceder a los datos recogidos por el LIDAR al subscribirse al topic LaserScan.

Con esta inicialización y la de las variables que se van a emplear, se puede analizar el código en profundidad. Se va a obviar en esta explicación las estructuras del código y el trabajo con ROS ya que no son el objetivo de este trabajo.

4.1.1 Funciones

En este primer punto, se van a analizar las funciones creadas para la gestión de los obstáculos.

```
def adelantamiento(lidar, control, msg):

  if np.min(msg.ranges[520:560]) < 1:
    lidar = 1
    print("obstaculo frontal")
  else:
    lidar = 0
  if np.min(msg.ranges[596:1080]) < 0.5:
    control = 1
  else:
    control = 0

  return lidar, control

# This function checks if we have a dynamic obstacle and if it is faster than our car
def obstaculo_movil(mindist, msg, j):

  if np.min(msg.ranges[530:550]) < 2 and mindist == 0:
    mindist = np.min(msg.ranges[530:550])
    print("Posible obstaculo ALERTA")
  elif np.min(msg.ranges[530:550]) > 1 and mindist != 0:
    mindist = 0
    j = 1
    print("OBSTACULO MOVIL A MAYOR VELOCIDAD, NO INTENTAR ADELANTAR")
  else:
    j = 0

  return j

# This function checks if we must change or not to the main road (We publish this information in a topic)
def verificar(msg):
  if np.min(msg.ranges[1:535]) < 0.8:
    continua = 1
  else:
    continua = 0
  return continua
```

Figura 17. Funciones detección. [Realización propia]

Como se puede ver en la *Figura 16*, en primer lugar, se ha desarrollado una función de adelantamiento. Esta función considera dos casos de la detección de obstáculos. El primero, es si hay un obstáculo frente al vehículo, esto se logra gracias a la compartimentación del LIDAR. El array que se genera en cada lectura del LIDAR tiene 1081 posiciones, luego, si se separa el rango 520-560, se estará procesado únicamente las mediciones tomadas frente al vehículo.

Por otro lado, también se comprueba el rango 596-1080, esto permite comprobar si hay un obstáculo u otro vehículo en el lateral, para no realizar un cambio de carril con el otro carril ocupado, de ahí que se haya seleccionado la palabra control para nombrar a la variable.

Ya se puede ver en esta función una aplicación práctica de la librería Numpy, ya que se seleccionada de todo el rango de lectura, aquel valor que sea mínimo, es decir, el obstáculo más próximo al vehículo en cada momento.

La segunda función de la *Figura 16,* se puede definir como una función de seguridad. Esta función, tiene el objetivo de detectar si el obstáculo que el vehículo tiene delante es un obstáculo móvil y si es así, si este va más rápido circulando o no.

Esta información resulta clave porque si no se tuviera esta barrera de seguridad al detectar un obstáculo, el vehículo intentaría adelantarlo de manera inmediata sin cuestionarse si puede llegar a hacerlo. De ahí que, si se inicia la maniobra y el otro vehículo va más rápido, nunca llegaría a adelantarlo y podría generar una situación de peligro en vías de doble sentido o bloquear por completo ambos carriles.

La última función de la *Figura 16,* también se puede considerar una función de seguridad y es igualmente importante e incluso más que la anterior. Esta función, actúa solamente cuando estamos en el carril de adelantamiento de tal manera que, cuando se ejecuta la maniobra, comprueba de manera activa todo el lateral derecho del vehículo verificando cuándo hemos adelantado al otro vehículo y si podemos volver al carril principal.

Esto es así, porque se podría plantear la situación de que, al adelantar al vehículo, se encontrase otro delante o toda la vía estuviese congestionada, luego será necesario continuar hasta que se verifique que el carril principal está libre para volver a él en condiciones de seguridad.

4.1.2 Definición del nodo

El siguiente paso es definir el nodo en sí. Para ello, primero se establece una clase, en este caso, se le ha llamado Master, en la que se indicará los topics a los que el usuario se quiere subscribir o aquellos en los que va a publicar.

```
class Master:

  def __init__(self):
    LIDAR_TOPIC = "/scan"
    DRIVE_TOPIC = "/vesc/high_level/ackermann_cmd_mux/input/nav_0"


    ####################################################
    # TODO >>>
    self.drive_msg = AckermannDriveStamped()


    ####################################################

    ####################################################
    # <<< TODO
    ####################################################

    # Initialize a publisher for drive messages
    self.drive_pub = rospy.Publisher(
        DRIVE_TOPIC,
        AckermannDriveStamped,
        queue_size=1)
    # Subscribe to the laser scan data
    rospy.Subscriber(
        LIDAR_TOPIC,
        LaserScan,
        self.callback)
```

Figura 18. Nodo Detección. [Realización propia]

Como se puede observar en la *Figura 17*, este nodo está subscrito al LIDAR y publica en el Drive_topic. Hay que realizar una aclaración en este punto, ya que, en este trabajo, aunque el nodo “Detección” esté subscrito como publicador al drive topic, no va a publicar nada en un principio.

Esto se ha establecido así para evitar colisiones en la publicación de órdenes de velocidad o giro. El hecho por el que no se quita entonces esta línea del código, es porque se considera que en futuras mejoras sí sería conveniente controlar la velocidad en este nodo a modo de barrera de seguridad en otros casos a considerar, como peatones o ciclistas.

```
def callback(self, msg):
  ##################################################
  # TODO >>>
  # Check all the possible obstacle and publish the information
  #
  ##################################################

  print("distancia frontal", np.min(msg.ranges[530:550]))
  print("distancia lateral", np.min(msg.ranges[600:1080]))
  # rango = msg.ranges
  # print(len(rango))
  global lidar, control, continua

  [lidar, control] = adelantamiento(lidar, control, msg)
  movil = obstaculo_movil(mindist, msg, j)
  if movil == 1:
    lidar = 0
    print("OBSTACULO MOVIL A MAYOR VELOCIDAD, NO INTENTAR ADELANTAR")
  continua = verificar(msg)
  #rospy.loginfo(lidar)
  print("LidarC: ", lidar)
  print("LidarT: ", control)
  print("Continuar en carril: ", continua)
  pub.publish(lidar)
  pub2.publish(control)
  pub3.publish(continua)

  ##################################################
  # <<< TODO
  ##################################################
```

Figura 19. Callback del LIDAR. [Realización propia]

En la *Figura 18,* se establece el callback diseñado para el LIDAR. Esto es, cada vez que llega una medición del LIDAR, se ejecuta este callback que se encarga de procesarla y mostrarla al usuario por pantalla. Desde aquí se llama a la función adelantamiento que actualiza las variables "*lidar*" y "*control*" que se explicaron con anterioridad y que son publicadas en los topics "*obstacle*" y "*adelantamiento*".

Adicionalmente, se genera la variable "continua" que será publicada en el topic "*verificación*" para indicar si el vehículo puede o no volver al carril principal.

```
if __name__ == "__main__":
  # Here we indicate that we are going to publish in three differents topics
  pub = rospy.Publisher('obstacle', Float32, queue_size=10)
  pub2 = rospy.Publisher('adelantamiento', Float32, queue_size=10)
  pub3 = rospy.Publisher('verificacion', Float32, queue_size=10)
  # We initialize the node
  rospy.init_node('Master', anonymous=True)
  rate = rospy.Rate(10)
  mover = Master()
  rospy.spin()
```

Figura 20. Main del código. [Realización propia]

Por último, en el apartado Main del código de la *Figura 19*, es cuando se crean los tres publicadores mencionados y se inicializa el nodo.

4.2 Dynamic_box

A continuación, se analizará el segundo de los códigos diseñados para el trabajo. Este código, es independiente a los otros dos y su objetivo principal es controlar el movimiento de uno de los obstáculos a lo largo del circuito.

Para la evitación de obstáculos estáticos, se recurrió a la utilización de cubos y cilindros que simularían vehículos estacionados o cualquier otro tipo de obstáculos. Es por ello por lo que, para la evitación dinámica de los mismo, lo más sencillo es aprovechar el ejecutable de uno de esos obstáculos estáticos, en este caso, uno de los cubos, y modificarlo para implementarle el pure pursuit y hacer que este se mueva alrededor del circuito a velocidad constante, simulando un vehículo que circula a menor velocidad que el principal.

Para lograr esto, hay que seguir la filosofía que se emplea con el coche, es decir, subscribirse al topic de odometría para obtener la localización del cubo y a partir de ahí hacer los cálculos y publicar una velocidad y un giro en el topic adecuado para que el cubo se desplace.

```
import numpy as np
import rospy
import re
import math
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
from tf.transformations import euler_from_quaternion
```

Figura 21. Librerias Dynamic_box. [Realización propia]

Como se puede ver en la *Figura 20*, como librerías adicionales necesitaremos la librería "*re*" que es una librería que permite localizar patrones de texto descritos con una sintaxis formal, es decir, va a ayudar a obtener los puntos de los txt donde se definen las posiciones donde se establece que pase el vehículo, para poder así almacenarlos en un array para su posterior procesamiento.

La librería math, permitirá realizar operaciones matemáticas complejas tipo seno y coseno que se emplearán en el cálculo del giro del vehículo o el objeto.

Por último, incluimos los mensajes tipo Twist que van a ser del tipo que se publicarán para el desplazamiento del objeto.

4.2.1 Funciones

Una vez incluidas las librerías, se establecen las funciones en las que se apoyará el código principal para su correcto funcionamiento, tal y como puede verse en la *Figura 21*.

```
#This code manage the movement of the dynamic obstacle that simulates a car o
# The Pure pursuit code is the same than our vehicule's
def PurePursuit(x_def, y_def, distancia, phi_error, self):
    "calculate de steering angle for the car"
    global cont, conty, len1, j
    if distancia > 0.3 and j == 0:
      i = 0
      px = math.cos(phi_error)*x_def + math.sin(phi_error)*y_def
      py = -math.sin(phi_error)*x_def + math.cos(phi_error)*y_def
      giro = 1.5*math.atan2(py,px)
      print("giro necesario: ", giro)
      self.drive_msg.linear.x = 0.3
      self.drive_msg.angular.z = giro

    else:
      print("punto alcanzado")
      print("siguiente punto")
      i = 1
      len = len1 - 1
      if conty == len:
        cont = 0
        conty = 0
      else:
        cont += 2
      j = 0
    return i
# This function takes the target points form the txt
def extraer_punto(filename):
    with open(filename, 'r') as f:
      Puntos = f.read()
    Puntos_1 = [float(s) for s in re.findall(r'-?\d+\.?\d*', Puntos)]
    return Puntos_1
```

Figura 22. Pure pursuit y extracción de puntos. [Realización propia]

En primer lugar, se establece la función del pure pursuit, que toma como variables de entrada la posición del vehículo en todo momento, la distancia al punto objetivo y el error en el ángulo de giro que lleva y con ello, calcula y publica la velocidad lineal y el giro necesario.

Es importante mencionar, que este es el único punto en todo el código donde se publica el giro del objeto y su velocidad para así evitar dos órdenes de giro contradictorias.

La variable "*i*" de retorno se activa cuando el punto objetivo se ha alcanzado, resulta útil para saber cuándo hay que mirar el siguiente punto en el txt.

Por otro lado, se establece la función "*extraer_punto*" que toma como entrada el nombre del txt donde se almacenan los puntos objetivos, los extrae, y los almacena en un array que devuelve como retorno.

4.2.2 Definición del nodo

Siguiendo el mismo esquema que el código de detección, se define el nodo con los topics en los que se quiere subscribir y publicar.

```
class Dynamic_box:


  def __init__(self):
    DRIVE_TOPIC = "/dynamic_box/cmd_vel"
    ODOMETRY_TOPIC = "/dynamic_box/odom"


    ################################################
    # TODO >>>
    self.drive_msg = Twist()


    ################################################

    ################################################
    # <<< TODO
    ################################################

    # Initialize a publisher for drive messages
    self.drive_pub = rospy.Publisher(
        DRIVE_TOPIC,
        Twist,
        queue_size=1)

    # Subscribe to the Odometry data
    rospy.Subscriber(
        ODOMETRY_TOPIC,
        Odometry,
        self.position)
```

Figura 23. Nodo Dynamic_box. [Realización propia]

En el caso del código observable en la *Figura 22,* hará falta subscribirse al topic odometry, pero esta vez, asociado al Dynamic_box y publicar en el tópico cmd_vel del Dinamic_box.

Puede surgir al lector aquí la duda de cómo conocer qué topic de posición y de velocidad están activos y asociados al Dinamic_box. Para resolver esto, ROS ofrece un comando que es rostopic_list que mostrará todos los topics activos cuando se ejecuta en una ventana de comandos.

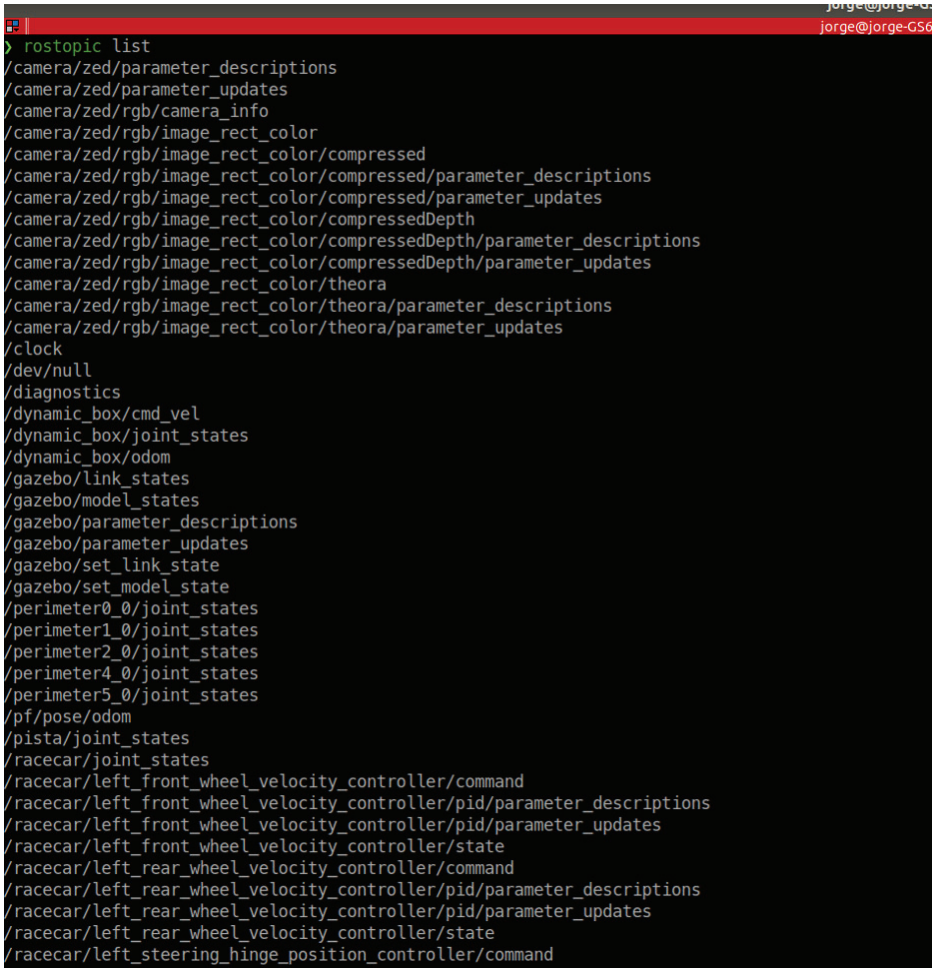


Figura 24. Nodos activos. [Realización propia]

Como puede verse en la *Figura 23*, en este caso, solo será necesario incluir un callback para el topic odometry tal y como se muestra a continuación.

```
  def position(self, msg):
    ################################################
    # TODO >>>
    # Take car position from Odometry topic
    # Take target position from txt
    # Make the car move to target position
    ################################################
    global i, Target_x, Target_y, cont, yaw, roll, pitch, conty, camino

    Pose_car_x = msg.pose.pose.position.x
    Pose_car_y = msg.pose.pose.position.y
    orientation_q = msg.pose.pose.orientation
    orientation_list = [orientation_q.x, orientation_q.y, orientation_q.z, orientation_q.w]
    (roll, pitch, yaw) = euler_from_quaternion (orientation_list)


    if i == 1:

      Target_x = camino[cont]
      conty = cont + 1
      Target_y = camino[conty]

    print("Punto objetivo en x: ", Target_x)
    print("Punto objetivo en y: ", Target_y)
    x_def = Target_x - Pose_car_x
    y_def = Target_y - Pose_car_y

    dist = math.sqrt(pow(x_def,2) + pow(y_def,2))
    print("distancia al punto objetivo: ", dist)

    i = PurePursuit(x_def, y_def, dist, yaw, self)
    self.drive_pub.publish(self.drive_msg)



    ################################################
    # <<< TODO
    ################################################

if __name__ == "__main__":
  # First, we initialize the node
  rospy.init_node("Dynamic_box")
  # In this case, we only take the main road target points
  Puntos_obj = extraer_punto('TargetPoints.txt')
  len1 = len(Puntos_obj)
  camino = Puntos_obj
  mover = Dynamic_box()
  rospy.spin()
```

Figura 25. Odometry callback and main. [Realización propia]

El callback que puede observarse en la *Figura 24*, toma la posición y la orientación del obstáculo y la transforma al sistema de Euler para poder calcular después el giro correctamente. También define el punto objetivo del txt según se va alcanzando y genera la variable distancia al punto objetivo necesaria para controlar cuando se alcanza el punto deseado.

Justo después de ejecutarse la función pure pursuit, se publican los mensajes de velocidad y giro, es por ello por lo que se incluye el self en la función, para que se vaya actualizado en cada ejecución de la función.

Por último, se define el main de la función que, al igual que el anterior, inicializa el nodo. Adicionalmente, este main extrae los puntos objetivos del txt y los almacena en un array. Esto se hace aquí y no en otro lugar porque así solo se tienen que extraer los puntos una vez y se está ahorrando procesamiento.

4.3 Safety

En este caso, se analizará el nodo principal. Este nodo, tiene la función de ejercer de master sobre el resto, es decir, es un nodo que se subscribe a los topics generados por los otros y actúa en consecuencia.

Se puede decir que este nodo es el cerebro del conjunto ya que controla lo que está pasando en todo momento, los otros, por el contrario, solamente realizan tareas concretas y son “ciegos” al resto de situaciones que se generan. Por ejemplo, el nodo detección se encarga de controlar si hay o no un obstáculo frente al vehículo, pero en ningún momento sabe si el coche está moviéndose o si está circulando correctamente.

El nodo safety, por el contrario, es el que gestiona el movimiento del vehículo aprovechándose de la información que le proporcionan los topics a los que se subscribe. En todos los diseños es importante que haya un master que sea consciente de todo lo que sucede para garantizar que todos trabajan de manera coordinada y que no se contradicen unos a otros.

Esto hace a su vez, que este sea el código más crítico ya que realmente es el que dirige al vehículo por los carriles, evitando los obstáculos según le llega la información de los diferentes topics.

Comprender este programa es comprender el cerebro del trabajo, es por ello por lo que se van a analizar las características esenciales del mismo. Como algunas funciones y elementos ya se han explicado en los códigos anteriores, se va a obviar su explicación en este punto.

4.3.1 Cabecera y funciones

Como cabecera, se deberán incluir todas las librerías y tipos de mensajes que se han mencionado en los programas anteriores ya que, como se ha indicado, este código es el más importante y necesitará de todas ellas para su correcto funcionamiento tal y como puede verse en la *Figura 25*.

```
#!/usr/bin/env python2
import numpy as np
import rospy
import re
import math
from std_msgs.msg import Float32
from nav_msgs.msg import Odometry
from ackermann_msgs.msg import AckermannDriveStamped
from tf.transformations import euler_from_quaternion
from sensor_msgs.msg import LaserScan
```

Figura 26. Cabecera safety. [Realización propia]

Una vez hecho esto e inicializado las distintas variables, se definen las funciones en las que se va a apoyar el programa. Como se podrá observar, muchas de ellas se reutilizan en los otros programas, esta es una de las ventajas de trabajar con funciones, las llamas solo cuando las necesitas y si poseen entradas genéricas, pueden ser reutilizadas.

```
def PurePursuit(x_def, y_def, distancia, phi_error, self):
    "calculate de steering angle for the car"
    global cont, conty, len1, len2, j, parada, giro
    if distancia > 0.3 and j == 0:
      i = 0
      px = math.cos(phi_error)*x_def + math.sin(phi_error)*y_def
      py = -math.sin(phi_error)*x_def + math.cos(phi_error)*y_def
      giro = 1.7*math.atan2(py,px)
      # print("giro necesario: ", giro)
      # The car will not move in case we have an emergency stop
      if parada == 0:
        self.drive_msg.drive.speed = 0.7
        self.drive_msg.drive.steering_angle = giro
      else:
        self.drive_msg.drive.speed = 0
    else:
      # print("punto alcanzado")
      # print("siguiente punto")
      i = 1
      len = len1 - 1
      if conty == len:
        cont = 0
        conty = 0
      else:
        cont += 2
      j = 0
    return i
```

Figura 27. Pure pursuit safety. [Realización propia]

Como se puede observar en la *Figura 26*, esta función es similar a la que se implementó en el Dynamic_box. Su objetivo es calcular el giro necesario en el vehículo para orientarlo hacia el punto objetivo. En este caso, se han tenido que añadir algunas líneas de código, ya que hay que considerar el hecho de que tenemos 2 carriles posibles. Además, en esta función es donde se publica la velocidad y el giro del vehículo y en ningún sitio más, para evitar publicaciones contradictorias. Por último, se puede ver cómo hay una función de protección que pone la velocidad a 0 para evitar colisiones.

```
# Here we get the points for the road of the txt files
def extraer_punto(filename):
    with open(filename, 'r') as f:
      Puntos = f.read()
    Puntos_1 = [float(s) for s in re.findall(r'-?\d+\.?\d*', Puntos)]
    return Puntos_1

# Here we decide if we must continue or not on the main road
def adelantar(lidarC):
    global camino, camino2, camino1, j, alt
    if camino1 == 1:
        if lidarC == 1:
          # print("Tomando ruta alternativa")
          j = 1
          camino = Puntos_alt
          camino2 = 1
          camino1 = 0
          alt = 0

# This function decides if we must do an emergency stop
def parada_emergencia(alt, msg):
    global lidarC, lidarT, giro, parada, camino2
    print("Punto alternativo: ", alt)
    if giro > 0.1 and alt > 1 and alt < 5 and camino2 == 1:
      if np.min(msg.ranges[540:1080]) < 0.3:
        parada = 1
        print("PARADA DE EMERGENCIA")
      else:
        parada = 0
    elif giro < 0.1 and alt > 1 and alt < 5 and camino2 == 1:
      if np.min(msg.ranges[400:1080]) < 0.5:
        parada = 1
        print("PARADA DE EMERGENCIA")
      else:
        parada = 0
```

Figura 28. Funciones Safety. [Realización propia]

La función "*extraer_punto*" que se puede ver en la *Figura 27*, es exactamente igual a la que se emplea en el código Dynamic_box. Por otro lado, la función "adelantar" se encarga de comprobar, en función de la información recibida, si es necesario cambiar de carril e iniciar la maniobra de adelantamiento.

Por último, la función "*parada_emergencia*", es la encargada de detectar un bloqueo en ambos carriles y mandar la orden para que la función "*PurePursuit*" establezca la velocidad a cero hasta que sea posible seguir circulando.

En esta función, se considera también la condición de la conducción en cada momento, es decir, no se estará en la misma situación si el coche está circulando en línea recta que en una curva cerrada, es por ello por lo que se han incluido modificaciones que adaptan la respuesta del vehículo en función del ángulo de giro que lleve.

4.3.2 Definición del nodo

Una vez definidas las funciones, el siguiente paso es crear el nodo y definir los distintos topics a los que se quiere subscribir ya sea como lector o como publicador. Al ser el nodo master, este nodo no va a publicar en ningún otro, solamente va a trabajar con la información que recibe de los topics.

```
class Safety:


  def __init__(self):
    DRIVE_TOPIC = "/vesc/high_level/ackermann_cmd_mux/input/nav_0"
    ODOMETRY_TOPIC = "/vesc/odom"
    LIDAR_TOPIC = "/scan"


    ################################################
    # TODO >>>
    self.drive_msg = AckermannDriveStamped()


    ################################################

    ################################################
    # <<< TODO
    ################################################

    # Initialize a publisher for drive messages
    self.drive_pub = rospy.Publisher(
        DRIVE_TOPIC,
        AckermannDriveStamped,
        queue_size=1)

    # Subscribe to the Odometry data
    rospy.Subscriber(
        ODOMETRY_TOPIC,
        Odometry,
        self.position)
    # Subcribe to obstacle topic
    rospy.Subscriber("obstacle",
        Float32,
        self.obstacle)

    # Subcribe to adelantamiento topic
    rospy.Subscriber("adelantamiento",
        Float32,
        self.adelantamiento)

    # Subcribe to verificacion topic
    rospy.Subscriber("verificacion",
        Float32,
        self.verificacion)
    # Subscribe to the laser scan data
    rospy.Subscriber(
        LIDAR_TOPIC,
        LaserScan,
        self.callback)
```

Figura 29. Nodo safety. [Realización propia]

Como se puede ver en la *Figura 28*, a parte de los tres nodos que se conocen de los programas anteriores (Drive_topic, Odometry y Lidar_topic), el nodo está subscrito a un topic llamado "*obstacle*", otro "*adelantamiento*" y un último llamado "*verificación*". Estos topics, son los generados y publicados en el nodo Detección luego, ya se puede ver cómo va a haber una comunicación unidireccional constante entre ambos nodos.

```
def callback(self, msg):
  ####################################################
  # TODO >>>
  # Check the LIDAR for the emergency stop
  #
  ####################################################
  global giro, lidarC

  if giro > 0.1:
    curva = 1
  else:
    curva = 0
  if np.min(msg.ranges[400:1080]) < 1 and curva == 1:
    lidarC = 1
    adelantar(lidarC)
  parada_emergencia(alt, msg)
  print("Parada: ", parada)
  ####################################################
  # <<< TODO
  ####################################################
```

Figura 30. Callback lidar. [Realización propia]

Analizando la *Figura 29*, se puede ver como este callback se encarga de evitar colisiones junto con la función "*parada_emergencia*" de manera independiente al resto de funciones, es decir, evitar una colisión es la máxima prioridad luego, si se genera dicha situación de peligro, el vehículo se detendrá independientemente de lo que digan otras funciones.

```
def position(self, msg):
  ################################################
  # TODO >>>
  # Take Robot position from Odometry topic
  # Take target position from txt
  # Make the robot move to target position
  ################################################
  global i, Target_x, Target_y, cont, yaw, roll, pitch, conty, camino, camino2, camino1, alt

  Pose_car_x = msg.pose.pose.position.x
  Pose_car_y = msg.pose.pose.position.y
  orientation_q = msg.pose.pose.orientation
  orientation_list = [orientation_q.x, orientation_q.y, orientation_q.z, orientation_q.w]
  (roll, pitch, yaw) = euler_from_quaternion (orientation_list)

  #print("Posicion coche en x: ", Pose_car_x)
  #print("Posicion coche en y: ", Pose_car_y)
  #print("Orientacion del coche en z: ", yaw)

  if i == 1:
    Target_x = camino[cont]
    conty = cont + 1
    Target_y = camino[conty]
    if camino2 == 1:
      alt += 1
  # print("Punto objetivo en x: ", Target_x)
  # print("Punto objetivo en y: ", Target_y)
  x_def = Target_x - Pose_car_x
  y_def = Target_y - Pose_car_y
  # Minimun distance to the target point
  dist = math.sqrt(pow(x_def,2) + pow(y_def,2))
  # print("distancia al punto objetivo: ", dist)

  i = PurePursuit(x_def, y_def, dist, yaw, self)
  self.drive_pub.publish(self.drive_msg)



  ################################################
  # <<< TODO
  ################################################
```

Figura 31. Callback Odometry. [Realización propia]

El callback del topic Odometry que se puede ver en la *Figura 30*, se va a encargar de controlar el desplazamiento del vehículo a lo largo del circuito. De manera similar que el del nodo Dynamic_box, calcula el punto objetivo en x e y, la distancia al punto objetivo y la orientación del vehículo y se lo manda a la función "*PurePursuit*", que es la encargada de orientar correctamente al vehículo y darle velocidad.

```
def obstacle(self, data):
  ##################################################
  # TODO >>>
  # This take de information form the Obstacle topic
  # And the let the function adelantar decides what to do
  #
  ##################################################
  global camino, lidarC, j, camino2, alt, curva
  if curva == 0:
    lidarC = data.data
  #print(camino)
  adelantar(lidarC)


  ##################################################
  # <<< TODO
  ##################################################
```

Figura 32. Callback obstacle. [Realización propia]

En este caso, el programa toma la información del topic “obstacle”, que se puede ver en la *Figura 31*, correspondiente a los obstáculos frontales y llama a la función “adelantar” para que interprete si es necesario no iniciar la maniobra de adelantamiento.

```
def adelantamiento(self, info):
  ##################################################
  # TODO >>>
  # This thakes the information from the adelantamiento topic
  # And then calculate if we can get back to the main road
  #
  ##################################################
  global camino, lidarC, lidarT, j, camino1, camino2, alt, volver
  lidarT = info.data
  #print(camino)
  if camino2 == 1:
    if lidarC == 0 and lidarT == 0 and alt == 4 and volver == 0:
      # print("Desvio al carril principal")
      camino = Puntos_obj
      j = 1
      alt = 0
      volver = 0
      camino2 = 0
      camino1 = 1


  ##################################################
  # <<< TODO
  ##################################################
```

Figura 33. Callback adelantamiento. [Realización propia]

Si se analiza la *Figura 32*, se puede ver como este callback toma la información correspondiente a los posibles obstáculos laterales traseros, lo que puede ser de utilidad para detectar obstáculos en curvas o para determinar si es posible volver al carril principal tras haber realizado un adelantamiento.

Esto no quiere decir que esta parte del programa esté verificando que el carril principal está libre porque este programa funciona constantemente. Simplemente, actúa como primera condición para poder iniciar la vuelta al carril principal.

```
def verificacion(self, info):
  ################################################
  # TODO >>>
  # This takes the information from the verificacion topic
  # And then change the value of volver variable
  # With this, the adelantamiento function can decide if we get back to the main road or not
  ################################################
  global camino2, alt, volver, lidarC
  # print("verificacion: ", info.data)
  if alt >= 2 and camino2 == 1:
    if info.data == 1:
      volver = 1
      alt = 0
    else:
      volver = 0


  ################################################
  # <<< TODO
  ################################################
```

Figura 34. Callback verificación. [Realización propia]

El callback “verificación” que se puede ver en la *Figura 33*, es uno de los callbacks más importantes de todo el programa ya que, su función, es la de comprobar de manera activa mientras se está adelantando, que el carril principal está libre para volver a él o no.

Como se mencionó anteriormente, hay una primera comprobación para volver al carril principal, que es la de no detectar obstáculos en la parte trasera con la variable “lidarT”, sin embargo, esta condición no es suficiente ya que el obstáculo en el carril principal puede ir avanzando a la vez.

Es por ello por lo que, este código, mete una protección adicional que se basa en obligar al vehículo continuar durante 4 puntos objetivos más en el carril secundario, y en ese momento volver a realizar la comprobación de si el lateral

derecho del vehículo está libre. De ser así, se volvería al carril principal, si no, se prolongaría durante otros 4 puntos más y así hasta poder volver en condiciones de seguridad.

```
if __name__ == "__main__":
  #initialize the node
  rospy.init_node("Safety")
  #Take the target points of the main road
  Puntos_obj = extraer_punto('TargetPoints.txt')
  len1 = len(Puntos_obj)
  camino = Puntos_obj
  camino1 = 1
  #Take the target points of the alternative road
  Puntos_alt = extraer_punto('AlternativeRoad.txt')
  len2 = len(Puntos_alt)
  print(Puntos_obj)
  mover = Safety()
  rospy.spin()
```

Figura 35. Main Safety. [Realización propia]

El último paso, que puede verse en la *Figura 34*, es definir el main del programa en el que se inicializa el nodo, se extraen los puntos que definen el carril principal y se almacenan en un array, se extraen también los puntos que definen el carril alternativo y se almacenan en otro array distinto y se ejecuta el nodo.

Capítulo 5 . Resultados y simulación

El objetivo de este capítulo es verificar el correcto funcionamiento de todo lo expuesto en los capítulos anteriores mediante las simulaciones y las pruebas experimentales. Al tratarse de un diseño compartimentado en nodos, para las simulaciones se deberán lanzar de manera simultánea varios códigos.

Antes de comenzar con las simulaciones, es necesario aprender a lanzar de manera correcta el entorno donde se va a realizar la simulación para lo que, se necesitarán, todos los paquetes que modelan el vehículo y el entorno, realizando algunos ajustes para este proyecto.

Como se mencionó anteriormente, se parte con el modelo del vehículo y con un mapa proporcionados por los organizadores de la competición. Sin embargo, para este trabajo se ha preferido emplear el modelo del vehículo que está en la escuela ya que, como finalmente se canceló la competición, puede resultar más útil para las pruebas experimentales.

En el *Anexo 3. Entorno de simulación* de esta memoria, se deja una guía para la correcta generación del entorno de trabajo y los enlaces para clonar los repositorios necesarios, tanto de diseño propio, como de GitLab.

En primer lugar, se seguirán los pasos explicados en el capítulo 4 para lanzar el entorno de simulación, junto con el coche y los obstáculos. En referencia a los obstáculos, al tener tres en un principio distribuidos por el circuito, se pueden ir moviendo con el ratón a distintos puntos del carril durante la simulación y así comprobar la respuesta del vehículo en condiciones activamente variables.

Por otro lado, se realiza una segunda simulación en la que se quitan dos de los tres obstáculos, dejando solo el que va a comportarse como obstáculo dinámico. Este tiene su propio código que habrá que lanzar en primer lugar para que comience a avanzar por el circuito antes que el propio vehículo. A continuación, se lanza el código de detección y el de safety (en ese orden) para que comience el vehículo a procesar la información del LIDAR y a avanzar por el circuito.

Es importante que en primer lugar se lance el código del objeto porque este se desplaza a menor velocidad. De no hacerlo así, habría que esperar a que el vehículo diese una vuelta completa para ver una maniobra de adelantamiento. También aclarar aquí que el objeto dinámico no tiene incorporados ningún tipo de sensores de detección, es por ello por lo que se retiran los otros dos objetos del circuito para realizar esta simulación, ya que no sería capaz de esquivarlos.

Una última opción para este caso es ir moviendo los otros dos obstáculos restantes según interese durante la simulación, para comprobar que el vehículo se mantiene en el carril secundario hasta que el principal está libre, pero habría que asegurarse de no interferir la trayectoria del obstáculo dinámico ya que, como se ha dicho, no tiene algoritmo de evitación de obstáculos.

Dicho esto, se van a comenzar las simulaciones según los objetivos de este trabajo. En primer lugar, se comprueba que realiza correctamente la evitación de obstáculos estáticos.

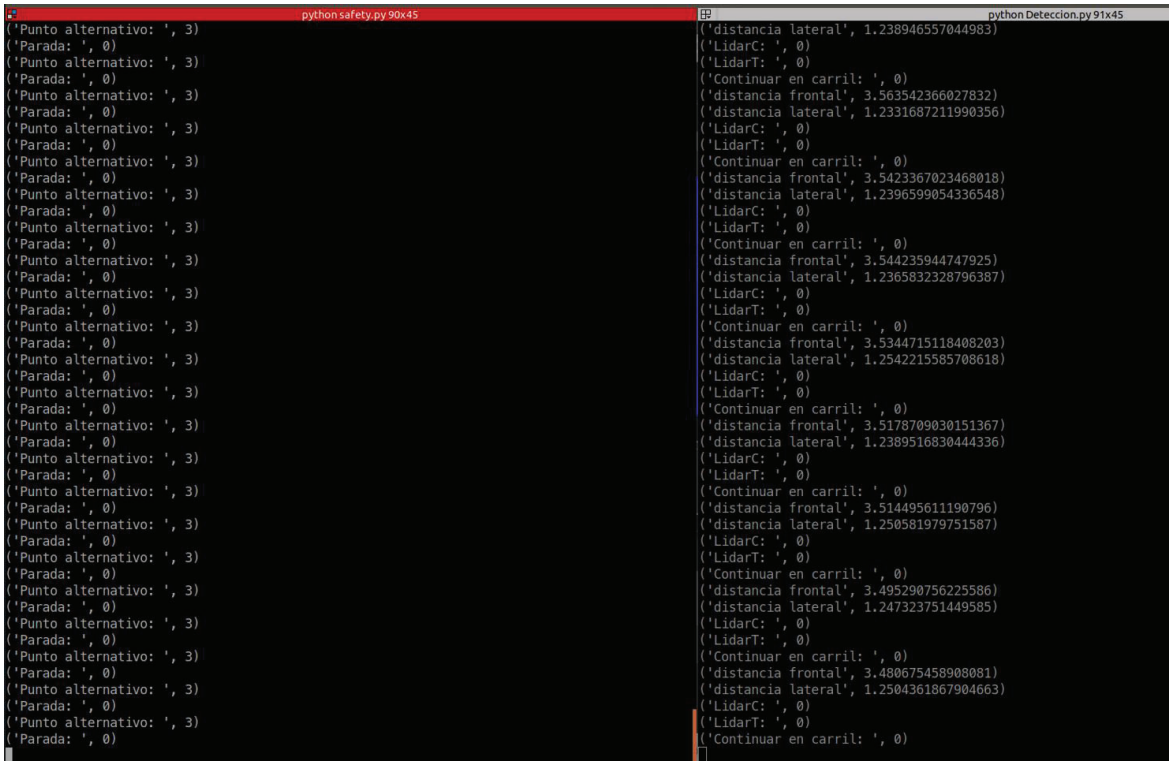


Figura 36. Obstáculos estáticos. [Realización propia]

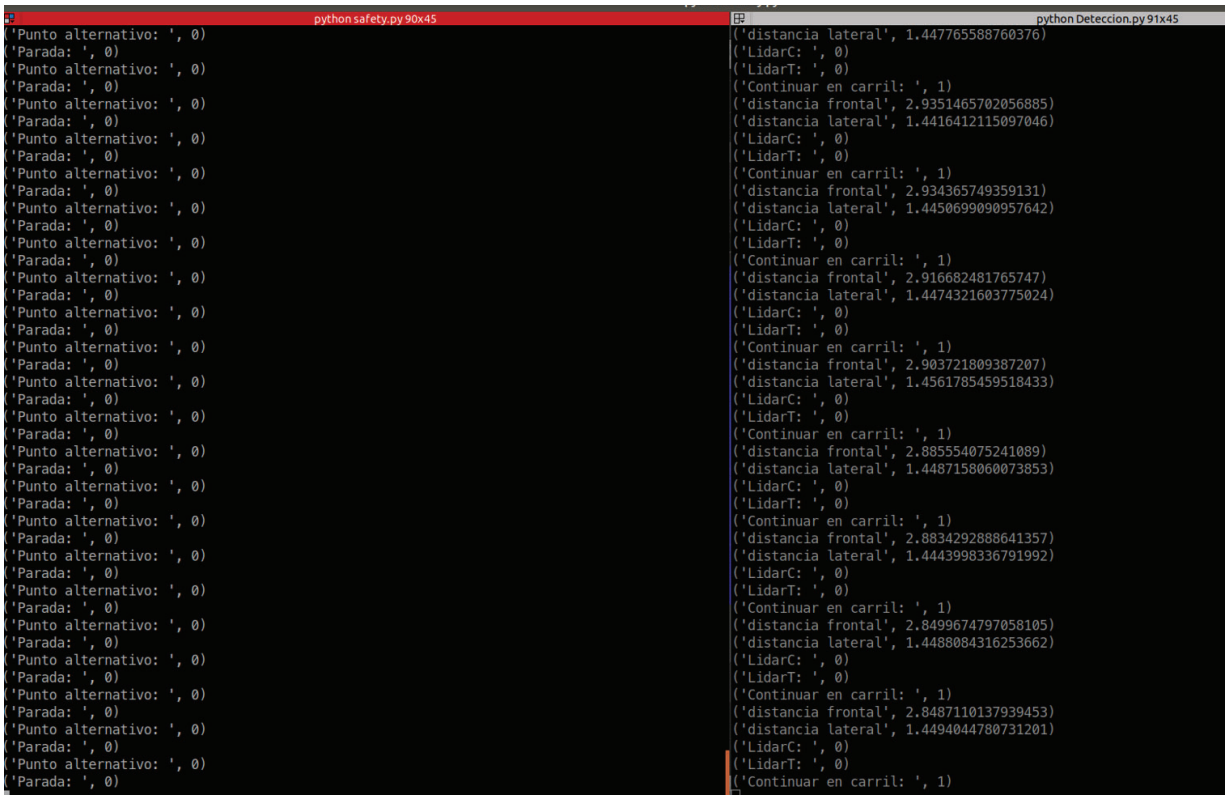


Figura 37. Obstáculos estáticos_2. [Realización propia]

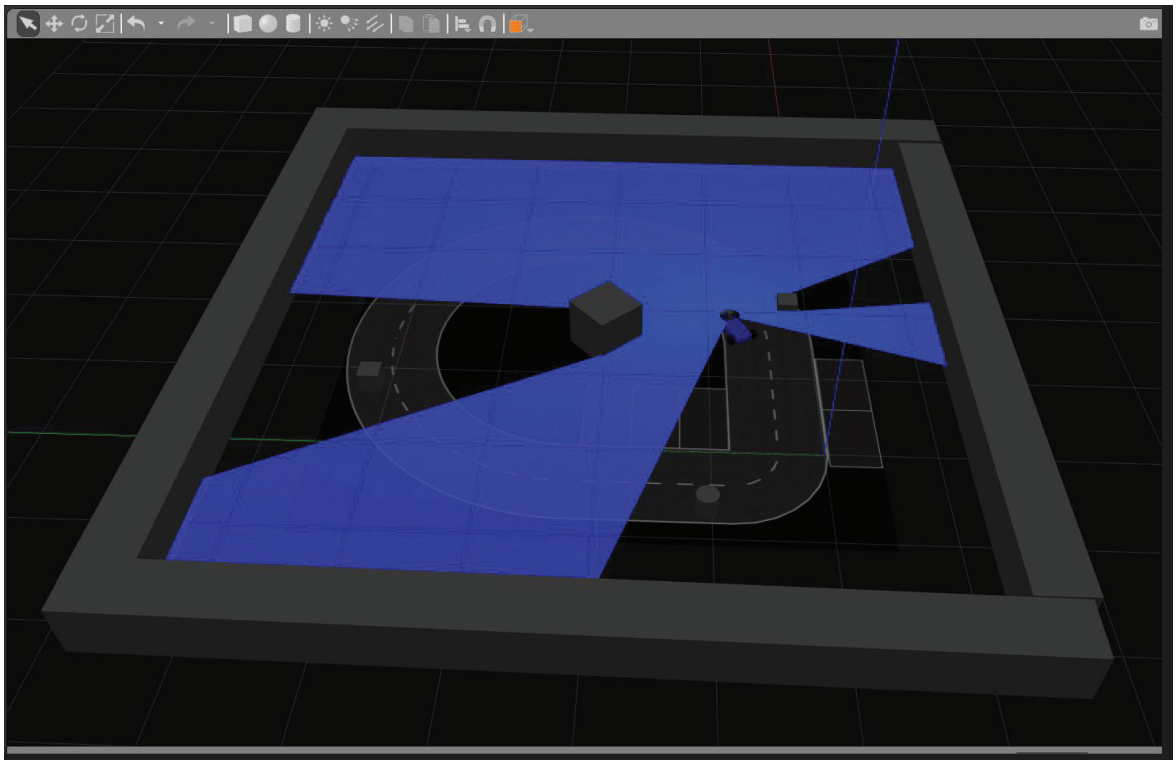


Figura 38. Simulación 1. [Realización propia]

En la *Figura 35 y Figura 36*, se puede ver como el vehículo es capaz de detectar sin problemas los obstáculos, mediante las variables “LidarT” y “LidarC”, además tenemos también otra variable que nos indica si puede o no volver al carril principal (continuar en carril). “Distancia frontal” y “distancia lateral” indican a qué distancia se encuentran los obstáculos detectados más cercanos al vehículo.

A parte de estas variables de control, hay otras muchas que no se han incluido para no llenar de variables la ventana de comando ya que pueden liar al lector si no está habituado al código. Estas variables están en el código comentadas, luego se pueden incluir en cualquier momento.

Ahora que se ha superado de manera satisfactoria el primer de los tres objetivos del trabajo, se puede pasar a la simulación del segundo caso considerado, evitación de obstáculos dinámicos.

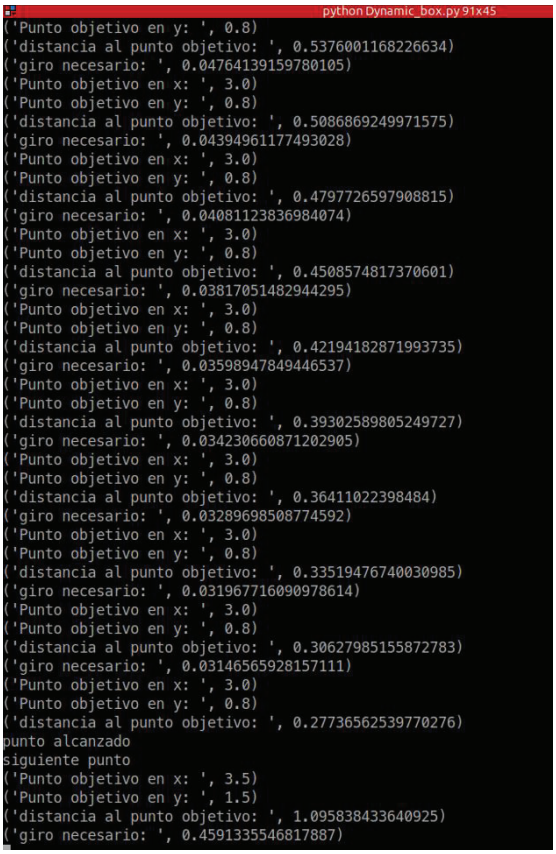


Figura 39. Obstáculo dinámico. [Realización propia]

Como se puede ver en la *Figura 38*, lo primero de todo es lanzar el código del obstáculo dinámico para que este comience a circular por el carril principal siguiendo la estrategia Pure_Pursuit ya explicada. Aquí se aprecia cómo esta estrategia se basa en ir de un punto a otro midiendo el giro necesario para orientar el obstáculo hacia el punto y la distancia al mismo.

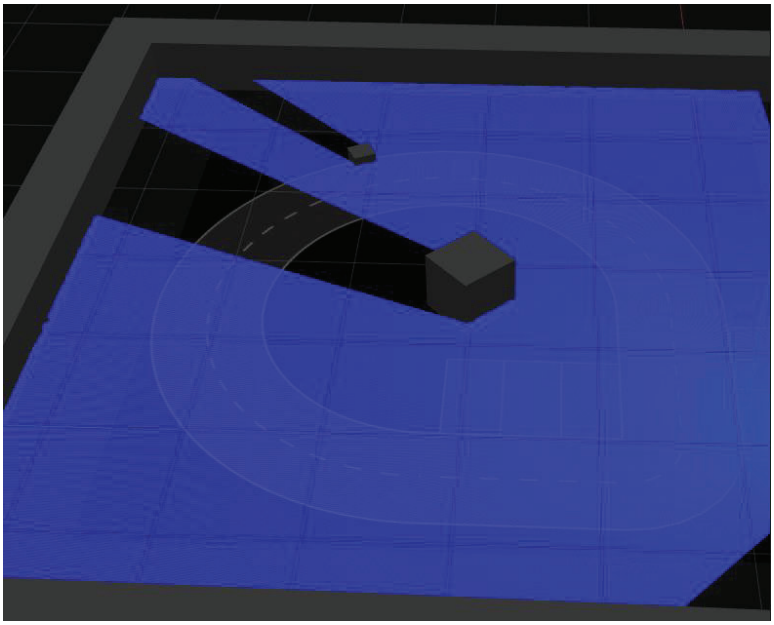


Figura 40. Obstáculo dinámico_2. [Realización propia]

Lo que se observa en la *Figura 39*, es el obstáculo dinámico circulando por el circuito de manera autónoma, siguiendo el código lanzado en la *Figura 38*. Una vez comprobado que funciona correctamente, se puede lanzar el código del vehículo y comprobar que realiza la maniobra de adelantamiento.

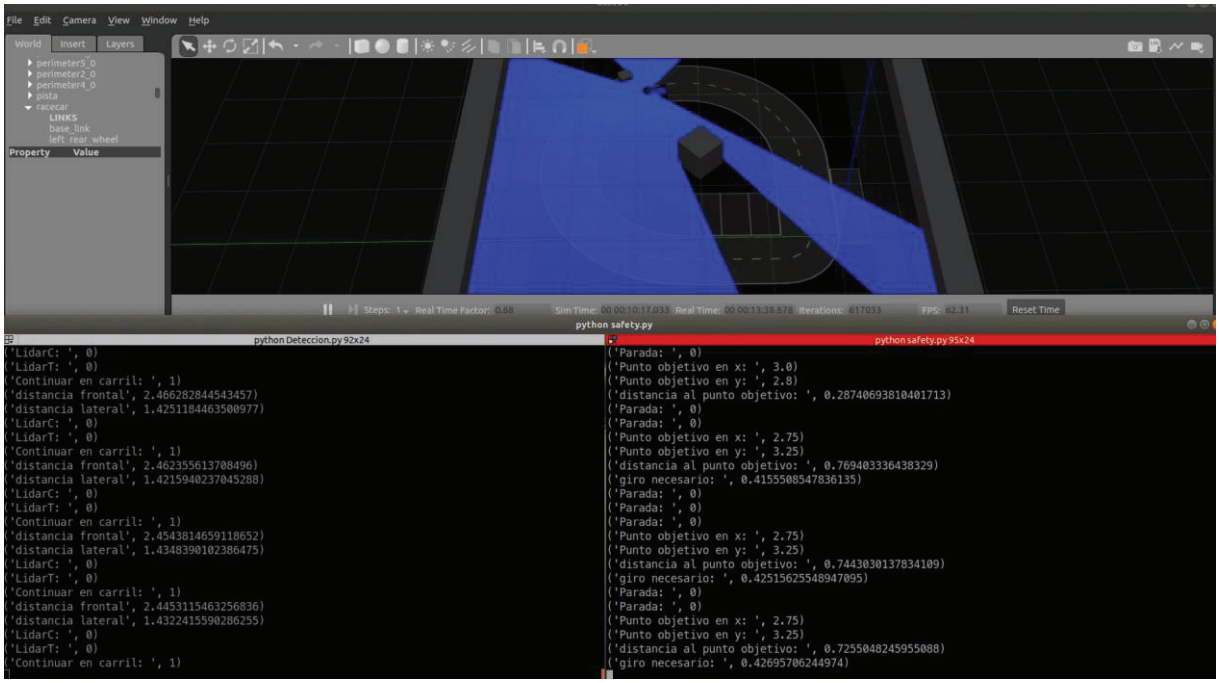


Figura 41. Adelantamiento dinámico. [Realización propia]

Con esto, queda superado el segundo de los tres objetivos del trabajo, luego se puede pasar a la simulación del último caso, la evitación de colisiones.

Este caso, se ha considerado como una situación extrema dentro de la conducción autónoma, es decir, una situación en la que no se puede adelantar, pero tampoco se puede volver al carril porque este está completamente bloqueado. Se puede pensar esta situación como dos coches adelantando que se ven obligados a frenar, una situación de tráfico intenso o un accidente que bloquea la vía, entre otros. Para simular esta situación, se combinan las dos anteriores de tal forma que se va a tener al obstáculo dinámico circulando por la vía, y en un momento dado, el vehículo iniciará la maniobra de adelantamiento siendo en ese preciso instante cuando se colocará un obstáculo estático en el carril secundario de tal forma que no podrá adelantar, pero tampoco podrá volver, luego no lo quedará otra opción que detenerse hasta que se retire el obstáculo estático.

Este código está preparado para que una vez que vuelva a estar libre el carril, el vehículo reanude la marcha de inmediato. Todo esto no se puede apreciar de

una manera tan clara en la *Figura 40 y Figura 41*, es por ello por lo que se incluirán enlaces a vídeos en los anexos de la presente memoria para poder tener una referencia visual de lo que se ha explicado.

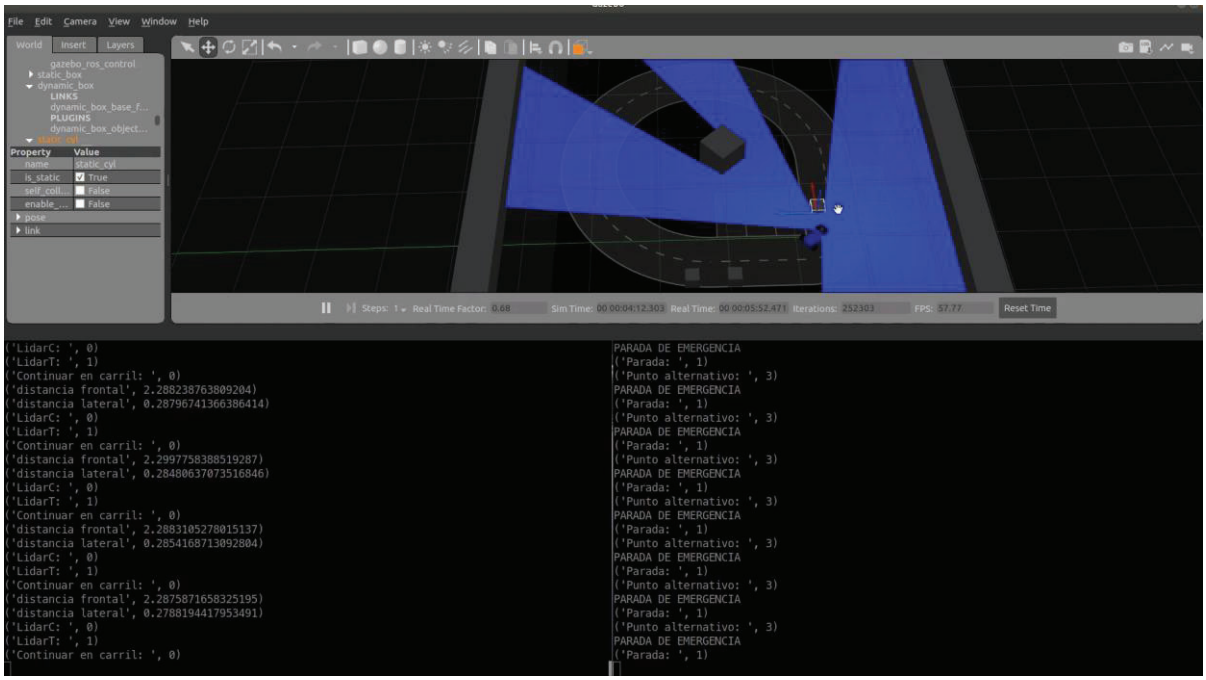


Figura 42. Evitación de colisiones. [Realización propia]

Con esto queda comprobado el correcto funcionamiento de todo el diseño, luego se pueden dar por satisfactorios los resultados y por finalizado el presente trabajo. En los anexos del presente documento, se dejan los pasos a seguir para realizar las pruebas experimentales con el vehículo UMA racecar, disponible en el departamento de sistemas y automática de la Escuela de Ingenierías Industriales de Málaga.

Capítulo 6 . Conclusiones y líneas futuras

Una vez finalizado el desarrollo y simulación del presente trabajo, se van a recoger una serie de conclusiones fruto de la experiencia vivida con la realización del proyecto.

Además, se recogen también posibles líneas de mejora que den lugar a futuros trabajos que partan de lo que se ha desarrollado en este, con el objetivo de mejorarlo o complementarlo.

6.1 Conclusiones

Me gustaría comenzar comentando que este trabajo se partió con un total desconocimiento de ROS y el entorno de simulación Gazebo. Además, nunca había trabajado con el sistema operativo Ubuntu y tenía conocimientos limitados de Python.

Por ello, la mayor dificultad la encontré en aprender a dominar las herramientas. Una vez había comprendido el funcionamiento de la filosofía de ROS, gracias al libro [20] *A gentle introduction to ROS,* pude comenzar a estructurar las bases de mi trabajo. Mi grado es Ingeniería Electrónica, Robótica y Mecatrónica y yo escogí la intensificación en Mecatrónica que es la rama que se orienta a la automatización de vehículos. Es por ello por lo que considero que en esta mención se debería enseñar algo acerca de ROS y de los lenguajes que se usan actualmente para la programación de estos sistemas. Es cierto que en el grado se aprende C/C++, pero un lenguaje como Python resulta indispensable en este campo y mi trabajo, que va sobre la automatización de un vehículo, se ha hecho con ROS y programando en Python.

Esta es la laguna que he encontrado realizando el proyecto y que habría sido de gran ayuda para haber podido avanzar más rápido en cosas básicas como entender el entorno de trabajo. En una mención como mecatrónica, creo que se debería incluir alguna parte práctica donde se estudiasen estos sistemas que son las bases para comprender la filosofía de la automatización en vehículos.

Si es cierto que he recibido una muy buena orientación de mi tutor de TFG y de otros compañeros y profesores que han hecho posible que llevase el trabajo a término y estoy muy agradecido por ello.

En relación con el proceso de trabajo, creo que ha sido indispensable el dominar el entorno de simulación Gazebo, para ir comprobando a medida que avanzaba que todo funcionaba correctamente. En un primer momento, me centré en conseguir que el vehículo se moviese alrededor del circuito siguiendo uno de los carriles, lo que no fue tarea fácil ya que no se me había enseñado ninguna técnica de seguimiento de carril. Mi tutor me recomendó el uso de la técnica Pure Pursuit, apoyándome en el artículo [11] *Pure-Pursuit Reactive Path Traking for Nonholonomic Mobile Robot with a 2D Laser Scanner,* luego, una vez comprendí en qué se basaba, la fui implementando en el programa hasta que conseguí que funcionase.

En ese periodo (de un par de meses aproximadamente), basé mis esfuerzos en compartimentar los objetivos e ir superándolos uno a uno, lo que fue clave para conseguir sacarlo todo adelante ya que, si no, habría intentado abarcar demasiado de una vez.

Adicionalmente, ha sido indispensable para poder realizar el proyecto, el disponer de los repositorios de la competición en el que se nos facilitaba un modelo del coche de la competición, circuitos reglamentarios y distintas herramientas que me han ayudado a comprender como estaban estructurados los elementos que conformaban el entorno de simulación (ver [18] *Iri_adc_workshop)*.

Entre ellos, en el caso de los obstáculos, se partió de un código que añadía tres obstáculos, uno de ello controlable mediante un panel y que se me ocurrió modificar para que ese obstáculo se moviese de manera autónoma como si fuese un vehículo, sin necesidad del panel. Como se puede ver, de no haber dispuesto del repositorio, no habría caído en el camino que podía tomar para abordar ese reto.

Además, modifiqué algunos archivos para poder combinar los circuitos y elementos de la competición con el modelo del vehículo de la escuela, ya que consideré que es más didáctico el desarrollar estas funcionalidades con el coche de la escuela, pensando en futuras pruebas prácticas presenciales con dicho vehículo.

6.2 Líneas futuras

Aun habiendo superado satisfactoriamente los objetivos del trabajo, aún hay muchas modificaciones y mejoras que pueden encaminar esta línea hacia una conducción autónoma óptima e implementable en la vida real.

En primer lugar, una buena opción sería orientar un trabajo en alcanzar unos objetivos similares a este, pero, en vez de hacerlo empleando el LIDAR, hacerlo con el uso de una cámara y una red neuronal entrenada. Una vez que se haya logrado este objetivo, se podría dejar otro trabajo en combinar el propuesto y el que yo he realizado, ya que se estarían aprovechando las ventajas de ambas herramientas. Por un lado, tendríamos a un LIDAR que tiene una respuesta rápida y un procesamiento bueno para detectar obstáculos improvistos o para el seguimiento vehicular y, por otro lado, tendríamos una cámara que permitiría al vehículo diferenciar si el obstáculo es un peatón, una bicicleta, un vehículo, etc.

Esto último se podría intentar desarrollar con el LIDAR, pero creo que daría unos resultados menos eficientes desde el punto de vista de la seguridad. En relación con el circuito empleado, creo que sería interesante el desarrollar un circuito que siga con la normativa actual en cuanto a tamaño de carril y líneas de carril y que incluyese en él vehículo en movimiento alrededor del circuito, edificios, peatones, semáforos, etc. De esta manera, se tendría un entorno de simulación complejo que permitiese llevar los diseños hasta sus extremos y detectar puntos de mejora o defectos.

Referencias

[1] *Waypoint-Based-2D-Navigation-in-ROS.* (2022). (disponible en: https://github.com/ArghyaChatterjee/waypoint-based-2D-navigation-in-ros).

[2] CARNET. *Autonomous driving challenge*. (2022). (disponible en: https://autonomous-driving-challenge.com/the-car/)

[3] ROS. *Robot Operating System*. (2022). (disponible en: https://www.ros.org/)

[4] Gazebo. *Simulate before you build.* (2022). *(disponible en: https://gazebosim.org/home)*

[5] AutoBild. *Coche autónomo: los seis niveles de conducción*. (2022). (disponible en: https://www.autobild.es/practicos/coche-autonomo-seis-niveles-conduccion-cuales-son-como-funcionan-cuando-llegaran-490005)

[6] Visual Studio Code. *Code editing.* (2022). (disponible en: https://code.visualstudio.com/)

[7] km77. *Conducción autónoma. Niveles y tecnologías.* (2022). (disponible en: https://www.km77.com/reportajes/varios/conduccion-autonoma-niveles)

[8] Junaid Muhammad. *Trajectory Tracking and Stabilization of Nonholonomic Wheeled Mobile Robot Using Recursive Integral Backstepping Control* (2021) (disponible en: https://www.mdpi.com/2079-9292/10/16/1992/htm)

[9] *Guía de funciones de Python con ejemplos* (disponible en: https://github.com/ArghyaChatterjee/waypoint-based-2D-navigation-in-ros).

[10] Jiménez Cuesta, Jaime (2012). *Simulación de vehículos eléctricos ligeros*. (disponible en: https://riunet.upv.es/bitstream/handle/10251/18173/Memoria.pdf?sequence=1).

[11] Morales, Jesús y Martínez, Jorge y Martínez, María y Mandow Anthony (2009*). Pure-Pursuit Reactive Path Traking for Nonholonomic Mobile Robot with a 2D Laser Scanner*. Hindawi. 10.

[12] Cuevas Castañeda, Cristian C. *ROS-Gazebo. Una Valiosa Herramienta de Vanguardia para el Desarrollo De La Robótica* (2015). UNAD. 145-160.

[13] Repiso Polo, Ely (2013). *Técnicas de detección de obstáculos y seguimiento de personas usando fusión de Lidar y otros sensores* (disponible en: https://digital.csic.es/handle/10261/98416).

[14] Regidor Vallcanera, Gabriel (2021). *ROS y algoritmos de navegación para un modelo a escala de un coche autónomo* (disponible en: http://rua.ua.es/dspace/handle/10045/115944).

[15] Del Águila Gómez, Alberto (2019). *Detección y evitación de obstáculos en conducción autónoma en un vehículo a escala 1:10*. UMA. 75.

[16] Gali, Marina (2016). *Implementación de localización geométrica para robots móviles bajo ROS* (disponible en: https://ravelpruebas.uc3m.es/handle/10016/27198).

[17] *Ubuntu 18.04 instalación desde cero* (disponible en: https://cambiatealinux.com/ubuntu-18.04-instalaci%C3%B3n-desde-cero).

[18] *Iri_adc_workshop* (disponible en: https://gitlab.iri.upc.edu/mobile_robotics/adc/adc_2021/iri_adc_workshop).

[19] Autonomous Driving Challenge. *ADC_2021*. (disponible en: https://gitlab.iri.upc.edu/mobile_robotics/adc/adc_2021).

[20] O'Ka.ne, J.M. (2014). *A gentle introduction to ROS*.

[21] Galli, M (2016). *Implementación de localización geométrica para robots móviles bajo ROS*.

[22] Morgan Quigley, Brian Gerkey, William D. Smart (2015). *Programming Robots with ROS, a practical introduction to Robot Operating System*.

[23] Cambil Calderón, José Luis (2022). *Navegación autónoma en circuito de un vehículo a escala con LIDAR utilizando ROS y GAZEBO*.

Anexos

Anexo 1. Códigos de simulación

- Detección

En este anexo, se presenta el código diseñado que se encarga de la detección en interpretación de los obstáculos que se encuentra el vehículo.

También genera las señales que publica para que el nodo Safety actúe en consecuencia.

```
import numpy as np
import rospy
from sensor_msgs.msg import LaserScan
from ackermann_msgs.msg import AckermannDriveStamped
from std_msgs.msg import Float32
lidar = 0
control = 0
mindist = 0
continua = 0
movil = 0
j = 0
# This code Take all the information of the sensors and publish variables in differents topics for the master code
# This function checks all the possible obstacles
def adelantamiento(lidar, control, msg):

  if np.min(msg.ranges[520:560]) < 1:
    lidar = 1
    print("obstaculo frontal")
  else:
    lidar = 0
  if np.min(msg.ranges[596:1080]) < 0.5:
    control = 1
  else:
    control = 0

  return lidar, control

# This function checks if we have a dynamic obstacle and if it is faster than our car
def obstaculo_movil(mindist, msg, j):

  if np.min(msg.ranges[530:550]) < 2 and mindist == 0:
    mindist = np.min(msg.ranges[530:550])
    print("Posible obstaculo ALERTA")
  elif np.min(msg.ranges[530:550]) > 1 and mindist != 0:
    mindist = 0
    j = 1
    print("OBSTACULO MOVIL A MAYOR VELOCIDAD, NO INTENTAR ADELANTAR")
  else:
    j = 0

  return j

# This function checks if we must change or not to the main road (We publish this information in a topic)
def verificar(msg):
  if np.min(msg.ranges[1:535]) < 0.8:
    continua = 1
  else:
    continua = 0
  return continua


class Master:

  def __init__(self):
    LIDAR_TOPIC = "/scan"
    DRIVE_TOPIC = "/vesc/high_level/ackermann_cmd_mux/input/nav_0"
```

```
  ################################################
  # TODO >>>
  self.drive_msg = AckermannDriveStamped()


  ################################################

  ################################################
  # <<< TODO
  ################################################

  # Initialize a publisher for drive messages
  self.drive_pub = rospy.Publisher(
      DRIVE_TOPIC,
      AckermannDriveStamped,
      queue_size=1)
  # Subscribe to the laser scan data
  rospy.Subscriber(
      LIDAR_TOPIC,
      LaserScan,
      self.callback)

def callback(self, msg):
  ################################################
  # TODO >>>
  # Check all the possible obstacle and publish the information
  #
  ################################################

  print("distancia frontal", np.min(msg.ranges[530:550]))
  print("distancia lateral", np.min(msg.ranges[600:1080]))
  # rango = msg.ranges
  # print(len(rango))
  global lidar, control, continua

  [lidar, control] = adelantamiento(lidar, control, msg)
  movil = obstaculo_movil(mindist, msg, j)
  if movil == 1:
    lidar = 0
    print("OBSTACULO MOVIL A MAYOR VELOCIDAD, NO INTENTAR ADELANTAR")
  continua = verificar(msg)
  #rospy.loginfo(lidar)
  print("LidarC: ", lidar)
  print("LidarT: ", control)
  print("Continuar en carril: ", continua)
  pub.publish(lidar)
  pub2.publish(control)
  pub3.publish(continua)

  ################################################
  # <<< TODO
  ################################################


   if __name__ == "__main__":
     # Here we indicate that we are going to publish in three differents topics
     pub = rospy.Publisher('obstacle', Float32, queue_size=10)
     pub2 = rospy.Publisher('adelantamiento', Float32, queue_size=10)
     pub3 = rospy.Publisher('verificacion', Float32, queue_size=10)
     # We initialize the node
     rospy.init_node('Master', anonymous=True)
     rate = rospy.Rate(10)
     mover = Master()
     rospy.spin()
```

- Dynamic_box

En este caso se presenta el código Dynamic_box. Esté código ha sido diseñado para encargarse de controlar uno de los obstáculos, para hacerlo circular alrededor del circuito, simulando ser un vehículo en marcha circulando a menor velocidad.

```
#!/usr/bin/env python2

import numpy as np
import rospy
import re
import math
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
from tf.transformations import euler_from_quaternion
i = 1
j = 0
camino = 0
len1 = 0
cont = 0
conty = 0
dist = 0
roll = pitch = yaw = 0.0

#This code manage the movement of the dynamic obstacle that simulates a car on the main road
# The Pure pursuit code is the same than our vehicule's
def PurePursuit(x_def, y_def, distancia, phi_error, self):
    "calculate de steering angle for the car"
    global cont, conty, len1, j
    if distancia > 0.3 and j == 0:
      i = 0
      px = math.cos(phi_error)*x_def + math.sin(phi_error)*y_def
      py = -math.sin(phi_error)*x_def + math.cos(phi_error)*y_def
      giro = 1.5*math.atan2(py,px)
      print("giro necesario: ", giro)
      self.drive_msg.linear.x = 0.3
      self.drive_msg.angular.z = giro

    else:
      print("punto alcanzado")
      print("siguiente punto")
      i = 1
      len = len1 - 1
      if conty == len:
        cont = 0
        conty = 0
      else:
        cont += 2
      j = 0
    return i
# This function takes the target points form the txt
def extraer_punto(filename):
    with open(filename, 'r') as f:
      Puntos = f.read()
    Puntos_1 = [float(s) for s in re.findall(r'-?\d+\.?\d*', Puntos)]
    return Puntos_1
```

```
class Dynamic_box:


  def __init__(self):
    DRIVE_TOPIC = "/dynamic_box/cmd_vel"
    ODOMETRY_TOPIC = "/dynamic_box/odom"


    ####################################################
    # TODO >>>
    self.drive_msg = Twist()


    ####################################################

    ####################################################
    # <<< TODO
    ####################################################

    # Initialize a publisher for drive messages
    self.drive_pub = rospy.Publisher(
        DRIVE_TOPIC,
        Twist,
        queue_size=1)

    # Subscribe to the Odometry data
    rospy.Subscriber(
        ODOMETRY_TOPIC,
        Odometry,
        self.position)



  def position(self, msg):
    ####################################################
    # TODO >>>
    # Take car position from Odometry topic
    # Take target position from txt
    # Make the car move to target position
    ####################################################
    global i, Target_x, Target_y, cont, yaw, roll, pitch, conty, camino

    Pose_car_x = msg.pose.pose.position.x
    Pose_car_y = msg.pose.pose.position.y
    orientation_q = msg.pose.pose.orientation
    orientation_list = [orientation_q.x, orientation_q.y, orientation_q.z, orientation_q.w]
    (roll, pitch, yaw) = euler_from_quaternion (orientation_list)



      if i == 1:

        Target_x = camino[cont]
        conty = cont + 1
        Target_y = camino[conty]

      print("Punto objetivo en x: ", Target_x)
      print("Punto objetivo en y: ", Target_y)
      x_def = Target_x - Pose_car_x
      y_def = Target_y - Pose_car_y

      dist = math.sqrt(pow(x_def,2) + pow(y_def,2))
      print("distancia al punto objetivo: ", dist)

      i = PurePursuit(x_def, y_def, dist, yaw, self)
      self.drive_pub.publish(self.drive_msg)



      ####################################################
      # <<< TODO
      ####################################################


  if __name__ == "__main__":
    # First, we initialize the node
    rospy.init_node("Dynamic_box")
    # In this case, we only take the main road target points
    Puntos_obj = extraer_punto('TargetPoints.txt')
    len1 = len(Puntos_obj)
    camino = Puntos_obj
    mover = Dynamic_box()
    rospy.spin()
```

- Safety

Este último caso presenta el código Safety. Es el código “master”, encargado de recibir toda la información del resto de nodos y topics para actuar en consecuencia. Es el único nodo capaz de poner en marcha o detener el vehículo, para evitar así órdenes contradictorias.

```
#!/usr/bin/env python2
import numpy as np
import rospy
import re
import math
from std_msgs.msg import Float32
from nav_msgs.msg import Odometry
from ackermann_msgs.msg import AckermannDriveStamped
from tf.transformations import euler_from_quaternion
from sensor_msgs.msg import LaserScan
i = 1
j = 0
camino = camino1 = camino2 = 0
len1 = len2 = 0
cont = conty = 0
dist = 0
roll = pitch = yaw = 0.0
lidarC = lidarT = alt = volver = parada = giro = curva = 0
#This is the master code. Is the only one that can move the car according to the information received from the topics
# This function calculates the Pure Pursuit movement
def PurePursuit(x_def, y_def, distancia, phi_error, self):
    "calculate de steering angle for the car"
    global cont, conty, len1, len2, j, parada, giro
    if distancia > 0.3 and j == 0:
      i = 0
      px = math.cos(phi_error)*x_def + math.sin(phi_error)*y_def
      py = -math.sin(phi_error)*x_def + math.cos(phi_error)*y_def
      giro = 1.7*math.atan2(py,px)
      # print("giro necesario: ", giro)
      # The car will not move in case we have an emergency stop
      if parada == 0:
        self.drive_msg.drive.speed = 0.7
        self.drive_msg.drive.steering_angle = giro
      else:
        self.drive_msg.drive.speed = 0
    else:
      # print("punto alcanzado")
      # print("siguiente punto")
      i = 1
      len = len1 - 1
      if conty == len:
        cont = 0
        conty = 0
      else:
        cont += 2
      j = 0
    return i
```

```
# Here we get the points for the road of the txt files
def extraer_punto(filename):
    with open(filename, 'r') as f:
      Puntos = f.read()
    Puntos_1 = [float(s) for s in re.findall(r'-?\d+\.?\d*', Puntos)]
    return Puntos_1

# Here we decide if we must continue or not on the main road
def adelantar(lidarC):
    global camino, camino2, camino1, j, alt
    if camino1 == 1:
        if lidarC == 1:
          # print("Tomando ruta alternativa")
          j = 1
          camino = Puntos_alt
          camino2 = 1
          camino1 = 0
          alt = 0

# This function decides if we must do an emergency stop
def parada_emergencia(alt, msg):
    global lidarC, lidarT, giro, parada, camino2
    print("Punto alternativo: ", alt)
    if giro > 0.1 and alt > 1 and alt < 5 and camino2 == 1:
      if np.min(msg.ranges[540:1080]) < 0.3:
        parada = 1
        print("PARADA DE EMERGENCIA")
      else:
        parada = 0
    elif giro < 0.1 and alt > 1 and alt < 5 and camino2 == 1:
      if np.min(msg.ranges[400:1080]) < 0.5:
        parada = 1
        print("PARADA DE EMERGENCIA")
      else:
        parada = 0

class Safety:


  def __init__(self):
    DRIVE_TOPIC = "/vesc/high_level/ackermann_cmd_mux/input/nav_0"
    ODOMETRY_TOPIC = "/vesc/odom"
    LIDAR_TOPIC = "/scan"


    ################################################
    # TODO >>>
    self.drive_msg = AckermannDriveStamped()


    ################################################

    ################################################
    # <<< TODO
    ################################################

    # Initialize a publisher for drive messages
    self.drive_pub = rospy.Publisher(
        DRIVE_TOPIC,
        AckermannDriveStamped,
        queue_size=1)
```

```
  # Subscribe to the Odometry data
  rospy.Subscriber(
      ODOMETRY_TOPIC,
      Odometry,
      self.position)
  # Subcribe to obstacle topic
  rospy.Subscriber("obstacle",
      Float32,
      self.obstacle)

  # Subcribe to adelantamiento topic
  rospy.Subscriber("adelantamiento",
      Float32,
      self.adelantamiento)

  # Subcribe to verificacion topic
  rospy.Subscriber("verificacion",
      Float32,
      self.verificacion)
  # Subscribe to the laser scan data
  rospy.Subscriber(
      LIDAR_TOPIC,
      LaserScan,
      self.callback)

def callback(self, msg):
  ################################################
  # TODO >>>
  # Check the LIDAR for the emergency stop
  #
  ################################################
  global giro, lidarC

  if giro > 0.1:
    curva = 1
  else:
    curva = 0
  if np.min(msg.ranges[400:1080]) < 1 and curva == 1:
    lidarC = 1
    adelantar(lidarC)
  parada_emergencia(alt, msg)
  print("Parada: ", parada)
  ################################################
  # <<< TODO
  ################################################

def position(self, msg):
  ################################################
  # TODO >>>
  # Take Robot position from Odometry topic
  # Take target position from txt
  # Make the robot move to target position
  ################################################
  global i, Target_x, Target_y, cont, yaw, roll, pitch, conty, camino, camino2, camino1, alt
```

```
  Pose_car_x = msg.pose.pose.position.x
  Pose_car_y = msg.pose.pose.position.y
  orientation_q = msg.pose.pose.orientation
  orientation_list = [orientation_q.x, orientation_q.y, orientation_q.z, orientation_q.w]
  (roll, pitch, yaw) = euler_from_quaternion (orientation_list)

  #print("Posicion coche en x: ", Pose_car_x)
  #print("Posicion coche en y: ", Pose_car_y)
  #print("Orientacion del coche en z: ", yaw)

  if i == 1:
    Target_x = camino[cont]
    conty = cont + 1
    Target_y = camino[conty]
    if camino2 == 1:
      alt += 1
  # print("Punto objetivo en x: ", Target_x)
  # print("Punto objetivo en y: ", Target_y)
  x_def = Target_x - Pose_car_x
  y_def = Target_y - Pose_car_y
  # Minimun distance to the target point
  dist = math.sqrt(pow(x_def,2) + pow(y_def,2))
  # print("distancia al punto objetivo: ", dist)

  i = PurePursuit(x_def, y_def, dist, yaw, self)
  self.drive_pub.publish(self.drive_msg)



  ####################################################
  # <<< TODO
  ####################################################

def obstacle(self, data):
  ####################################################
  # TODO >>>
  # This take de information form the Obstacle topic
  # And the let the function adelantar decides what to do
  #
  ####################################################
  global camino, lidarC, j, camino2, alt, curva
  if curva == 0:
    lidarC = data.data
  #print(camino)
  adelantar(lidarC)


  ####################################################
  # <<< TODO
  ####################################################
```

```
  def adelantamiento(self, info):
    ##################################################
    # TODO >>>
    # This thakes the information from the adelantamiento topic
    # And then calculate if we can get back to the main road
    #
    ##################################################
    global camino, lidarC, lidarT, j, camino1, camino2, alt, volver
    lidarT = info.data
    #print(camino)
    if camino2 == 1:
      if lidarC == 0 and lidarT == 0 and alt == 4 and volver == 0:
        # print("Desvio al carril principal")
        camino = Puntos_obj
        j = 1
        alt = 0
        volver = 0
        camino2 = 0
        camino1 = 1



    ##################################################
    # <<< TODO
    ##################################################

  def verificacion(self, info):
    ##################################################
    # TODO >>>
    # This takes the information from the verificacion topic
    # And then change the value of volver variable
    # With this, the adelantamiento function can decide if we get back to the main road or not
    ##################################################
    global camino2, alt, volver, lidarC
    # print("verificacion: ", info.data)
    if alt >= 2 and camino2 == 1:
      if info.data == 1:
        volver = 1
        alt = 0
      else:
        volver = 0



    ##################################################
    # <<< TODO
    ##################################################

if __name__ == "__main__":
  #initialize the node
  rospy.init_node("Safety")
  #Take the target points of the main road
  Puntos_obj = extraer_punto('TargetPoints.txt')
  len1 = len(Puntos_obj)
  camino = Puntos_obj
  camino1 = 1
  #Take the target points of the alternative road
  Puntos_alt = extraer_punto('AlternativeRoad.txt')
  len2 = len(Puntos_alt)
  print(Puntos_obj)
  mover = Safety()
  rospy.spin()
```

Anexo 2. Videos de simulación

En este anexo se facilitan enlaces a los vídeos de las simulaciones realizadas, para poder entender de una manera más didáctica la filosofía de este trabajo.

- Evitación de obstáculos estáticos:

  https://youtu.be/s-WaDRNt8wc

- Evitación de obstáculos dinámicos:

  https://youtu.be/Y-Jt324XwG8

- Evitación de colisiones:

  https://youtu.be/q9BrxY4vKWw

Anexo 3. Entorno de simulación

Para las simulaciones, hay que generar un entorno de trabajo en ROS donde se irán incluyendo los archivos y carpetas con los que se va a trabajar. Abriendo una ventana de comandos en el terminal (recordar que se trabaja con Ubuntu) el procedimiento a seguir es el siguiente:

1. Crear y compilar el entorno de trabajo (*ver Figura 42*):

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/
$ catkin_make
```

Figura 43. Crear entorno de trabajo. [Realización propia]

El comando catkin_make es una herramienta que, al ejecutarla, creará un enlace CMakeList.txt en la carpeta src. Además, también deberán haberse generado carpetas 'build' y 'devel'.

2. A continuación, se compila de nuevo el archivo setup.*sh (ver *Figura 43*):

```
$ source devel/setup.bash
```

*Figura 44. Setup.*sh. [Realización propia]*

3. Una vez hecho esto quedaría comprobar que el espacio de trabajo se ha superpuesto correctamente por el script de la configuración. Para hacer esto, hay que comprobar que la variable de entorno ROS_PACKAGE_PATH incluya el directorio en el que se encuentra (ver *Figura 44*):

```
$ echo $ROS_PACKAGE_PATH
/home/youruser/catkin_ws/src:/opt/ros/kinetic/share
```

Figura 45. ROS_PACKAGE_PATH. [Realización propia]

Con esto quedaría configurado y listo el entorno de trabajo con el que se va a trabajar.

Anexo 3.1 Descargar carpetas del repositorio

Para lanzar el entorno de simulación, primero, hay que descargarse la carpeta de trabajo que ha sido colgada en Github para que cualquiera con el enlace, pueda probar este diseño o por si resulta de utilidad en futuros trabajos.

- ➔ https://github.com/JorgeRuRi/ObstacleAvoidance.git
- ➔ https://gitlab.iri.upc.edu/mobile_robotics/adc/adc_2021

Abriendo una ventana de comandos en el terminal, se puede clonar el repositorio en la carpeta de trabajo que se ha generado anteriormente para que se copien automáticamente todos los archivos. De esta forma, se ahorra el tener que descargarse todas las carpetas de la competición y unirlas con el modelo del coche de la escuela.

En el primer enlace, están todos los archivos de diseño propio necesarios para poder probar el proyecto en el entorno proporcionado por la competición. Este entorno se encuentra en el segundo de los enlaces junto con los obstáculos y demás librerías necesarias para las simulaciones. Abriendo la ventana de comandos, lo primero de todo es situarse en la carpeta src del entorno de trabajo, como se puede ver en la *Figura 45*:

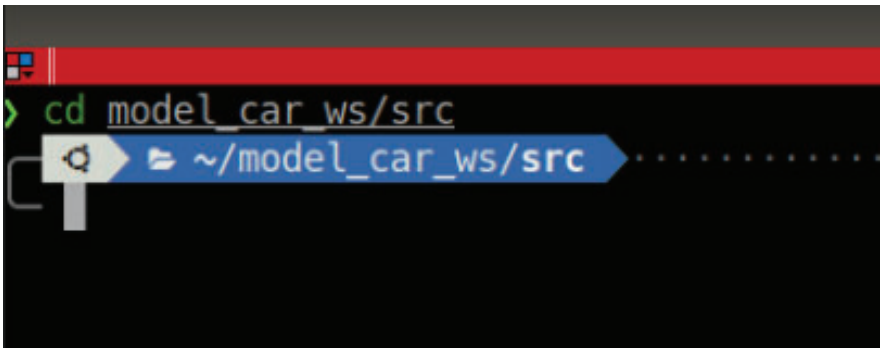


Figura 46. Entrono de trabajo. [Realización propia]

Una vez hecho eso, quedaría clonar todas las carpetas que hagan falta para la simulación con el enlace anteriormente compartido tal y como se muestra en la *Figura 46*:

Figura 47. Clonar repositorio. [Realización propia]

Antes de lanzar el entorno de simulación hay que volver a ejecutar el comando catkin_make desde la carpeta src, para que actualice el archivo CMake_List y reconozca todas las carpetas que se han clonado, como puede verse en la *Figura 47:*

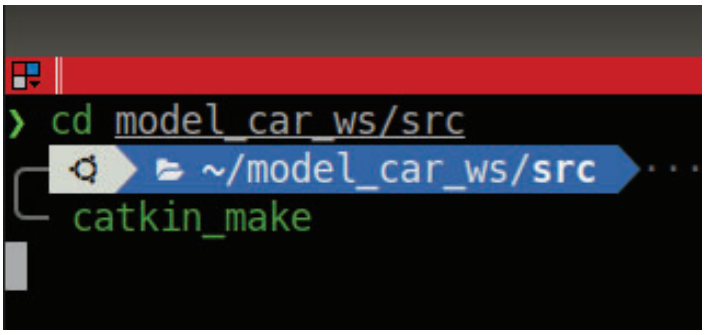


Figura 48. Catkin Make. [Realización propia]

Para lanzar el entorno de simulación, ternemos que estar situados en la carpeta src en la ventana de comando. Desde aquí, hay que lanzar el siguiente comando que abrirá el mapa en Gazebo con el circuito a escala proporcionado por la competición que puede verse en la *Figura 48:*

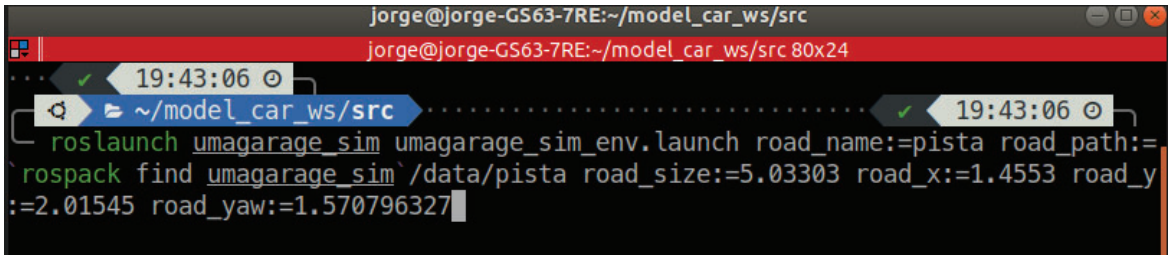


Figura 49. Lanzar circuito a escala. [Realización propia]

Este circuito es útil, ya que las líneas de carril están diseñadas a escala, luego permite tener un entorno más realista para las simulaciones.

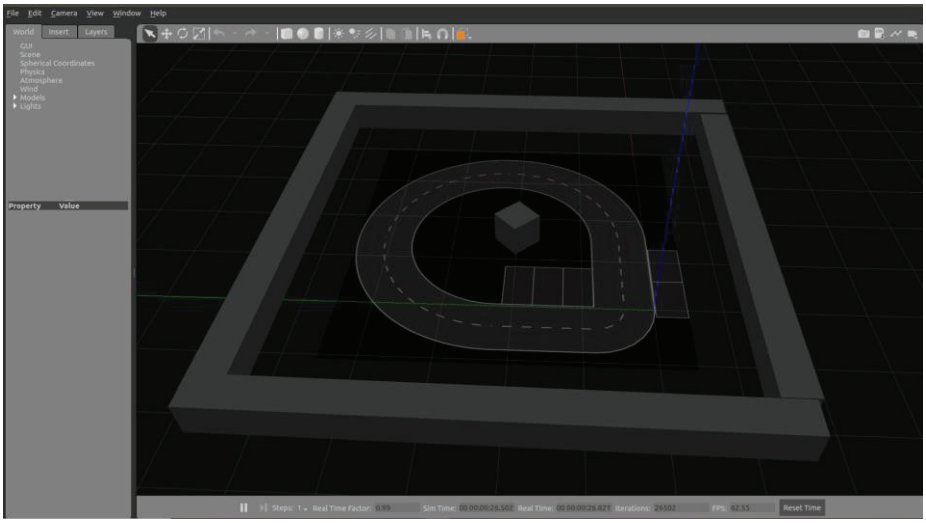


Figura 50. Circuito a escala. [Realización propia]

Si se han seguido los pasos correctamente, se debería abrir un mapa como el que se muestra en la *Figura 49*. Si es así, ya se puede probar el software de evitación de obstáculos.

Anexo 4. Pruebas experimentales

En este anexo, se explicará cómo poner en marcha el vehículo físico disponible en el laboratorio del departamento para las pruebas experimentales.

Para la puesta en marcha del vehículo, se pueden seguir los pasos especificados en el TFG [23] *Navegación autónoma en circuito de un vehículo a escala con LIDAR utilizando ROS y GAZEBO,* donde se incluyen referencias visuales para arrancar el vehículo de manera correcta, sin dañar los componentes ni las baterías.

En segundo paso para poder realizar las pruebas experimentales es conectar el mando de teleoperación con el vehículo. Para ello, hay que apoyarse en un archivo llamado “teleop” que contiene la información que permite al usuario controlar la velocidad y giro del vehículo con un mando ps3. Para realizar este emparejamiento de forma satisfactoria, hay que abrir una ventana de comandos e introducir:

➔ *Sudo sixad -s*

Una vez hecho esto, se pedirá introducir la contraseña del equipo raíz. En ese momento el equipo se quedará en espera para conectarse con el mando, momento en el que se debe mantener pulsado el botón ps del mando para ponerlo en modo emparejamiento. Cuando se note una vibración en el mando, significará que el emparejamiento se ha realizado con éxito.

El tercer paso para realizar las pruebas experimentales es conectar el pc de trabajo con el vehículo para poder cargarle los códigos diseñados. Para este paso, fue necesario realizar unas modificaciones en los archivos “.zshrc”, que permitiesen un correcto reconocimiento entre los equipos. Estas modificaciones, fueron realizadas por un compañero en su tfg y están los pasos detallados y explicados en el mismo [23] *Navegación autónoma en circuito de un vehículo a escala con LIDAR utilizando ROS y GAZEBO*. Una vez cumplimentados estos pasos, basta con introducir en una ventana de comandos del equipo de trabajo el comando “ssh 10.42.0.1” para estar conectados con el vehículo de manera

remota y poder lanzarle los nodos y controlarlo, sin necesidad de tenerlo conectado a una pantalla en todo momento.

Si se han seguido todos los pasos correctamente, el vehículo debería estar listo para ser utilizado y controlado por el usuario sin inconvenientes.

Desde el equipo personal de trabajo y en una ventana ssh hay que situarse en la carpeta dl proyecto con el comando:

➔ cd Development/racecar-ws/src/tfg_jorge

Desde esta dirección, se lanzarán los códigos diseñados en este proyecto para las pruebas experimentales. Estos códigos que están almacenados en el UMA racecar han sido adaptados para funcionar aquí correctamente. La modificación realizada más importante que se debe mencionar es que, durante las pruebas experimentales, me percaté de que e LIDAR en este caso, no devuelve un “inf” cuando no detecta ningún obstáculo o detecta uno muy lejano, sino que devuelve un valor extremadamente pequeño.

Este hecho supuso un inconveniente, debido a que, al lanzar el código, siempre detectaba un obstáculo demasiado cerca, luego se activaban las medidas de emergencia y no funcionaba correctamente.

Para solucionar esto, hay que almacenar los valores del LIDAR en un array al principio de su callback quedando:

➔ Lectura = np.asarray(msg.ranges)

Con esto le estamos diciendo al programa que todos los datos de LIDAR me los almacene en Lectura y que interprete la variable “Lectura” como un array.

➔ Lectura[Lectura < 0.01] = 20

```
def callback(self, msg):
  ################################################
  # TODO >>>
  # Check the LIDAR for the emergency stop
  #
  ################################################
  global giro, lidarC

  Lectura = np.asarray(msg.ranges)
  Lectura[Lectura < 0.01] = 20

  if giro > 0.1:
    curva = 1
  else:
    curva = 0
  if np.min(Lectura[400:1079]) < 1 and curva == 1:
    lidarC = 1
    adelantar(lidarC)
  parada_emergencia(alt, Lectura)
  print("Parada: ", parada)
  ################################################
  # <<< TODO
  ################################################
```

Figura 51. Modificaciones en el código. [Realización propia]

Con esta segunda modificación, se consigue que todos esos valores extremadamente pequeños del LIDAR, los cambie por un 20 que es un valor muy grande, lo que se interpreta como que no hay obstáculos próximos en esa medida, siendo esto lo deseado.

Hecho esto, en una primera ventana de comandos, se lanza un roscore para que el vehículo detecte los nodos de trabajo diseñados y los topics con la información del vehículo.

Seguidamente, se lanza un archivo launch llamado teleop.launch que va a permitir controlar al vehículo con el mando y además está adaptado para activar los topics como nav0, que permite controlar la velocidad y giro del vehículo o el que contiene la información del LIDAR.

```
done
~/Development/racecar-ws/src/tfg_jorge ····· 1m 48s 16:24:13
roslaunch racecar teleop.launch
```

Figura 52. Pruebas experimentales. [Realización propia]

Si este paso se ha realizado correctamente, ya estarán publicándose información el topic Odometry y el en el LIDAR, luego ya se pueden lanzar los diseños realizados en este proyecto.

El orden es importante, porque primero interesa que el vehículo empiece a procesar la información del LIDAR para luego comenzar a desplazarse, de lo contrario se comenzaría a desplazar de manera “ciega” y podría ocasionar situaciones de peligro.

Se lanza ahora el código de detección, para que comience el coche a procesar la información de los distintos topics y a publicar las variables de control.

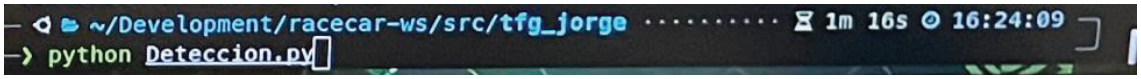


Figura 53. Pruebas experimentales 2. [Realización propia]

Cuando este nodo ya esté funcionando, lanzamos el nodo safety, que pondrá al vehículo en marcha.

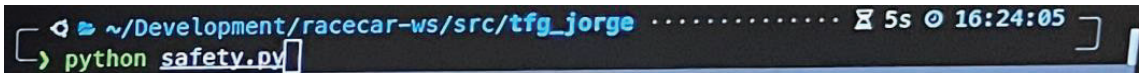


Figura 54. Pruebas experimentales 3. [Realización propia]

El nodo Dynamic_box, no podrá ser utilizado en las pruebas experimentales, ya que haría falta un segundo vehículo para las pruebas experimentales del que actualmente no se dispone en la Escuela.