UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola d'Enginyeria de Barcelona Est

TRABAJO FINAL DE GRADO

Grado en Ingeniería Electrónica Industrial y Automática

DISEÑO E IMPLEMENTACIÓN DEL CONTROL DE UN COCHE AUTÓNOMO CON COMUNICACIÓN EXTERNA

Memoria y Anexos

Autora: Elena Villalba Aguilera
Director: Joaquim Blesa Izquierdo
Co-Director: Alejandro Rolán Blanco

Convocatoria: junio 2022

Resumen

El presente proyecto se centra en el diseño y la implementación del control de un vehículo autónomo y en la comunicación entre éste y un ordenador. Adicionalmente, de manera independiente, se incluye el control manual con un mando externo Bluetooth.

El principal objetivo es lograr desarrollar un óptimo algoritmo capaz de manipular al automóvil con la finalidad de recorrer un circuito, acotado por dos líneas, sin salirse y de manera completamente autónoma.

Para tales efectos, se emplea un sistema ultrasónico de navegación idóneo para la localización del vehículo en un plano 2D. Mediante un ordenador externo al vehículo se procesa la información captada y se realizan una serie de cálculos con el fin de determinar la velocidad y la dirección pertinentes. Por último, se envían los resultados a una *Raspberry Pi 4 Model B*, a bordo del vehículo, que se encarga de actuar sobre los motores para obtener el movimiento deseado.

Destacar que tanto los códigos programados como la comunicación entre la Raspberry Pi y el ordenador se implementan en ROS (*Robot Operating System*), mediante el lenguaje de programación de alto nivel Python.

Palabras claves: *navegación autónoma, Raspberry Pi, Robot Operating System, sistema ultrasónico de navegación, vehículo autónomo.*

Resum

Aquest projecte es centra en el disseny i la implementació del control d'un vehicle autònom i en la comunicació entre aquest i un ordinador. Addicionalment, de manera independent, s'hi inclou el control manual amb un comandament extern Bluetooth.

El principal objectiu és aconseguir desenvolupar un òptim algorisme capaç de manipular l'automòbil amb la finalitat de recórrer un circuit, acotat per dues línies, sense sortir-se'n i de manera completament autònoma.

D'aquesta manera, es fa servir un sistema ultrasònic de navegació idoni per a la localització del vehicle en un pla 2D. Mitjançant un ordinador extern es processa la informació captada i es realitzen una sèrie de càlculs per tal de determinar la velocitat i la direcció pertinents. Per acabar, s'envien els resultats a una *Raspberry Pi 4 Model B* que s'encarrega d'actuar sobre els motors per obtenir el moviment desitjat.

Cal destacar que tant els codis programats com la comunicació entre la Raspberry Pi i l'ordinador s'implementen en ROS (*Robot Operating System*), mitjançant el llenguatge de programació d'alt nivell Python.

Paraules claus: *navegació autònoma, Raspberry Pi, Robot Operating System, sistema ultrasònic de navegació, vehicle autònom.*

Abstract

This project focuses on the design and the implementation of the control of an autonomous vehicle and the communication between it and a computer. Additionally, manual control is included with an external Bluetooth control.

The main objective is to develop an optimal algorithm capable of manipulating the automobile in order to follow a track, bounded by two lines, without diverting and completely autonomously.

In this way, an ultrasonic navigation system is used with the purpose of locating the vehicle on a 2D plane. An external computer processed the information captured by the sensors, executing the implemented algorithm so as to determine the relevant speed and direction. Finally, the results are sent to a *Raspberry Pi 4 Model B* on board the vehicle, which is responsible of acting on the motors to obtain the desired movement.

The programmed codes and the communication between the Raspberry Pi and the computer are implemented in ROS (*Robot Operating System*), using the high-level programming language Python.

Keywords: *autonomous navigation, autonomous vehicle, Raspberry Pi, Robot Operating System, ultrasonic navigation system.*

Agradecimientos

En primer lugar, me gustaría agradecer a mis tutores, Joaquim Blesa y Alejandro Rolán, por darme la oportunidad de realizar este trabajo, brindarme los recursos necesarios para llevar a cabo el vehículo autónomo y por la orientación y consejos prestados a lo largo de las diferentes etapas de este proyecto.

También, me gustaría agradecer la ayuda proporcionada por parte de los maestros de laboratorio a lo largo de estos cuatro meses.

A mi familia, por apoyarme, animarme y ser comprensivos por el esfuerzo, sacrificio y trabajo que he realizado durante todos estos años.

Y en general, a toda aquella persona que ha podido contribuir de alguna forma en mi camino hasta el último momento de esta etapa.

Glosario

ESC (*Electronic Speed Control*), también conocido como variador, es un controlador capaz de controlar la velocidad y la dirección de motores, a través del ancho de pulso de una señal PWM.

GPIO (*General Purpose Input/Output*). Propósito general de entrada y salida. En otras palabras, son una serie de pines cuyo comportamiento, como entrada o salida, es establecido por el programador. Normalmente, son empleados para el control de LEDs, sensores o actuadores.

Master-slave, o maestro-esclavo traducido al castellano, es una arquitectura de comunicación en la que el dispositivo maestro empieza la comunicación con los esclavos y permite el envío y la recepción de datos entre ellos. El maestro solo puede ser una única máquina mientras los esclavos pueden ser más de una.

Node, o nodo en castellano, es un término utilizado en ROS que describe la instancia de un ejecutable. Dicho ejecutable puede equivaler a un sensor, un actuador, un algoritmo o un programa. Su comportamiento viene determinado en función de si publica y/o se suscribe a diferentes tópicos.

Package. Terminología utilizada en ROS para referirse al conjunto de paquetes generados por los usuarios. Un paquete puede contener uno o más nodos, una librería, un conjunto de archivos de configuración, entre otros. El objetivo de su uso es organizar de manera eficiente el software para poder reutilizarlo en futuros proyectos.

PWM (*Pulse Width Modulation*), o modulación de ancho de pulso en castellano, es una técnica comúnmente utilizada en electrónica para emular una señal analógica, a partir de una señal digital. Modificando el ancho de pulso de la señal digital se consigue variar la media de la tensión de salida.

ROS (*Robot Operating System*). *Framework* de código abierto libre que suministra un conjunto de herramientas y librerías para el desarrollo del software de robots.

TOF (*Time of Flight).* Método empleado para estimar la distancia entre cuerpos. Se basa en calcular el tiempo transcurrido ente la emisión de un pulso ultrasónico y la recepción de éste.

Topic, o tópico, es un término empleado en ROS que hace referencia al canal de transporte de datos que permite la comunicación entre nodos.

Trilateración. Técnica geométrica utilizada para determinar la posición exacta de un objeto a partir de la distancia hallada entre éste y tres puntos de referencia como mínimo.

Índice

RESUMEN ____ I

RESUM ____ II

ABSTRACT ____ III

AGRADECIMIENTOS ____ V

GLOSARIO ____ VII

1. PREFACIO ____ 17
   1.1. Origen del trabajo 17
   1.2. Motivación 17
   1.3. Requerimientos previos 18

2. INTRODUCCIÓN ____ 19
   2.1. Objetivos del trabajo 20
   2.2. Alcance del Trabajo 21
   2.3. Metodología 21

3. ESTADO DEL ARTE ____ 23
   3.1. Vehículo autónomo 23
      3.1.1. Niveles de conducción 23
      3.1.2. Ventajas y desventajas 24
      3.1.3. Actualidad y futuro 24
   3.2. Encaje del proyecto en la situación actual 24

4. HARDWARE ____ 26
   4.1. Diagrama de bloques 26
   4.2. Raspberry Pi 27
      4.2.1. Elección y comparativa de diferentes modelos 27
   4.3. Periféricos 30
      4.3.1. Balizas ultrasónicas y módem 30
      4.3.2. Mando Bluetooth 32
      4.3.3. Motor DC y controlador 33
      4.3.4. Servomotor 36
      4.3.5. Batería 37
      4.3.6. Reguladores de tensión 37

4.4. Esquema electrónico ... 39

5. SOFTWARE ... 40

5.1. Ubuntu ... 40

5.2. Robot Operating System (ROS) ... 41

5.2.1. Conceptos básicos ... 42

5.2.2. Comandos ... 45

5.3. Python ... 46

5.4. Marvelmind Dashboard ... 46

6. IMPLEMENTACIÓN ... 47

6.1. Montaje del vehículo ... 47

6.2. Posicionamiento y calibración de las balizas ultrasónicas ... 50

6.2.1. Colocación de las balizas ... 50

6.2.2. Montaje de las balizas estacionarias ... 52

6.2.3. Configuración de los parámetros de las balizas ... 53

6.3. Comunicación Master-Slave ... 57

6.4. Programación ... 58

6.4.1. Estrategia ... 58

6.4.2. Ordinogramas ... 61

6.4.3. Descripción de los tópicos ... 61

6.4.4. Control manual mediante un mando Bluetooth. Nodo ps4_controller ... 63

6.4.5. Control de las balizas. Nodo hedge_rcv_bin ... 68

6.4.6. Guardado de los waypoints. Nodo save_waypoints ... 68

6.4.7. Detector de curvas. Nodo curves_detector ... 70

6.4.8. Control autónomo. Nodo autonomous_control ... 75

6.4.9. Control del vehículo. Nodo car_control ... 82

7. FUNCIONAMIENTO Y RESULTADOS ... 88

7.1. Balizas ultrasónicas ... 88

7.1.1. Exactitud de las balizas ... 89

7.1.2. Precisión de las balizas ... 91

7.2. Algoritmo de control ... 92

8. PROPUESTAS DE MEJORA ... 93

8.1. NVIDIA Jetson ... 93

8.2. LIDAR ... 94

8.3. SLAM ... 95

8.4. Adición de funcionalidades ... 96
8.4.1. Detección de señales de tráfico ... 96
8.4.2. Detección del estado de carga de la batería ... 96
8.4.3. Aparcamiento autónomo ... 97

9. ANÁLISIS DEL IMPACTO AMBIENTAL ... 98

CONCLUSIONES ... 101

PLANIFICACIÓN TEMPORAL ... 103

PRESUPUESTO ... 105

Mano de obra ... 105
Componentes y materiales ... 105
Total ... 106

BIBLIOGRAFÍA ... 107

ANEXO A: PUESTA EN FUNCIONAMIENTO DEL SISTEMA DE NAVEGACIÓN DE INTERIORES ... 111

A1. Introducción ... 111
A1.1. Objeto ... 111
A2. Puesta en funcionamiento del Super-NIA-3D ... 112
A2.1. Material y requisitos del sistema de navegación ... 112
A2.2. Instalación del programa Dashboard ... 113
A2.3. Actualización del software vía USB ... 115
A2.4. Configuración del sistema ... 117
A2.5. Uso del sistema ... 120

ANEXO B: CÓDIGOS ... 122

B1. Nodo ps4_controller ... 122
B2. Nodo hedge_rcv_bin ... 127
B3. Nodo save_waypoints ... 136
B4. Nodo curves_detector ... 138
B5. Nodo autonomous_control ... 145
B6. Nodo car_control ... 159

Índice de figuras

Figura 1.1. Robot móvil equipado con un sensor ultrasónico, desarrollado para el trabajo de investigación ___ 18
Figura 2.1. Circuito situado en el laboratorio A5.4 de la EEBE ___ 20
Figura 4.1. Diagrama de bloques con los componentes utilizados ___ 26
Figura 4.2. Raspberry Pi 3 Modelo B+ ___ 28
Figura 4.3. Raspberry Pi 4 Modelo B ___ 28
Figura 4.4. Sistema de navegación: balizas y módem ___ 30
Figura 4.5. Representación gráfica del método *Time of Flight* ___ 31
Figura 4.6. Representación gráfica del método de trilateración. ___ 32
Figura 4.7. Mando inalámbrico DUALSHOCK 4 ___ 32
Figura 4.8. Motor DC ___ 33
Figura 4.9. Controlador L298N ___ 33
Figura 4.10. Principio de funcionamiento del puente-H ___ 34
Figura 4.11. TAS-202 de Avioracing ___ 35
Figura 4.12. Servomotor PDI-1181MG ___ 36
Figura 4.13. Batería LiPo de 11,1 V y 5200 mAh ___ 37
Figura 4.14. Convertidor JZK ___ 38
Figura 4.15. Regulador S13V30F5 ___ 38
Figura 4.16. Esquema electrónico ___ 39
Figura 5.1. Logotipo de Ubuntu ___ 40
Figura 5.2. Logotipo de ROS ___ 41
Figura 5.3. Logotipo de *ROS Noetic Ninjemys* ___ 41
Figura 5.4. Comunicación entre nodos utilizando topics ___ 44
Figura 5.5. Logotipo de Python ___ 46
Figura 6.1. Automóvil D12 Kei Truck ___ 47
Figura 6.2. Componentes que incorpora el D12 Kei Truck con su respectiva ubicación ___ 48
Figura 6.3. Distribución de los componentes en la madera ___ 49
Figura 6.4. Resultado final del montaje del vehículo ___ 49
Figura 6.5. Plano del circuito con la posición de las balizas ___ 51
Figura 6.6. Montaje de las balizas estacionarias ___ 52
Figura 6.7. Distancias entre las balizas estacionarias. ___ 53
Figura 6.8. Introducción de las coordenadas de la baliza 2 ___ 53
Figura 6.9. Menú de configuración del submapa donde se inhabilita la opción de *3D navigation* ___ 54
Figura 6.10. Mapa resultante tras introducir las coordenadas de las balizas estacionar y delimitar el mapa ___ 54
Figura 6.11. Configuración de los transmisores de las balizas ___ 55
Figura 6.12. Configuración de la señal ultrasónica de las balizas ___ 55
Figura 6.13. Opción de *Real-time* ___ 55
Figura 6.14. Menú de configuración del módem donde se encuentran los parámetros *Window of averaging*, *Distance filter* y *High resolution mode* ___ 56
Figura 6.15. Configuración de la frecuencia de trabajo ___ 56

Figura 6.16. Menú de configuración de los parámetros de radio de las balizas 56
Figura 6.17. Resultado obtenido tras ejecutar el ip a en la terminal del dispositivo master 57
Figura 6.18. Esquema del circuito del laboratorio A5.4 de la EEBE con los puntos de referencia guardados 59
Figura 6.19. Primer conjunto de códigos que permiten controlar el vehículo de manera manual y guardar los puntos de referencia 61
Figura 6.20. Segundo conjunto de códigos que permiten controlar el vehículo de manera autónoma y manual 61
Figura 6.21. Esquema de actuación de los joysticks izquierdo y derecho, y de los botones L1 y R1 64
Figura 6.22. Ángulos de giro del servomotor en azul que deberá girar en los puntos 2 y 3 71
Figura 6.23. Máquina de estados finitas implementada en el nodo *autonomous_control* 76
Figura 6.24. Márgenes calculados en el caso del estado *x Range* 77
Figura 6.25. Márgenes calculados en el caso del estado *y Range* 78
Figura 6.26. Márgenes calculados en el caso del estado *b Range* 78
Figura 8.1. Placa NVIDIA Jetson Nano 93
Figura 8.2. Muestra de un LIDAR 94
Figura 8.3. Mapeado resultante tras aplicar herramientas de ROS y utilizar un LIDAR 95
Figura 8.4. Ejemplo de reconocimiento de señales de tráfico en diferentes situaciones 96
Figura 9.1. Logotipo de RoHS 98
Figura A1. Componentes del set de inicio *Super-NIA-3D*: módem y Super-Beacons 112
Figura A2. Paquete a descargar que contiene el instalador del programa y la última versión de software del módem y de los *Super-Beacons* 113
Figura A3. Pasos para la instalación del programa Dashboard 113
Figura A4. Archivo con el que se ejecuta el programa Dashboard 115
Figura A5. Encendido del *Super-beacon* 115
Figura A6. Ilustración del paso 5 y 13 116
Figura A7. Selección del fichero .HEX del módem 116
Figura A8. Selección del fichero .HEX del beacon 117
Figura A9. Botón *Default* ubicado en la esquina inferior derecha del programa Dashboard 118
Figura A10. Configuración de la radio 118
Figura A11. Configuración de la dirección del *beacon* 118
Figura A12. Configuración de la frecuencia del ultrasonido 119
Figura A13. Configuración del beacon como móvil 119
Figura A14. Comprobación del conexionado del módem conectado 120
Figura A15. Panel Dashboard que indica el modo de funcionamiento de los beacons 120
Figura A16. Configuración de la altura en se encuentra los beacons 121
Figura A17. Tabla de posiciones 121
Figura A18. Botón para congelar el mapa y el submapa 121

Índice de tablas

Tabla 4.1. Comparativa entre la *Raspberry Pi 3 Model B+* y *Raspberry Pi 4 Model B* 29
Tabla 4.2. Comportamiento del motor según el estado de los pines IN1 y IN2 (IN3 y IN4 para la segunda salida del motor) 34
Tabla 4.3. Comparativa entre el L298N y el TAS-202 35
Tabla 5.1. Mensajes primitivos de ROS con su correspondencia en el lenguaje C++ y Python 43
Tabla 6.1. Posición entre balizas 50
Tabla 6.2. Coordenadas de las balizas estacionarias 51
Tabla 7.1. Distancias reales respecto al origen de coordenadas de los puntos de referencia 88
Tabla 7.2. Distancias obtenidas con las balizas ultrasónicas 89
Tabla 7.3. Errores relativos y absolutos del sistema de navegación para interiores 90
Tabla 7.4. Promedio de errores absolutos y relativos 90
Tabla 7.5. Errores relativos y absolutos del sistema de navegación para interiores respecto a cada lectura realizada y a la media de las tres 91
Tabla 7.6. Promedio de errores absolutos y relativos 91
Tabla 9.1. Materiales de los dispositivos utilizados 99
Tabla 9.2. Impacto ambiental y reciclado de cada material 99
Tabla 0.1. Coste de ingeniería, impuestos incluidos 105
Tabla 0.2. Coste total de material y componentes utilizados, impuestos incluidos 105
Tabla 0.3. Presupuesto total del proyecto, impuestos incluidos 106

1. Prefacio

1.1. Origen del trabajo

La elaboración del presente trabajo de final de grado surgió de una oferta planteada por parte del profesorado de la universidad sobre el diseño e implementación del control de un vehículo autónomo. Las facilidades ofrecidas por el profesorado y el interés personal de profundizar sobre la robótica, uno de los campos más en auge y desafiantes en la tecnología actual, me convencieron para sumergirme en este tema.

Por otro lado, los conocimientos adquiridos en PUCRA (*Polytechnic University of Catalonia Robotics Association*), durante estos tres últimos años, me han brindado la posibilidad de atender a esta propuesta, ya que en la asociación he podido explorar las diferentes áreas de la robótica, permitiéndome obtener unas competencias básicas imprescindibles para la correcta ejecución de este proyecto.

En PUCRA tomé contacto por primera vez con Linux y ROS y vi en este proyecto la única vía para seguir explorando estos softwares. A través de las empresas que patrocinan la asociación descubrí que estas herramientas son las más utilizadas actualmente. Sin embargo, existe muy poco personal cualificado en este ámbito debido a que en los grados de ingeniería todavía no se estudian estos entornos.

1.2. Motivación

Mi interés hacia este sector proviene de niña, gracias a un regalo de mis padres por mi décimo cumpleaños: un Kit de Arduino. Así, entré en contacto con la electrónica y surgió en mí una curiosidad creciente por este tema.

Cuando comencé bachillerato ya tenía algunas nociones básicas de Arduino y las decidí aprovechar para llevar a cabo el trabajo de investigación, exigido en este periodo de formación, sobre vehículos autónomos. En concreto, se trataba del diseño y construcción de un robot móvil cuya finalidad era trazar una ruta y evadir los obstáculos que se iba encontrando por el camino con la ayuda de un sensor ultrasónico.

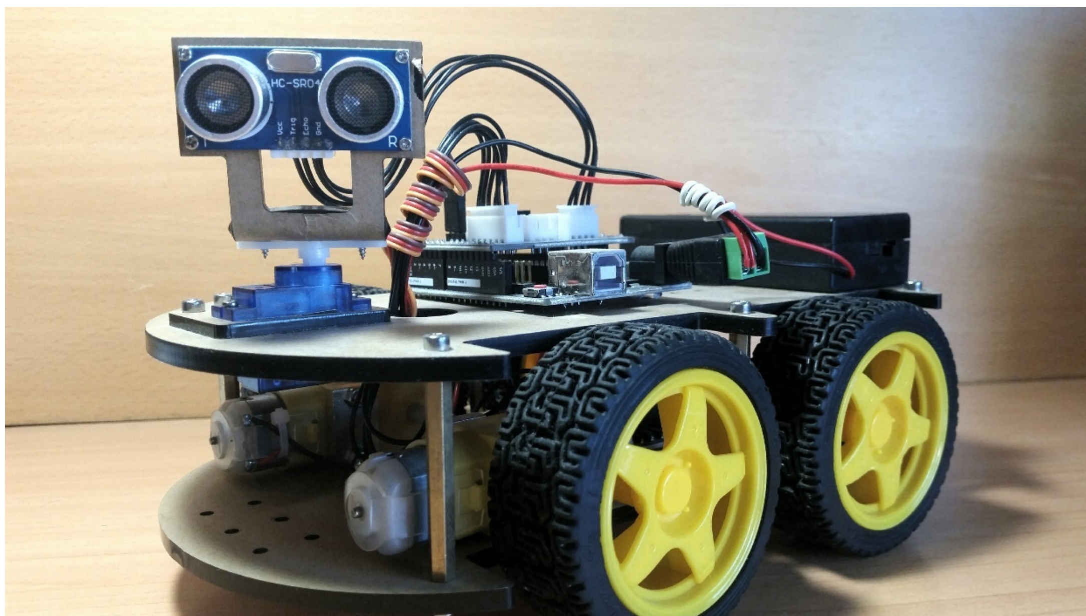


Figura 1.1. Robot móvil equipado con un sensor ultrasónico, desarrollado para el trabajo de investigación (Fuente: propia).

Como se puede observar en la imagen anterior, el resultado final del trabajo de bachillerato muestra cierta similitud con el proyecto actual debido a que el propósito de los dos es el mismo: diseño e implementación de un vehículo autónomo. De este modo, la realización de estos dos proyectos se puede enfocar como una evolución en mi carrera, mostrando las habilidades y competencias adquiridas durante el transcurso del grado de ingeniería electrónica industrial y automática.

1.3. Requerimientos previos

En términos generales, es aconsejable el aprendizaje de los pilares fundamentales de la robótica para un adecuado desarrollo de este trabajo. Por tanto, es recomendable conocer Linux, ya que es el sistema operativo utilizado por el ordenador externo y por el procesador incorporado en el vehículo. Se sugiere tener nociones básicas sobre ROS, puesto que es el entorno utilizado para la comunicación entre ordenador y automóvil y, cierta habilidad con Python debido a que es el lenguaje de programación empleado. Además, hacen falta conocimientos de electrónica para la adecuada elección e implementación de los diferentes componentes electrónicos que incorpora el vehículo.

2. Introducción

Este proyecto se enmarca dentro del ámbito de la navegación autónoma, un tema presente tanto en los automóviles autoconducidos como en la robótica.

Los vehículos autónomos y la robótica abarcan distintas disciplinas tecnológicas, como la mecánica, la electrónica, la informática, entre otras, con la finalidad de diseñar e implementar automóviles y robots capaces de realizar determinadas operaciones de forma autónoma, sin que el ser humano intervenga y sustituyendo al mismo.

La navegación autónoma se considera una de las tareas ejercidas por robots y vehículos autónomos y, se puede definir como el desplazamiento independiente a través de un escenario siguiendo la trayectoria más óptima desde un punto inicial hasta un punto final, tomando decisiones propias basándose en el entorno que los rodea [1]. Los autómatas están dotados con inteligencia artificial con la intención de efectuar acciones con suma rapidez en función de los datos extraídos a partir de diferentes sensores, cámaras y sistemas de navegación. Este campo de trabajo es uno de los más complejos y desafiantes, ya que son necesarios complejos algoritmos matemáticos y potentes sistemas informáticos para detectar el entorno y los obstáculos, elaborar un mapa con la información capturada, planificar un recorrido despejado y navegar por él.

Como se ha comentado, los vehículos autónomos, al igual que los robots, agrupan diferentes campos para llevar a cabo diversas tareas, pero la navegación es una de las más importantes y de las que actualmente se encuentra en investigación y evolución por su elevada complejidad. Es por este motivo que este trabajo se centra en cómo un vehículo es capaz de desenvolverse por un entorno conocido con la finalidad de recorrer un circuito, acotado por dos líneas, sin salirse.

Un punto importante a tener en consideración es la localización del automóvil. En el sector de la automoción, la obtención de la posición y la trayectoria se resuelve con la tecnología GPS (*Global Positioning System*). No obstante, los robots normalmente desempeñan sus funciones en espacios interiores, donde la señal GPS es inestable. Por ello, es inevitable implementar otra clase de tecnologías que proporcionen un posicionamiento preciso en espacios cerrados. En este proyecto, se analiza y se utiliza un sistema de navegación ultrasónico, especialmente diseñado para interiores, con el que se pretende determinar la posición exacta, en todo momento, del vehículo.

Por último, resaltar que el desarrollo del software se realiza mediante ROS dado que es un sistema operativo de código abierto que se encuentra en constante mantenimiento por la empresa *Open Source Robotics Foundation* (OSRF). Además, es capaz de ejecutar diferentes programas al mismo tiempo, posibilitando la interacción entre ellos.

2.1. Objetivos del trabajo

El objetivo de este proyecto es desarrollar e implementar un algoritmo de control para un vehículo autónomo, de forma que logre desenvolverse por cuenta propia dentro de un circuito, acotado por dos líneas. En la **figura 2.1** se muestra el circuito que debe recorrer, localizado en el laboratorio A5.4 de la *Escola d'Enginyeria de Barcelona Est* (EEBE), situada en el Campus Diagonal-Besós de la *Universitat Politècnica de Catalunya* (UPC).

Figura 2.1. Circuito situado en el laboratorio A5.4 de la EEBE (Fuente: propia).

Otro objetivo principal es implementar la comunicación externa con la finalidad de ejecutar el algoritmo de control en un ordenador externo, enviando solamente la dirección y la velocidad pertinentes al procesador a bordo del vehículo. De este modo, la utilización de un ordenador externo libera de carga a dicho procesador haciendo que el tiempo de respuesta sea más rápido. Este punto es de vital importancia debido a que este procesador presenta ciertas limitaciones de computación.

Por último, dado que el algoritmo y la comunicación se desarrollan en ROS es necesario profundizar y mejorar las habilidades con este entorno.

2.2. Alcance del Trabajo

El presente proyecto se centra en el control, tanto manual como automático, de un vehículo y en la comunicación entre éste y un ordenador externo. Así pues, se pretende:

- Controlar el vehículo de forma manual mediante un mando Bluetooth con el que se ajusta la velocidad y la dirección. Además, por medio de este mando se pretende elegir entre el modo manual o automático, ejecutar una parada de emergencia y un reinicio del algoritmo de control cuando se necesite.
- Que el vehículo sea capaz de seguir un determinado circuito de forma autónoma, sin salirse de las líneas, gracias al uso de un sistema de navegación ultrasónico para interiores.
- Gestionar la información recibida por parte de los sensores a través de un ordenador externo para liberar de carga al procesador a bordo del vehículo. De esta forma, se pretende que el ordenador gestione la información recibida por parte de los sensores y el mando y ejecute el algoritmo desarrollado para obtener y enviar la velocidad y dirección pertinentes al procesador a bordo.

2.3. Metodología

La metodología escogida es de tipo ágil. Por definición, este tipo de metodología permite adaptar la forma de trabajo a las condiciones del proyecto, consiguiendo flexibilidad e inmediatez en la respuesta para amoldar el proyecto y su desarrollo a las circunstancias específicas del entorno [2]. En otras palabras, el proyecto se divide en pequeños objetivos que tienen que lograrse en un corto plazo de tiempo permitiendo flexibilidad y modelación.

Este tipo de metodología es la que se acostumbra a utilizar en proyectos relacionados con el software debido a que buscan entregar en poco tiempo pequeños programas en funcionamiento con el fin de aumentar la satisfacción del desarrollador.

El proyecto se ha dividido en las fases mostradas a continuación, algunas de las cuales han precisado de más tiempo para el aprendizaje y/o por los imprevistos surgidos. Destacar que se han incorporado diferentes elementos a medida que se iba avanzando, ya que este tipo de metodología lo permite.

Fases del proyecto:

1. Puesta a punto del entorno:
    - Instalación de Ubuntu en el ordenador y en el procesador a bordo del vehículo.
    - Instalación de ROS.
2. Implementación del hardware:
    - Elección y prueba de los componentes electrónicos.
    - Diseño de una PCB.
    - Montaje del vehículo.
3. Implementación del sistema de navegación ultrasónico para interiores:
    - Aprendizaje del software y hardware utilizado para su control.
    - Estudio del mejor posicionamiento.
    - Implementación del sistema en el laboratorio.
4. Desarrollo de los códigos:
    - Control del vehículo (control del driver del motor y el servomotor).
    - Control del mando Bluetooth.
    - Implementación del algoritmo encargado de la navegación autónoma.
5. Implementación en ROS:
    - Comunicación entre ordenador y vehículo.
    - Sistema de navegación.
    - Programas de control.
6. Redacción de la memoria.

3. Estado del arte

La tecnología siempre ha jugado un papel muy importante en la existencia del ser humano. Con ella, se ha intentado buscar soluciones a diferentes problemas con el fin de facilitar la vida en la sociedad. A día de hoy, la evolución de la ingeniería ha sido tan colosal hasta el punto que ha transformado la forma de vivir, comunicarse y relacionarse. El sector del transporte tampoco se queda atrás. En la automoción, la llegada de las nuevas tecnologías está ayudando al desarrollo de los vehículos autónomos cambiando la forma de desplazarse de un destino a otro.

3.1. Vehículo autónomo

Antaño, el sector de la automoción solo necesitaba de la parte mecánica y, gradualmente, se ha ido incorporando la electrónica y la informática a este sector. Hoy en día, la tendencia que se está siguiendo es capacitar al vehículo con tecnologías innovadoras con el propósito de ser capaces de hacerlo funcionar de manera autónoma y desde un punto de vista eléctrico, sin depender de fuentes de energía contaminantes.

3.1.1. Niveles de conducción

Hasta llegar a un vehículo totalmente autónomo, en el que el conductor pase a ser un ocupante más, sin estar pendiente de la conducción, la Sociedad de Ingenieros Automotrices (SAE) establecieron en 2015 los siguientes seis niveles de autonomía [3]:

- **Nivel 0.** Vehículos controlados 100% por humanos.
- **Nivel 1.** Vehículos con asistencia al conductor. Se incluyen los sistemas de frenado de emergencia y de velocidad de crucero y alerta de un cambio de carril involuntario.
- **Nivel 2.** Vehículo con asistencia avanzada al conductor. En este nivel, el automóvil tiene la posibilidad de controlar tanto la velocidad como la dirección por periodos cortos de tiempo cuando detecta peatones, obstáculos en puntos ciegos o señales de tráfico.
- **Nivel 3.** En este punto, el automóvil es capaz de conducir sin la interacción humana. Ahora bien, es necesario que el conductor esté en alerta por si tiene que tomar control si pasa algún imprevisto.
- **Nivel 4.** Vehículo autónomo en ciertas situaciones y ambientes previamente definidos, como, por ejemplo, un pueblo o una ciudad. En este nivel, es recomendable que el conductor se mantenga al tanto para intervenir cuando sea necesario. Además, el automóvil debe monitorizar en todo momento lo que va ocurriendo a lo largo del trayecto.
- **Nivel 5.** Vehículo completamente autónomo, en el que no existen ni volante ni pedales y donde el conductor es un ocupante más.

3.1.2. Ventajas y desventajas

Una de las grandes ventajas que presentan los vehículos autónomos es que al suprimir la interacción humana se eliminarán los errores cometidos por los conductores, de tal manera que los accidentes de tráfico se reducirán. Asimismo, el tráfico disminuirá debido a que es un problema surgido por la falta de coordinación. Otro punto importante, es que no habrá que preocuparse por encontrar aparcamiento, ya que el automóvil se mantendrá buscando sitio una vez que el pasajero se baje en su destino. Por último, se aprovecharía el tiempo debido a que no es necesario estar pendientes de los sucesos que van transcurriendo a lo largo del trayecto facilitando la elaboración de otras tareas.

En contrapartida, la fiabilidad y seguridad pueden ser una gran preocupación. Por una parte, pueden surgir problemas en el código o situaciones imprevistas, provocando que el coche reaccione inadecuadamente. Por otro lado, siempre existe el riesgo de los ataques informáticos y la pérdida de la privacidad, puesto que se necesita de conexión a la red donde se publica la ubicación del vehículo en tiempo real.

3.1.3. Actualidad y futuro

Actualmente, no existe ningún vehículo comercial que presente un nivel de autonomía 4 o 5, pero son muchos los fabricantes que están dedicando sus recursos a su investigación [4]. No obstante, cada vez es más usual encontrar vehículos de conducción semiautónoma que ofrecen asistencia al conductor. Los automóviles de última generación están equipados con todo tipo de sistemas de navegación, radares, sensores y dispositivos que ayudan a conocer lo que ocurre y hay a su alrededor. Con la información extraída, el vehículo es capaz de actuar en ciertas situaciones como, por ejemplo, cambiar la dirección si se desvía del carril o reducir la velocidad o incluso frenar si se detecta que se va a producir una colisión.

En relación con el vehículo completamente autónomo, a pesar de que no existe una fecha para que se transforme en una realidad, no cabe ninguna duda que su llegada va a suponer grandes cambios en el transporte. Se insinúa que estos se utilizarán más que los vehículos actuales [4]. Además, podría cambiar la costumbre de tener coches en propiedad y podrían pasar a ser de uso compartido.

3.2. Encaje del proyecto en la situación actual

El vehículo autónomo que se desarrolla en este proyecto encajaría dentro del nivel 4 de autonomía, puesto que funciona en escenarios conocidos y no es necesario un piloto que maneje el coche, aunque se incorpora un control manual mediante un mando inalámbrico para retomar el control cuando sea necesario o se desee. Asimismo, se monitoriza, en cada instante de tiempo, la velocidad, la dirección y las coordenadas del vehículo a través de un ordenador externo.

Lograr una conducción autónoma en espacios controlados, es un paso previo e imprescindible para llegar al vehículo de nivel 5 de autonomía. Por ello, los trabajos de investigación en este campo son esenciales si se desea en un futuro no muy lejano, conseguir una conducción en las ciudades sin accidentes, sin depender de fuentes de energía contaminantes y sin atascos.

En la industria y el sector empresarial, también, son muchas las expectativas puestas en los vehículos autoconducidos, ya que la navegación autónoma se puede aplicar tanto en automóviles que circular por carreteras o como en robots que se mueven libremente dentro de una fábrica o cualquier otro ambiente controlado, como es el propósito de este proyecto. Por ejemplo, dentro de una fábrica, se pueden emplear vehículos, como el de este trabajo, para realizar tareas de inspección de máquinas o movimientos de mercancías dentro de un circuito predeterminado.

4. Hardware

En este apartado se muestra un diagrama de bloques en el que se visualiza de manera gráfica los diferentes sistemas que forman el control del vehículo con sus respectivos componentes electrónicos. Posteriormente, se detallan los componentes empleados acompañados de un breve argumento de su utilización. Resaltar que, en el procesador a bordo del vehículo y el controlador del motor, se expone una comparativa entre los diferentes modelos planteados por tal de complementar la justificación de su elección. Finalmente, se presenta el esquema electrónico del vehículo.

4.1. Diagrama de bloques

En la **figura 4.1** se muestra el diagrama de bloques que representa los diferentes sistemas requeridos para la implementación de un vehículo autónomo.

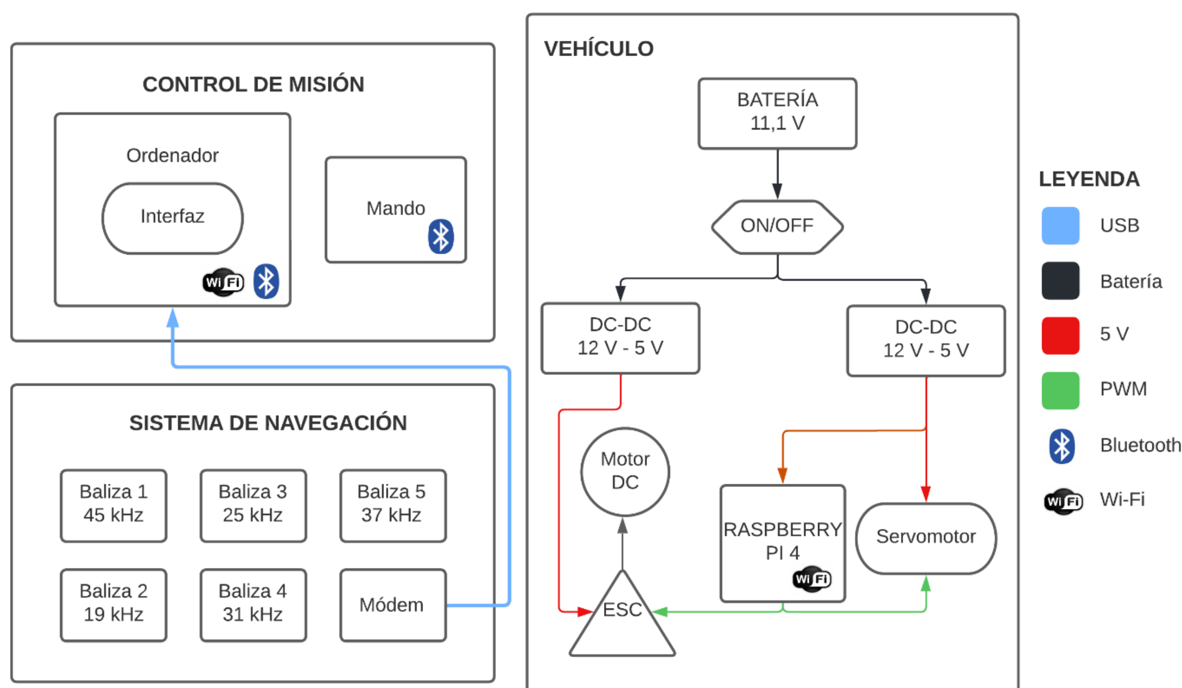


Figura 4.1. Diagrama de bloques con los componentes utilizados (Fuente: propia).

El control de misión está formado por un ordenador y un mando de la *Play Station 4* (PS4). El ordenador se ocupa de ejecutar los diferentes programas encargados del funcionamiento del sistema de navegación, del mando y del control autónomo del coche, y mediante una interfaz gráfica muestra los acontecimientos que van ocurriendo a lo largo del recorrido. A través del mando Bluetooth se elige el modo de funcionamiento (manual o automático), se puede realizar una parada de emergencia o un reinicio del algoritmo de control cuando se necesite y se cambia la dirección y la velocidad del vehículo cuando se esté en el modo manual.

El sistema de navegación está compuesto por cinco balizas ultrasónicas, cuatro de ellas están distribuidas por el circuito y son capaces de identificar la posición de la quinta baliza instalada en el vehículo. Además, se incluye un módem que se encarga de recibir la información captada por las balizas y enviarla al ordenador por USB.

Por último, los componentes incluidos en el coche son: una batería, dos convertidores DC-DC de 12 V a 5 V, un motor DC con su correspondiente controlador, un servomotor, una baliza y una *Raspberry Pi 4 Model B* encargada del control de los actuadores. Cabe destacar que la comunicación entre el ordenador y vehículo se realiza con wifi mediante ROS.

4.2. Raspberry Pi

La Raspberry Pi es un computador compacto de bajo coste destinado al desarrollo de proyectos electrónicos e informáticos. Es capaz de llevar a cabo la mayoría de las tareas características de un ordenador de escritorio, ya que puede ser conectada a un monitor o a una televisión y usarse con un teclado y un ratón. Además, está dotado con pines GPIO (*General Purpose Input/Output*) empleados para conectar todo tipo de periféricos, como sensores, actuadores o cámaras [5].

Este microordenador fue desarrollado en 2012 por la *Raspberry Pi Foundation* con la finalidad de fomentar y enseñar los pilares básicos de la computación en las escuelas y universidades. Sus diseños se basan en el hardware libre y utilizan *Raspberry Pi OS*, un sistema operativo de código abierto basado en Linux. Aunque para este proyecto se instala otra distribución diferente que se verá más adelante.

Esta placa es de suma importancia incorporarla en este proyecto, puesto que es la encargada de controlador los actuadores del vehículo a partir de las órdenes recibidas por el ordenador externo.

4.2.1. Elección y comparativa de diferentes modelos

Se ha decantado por la utilización de la *Raspberry Pi 4 Model B* de 8 GB de RAM debido a que proporciona la potencia necesaria para dotarla con Ubuntu, una distribución de Linux en la que se entrará más en detalle en el siguiente apartado. De todas formas, a continuación, se comparan los dos modelos estudiados para justificar la elección.

- **Raspberry Pi 3 Modelo B+**

La *Raspberry Pi 3 Model B+* incorpora un procesador ARM Cortex-A53, en concreto, el BCM2837 de 64 bits con cuatro núcleos que trabajan a una frecuencia de 1,4 GHz, una GPU (*Graphics Processing Unit*) VideoCore IV 400 MHz y una memoria RAM de 1 GB. Cuenta con soporte Bluetooth 4.2 y Wi-Fi 802.11ac de doble banda que permite conectarse a las redes 2,4 y 5 GHz. Además, dispone de 40 pines GPIO,

puertos HDMI, entrada Ethernet, salida CSI y DSI para la cámara y pantalla táctil, toma de auriculares, entrada micro SD, cuatro puertos USB 2.0 y entrada de alimentación de 5 V y 2,5 A en DC. [6]

Figura 4.2. Raspberry Pi 3 Modelo B+ (Fuente: [6]).

- **Raspberry Pi 4 Modelo B**

La *Raspberry Pi 4 Model B* es la última versión que se ha desarrollado y comercializado a día de hoy. Esta versión incorporar un procesador BCM2711B0 de 64 bits con cuatro núcleos Cortex-A72 de 1,5 GHz, una GPU VideoCore VI 500 MHz y una RAM de 8 GB. Dispone de Bluetooth 5.0 y Wi-Fi 802.11ac de doble banda. Asimismo, incluye 40 pines GPIO, dos micros HDMI, dos USB 2.0, dos USB 3.0, interfaz para cámara y pantalla táctil, toma de auriculares, lector micro SD. La alimentación puede ser suministrada o bien vía USB de tipo C o a través de los GPIO a 5 V y como mínimo 3 A. [7]

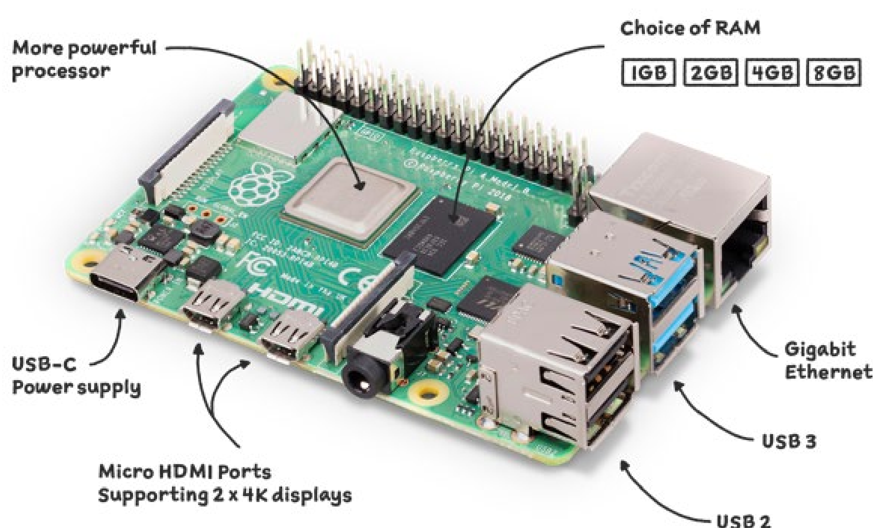


Figura 4.3. Raspberry Pi 4 Modelo B (Fuente: [7]).

- **Comparativa y elección entre los dos modelos**

Tabla 4.1. Comparativa entre la *Raspberry Pi 3 Model B+* y *Raspberry Pi 4 Model B (Fuente: propia).*

	RASPBERRY PI 3 MODEL B+	**RASPBERRY PI 4 MODEL B**
PROCESADOR	BCM2837B0 de 64 bits y con 4 núcleos	BCM2711B0 de 64 bits con 4 núcleos
FRECUENCIA DE RELOJ	1,4 GHz	1,5 GHz
GPU	VideoCore IV 400 MHz	VideoCore VI 500 MHz
RAM	1 GB	8 GB
CONECTIVIDAD INALÁMBRICA	Wi-Fi 2,4 GHz / 5 GHz Bluetooth 4.2	Wi-Fi 2,4 GHz / 5 GHz Bluetooth 5.0
PUERTOS	40 pines GPIO HDMI 4 x Micro USB 2.0 CSI (cámara Raspberry Pi) DSI (pantalla táctil) Toma de auriculares Lector Micro SD	40 pines GPIO HDMI 2 x Micro USB 2.0 2 x USB 2.0 2 x USB 3.0 CSI (cámara Raspberry Pi) DSI (pantalla táctil) Toma de auriculares Lector Micro SD
ALIMENTACIÓN	5 V DC vía micro USB (min 2,5 A)	5 V DC vía USB tipo C (min 3 A) 5 V DC vía GPIO (min 3 A)
PRECIO	35,15 €	67,76 €

Fijándose en el aspecto físico de las placas, no se halla prácticamente ninguna diferencia, ya que la posición y la disposición de los chips y conectores son la misma. Sin embargo, en las especificaciones de la **tabla 4.1** se observan diferentes cambios que hacen que la versión 4 sea más potente.

Como en este microordenador se instala ROS (un software que necesita como mínimo 1 GB de RAM) con el propósito de recibir la información enviada por el ordenador externo, es importante escoger la versión que disponga de un mejor procesador y más memoria RAM. De este modo, la velocidad de procesamiento será más elevada y, tanto el sistema operativo como ROS funcionarán de manera fluida. Por este motivo, los factores claves de la elección de la *Raspberry Pi 4 Model B* son la mejora del procesador que hace que la frecuencia de trabajo aumente y la alta memoria RAM que dispone.

4.3. Periféricos

En este apartado se explican los diferentes dispositivos utilizados en el sistema de navegación y en el vehículo.

4.3.1. Balizas ultrasónicas y módem

El sistema de navegación está compuesto por cinco balizas ultrasónicas que permiten localizar inalámbricamente robots, vehículos o personas dentro de un edificio. Además, incorpora un módem que recoge y envía al PC, mediante USB, la información captada por el conjunto de dispositivos.

Figura 4.4. Sistema de navegación: balizas y módem. Esquina inferior derecha se encuentra el módem y el resto son las balizas (Fuente: [8]).

En este proyecto se utilizan los siguientes dispositivos:

- 4 x balizas estacionarias distribuidas por el laboratorio
- 1 x Baliza móvil equipada en el vehículo
- 1 x Módem conectado al ordenador externo

El conjunto destaca por la exactitud en el posicionamiento de las balizas debido a que proporciona una precisión de hasta ±2 cm. Adicionalmente, presenta un batería de 1.000 mAh con la que se garantiza una autonomía de dos a seis días, según el modo de funcionamiento en el que trabaje el sistema.

La localización del vehículo se determina con el método matemático de trilateración. La trilateración se realiza a partir de la posición, previamente determinada, de las balizas estacionarias y el cálculo, en cada instante del tiempo, de la distancia entre cada baliza estacionaria y la baliza móvil mediante el principio *Time of Flight* (TOF).

El *Time of Flight* se basa en calcular la distancia con la diferencia de tiempo ente la emisión de un pulso ultrasónico y la recepción de éste [9]. Conociendo la velocidad del sonido, que es de 343,2 m/s a una temperatura de 20 ºC y una humedad relativa del 50 % [10], y la **ecuación 4.1** se obtiene la distancia real del automóvil.

$$d\ [m] = \frac{c\ \left[\frac{m}{s}\right] * t\ [s]}{2} \qquad \textbf{(Ec. 4.1)}$$

Donde *d* es la distancia, *c* la velocidad del sonido y *t* la diferencia de tiempo entre la emisión y recepción de la señal. El dividido entre dos se debe a que el pulso recorre el camino dos veces, ya que primero se dirige al vehículo hasta que rebota y vuelve al sensor que lo ha emitido. La **figura 4.5** es una representación gráfica de lo que sucede con la señal ultrasónica.

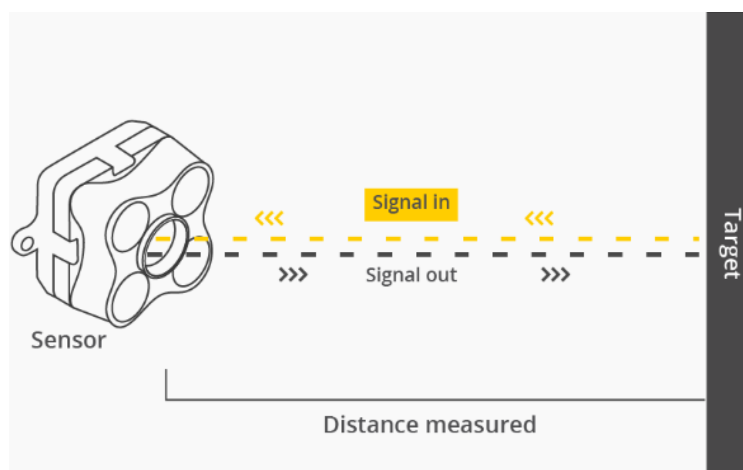


Figura 4.5. Representación gráfica del método *Time of Flight* (Fuente: [9]).

Tras calcular la distancia entre una baliza estacionaria y la móvil aplicando el TOF, se conoce que el vehículo se encuentra sobre el perímetro de una circunferencia cuyo centro sería la propia baliza y el radio, la distancia total hasta la baliza móvil.

Obteniendo la distancia de una segunda baliza estacionaria con la baliza móvil, se describe una segunda circunferencia que intersecciona en dos puntos con la anterior. En alguno de estos dos puntos se ubica el vehículo.

Teniendo la información de una tercera baliza que establece una tercera circunferencia, se determina la ubicación exacta del vehículo debido a que las tres circunferencias interseccionan en un único punto.

Añadiendo más balizas al sistema de navegación se obtiene un resultado más preciso. En la **figura 4.6** se muestra de forma gráfica el método de trilateración, en el que los nodos blancos son las posiciones conocidas de las balizas estacionarias y el nodo negro es la baliza móvil.

Destacar que se requiere de tres o más balizas estacionarias para el cálculo de la posición de la baliza móvil en dos dimensiones, y de cuatro o más si se desea localizar el vehículo en tres dimensiones. El número máximo de balizas utilizadas dependerá de la precisión y de las dimensiones del espacio interior en el que se tenga que implementar el sistema de navegación. Las balizas que se utilizan este proyecto garantizan una precisión de hasta ±2 cm a una distancia máxima entre balizas estacionarias de 10 m, siempre y cuando no haya obstáculos de por medio que interfieran en la señal ultrasónica. Así, si se quiere conservar la precisión, cuanto mayor son las dimensiones del espacio interior, mayor es el número de balizas a emplear.

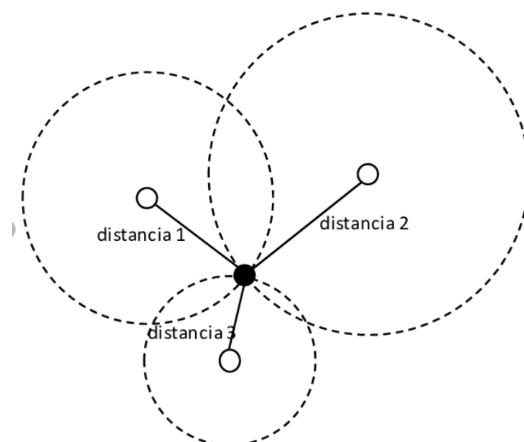


Figura 4.6. Representación gráfica del método de trilateración. Posición de las balizas estacionarias (nodos blancos) y posición de la baliza móvil (nodo negro) (Fuente: [12]).

4.3.2. Mando Bluetooth

El control manual del vehículo se realiza mediante el uso de un mando externo con conectividad Bluetooth. En particular, se dispone del mando inalámbrico DUALSHOCK 4 de la Play Station 4. Su elección se debe a que hay una librería en Python que ofrece facilidad a la hora de su control.

Figura 4.7. Mando inalámbrico DUALSHOCK 4 (Fuente:[13]).

Según se observa en la **figura 4.7**, en la parte izquierda del mando inalámbrico se encuentran cuatro botones de dirección, el botón *SHARE* y los botones L1 y L2. En lado derecho están situados, el botón *OPTIONS*, los botones R1 y R2 y cuatro botones representados con figuras geométricas: un triángulo, un círculo, una cruz y un cuadrado. Además, dispone de dos joysticks, el botón PS y un panel táctil.

4.3.3. Motor DC y controlador

Los motores DC, o motores de corriente continua, convierten la energía eléctrica en mecánica, provocando un movimiento rotatorio que gracias a una serie de engranajes hacen que el vehículo se mueva. El motor utilizado, mostrado en la **figura 4.8**, está preparado para funcionar a una tensión máxima de 12 V, pero por sí solo no se consigue regular la velocidad. Se necesita de un controlador para ir variando la tensión suministrada al motor y hacer que éste gire a la velocidad deseada y cambiar su sentido de giro.

Figura 4.8. Motor DC (Fuente: [14]).

El controlador es el encargado de cambiar la dirección de giro y generar una señal PWM (*Pulse Width Modulation*) que mediante el ancho de pulso establecido varía la velocidad. En concreto, se ha decantado por un ESC (*Electronic Speed Control*), también conocido como variador, debido a que proporciona la potencia adecuada para este tipo de aplicaciones. De todos modos, a continuación, se comentan y se comparan las dos opciones que se barajaban desde un principio, justificando su elección.

- **L298N**

El L298N permite controlar hasta dos motores de corriente continua o motores paso a paso desde un microcontrolador o un microprocesador, variando la velocidad al igual que la dirección de giro. En la **figura 4.9** se muestra una imagen de éste.

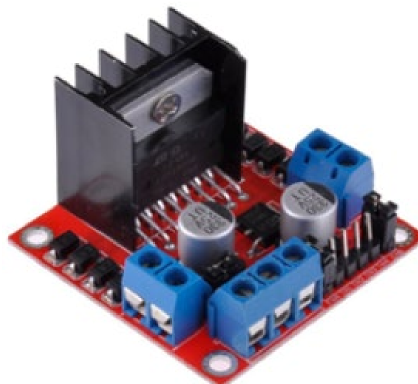


Figura 4.9. Controlador L298N (Fuente: [15]).

El rango de tensión que suministra al motor va desde los 3 V hasta los 35 V, con una corriente de salida constante de 2 A y de pico de 3 A. Integra protecciones contra sobrecorrientes y sobrecalentamientos, y está equipado con diodos *flyback* que protegen contra las intensidades inducidas [15].

El funcionamiento de este controlador se base en dos puentes-H, uno para cada salida de motor, con los cuales se consigue invertir el sentido de la corriente, de tal manera que haga cambiar la dirección de giro del motor.

El puente-H está constituido por cuatro transistores, conectados entre Vcc y GND, con la carga a alimentar entre ellos, dibujando una “H”. Por vía de los cuatros transistores se cambia el sentido de la intensidad, encendiendo los transistores opuestos en diagonal de cada rama, tal y como muestra la **figura 4.10**. De este modo, se consigue cambiar el sentido de giro del motor. Adicionalmente, si se conectan los transistores inferiores o superiores al mismo tiempo se activará el freno.

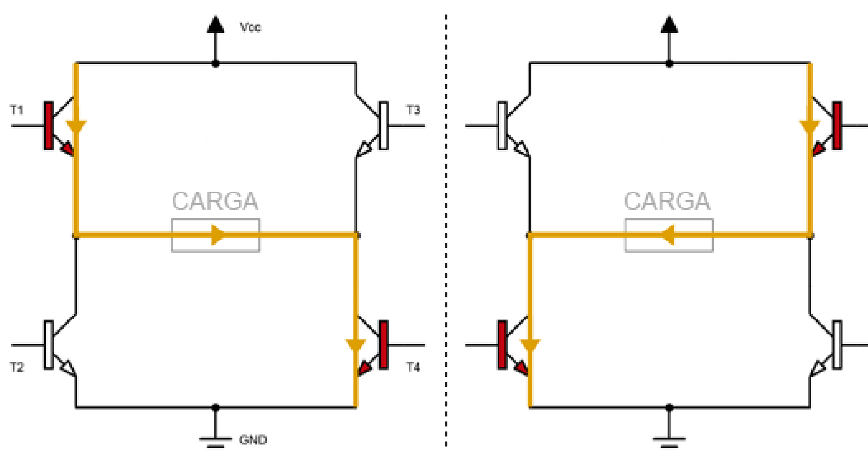


Figura 4.10. Principio de funcionamiento del puente-H (Fuente: [15]).

El controlador L298N, dispone de dos pines, IN1 y IN2 (IN3 y IN4 para la salida del segundo motor), que controlan la activación de los transistores. Además, tiene un tercer pin, IEA (IEB para la segunda salida) que desactiva los cuatro transistores, desconectando la carga por completo, y controla la velocidad de giro del motor si la señal recibida es un PWM [15].

En la siguiente tabla, se muestra un resumen del comportamiento del motor según el estado de los pines de control de los transistores.

Tabla 4.2. Comportamiento del motor según el estado de los pines IN1 y IN2 (IN3 y IN4 para la segunda salida del motor) (Fuente: propia).

IN1 (o IN3)	IN2 (o IN4)	Estado del motor
HIGH	LOW	Giro hacia adelante
LOW	HIGH	Giro hacia atrás
LOW	LOW	Freno

- **ESC (*Electronic Speed Control*)**

Un ESC es un dispositivo capaz de convertir la tensión DC de entrada en una tensión DC ajustable a la salida en función de la señal PWM que se aplique.

El TAS-202 de Avioracing, es el ESC seleccionado, mostrado en la **figura 4.11**. Se trata de un controlador fabricado para vehículos de radiocontrol que se alimenta entre 4,8 V y 7 V y suministra un corriente de hasta 10 A. Su frecuencia de trabajo es de 50 Hz y según el ancho de pulso establecido por su pin de control se consigue el control de la velocidad y del sentido de giro del motor. Aplicando un ancho de pulso de 1,5 ms el motor permanecerá parado, ante un pulso de entre 1,6 a 1,9 ms girará hacia adelante, incrementando la velocidad a medida que el ancho incremente y con un pulso entre 1,2 y 1,45 ms, girará hacia atrás, decreciendo la velocidad a medida que se incremente el pulso.

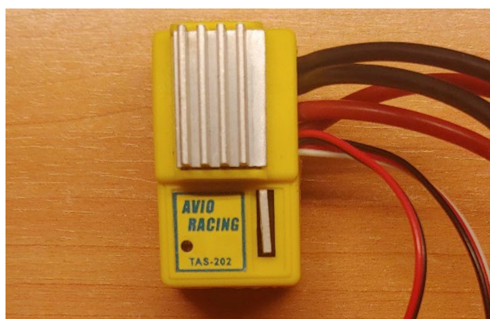


Figura 4.11. TAS-202 de Avioracing (Fuente: propia).

- **Comparativa y elección**

Tabla 4.3. Comparativa entre el L298N y el TAS-202 (Fuente: propia).

	L298N	**TAS-202**
TENSIÓN DE ALIMENTACIÓN	6 V a 35 V	4,8 V a 7 V
CORRIENTE PROPORCIONADA	2 A constantes 3 A de pico	10 A constantes 20 A de pico
PRINCIPIO DE FUNCIONAMIENTO	Puente-H cambio de dirección PWM variación velocidad	PWM variación velocidad y dirección
PINES	IN1, IN2, IEA motor uno IN3, IN4 y IEB motor	Un único pin para el control del PWM
ESTADOS DEL MOTOR	Hacia adelante, hacia atrás y freno	Hacia adelante, hacia atrás y freno
PRECIO	7,99 €	10,99 €

A partir de las características comparadas en la **tabla 4.3** y de la realización de una prueba experimental en la que se han implementado los dos controladores en el vehículo con la finalidad de averiguar la velocidad que podían proporcionar, se ha determinado que el mejor controlador para este proyecto es el ESC TAS-202.

A pesar del gran rango de tensión de alimentación y el bajo precio del L298N, se ha optado por implementar el ESC TAS-202. El L298N presenta una eficiencia baja debido a que la tensión recibida por el motor se encuentra 3 V por debajo de la tensión de alimentación. Además, la potencia suministrada es mucho menor que la del TAS-202. Por otro lado, durante la prueba realizada, se ha podido observar que el TAS-202 proporcionaba un rango mayor de velocidad, siendo la velocidad mínima y máxima inferior y superior a la proporcionada por el L298N.

4.3.4. Servomotor

El servomotor elegido es el modelo PDI-1181MG, tal y como se muestra en la **figura 4.12**. Éste se encarga del control de la dirección del vehículo y funciona con un rango de tensión entre 4,8 V y 6 V, aunque es recomendable alimentarlo a 5 V. Su frecuencia de trabaja es de 300 Hz y el par de bloqueo ofrecido es de 3,5 kg/cm [16].

Figura 4.12. Servomotor PDI-1181MG (Fuente: [16]).

El posicionamiento del servomotor se determina a partir del ancho de pulso de la señal PWM generada por la Raspberry Pi. De esta manera, para conseguir una orientación de 0º el pulso ha de tener un ancho de 1 ms, con un ancho de 1,5 ms se situará en 90º y ante 2 ms en 180º.

Por último, como la corriente que puede suministrar el microordenador no es suficiente, se alimenta a través de un convertidor DC-DC.

4.3.5. Batería

La batería es el elemento que almacena la energía que alimentará al vehículo. Está compuesta por una serie de celdas electroquímicas que convierten la energía química en eléctrica.

La elección de la batería se ha realizado mediante un estudio en el que se ha observado que las baterías de LiPo son las más utilizadas en la actualidad en el mundo de los vehículos radio control debido a su alta densidad energética y su ratio de descarga. Por ello, se selecciona una batería de LiPo de 11,1 V y 5200 mAh [17].

Figura 4.13. Batería LiPo de 11,1 V y 5200 mAh (Fuente: [17]).

A partir de la capacidad de la batería, una aproximación del consumo del coche de 5 A (a máxima potencia) y la **ecuación 4.2** se deduce su duración.

$$Duración\ [h] = \frac{Capacidad\ Bateria\ [mAh]}{Consumo\ [mA]} \quad \textbf{(Ec. 4.2)}$$

Así, la duración obtenida de la batería es de alrededor una hora.

4.3.6. Reguladores de tensión

El regulador de tensión es preciso a la hora de suministrar una apropiada tensión a la Raspberry Pi, al servomotor y al controlador del motor DC. Además, se encarga de proteger los diferentes equipos electrónicos de sobretensiones y sobrecorrientes y de estabilizar eficientemente el voltaje.

En este proyecto se utilizan dos reguladores con la finalidad de reducir y estabilizar el voltaje de la batería a 5 V. Se emplean dos diferentes, puesto que la Raspberry Pi y el ESC presentan un consumo tan elevado que un solo regulador no es capaz de proporcionar. Además, se consigue que el motor no introduzca posibles perturbaciones o ruidos en la alimentación que perjudiquen al resto de componentes.

- **Regulador JZK**

El modelo escogido para suministrar la correcta tensión al servomotor y a la Raspberry Pi es el JZK. Éste es un convertidor reductor que presenta dos entradas que soportan una tensión de 9 a 36 V y dos salidas de 5 V y 5 A cada una.

Figura 4.14. Convertidor JZK (Fuente: [18]).

- **Regulador S13V305F5**

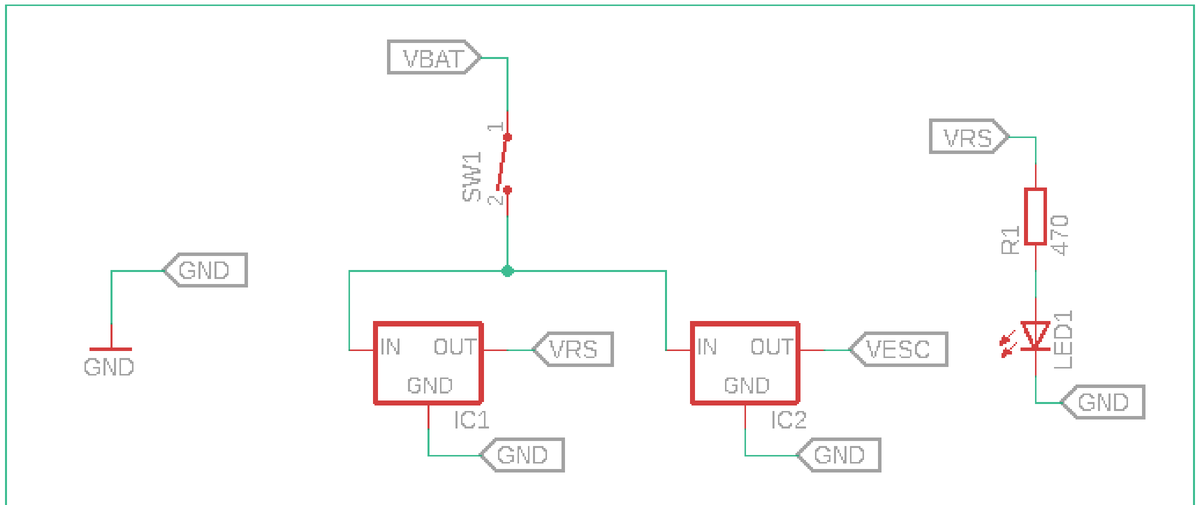
El ESC lleva su propio regulador por su elevada potencia de consumo. En concreto, se utiliza el convertidor S13V30F5 que es capaz de suministrar eficientemente 5 V y hasta 4 A de corriente continua a partir de un voltaje de entrada de entre 2,8 V a 22 V. Además, incorpora protección de voltaje inverso, bloqueo por bajo voltaje, protección contra sobretensiones y sobrecorrientes y una función de apagado térmico [19].

Figura 4.15. Regulador S13V30F5 (Fuente: [19]).

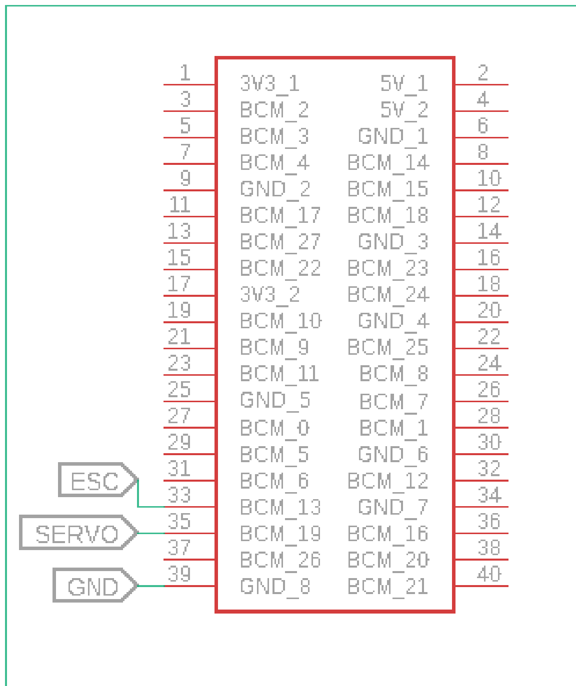
4.4. Esquema electrónico

A continuación, se presenta el esquema electrónico del vehículo, realizado mediante el software Eagle.

Batería y reguladores



Raspberry Pi 4 Model B



Actuadores y ESC

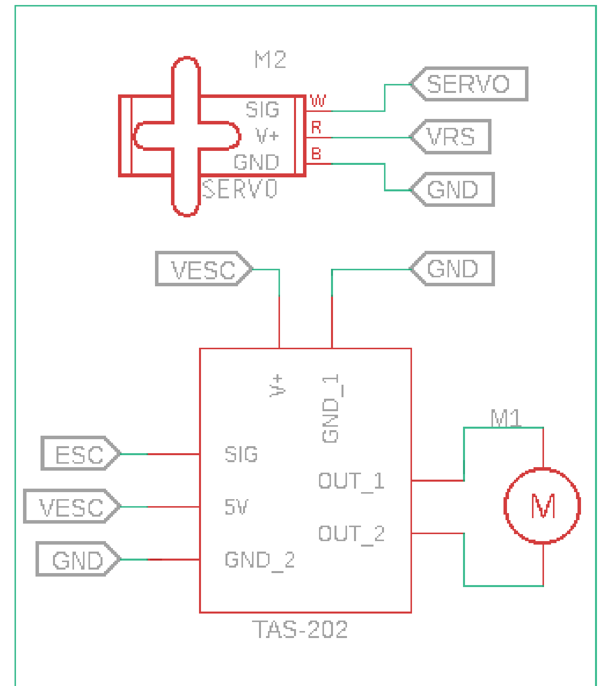


Figura 4.16. Esquema electrónico (Fuente: propia).

5. Software

En este apartado se explican brevemente los diversos softwares utilizados, detallando los conceptos claves para el adecuado uso de estos y el entendimiento de los posteriores códigos desarrollados.

5.1. Ubuntu

La elección y la instalación del sistema operativo en la *Raspberry Pi 4 Model B* y en el ordenador externo es fundamental. Con éste se consigue coordinar el hardware y administrar los directorios y recursos de un equipo. Adicionalmente, se facilita la interacción entre el usuario y los programas del computador a través de una interfaz gráfica.

Como ya se ha comentado previamente, aunque la Raspberry Pi dispone de su propio sistema operativo, *Raspberry Pi OS,* se opta por emplear *Ubuntu 20.04.4 LTS*. Esto se debe a que éste es uno de los pocos sistemas operativos capaz de ejecutar ROS.

Figura 5.1. Logotipo de Ubuntu (Fuente: [20]).

Ubuntu es una distribución de código abierto basada en Debian, otro sistema operativo desarrollado a partir de Linux, que destaca por su facilidad de uso y simplicidad. Este software se lanzó por primera en 2006 por la empresa Canonical, quien actualmente se ocupa de su mantenimiento y del desarrollo de nuevas versiones.

5.2. Robot Operating System (ROS)

Robot Operating System (ROS) es un *framework* [1] de código abierto libre que suministra un conjunto de herramientas y librerías para el desarrollo del software de robots. Surgió en 2007 en la Universidad de Stantford a manos de Eric Berger y Keenan Wyrobek, dos estudiantes que estaban trabajando en su tesis. Desde el 2008 el desarrollo y mantenimiento continua gracias a la *Open Source Robotics Foundation* (OSRF).

Figura 5.2. Logotipo de ROS (Fuente: [23]).

A día de hoy, ROS cuenta con diferentes versiones que son utilizadas dependiendo del sistema operativo seleccionado. Por ello, como la versión de Ubuntu escogida es la 20.04.4 LTS se recomienda instalar la versión *ROS Noetic Ninjemys* [24].

Figura 5.3. Logotipo de *ROS Noetic Ninjemys* (Fuente: [24]).

ROS está compuesto por dos partes esenciales: el sistema operativo y ROS *packages*.

Aunque ROS no es un sistema operativo como tal, ofrece los mismos servicios: abstracción del hardware, gestión de directorios, control de periféricos externos, comunicación entre procesos del mismo y/u otros dispositivos e implementación de interfaces gráficas. Su principal fundamento se basa en ejecutar un gran número de programas en paralelo que intercambian información entre sí en

[1] *Framework*: estructura de trabajo empleada comúnmente por programadores para el desarrollo de software. Contribuye a organizar el desarrollo de programas informáticos, permitiendo reutilizar códigos de varios proyectos [22].

tiempo real, garantizando un acceso eficiente a los recursos del autómata. Por ejemplo, en un robot existen diferentes ejecutables que se encargan de la lectura de los sensores, del proceso de datos y del control de los actuadores. Estas tareas deben realizarse de forma continuada y en paralelo, enviando y recibiendo información entre ellas para el correcto funcionamiento del robot.

Por otro lado, el ROS *packages* es el conjunto de paquetes aportados por los usuarios, generalmente independientes a los robots, que implementan una serie de capacidades como la navegación, la localización y el mapeo.

5.2.1. Conceptos básicos

Seguidamente, se explican los conceptos básicos de ROS necesarios a la hora de su uso y entendimiento de los programas desarrollados.

- **Nodes**

Un *node*, o nodo en castellano, es una instancia de un ejecutable que puede equivaler a un sensor, un actuador, un algoritmo, etcétera. Los programas en ROS están formados por un conjunto de nodos capaces de comunicarse entre sí.

- **Master**

El *master*, o maestro, es un nodo intermediario que ayuda a conexionar los diferentes nodos, estableciendo la comunicación entre ellos. Éste conoce todos los detalles sobre los nodos existentes y sin él, los nodos no lograrían localizarse e intercambiar mensajes. Cabe destacar que solamente puede existir un nodo *master*.

- **Topics**

Un *topic*, o tópico, es el canal de transporte de datos que permite la comunicación entre nodos. A cada canal se le asigna su propio nombre y uno o más nodos pueden enviar datos en un tópico, mientras que uno o más pueden recibirlos.

- **Messages**

Los mensajes, o *messages* en inglés, son unas estructuras de datos empleadas por los nodos para intercambiar información a través de los *topics*. En ROS se permiten tanto los mensajes de tipo primitivos (booleanos, caracteres, enteros, reales, etc) como las matrices compuestas a partir de los primitivos.

Los tipos de datos primitivos que se admiten en ROS están expuestos en la **tabla 5.1.**

Tabla 5.1. Mensajes primitivos de ROS con su correspondencia en el lenguaje C++ y Python (Fuente: [26]).

ROS	C++	Python
bool	uint8_t	bool
int8	int8_t	int
uint8	uint8_t	int
int16	int16_t	int
uint16	uint16_t	int
int32	int32_t	int
uint32	uint32_t	int
int64	int64_t	int
uint64	uint64_t	int
float32	float	float
float64	double	float
string	std::string	byte
time	ros::Time	rospy.Time
duration	ros::Duration	rospy.Duration

- **Publisher y Subscribers**

Siendo conscientes de los conceptos vistos hasta ahora: *node*, *master*, *topic* y *messages,* se introducen el *Publisher* y el *Subscriber*, términos imprescindibles para comprender como se lleva a cabo la comunicación entre los diferentes nodos mediante ROS.

Un nodo puede comportarse como *Publisher* y/o *Subscriber* según si publica y/o envía información. Por consiguiente, un nodo *Publisher* tiene la capacidad de enviar mensajes del tipo estandarizado a uno o varios *topics*. Por otro lado, el *Subscriber* se subscribe a uno varios *topics* para recoger los mensajes que se vayan publicando.

A continuación, se muestra un ejemplo en el que dos nodos se comunican entre sí.

Como se puede observar en la **figura 5.4**, se crean dos nodos llamados *Talker* y *Listener*. El *Talker* publica el mensaje *Hello World* de tipo *string* en el topico */talker* y el *Listener* se subscribe a éste.

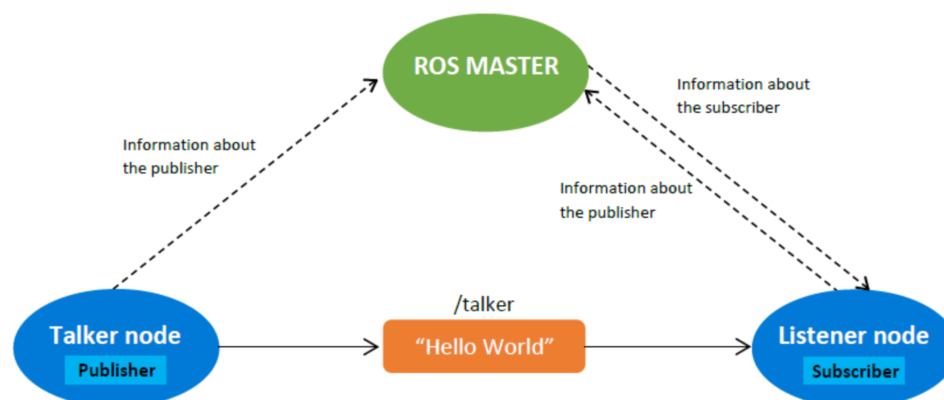


Figura 5.4. Comunicación entre nodos utilizando *topics* (Fuente: propia).

Con solo ejecutar estos dos nodos, el proceso no funcionará. Por esta razón, antes de empezar es importante ejecutar el *ROS Master*. Cuando se inicia el nodo *Talker*, primero se conecta al *ROS Master* y le enseña las características del tópico creado: nombre del tópico, tipo del mensaje y dirección URI[2]. Seguidamente, se ejecuta el nodo *Listener* que se vincula al *ROS Master* para intercambiar los detalles del nodo, como el tópico al que se va a subscribir, el tipo de mensaje a recibir y su URI. De esta manera, en el caso de que haya un *Publisher* y un *Subscriber* en el mismo tópico, el master se encarga de intercambiar la URI del publicador con la del subscritor, hecho que ayuda a que los dos nodos se conecten y empiezan a traspasarse información.

- **Comunicación master-slave**

ROS da la posibilidad de ejecutar nodos en diferentes dispositivos siempre y cuando estos se encuentren en la misma red. Así, un nodo puede inicializarse en la máquina cuyos recursos sean los suficientes para poder llevar a cabo las acciones descritas en éste. Sin embargo, hay que tener en cuenta algunas excepciones, por ejemplo, no se va a poder iniciar un nodo que controle algún actuador especifico de esa máquina.

La comunicación en ROS se basa en la arquitectura *master-slave*, o maestro-esclavo. El dispositivo maestro empieza la comunicación con los esclavos y permite el envío y la recepción de datos entre ellos. El maestro solo puede ser una única máquina mientras los esclavos puede ser más de una. Lo único necesario para esta comunicación es saber la dirección IP de los dispositivos.

- **Archivo Launch**

Es una herramienta para iniciar, gestionar y detener cómodamente el *master* y los múltiples nodos.

[2] URI (*Uniform Resource Identifier*): identificador utilizado para todo tipo de direcciones que se refieren a objetos de Internet, como páginas, documentos, imágenes, videos, etcétera [27].

5.2.2. Comandos

A continuación, se describen los comandos más típicos de ROS que se emplean a través de la terminal[3] de Ubuntu. Hay que tener presente que estos son sensibles al uso de las mayúsculas.

- **roscore.** Es el primer comando que hay que ejecutar siempre, ya que es el que se encarga de la inicialización del *master*.
- **rosrun [nombre del paquete] [nombre del ejecutable].** Lanza el programa ejecutable y crea el nodo correspondiente.
- **roslaunch [nombre del paquete] [nombre del archivo *launch*].** Ejecuta el *master* y los nodos a partir del archivo *launch* e incluye soporte para la ejecución de la comunicación maestro-esclavo.
- **rosnode.** Proporciona información de depuración sobre los nodos, esto incluye el tipo de nodo y sus conexiones. Este comando debe ser acompañado por uno de los siguientes parámetros:
    - **list.** Muestra una lista con los nodos activos.
    - **machine.** Muestra una lista de los nodos activos en el dispositivo.
    - **info [nombre del nodo].** Muestra la información del nodo.
    - **kill [nombre del nodo].** Para el nodo.
- **rostopic.** Muestra por la terminal la información de los tópicos, incluyendo el nodo *Publisher*, el *Subscriber*, la frecuencia de envió y el mensaje. Éste también debe ser acompañado por uno de los siguientes parámetros:
    - **list.** Muestra una lista de los tópicos activos.
    - **info [nombre del nodo].** Muestra la información del tópico.
    - **find [nombre del tópico].** Encuentra el tópico por el tipo.
    - **echo [nombre del tópico].** Muestra el mensaje que contiene el tópico.
    - **hz [nombre del tópico].** Muestra la frecuencia de publicación del mensaje.
    - **pub [nombre del tópico] [tipo de mensaje] [mensaje].** Publica un mensaje.
- **rqt_graph.** Muestra un gráfico de los nodos activos con sus correspondientes comunicaciones a través de los tópicos conectados.
- **Roswtf.** Diagnostica los problemas surgidos mientras se ejecuta un sistema de ROS.

[3] Terminal: herramienta que permite la comunicación entre usuario y máquina a partir de unas líneas de comandos escritas por el usuario. Antes de la llegada de las interfaces gráficas, éste era el único sistema por el cual el usuario interactuaba con el computador [28].

5.3. Python

El lenguaje de programación seleccionado para el desarrollo de los diferentes códigos es Python, ya que se tiene experiencia previa por estudiar los conceptos básicos en primero de ingeniería y se ha utilizado en la elaboración de otros proyectos transcurridos a lo largo de la carrera.

Figura 5.5. Logotipo de Python (Fuente: [29]).

Python es un lenguaje de alto nivel orientado, principalmente, a la programación de objetos[4] muy popular por su clara sintaxis y legibilidad. ROS facilita la librería *rospy*, la cual capacita a Python a interactuar con los nodos, tópicos y diversos parámetros de ROS.

5.4. Marvelmind Dashboard

El Dashboard es una aplicación de Windows y Linux desarrollada por la empresa Marvelmind con el propósito de configurar y sintonizar el sistema de navegación para interiores, visto anteriormente. Además, también admite realizar un seguimiento de los datos obtenidos por las balizas y visualizarlos por medio de una interfaz gráfica.

Esta aplicación solo se usará para la configuración inicial del sistema de navegación, puesto que la empresa de las balizas proporciona un paquete, que incluye librerías y códigos, para poder volcar diferentes datos a ROS.

[4] Programación Orientada a Objetos (OOP): modelo o estilo de programación informático que estructura el código en torno a objetos, en vez de funciones. Se utiliza para organizar un programa en piezas simples y reutilizables con el fin de establecer instancias individuales de objetos. Un objeto se puede definir como un campo de datos que presenta un estado y un comportamiento único [30].

6. Implementación

Seguidamente, se especifica con todo detalle el proceso de implementación. Se describe el montaje del vehículo, el posicionamiento y la calibración del sistema de posicionamiento para interiores, la configuración de la comunicación *master-slave* y los códigos que dan forma a los algoritmos de control del movimiento manual y autónomo del vehículo.

6.1. Montaje del vehículo

El montaje del vehículo se realiza sobre el D12 Kei Truck, una camioneta de radiocontrol, mostrada en la **figura 6.1**. A partir de su estructura se realizan una serie de modificaciones electrónicas con el fin de incrementar su autonomía y lograr el control de la velocidad y dirección por medio de la *Raspberry Pi 4 Model B*.

Figura 6.1. Automóvil D12 Kei Truck (Fuente: [31]).

Para ello, primero se estudia la distribución de sus componentes electrónicos. Como se puede observar en la **figura 6.2**, la camioneta, está equipada con un interruptor, dos LEDs frontales, un servomotor, un motor de corriente continua, una batería de 7,4 V y 500 mAh y una placa encargada del control de los actuadores y de la correcta alimentación de los diferentes periféricos, además de la comunicación por radio.

Así que, para poder alcanzar los objetivos impuestos, se inhabilita el uso del interruptor y se extrae la placa electrónica y la batería, dejando el servomotor de dirección y el motor DC.

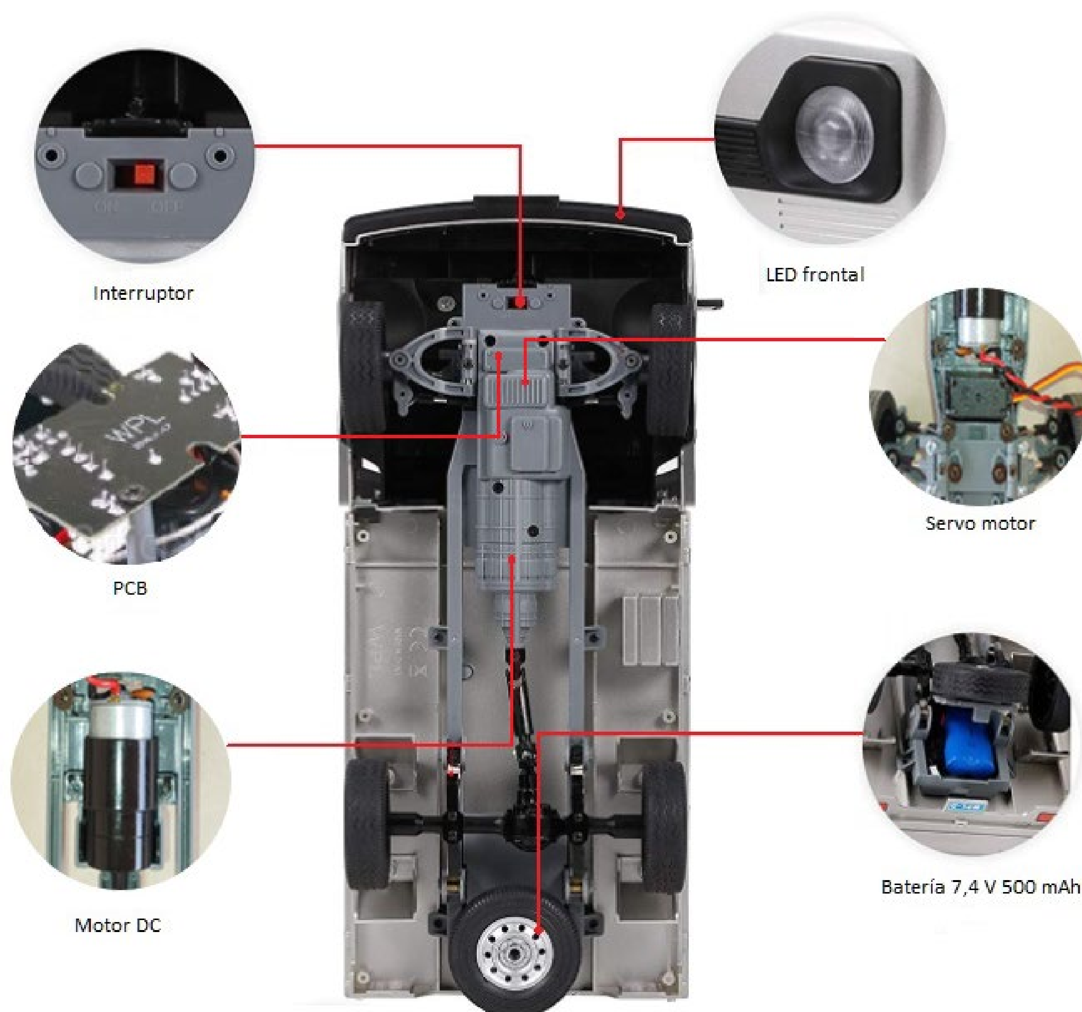


Figura 6.2. Componentes que incorpora el D12 Kei Truck con su respectiva ubicación (Fuente: propia).

Seguidamente, se suelda sobre una placa de topos un conjunto de conectores con la finalidad de agrupar las diferentes conexiones y evitar desconexiones durante el recorrido del vehículo. Adicionalmente, se suelda un interruptor para facilitar el encendido y el apagado de los equipos electrónicos, y se añade un LED de color rojo con su respetiva resistencia para indicar de forma visual que los dispositivos se encuentran alimentados. Resaltar que la placa de topos se ha elaborado siguiendo el esquema electrónico del **apartado 4.4.**

Posteriormente, se recorta una madera con unas dimensiones de 13,5x16 cm y se distribuyen encima de ella la placa de topos realizada, la *Raspberry Pi 4 Model B*, el controlador TAS-202 y los reguladores. Los componentes se anclan con tornillos o con cinta adhesiva de doble cara y velcro. De esta manera, se evita que los componentes se muevan y se desconecten. En la **figura 6.3** se observa el resultado de este paso.

Figura 6.3. Distribución de los componentes en la madera (Fuente: propia).

Finalmente, como se muestra en la **figura 6.4**, se sitúa la madera en la parte trasera del vehículo, se posiciona la batería en el espacio sobrante, se realizan las conexiones pertinentes y se coloca la baliza móvil en la parte más alta del vehículo (resaltar que la baliza móvil es la que se observa, en la **figura 6.4** en la parte derecha del vehículo, justo encima de la cabina).

Figura 6.4. Resultado final del montaje del vehículo (Fuente: propia).

6.2. Posicionamiento y calibración de las balizas ultrasónicas

En este apartado se presenta la implementación de las balizas una vez se han configurado tal y como se indica en el **anexo A**. En dicho anexo se detalla paso a paso cómo instalar la aplicación Marvelmind Dashboard para la configuración del sistema de navegación y cómo actualizar el software de las balizas. Adicionalmente, se enseña a cambiar los parámetros más importantes de las balizas para obtener mejores resultados

6.2.1. Colocación de las balizas

Antes de empezar a posicionar las balizas estacionarias por el laboratorio que se encuentra el circuito, es de suma importancia indicar, primeramente, al programa Marvelmind Dashboard cuál va a ser la baliza móvil y cuales las estacionarias. En el **apartado A2.4** se indica cómo hacerlo.

En este proyecto, se establece que las balizas cuyas direcciones son 1, 2, 3 y 5 serán las estacionarias, mientras que la 4 será la móvil. De esta manera, las balizas 1, 2, 3, y 5 se distribuyen por el laboratorio y la baliza 4 se coloca horizontalmente en la posición más alta del vehículo, como se puede observar en la **figura 6.4**, para evitar que el propio vehículo obstaculice la señal ultrasónica.

Por otro lado, las balizas estacionarias se deben de situar verticalmente a una altura superior a la móvil y con una cierta inclinación, de modo que miren hacia el suelo. Así, proporcionaran cobertura a todo el circuito, evitando los objetos que puedan ocasionar obstrucciones en la señal ultrasónica.

Otro punto a tener en cuenta es la distancia entre las balizas estacionarias. Según el *datasheet*, la máxima distancia para asegurar una precisión de ±2 cm es de 10 m [8]. El circuito del laboratorio mide 2,8 m de ancho y 8,80 m de largo, así que en un principio se podría cubrir todo el circuito. Sin embargo, tras realizar unas series de pruebas en el laboratorio se ha observado que la distancia máxima a la que se pueden colocar es de 5 m. Este hecho se debe a que hay un armario metálico en uno de los laterales que provoca rebotes de la señal ultrasónica, originado ecos y ruidos. Por lo tanto, se cubre la mitad del circuito, colocando las balizas estacionarias tal y como se muestra en la **figura 6.5**. En la siguiente tabla se recogen las distancias entre balizas, que es importante calcular y apuntar para a posteriori anotarlas en la aplicación Marvelmind Dashboard y poder aplicar la trilateración adecuadamente.

Tabla 6.1. Posición entre balizas (Fuente: propia).

Dirección	1	2	3	5
1		3,40 m	4,16 m	2,80 m
2	3,40 m		2,00 m	4,16 m
3	4,16 m	2,00 m		3,40 m
5	2,80 m	4,16 m	3,40 m	

En la siguiente figura se muestra la mitad del circuito que recorrerá el vehículo con la posición de las balizas estacionarias.

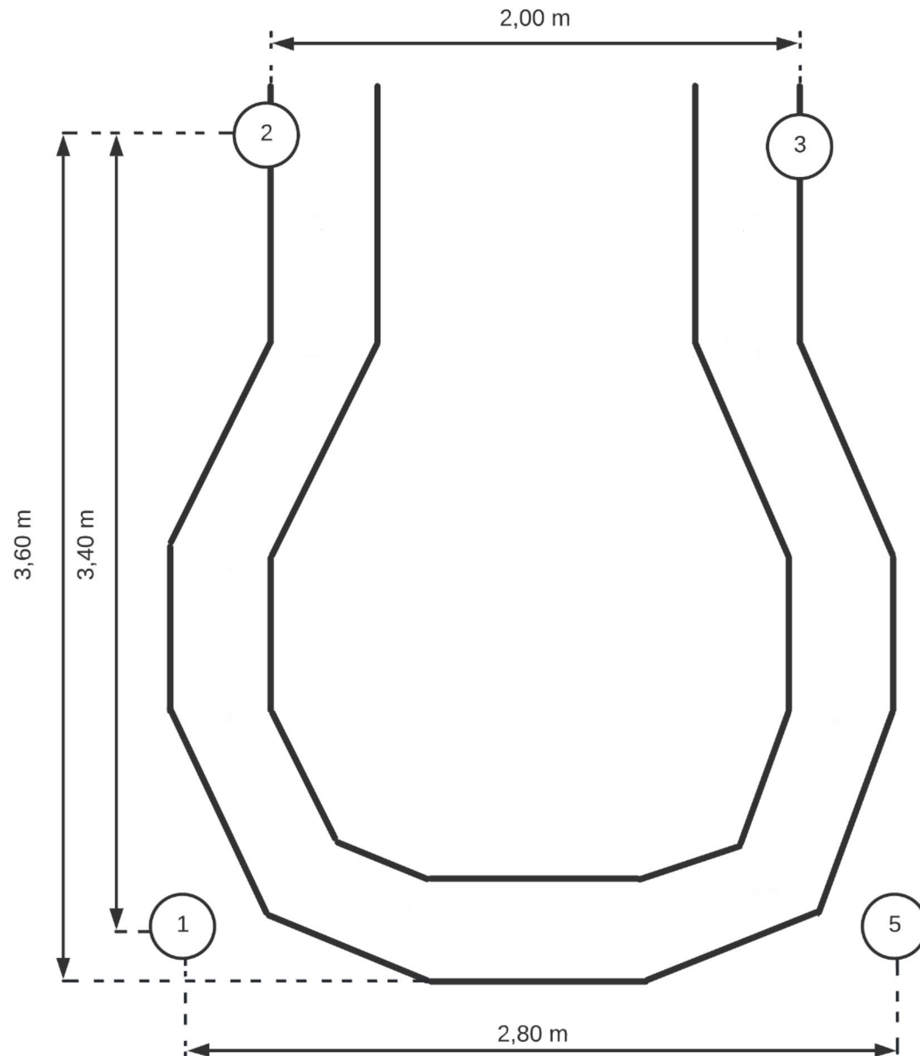


Figura 6.5. Plano del circuito con la posición de las balizas (Fuente: propia).

Por último, es importante establecer y anotar, también, las coordenadas de las balizas estacionarias sobre un plano XY. Éstas deben ser acordes con las posiciones de las balizas de la **figura 6.5** y tener coordenadas positivas, éste último requisito se impone para facilitar la programación. Así, las coordenadas de cada baliza estacionaria se recogen en la siguiente tabla.

Tabla 6.2. Coordenadas de las balizas estacionarias (Fuente: propia).

	1	2	3	5
X	0,00 m	0,40 m	2,40 m	2,80 m
Y	0,20 m	3,60 m	3,60 m	0,20 m

6.2.2. Montaje de las balizas estacionarias

Como se ha comentado en el apartado anterior, las balizas estacionarias deben estar a una altura superior a la del vehículo y con una cierta inclinación para aumentar la cobertura. Para ello, se utiliza un listón de abeto de 25x25mm x 2,4m y una escuadra plana de ángulo doble como pilar de sujeción de las balizas y un tablero de madera de 30x60x1 cm como base del pilar.

El listón se corta en cuatro trozos iguales de 30 cm de alto y se realiza dos agujeros: uno a una altura de 28 cm para anclar la escuadra con un tornillo y el otro en el centro de la base. El tablero de madera se corta en cuatro trozos de 15x15 cm y se taladra un agujero en el centro. La unión del listón con el tablero se hace pasando un tornillo por el agujero del tablero y del listón. Por último, las balizas se enganchan a las escuadras con cinta de doble cara. Esta escuadra presenta un cierto ángulo que permite posicionar la baliza verticalmente e inclinarla con el ángulo deseada. En la **figura 6.6** se muestra el resultado final.

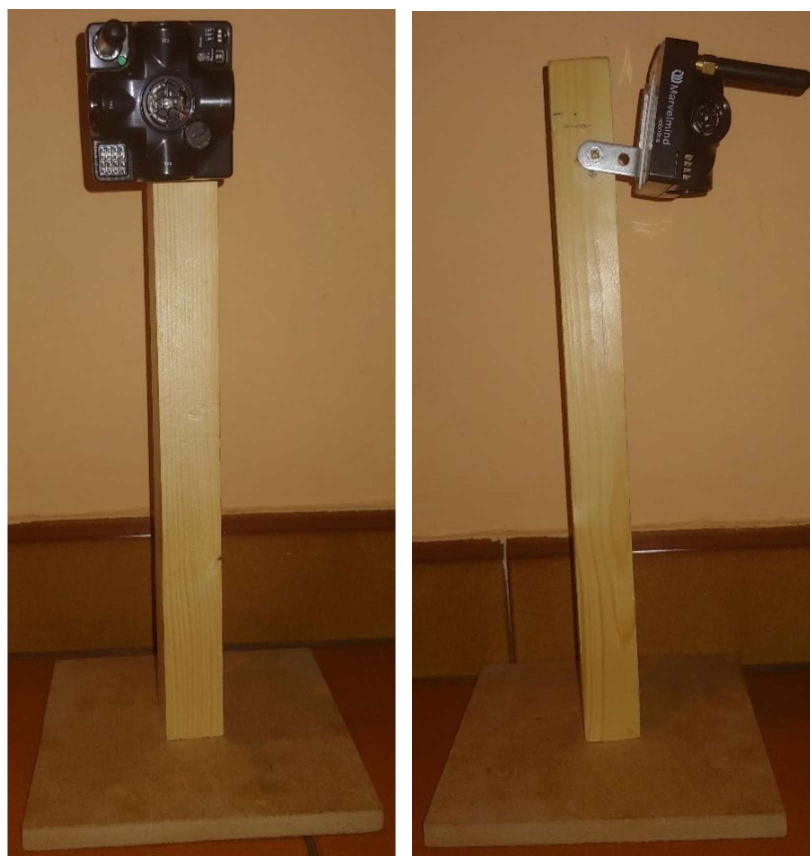


Figura 6.6. Montaje de las balizas estacionarias (Fuente: propia).

6.2.3. Configuración de los parámetros de las balizas

Una vez posicionadas las balizas en el laboratorio según la **figura 6.5**, se procede a configurar las balizas a través de la aplicación de Marvelmind Dashboard. Para ello, se guardan las posiciones exactas de las balizas estacionarias, se crea un mapa y se ajustan los parámetros de todas las balizas con la finalidad de mejor su precisión y aumentar la velocidad de envío de datos.

Primero, en la aplicación, se anotan las distancias entre las balizas estacionarias. Haciendo clic derecho en la tabla superior izquierda del programa Marvelmind Dashboard se abre un menú en el que se elige la opción de *enter distance for pair* para introducir, una a una, las distancias de la **tabla 6.1**. Una vez introducidas las distancias, el color de fondo de la tabla cambiará de blanco a verde, indicando que las balizas están bien posicionadas, sin obstrucciones en la señal ultrasónica. En la siguiente figura se muestra el resultado de este paso.

P01	1	2	3	5
1		3.40	4.16	2.80
2	3.40		2.00	4.16
3	4.16	2.00		3.40
4				
5	2.80	4.16	3.40	

Figura 6.7. Tabla de la aplicación Marvelmind Dashboard que contiene las distancias entre las balizas estacionarias (Fuente: propia).

Para llevar a cabo la trilateración, como se ha visto en el **apartado 4.3.1**, es de suma importancia indicar a la aplicación las coordenadas de las balizas estacionarias. Así que, en la opción de *Manual setup coordinates* que aparece tras clicar con el botón derecho en una de las balizas del menú inferior del programa, se introducen la posición sobre el eje X e Y, según la **tabla 6.2**. Además, es conveniente introducir la altura a la que se encuentran respecto el suelo con el fin de mejorar la precisión de la localización del vehículo.

Coordinates of beacon 2

X, m
0.400
Y, m
3.600
Z, m
0.280

Set manual | Auto | Cancel

Figura 6.8. Introducción de las coordenadas de la baliza 2 (Fuente: propia).

Como se trabaja con cuatro balizas estacionarias, la aplicación da por hecho que la navegación se realiza sobre un plano en 3D. Por este motivo, en la configuración del submapa se deshabilita la opción de *3D navigation*, tal y como se observa en la **figura 6.9**.

Starting beacon trilateration (0..255)	0
Starting set of beacons	1; 5;2;0
Helper beacon (0..255)	n/a
Max. helper distance deviation, m (0.00..2.50)	n/a
3D navigation	disabled
Only for Z coordinate	disabled

Figura 6.9. Menú de configuración del submapa donde se inhabilita la opción de *3D navigation* (Fuente: propia).

Seguidamente, se delimita el mapa para evitar que calcule posiciones en las que el vehículo nunca va a estar. La zona en la que se encuentra el circuito se delimita manualmente pulsando la tecla *Shift* y haciendo clic izquierdo en el ratón al mismo tiempo, acción que crea los vértices del rectángulo creando automáticamente el perímetro del circuito. En la **figura 6.10** se muestra el resultado de este paso.

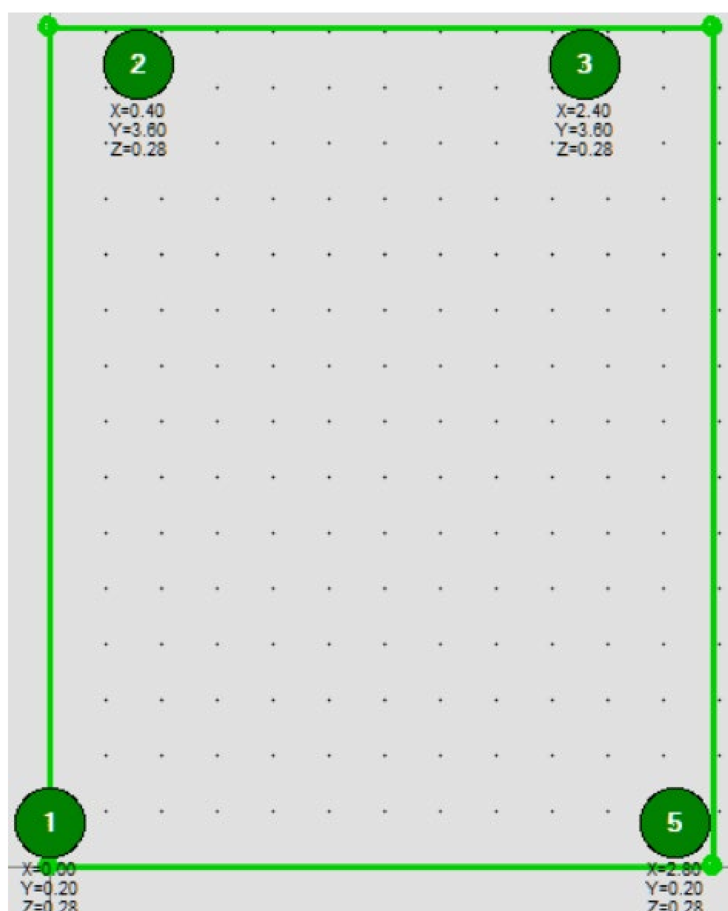


Figura 6.10. Mapa resultante tras introducir las coordenadas de las balizas estacionar y delimitar el mapa. Las líneas verdes representan el límite del mapa (Fuente: propia).

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola d'Enginyeria de Barcelona Est

A continuación, se inhabilitan los transmisores de las balizas estacionarias que no apuntan hacia el circuito. De este modo, se evita que haya eco y ruido. Los transmisores que se apagan son el 1 y el 5 de cada baliza, ya que son los que miran hacia el techo y al lado contrario del circuito, tal y como se muestra en la **figura 6.11.**

TX1	TX2	TX3	TX4	TX5	HIDE
					Normal
					Frozen

Figura 6.11. Configuración de los transmisores de las balizas (Fuente: propia).

Adicionalmente, tanto en el menú de configuración del ultrasonido de las balizas estacionarias como en el de la baliza móvil, se cambian los parámetros de *Number of periods* y *Receiver amplifier* con el objetivo de filtrar la señal y disminuir el eco. Por consiguiente, se estable el número de periodo a 30 y se habilita la opción de manual en el *Amplification* para cambiar la amplificación de la recepción a 10. En la **figura 6.12** se observa donde se encuentran estos parámetros.

Ultrasound	(-) collapse
Mode of work	Normal
Analog power in sleep	disabled
Power after transmission	turn off
Transmitter mode	PWM
Ultrasonic frequency, Hz (100..65000)	45000
Duty, % (1..99)	50
Number of periods (1..255)	30
Code for jitter TX (0..63)	0
Amplification	manual
Receiver amplifier (0..4095)	10
Time gain control	disabled
Mode of threshold	automatic

Figura 6.12. Configuración de la señal ultrasónica de las balizas (Fuente: propia).

Por otro lado, se disminuye la latencia[5] de la información captada para que el vehículo le dé tiempo a reaccionar, antes de salirse del circuito. Para tales efectos, se inhabilita la opción de *Real-time Player,* que aparece en la parte izquierda de la aplicación (**figura 6.13**). Además, en el menú de configuración del módem se ponen los parámetros de *Window of averaging* y de *Distance filter* a 4 y a 0, respectivamente, y se inhabilita la opción de *High resolution mode* (**figura 6.14**).

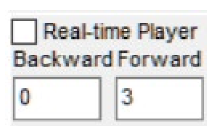


Figura 6.13. Opción de Real-time (Fuente: propia).

[5] Latencia: tiempo que transcurre entre el procesado y envío de la información y la recepción de los datos generados.

Radio frequency band	"915 MHz"
Carrier frequency, MHz	919.0
Radio channel	0
Device address (1..254)	10
Long time sleep	disabled
Window of averaging (0..16)	4
Distance filter (0..16)	0
Max. hedges readout per cycle (1..255)	n/a
High resolution mode (mm)	disabled
Detect new devices in frozen mode	enabled
Temperature of air, °C (-20..60)	23
Update location mode	License SW v7.1 required

Figura 6.14. Menú de configuración del módem donde se encuentran los parámetros *Window of averaging*, *Distance filter* y *High resolution mode* (Fuente: propia).

Por último, se modifica la frecuencia de trabajo a 16 Hz (**figura 6.15**), y se cambia el perfil de radio de las balizas a 153 kbps (**figura 6.16**), con el fin de aumentar la velocidad de envío de información.

Firmware version	V7.040 Modem HW v5.1
Architecture	NIA
Location update rate	16+ Hz
Supply voltage, V (4.5..5.5)	4.99
Time from reset, h:m:s	00:35:45 / 13:02:25 / 0
RSSI from 001, dBm	-39
RSSI to 001, dBm	-43
Profile	General

Figura 6.15. Configuración de la frecuencia de trabajo (Fuente: propia).

Parameters of radio	(-) collapse
Base frequency, MHz	919.000
Radio profile	153 Kbps
Device address (1..254)	1
Radio channel	0
Modulation	GFSK
Power of TX, dBm (-16..14)	10
Channel spacing, kHz (25.391..405.457)	49.190
Intermediate frequency (ID), kHz	417
Offset frequency, kHz (-101.56..100.77)	1.22
Deviation frequency, kHz	151.985
Channel bandwidth, kHz	555.583
CCA/LBT mode	always
DC blocking filter	disabled
Manchester	disabled
Whitening	enabled
FEC	disabled

Figura 6.16. Menú de configuración de los parámetros de radio de las balizas (Fuente: propia).

6.3. Comunicación Master-Slave

La implementación de la comunicación *master*-slave, a través de ROS, se realiza mediante los pasos descritos a continuación.

Se considera que el ordenador externo es el *master* por sus elevadas prestaciones y capacidad de procesamiento, mientras que las *Raspberry Pi 4 Model B* es el *slave*.

1) Se obtiene la dirección IP del *master* y del *slave* ejecutando la siguiente línea de comandos en sus respectivas terminales.

```
$ ip a
```

En la **figura 6.17** se observa donde se localiza la dirección IP tras ejecutar el comando *ip a*.

```
elena@Elena: ~
elena@Elena:~$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group defaul
t qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp3s0f1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP
group default qlen 1000
    link/ether 70:4d:7b:be:5c:27 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.96/24 brd 192.168.1.255 scope global dynamic noprefixroute en
p3s0f1
       valid_lft 86247sec preferred_lft 86247sec
    inet6 fe80::7a1e:78c7:101c:dc4d/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
elena@Elena:~$
```

Figura 6.17. Resultado obtenido tras ejecutar el *ip a* en la terminal del dispositivo master (Fuente: propia).

En este caso las direcciones son:

- Master PC IP: 192.168.1.96
- Slave Raspberry Pi IP: 192.168.1.69

2) Se configura el archivo *bashrc*[6], indicando al ordenador externo que es el maestro y a la Raspberry Pi que es el esclavo. Para ello, primero se abre el archivo con el siguiente comando.

```
$ gedit .bashrc
```

Al final del archivo del dispositivo maestro se escriben las siguientes líneas.

```
export ROS_IP = 192.168.1.96
export ROS_MASTER_URI = http://192.168.1.136:11311
```

Mientras que al final del archivo *bashrc* del esclavo se escribe lo siguiente.

```
export ROS_IP = 192.168.1.69
export ROS_MASTER_URI = http://192.168.1.136:11311
```

3) Finalmente, se escribe en la terminal de cada dispositivo la siguiente instrucción que lee y ejecuta el archivo que se ha modificado.

```
$ source .bashrc
```

Con estos tres simples pasos la comunicación *master-slave* se llevará a cabo cada vez que se inicialice ROS en el dispositivo maestro con el comando *roscore*.

6.4. Programación

Este apartado se introduce con una explicación de la estrategia seguida con el fin de tener una visión general del funcionamiento de los diferentes códigos programados. Seguidamente, se muestran una serie de ordinogramas que representan la comunicación entre los nodos implementados mediante la suscripción o envío de mensajes a través de los tópicos. Se enumeran y se indican el tipo de variables de los diferentes mensajes que contienen los tópicos y, finalmente, se detalla el funcionamiento de cada uno de uno de los nodos.

6.4.1. Estrategia

El algoritmo de control implementado es independiente al circuito a recorrer. En otras palabras, los diferentes códigos se han programado con el objetivo de que el vehículo circule por cualquier circuito, calculando y trazando por sí solo la trayectoria. Esto se podrá llevar a cabo siempre y cuando las balizas estacionarias se sitúen sobre el primer cuadrante, es decir, que sus coordenadas sean positivas. Además, antes de empezar la conducción autónoma es necesario guardar unos puntos de referencia

[6] Archivo *bashrc*: es un archivo el cual reúne un conjunto de comandos que se ejecutan cuando un usuario inicia sesión [32]

que describan la forma del circuito. Así, estos puntos indicarán el origen y final de las rectas y de las curvas.

En el caso del circuito del laboratorio A5.4 de la EEBE, se guardan 14 puntos de referencia que describen la forma de la mitad del circuito. Cabe recordar que no se realiza el recorrido de todo el circuito porque la señal ultrasónica es inestable, provocando inexactitudes en las medidas. Guardando los 14 puntos que se muestran en la **figura 6.18**, el vehículo es capaz de realizar el número de vueltas que se le indiquen.

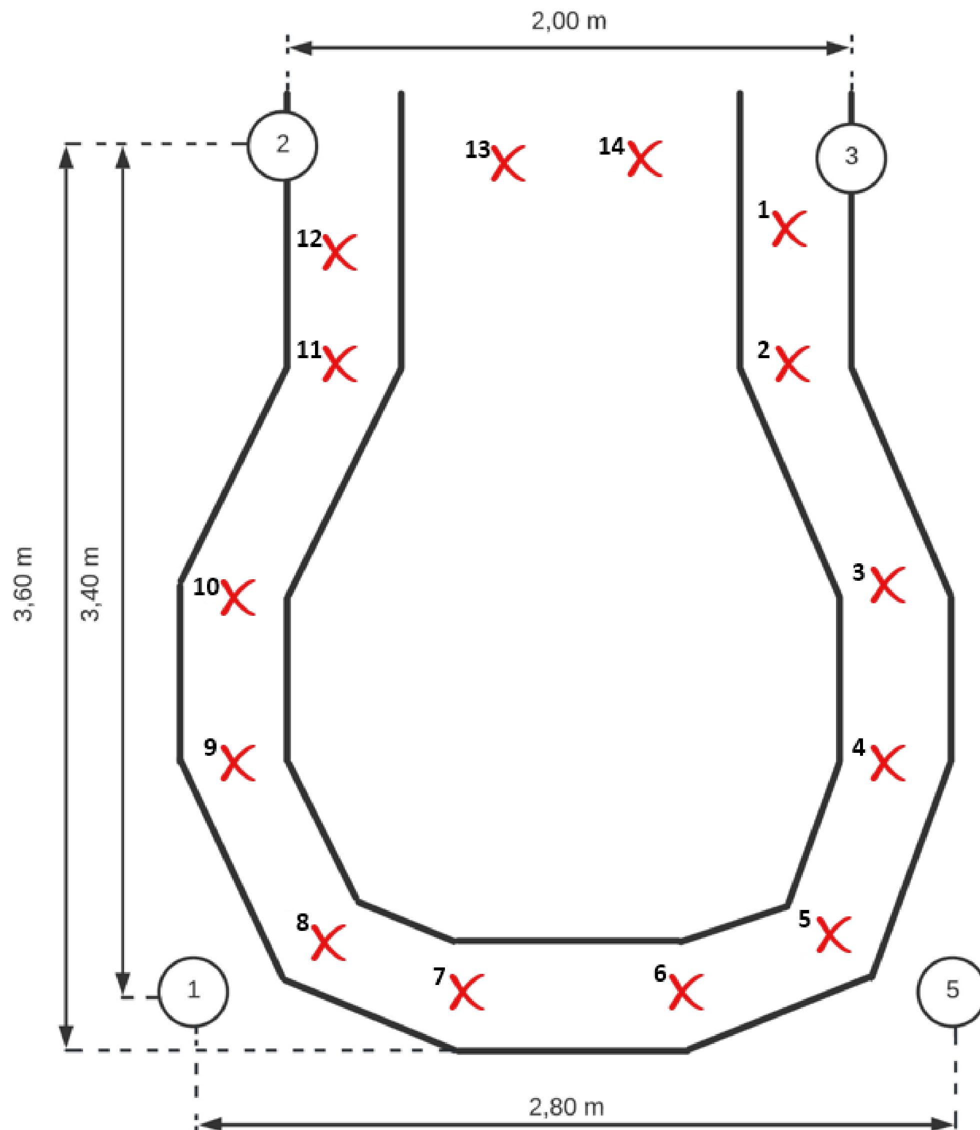


Figura 6.18. Esquema del circuito del laboratorio A5.4 de la EEBE con los puntos de referencia guardados (Fuente: propia).

A partir de los puntos guardados, el algoritmo de control es capaz de determinar la trayectoria que deberá seguir el vehículo y de corregir su dirección y su velocidad si se desvía, actuando sobre el servomotor y el motor DC.

La trayectoria se obtiene con las distancias que hay entre dos puntos de referencia consecutivos y con los ángulos que se dibujan en la intersección entre dos rectas formadas por:

- La unión de los dos primeros puntos de referencia entre los cuales se encuentra el vehículo.
- La unión del segundo punto con el tercero, siendo el segundo el cogido anteriormente y el tercero el siguiente punto de referencia del circuito. Es decir, la unión de los dos próximos puntos a los que el coche se dirigirá.

En función de la localización del coche y el ángulo de la recta donde se encuentre, se actúa sobre el servomotor para modificar la dirección. Además, se aumenta o disminuye la velocidad dependiendo del ángulo del servomotor, puesto que en los ángulos extremos se requiere de más par para mover el vehículo.

En relación con la corrección de la trayectoria del vehículo, ésta se realiza comparando dos ecuaciones características de la recta. La primera ecuación se haya con los dos puntos de referencias consecutivos entre los que se encuentra el coche, mientras que la segunda describe la recta formada por las coordenadas del vehículo y el punto de referencia más próximo a él. Mediante esta comparación se determina si es necesario disminuir o aumentar el ángulo actual del servomotor por tal de seguir la trayectoria previamente calculada.

Por otro lado, se añade el uso de un mando inalámbrico con la finalidad de controlar el vehículo de forma manual, de realizar paradas de emergencia y reinicio de los códigos de forma más rápida y de facilitar el guardado de los puntos de referencia.

6.4.2. Ordinogramas

El funcionamiento del algoritmo se basa en dos conjuntos de códigos. El primero permite controlar el vehículo de forma manual y guardar los puntos de referencia, mediante el mando inalámbrico, para que posteriormente el vehículo pueda recorrer el circuito correctamente en modo automático.

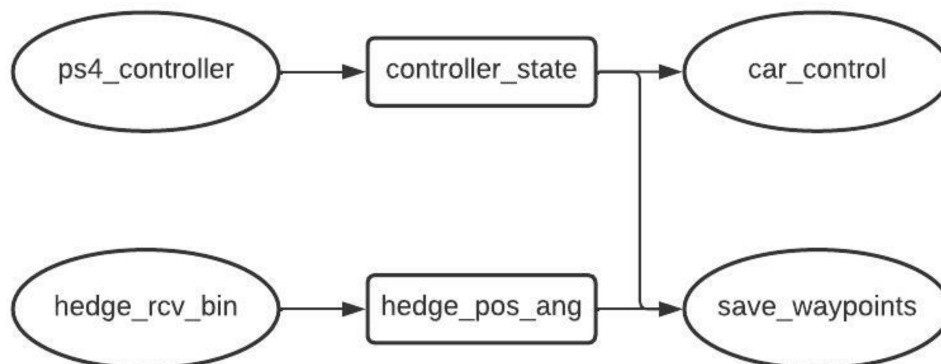


Figura 6.19. Primer conjunto de códigos que permiten controlar el vehículo de manera manual y guardar los puntos de referencia (Fuente: propia).

El segundo conjunto de programas permite conducir el vehículo con el mando inalámbrico o controlarlo de manera autónoma a partir de los puntos de referencias guardados.

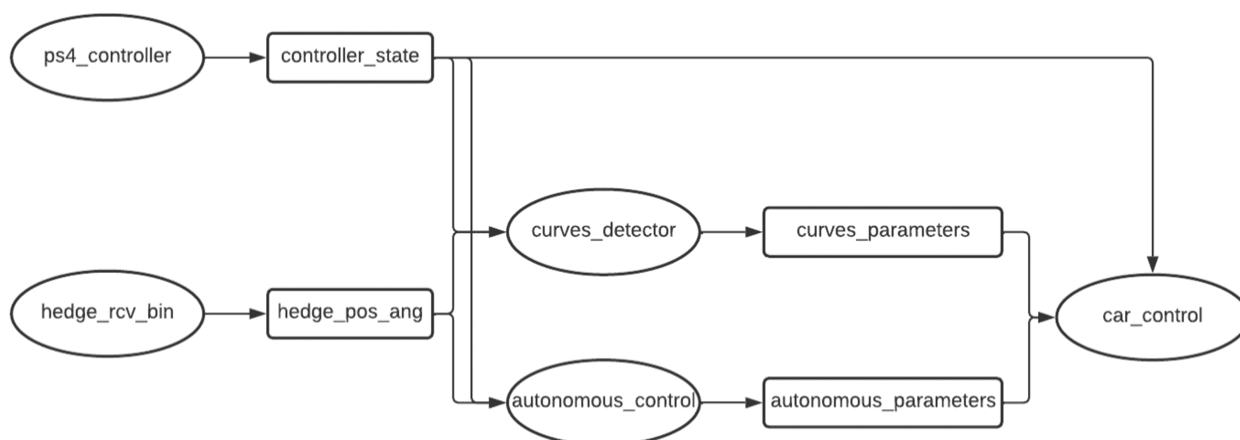


Figura 6.20. Segundo conjunto de códigos que permiten controlar el vehículo de manera autónoma y manual (Fuente: propia).

6.4.3. Descripción de los tópicos

A continuación, se enumeran los mensajes que circulan a través de los tópicos, indicando los valores que pueden tomar y explicando brevemente su utilidad. En el interior del paréntesis que aparece justo al lado del nombre de cada mensaje se indica el tipo de variable.

- **Tópico controller_state**

El nodo *ps4_controller* publica mensajes en el tópico *controller_state*, mientras que los nodos *car_control*, *save_waypoints*, *curves_detector* y *autonomous_control* se suscriben. Los mensajes enviados y recibidos son los siguientes, donde se indica con un paréntesis su tipo.

- ***operating_mode*** *(string).* Si el mensaje es "*manual*" indica que el vehículo se va a controlar de manera manual y si es "*autonomous*" de forma autónoma.
- ***movement*** *(string)*. Indica si el vehículo debe ir hacia adelante, marcha atrás o debe de parar cuando el modo de operación sea el manual. Así, este mensaje puede ser "*forward*", "back" o "*stop*", respectivamente.
- ***direction*** *(string)*. Dirección del vehículo en modo manual, siendo "*right*" giro hacia la derecha, "*left*" giro hacia la izquierda y "*straight*" recto.
- ***speed*** *(int64)*. Velocidad del vehículo en modo manual. Este mensaje obtiene los valores del 0 al 100. Con el 0, se para y el 100 corresponde al 100 % de velocidad.
- **angle** *(int64)*. Ángulo de giro del servomotor en modo manual. El rango de valores que obtiene se comprende entre el 45 y el 135. Del número 45 al 89 el vehículo gira hacía la derecha, del 91 al 135, gira hacía la izquierda y el 90 indica que el vehículo va recto.
- ***flag_square_press*** *(int64)*. *Flag* que se pone a uno cuando se ha pulsa el botón representado con un cuadrado en el mando inalámbrico. De esta manera, se indica cuando se debe de guardar un punto de referencia o reiniciar el algoritmo de control autónomo, como ya se verá más adelante.
- ***flag_emergency_stop*** *(int64)*. *Flag* que se levanta cuando se pulsa el botón con el símbolo de un círculo con el objetivo de indicar que se debe de realizar una parada de emergencia.

- **Tópico hedge_pos_ang**

El nodo *hedge_rcv_bin*, crea el tópico *hedge_pos_ang* donde publica los siguientes mensajes relacionados con los datos de localización del vehículo.

- ***address*** (uint8). Dirección de la baliza móvil
- ***x_m*** (float64). Coordenada X en metros.
- ***y_m*** (float64). Coordenada Y en metros.
- ***z_m*** (float64). Coordenada Z en metros.

De los anteriores menajes que se publican solo se utilizan el *x_m* y el *y_m* con el fin de conocer las coordenadas del vehículo.

A este tópico se suscribe el nodo de *save_waypoints* en el primer conjunto de programas, mientras que en el segundo se suscriben los nodos *curve_detector* y *autonomous_control*.

- **Tópico curves_parameters**

El nodo *curves_detector* publica mensajes en el tópico *curves_parameters* y el nodo *car_control* recibe los mensajes.

- ***movement*** *(string)*. Indica si el vehículo debe ir hacia adelante, marcha atrás o debe de parar en el modo autónomo. Así, este mensaje puede ser "*forward*", "back" o "*stop*", respectivamente.
- ***direction*** *(string)*. Dirección del vehículo en modo autónomo, siendo "*right*" giro hacia la derecha, "*left*" giro hacia la izquierda y "*straight*" recto.
- ***speed*** *(int64)*. Velocidad del vehículo en modo automático. Este mensaje obtiene los valores del 0 al 100. Con el 0, se para y el 100 corresponde al 100 % de velocidad.
- ***angle*** *(int64)*. Ángulo del servomotor determinado a partir de la inclinación de la recta formada a partir de los dos puntos de referencia entre los que se encuentra el vehículo.

- **Tópico autonomous_parameters**

Los mensajes del tópico *autonomous_parameters* son enviados por el nodo *autonomous_control* y recibidos por el *car_control*.

- ***angle*** *(int64)*. Ángulo correctivo que se suma al ángulo del *curve_angle* para corregir la trayectoria si se desvía el vehículo de su ruta.

6.4.4. Control manual mediante un mando Bluetooth. Nodo ps4_controller

El nodo *ps4_controller* es el encargado del control del mando inalámbrico DUALSHOCK 4 de la Play Station 4. Mediante su programación se asigna una acción a algunos de los botones o joysticks[7] cuando se pulsan o cambian de estado por tal de controlar el vehículo manualmente.

De todos los joysticks y botones que presenta el mando, solo se utilizan los dos joysticks de la parte frontal, los botones representados con figuras geométricas del lado derecho y los botones L1 y R1.

El joystick izquierdo se encarga del control del estado del motor. Cuando éste se accione sobre el eje Y, el vehículo se moverá hacia adelante o hacia atrás en función de si los valores leídos del potenciómetro son positivos o negativos. Si son positivos el coche se moverá hacia adelante y si son negativos irá marcha atrás.

---

[7] Joystick: periférico de entrada basado en una palanca que se mueve sobre su base y es capaz de detectar la dirección a la cual se está accionando. Cuenta con dos potenciómetros lineales que son los encargados de cuantificar en valores analógicos su desplazamiento tanto en el eje x como en el y.

El joystick derecho es el responsable del control del servomotor, por lo tanto, mediante su manipulación se cambiará la dirección del vehículo en función de los valores leídos del potenciómetro del eje X. Así, si se detecta que los valores leídos son positivos el coche girará hacia la derecha y si son negativos, hacia la izquierda. Además, si el joystick se encuentra en reposo, el vehículo irá recto.

Con los botones L1 y R1 se controla la velocidad del vehículo. Se establecen diez velocidades diferentes que se cuantifican en forma de porcentaje y van del 16% al 100%. Siendo la velocidad mínima con la cual se empieza a mover el coche de 16% y la máxima de 100%. Si se pulsa el botón R1 se aumenta la velocidad y si se pulsa el botón L1 se disminuye.

En la **figura 6.21** se observa un resumen del funcionamiento del joystick izquierdo y derecho, y de los botones L1 y R1.

Figura 6.21. Esquema de actuación de los joysticks izquierdo y derecho, y de los botones L1 y R1 (Fuente: propia).

Por último, haciendo mención a los botones del lado derecho del mando, al pulsar el botón representado con un triángulo, se selecciona el modo manual. En cambio, si se pulsa el botón con una cruz se cambia al modo autónomo. En relación con el botón definido con un cuadrado, al pulsarlo cuando se está ejecutando el primer conjunto de códigos se guardará la posición en la que se encuentra el vehículo, mientras que si se pulsa con el segundo y el vehículo se controla autónomamente, se reiniciará los programas. Adicionalmente, al pulsar el botón representado con un círculo se ejecuta una parada de emergencia independientemente del modo en el que se encuentre.

A la hora de programar el nodo *ps4_controller*, se emplea la librería *pyPS4Controller.controller* por tal de facilitar la programación. Mediante el uso de esta librería se detecta los eventos que le van sucediendo al mando y se les asocia una acción. En otras palabras, se crea una función para cada uno de los botones y joysticks utilizados, la cual se ejecuta cada vez que cambian de estado.

La organización del código se realiza mediante la creación de una única clase[8] que contienen las funciones asociadas a un botón o a un joystick y una función para el envío de mensajes a través del tópico *controller_status*.

Esta clase se inicializa configurando el mando con una función incluida en la librería, asignando el estado inicial de los mensajes y enviándolos al tópico *controller_state*. Además, se declaran las variables que se van a utilizar a lo largo del programa.

```
def __init__(self, **kwargs):
        Controller.__init__(self, **kwargs) # Controller initialization

        # Variables definition with its initial status
        msg.operating_mode = "manual"
        msg.movement = "stop"
        msg.direction = "straight"
        msg.speed = 0  # stop
        msg.angle = 90 # middle angle
        msg.flag_emergency_stop = 0
        msg.flag_square_press = 0
        pub.publish(msg) # Send to the topic the initial status

        # Variable to count the number of times the R1 and L1 buttons
        # have been pressed to set the motor speed
        self.speed_counter = 0

        # Variable to send information
        self.prevMovement = ""
        self.prevDirection = ""
        self.prevSpeed = 10
        self.prevAngle =  0
```

Las funciones que se ejecutan tras detectar un evento del mando son:

- **on_L3_up.** Función que detecta el estado del joystick izquierdo y se encarga de que el vehículo vaya hacia adelante.
- **on_L3_down.** Detecta el joystick izquierdo y controla que el motor vaya hacía atrás.
- **on_L3_y_at_res.** Asociado, también, al joystick izquierdo y hace que el motor frene.
- **on_L2_press.** Función asociada al joystick izquierdo que controla la dirección del servomotor.
- **on_L1_press.** Detecta si se ha pulsado el botón L1 y cuando se acciona baja la velocidad del vehículo.
- **on_R1_press.** Registra el evento del pulsado del botón R1 y aumenta la velocidad cuando sucede.

[8] Clase: es una herramienta fundamental en la Programación Orientada a Objetos que permite empaquetar variables, funciones y datos juntos [33].

- **on_square_press.** Función que detecta el pulsado del botón representado con forma de triángulo y se encarga de la selección del modo manual.
- **on_circle_press.** Asociado al botón con una cruz y selecciona el modo autónomo.
- **on_x_press y on_x_release.** Funciones asociadas al botón con un cuadrado e indica cuándo se debe guardar la posición del vehículo o se debe reiniciar los programas. Se utiliza tanto el evento de pulsado como el soltado del botón con la finalidad de programar un filtro antirrebote[9] y asegurar que el guardado de la información y el reinicio se realice una única vez tan solo cuando se ha pulsado el botón con una cruz.
- **on_trinagle_press.** Detecta el botón representado con un círculo y ejecuta la parada de emergencia.

A continuación, se detallan dos funciones que se ejecutan tras detectar un evento asociado a un botón y en a un joystick. Con estos dos ejemplos se facilita el entendimiento de las demás funciones sin la necesidad de ser explicadas.

En la función de *on_square_press* cada vez que se detecta que el botón representado con un triángulo ha sido pulsado, se actualiza los mensajes *operating_mode* con el *string* “manual” y el *flag_emergency_stop* se establece a 0 para evitar que el vehículo vuelva a ejecutar la parada de emergencia en caso de haber sido accionada previamente. Finalmente, con la línea de código *pub.publish(msg)* se envía los mensajes correspondientes al tópico *controller_status*.

```
def on_square_press(self):    # Triangle button
        msg.operating_mode = "manual"
        msg.flag_emergency_stop = 0

        # Send topics messages
        pub.publish(msg)
```

Por otro lado, la función asociada a los joysticks contiene el parámetro *value*, que indica el valor analógico exacto del joystick. Esta variable puede tomar los valores de entre un rango de -4095 a 4095 para el eje X e Y. En la función *on_L2_press* se programa que si dicha variable toma valores menores a -8000 el vehículo debe girar hacia la izquierda, mientras que si toma valores superiore a 8000 deberá girar hacia la derecha. Además, entre el rango de -8000 a 8000, el vehículo deberá de permanecer recto.

```
def on_L2_press(self, value): # Right joystick
        if value <= -8000:
            msg.direction = "left"
        elif value >= 8000:
            msg.direction = "right"
```

[9] Filtro antierrebote: técnica utilizada para evitar que el botón rebote al pulsarlo una única vez, ya que al presionarlo o soltarlo se produce una fluctuación en la señal que ocasiona interpretar más de un único pulsado.

```
        else:
            msg.direction = "straight"

        # Angle selection based on right joystick value
        msg.angle = self.select_angle(value)

        self.SendInfo()# Send information
```

El ángulo de giro se determina con la función *select_angle*, la cual retorna el ángulo al que debe ir el servomotor en función de la variable *value*. En este caso, se determinan 7 posibles ángulos: tres correspondientes al giro izquierdo, tres al derecho y una a la posición que hace que el vehículo vaya recto.

```
def select_angle(self,joystick_value):
    # ---------------------- Turn to the left ----------------------- #
        if joystick_value <= -8000 and joystick_value > -16000:
            return 105
        elif joystick_value <= -16000 and joystick_value > -24000:
            return 120
        elif joystick_value <= -24000:
            return 135 # Max value
    # ---------------------- Turn to the right ---------------------- #
        elif joystick_value >= 8000 and joystick_value < 16000:
            return 75
        elif joystick_value >= 16000 and joystick_value < 24000:
            return 60
        elif joystick_value >= 24000:
            return 45 # Max value
    # ------------------------- Go straight ------------------------- #
        else:
            return 90 # Middle position
```

Finalmente, con la función *SendInfo* se envía los mensajes al tópico *controller_status*. Destacar que estos solo se envían si alguno de los mensajes a cambiando respecto al último envío. De esta manera, se evita saturar al tópico con mensajes repetitivos.

```
def SendInfo(self):
        if ((self.prevMovement != msg.movement) or (self.prevDirection
!= msg.direction) or (self.prevSpeed != msg.speed) or (self.prevAngle
!= msg.angle)):
            # Save previous messages
            self.prevMovement = msg.movement
            self.prevDirection = msg.direction
            self.prevAngle =  msg.angle
            self.prevSpeed =  msg.speed
            pub.publish(msg) # Send topics messages
```

En referencia a la creación del nodo y los tópicos se realiza de la siguiente forma.

```
# Initialization of topic, node and messages
pub =
rospy.Publisher('controller_state',controller_state,queue_size=10)
rospy.init_node('ps4_controller', anonymous=True)
rospy.loginfo("PS4 Controller Node Started")
msg = controller_state()
```

6.4.5. Control de las balizas. Nodo hedge_rcv_bin

El programa referente a este nodo lo proporciona la empresa de las balizas ultrasónicas. Este código procesa los datos recibidos por el módem del sistema de navegación ultrasónico y obtiene la localización de la baliza móvil.

El módem que se conecta con un cable USB al ordenador externo, envía por el puerto serial las distancias halladas, con el principio de TOF, entre la baliza móvil y cada una de las balizas estacionarias. El programa es el encargado de aplicar el algoritmo de trilateración con el fin de encontrar la posición exacta de la baliza móvil.

Este nodo es un *publisher* y a través del tópico *hedge_pos_ang* envía las coordenadas de la baliza móvil. También, envía cada minuto la posición de cada baliza estacionaria con su respectiva dirección para poder identificarlas correctamente.

6.4.6. Guardado de los waypoints. Nodo save_waypoints

Este nodo es un suscritor y el objetivo de su implementación es guardar de forma cómoda, a través del mando inalámbrico, los trece puntos de referencia necesarios para el algoritmo de control autónomo.

En este código se emplea la librería csv que permite manipular de forma sencilla los ficheros con extensión CSV[10]. Con ella, se pueden crear, abrir y guardar archivos CSV y guardar y consultar diferentes datos dentro de dicho fichero.

La organización del código se realiza mediante la creación de una única clase, denominada *Save_Waypoints*, que contienen dos funciones: una detecta el pulsado del botón con el símbolo de un cuadrado y la segunda guarda la localización del vehículo.

Cuando se ejecuta el código, lo primero que se realiza es la inicialización de la clase. En ella, se declaran las variables que se van a utilizar, se crea, se abre y se indica que se van a guardar datos en un archivo CSV y se inicializa el nodo con los tópicos a los que se va a suscribir, denotando la función que se va a ejecutar cada vez que se actualicen sus mensajes.

```
def __init__(self):

        # Declaration of global variables
        self.num_waypoints = 3 # Number of waypoints we want to save
        self.operating_mode = "manual" # Current operating mode
```

[10] Un archivo CSV (*Comma Separated Values*) es un tipo de archivo que organiza los datos en filas y en columnas, separando las filas por comas, muy utilizado para el intercambio de grandes cantidades de datos.

```
        # Flag to indicate when we have to save the waypoint
        self.flag_save_waypoint = 0

        # Flag to indicate the previous state of square button
        self.flag_previous_press = 0

        # Open csv file where the waypoints will be saved
        self.csv_file = open('waypoints','w', newline = '')
        self.csv_write = csv.writer(self.csv_file)

        # autonomous_control node initialitzation
        rospy.init_node('save_waypoints', anonymous=True)
        rospy.Subscriber("hedge_pos_ang", hedge_pos_ang,
self.save_waypoints)
        rospy.Subscriber("controller_state", controller_state,
self.ps4_controller)

        rospy.loginfo("Save Waypoints Node Started")
```

En la primera función se programa el antirrebote del botón representado con un cuadrado, mencionado en el anterior subapartado. Cada vez que se actualiza el tópico *controller_state*, se revisa si el botón ha cambiado de estado respecto su estado anterior. Cuando este hecho suceda, querrá decir que el botón ha sido pulsado y se levantará el *flag save_waypoints* para indicar que en la siguiente actualización de los mensajes del tópico *hedge_pos_ang* se podrá guardar la posición de la baliza móvil.

```
def ps4_controller(self,data):

        # These statements prevent the square button bounce
        if self.flag_previous_press != data.flag_square_press:
            self.flag_save_waypoint = data.flag_square_press
            self.flag_previous_press = data.flag_square_press
```

La segunda función se lleva a cabo con cada actualización de la posición de la baliza móvil. Se encarga de guardar la posición actual del vehículo en un fichero CSV cuando el botón representado con un cuadrado ha sido pulsado y el modo de operación del vehículo es el manual.

```
def save_waypoints(self,data):
        if self.operating_mode == "manual":

            # Save waypoints under a certain condition
            if self.num_waypoints >= 1 and self.flag_save_waypoint==1:
                self.flag_save_waypoint = 0
                self.num_waypoints -= 1

                # Save and print the waypoint it has saved
                self.csv_write.writerow([data.x_m, data.y_m])
                print("waypoint saved: ", data.x_m, data.y_m)

            # Indicate that all waypoints have been save and close csv
              file
            elif self.num_waypoints == 0:
                print("All waypoints are saved")
                self.csv_file.close()
```

6.4.7. Detector de curvas. Nodo curves_detector

El nodo *curves_detector* se suscribe y publica a diferentes tópicos al mismo tiempo. Su propósito es calcular el ángulo de giro del servomotor, detectando la próxima curva o recta del circuito a partir de los puntos de referencias previamente guardados.

Este nodo se suscribe a los tópicos *controller_state* y *hedge_rcv_bin* con el fin de saber si se ha pulsado el botón que contiene el símbolo de un cuadrado y actualizar la posición actual del vehículo, mientras que se suscribe al tópico *curves_parameters* para enviar la dirección y velocidad pertinentes del vehículo.

En este programa también se emplea la librería csv de Python para poder leer los datos guardados y al igual que los códigos de los primero dos nodos, la organización del código se realiza mediante la creación de una única clase, llamada *CurvesDetector*, que agrupa el conjunto de funciones programadas.

En la inicialización de la clase, se realizan cinco acciones diferentes con el objetivo de disminuir el tiempo de repuesta durante el transcurso del programa y, así, hacer que el vehículo reaccione a su debido tiempo. La primera acción que se realiza es la declaración de las variables globales que se van a utilizar a lo largo del código y la declaración de las variables locales que se van a emplear en esta inicialización.

Seguidamente, se abre el fichero CSV previamente creado y se guarda en una matriz los catorce puntos de referencia. En cada fila de la matriz se almacenan las coordenadas de cada punto de referencia, de tal forma que en la primera columna quede la distancia respecto al eje X y en la segunda columna las del eje Y.

```
# Open csv file where the waypoints are saved
csv_file = open('waypoints','r', newline = '')
csv_read = csv.reader(csv_file)

# Read and save all the waypoints and calculates the angles
for row in csv_read:
   self.Waypoints.append(row) # Save rows

# x and y coordinates are converted from string to float
   for j in range(2):
      self.Waypoints[i][j] = float(self.Waypoints[i][j])

   i += 1 # Update de row number

# Close csv file
csv_file.close()
```

La tercera tarea que se realiza en la inicialización calcula y guarda en una matriz los ángulos que deberá girar el servomotor. Como en cada punto guardado se produce un cambio de trayectoria, ya que estos son los que indican el comienzo o fin de una recta o una curva, se calcula el ángulo que hay entre la trayectoria actual y la trayectoria siguiente. De esta manera, se determinar el ángulo que hay que hacer girar al servomotor en cada punto de referencia. En la siguiente figura se muestra en azul los ángulos que deberá girar el servomotor en los puntos 2 y 3.

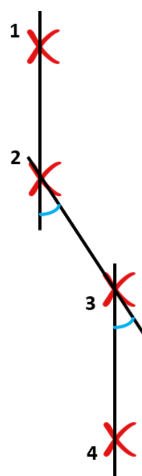


Figura 6.22. Ángulos de giro del servomotor en azul que deberá girar en los puntos 2 y 3. Las cruces rojas representa los puntos de referencia guardados (Fuente: propia).

El código correspondiente a esta tarea se agrupa en una función llamada *CalCarAngle*, que calcula los ángulos tangentes correspondiente al vector director formado por el punto donde se va a realizar el cambio de dirección y el siguiente punto.

```
for i in range(self.WaypointsNum + 1):
   if (i == 1): # Start angle
      V[0] = self.Waypoints[i][0] - self.Waypoints[i-1][0]
      V[1] = self.Waypoints[i][1] - self.Waypoints[i-1][1]
      cal_angle = math.degrees(math.atan(V[0]/V[1]))
      corr_angle.append(cal_angle)
      self.CarAngle.append(corr_angle[i-1])

    if (i >= 1):
       if (i < self.WaypointsNum):
          V[0] = self.Waypoints[i][0] - self.Waypoints[i-1][0]
          V[1] = self.Waypoints[i][1] - self.Waypoints[i-1][1]

        else:
           V[0] = self.Waypoints[0][0] - self.Waypoints[i-1][0]
           V[1] = self.Waypoints[0][1] - self.Waypoints[i-1][1]

        cal_angle = math.degrees(math.atan(V[1]/V[0]))
```

Una vez calculado el ángulo, en la misma función *CalCarAngle* se determinar si el vehículo debe de girar hacia la derecha o izquierda. Para ello, se encuentra el cuadrante donde se encuentra el vector director

que indica la trayectoria y se suma al ángulo encontrado 180º si está en el segundo o tercer cuadrante 0 360º si está en el cuarto cuadrante. Además, se resta el ángulo actual con el anterior, obteniendo el ángulo de giro con su correspondiente signo. Si el signo sale negativo indicará que se deberá girar a la izquierda y, por el contrario, si es positivo deberá girar a la derecha.

```
# Determine the quadrant of the vector and the sum of the degrees that
# belong to it
   if ((V[0] > 0) and (V[1] > 0)):   # First quadrant
      corr_angle.append(cal_angle)
   elif ((V[0] < 0) and (V[1] > 0)): # Second quadrant
       corr_angle.append(cal_angle + 180)
   elif ((V[0] < 0) and (V[1] < 0)): # Third quadrant
        corr_angle.append(cal_angle + 180)
   else:                              # Fourth quadrant
        corr_angle.append(cal_angle + 360)

# Subtract current angle from previous
for i in range (len(corr_angle)):
   if ((i >= 2)):
      self.CarAngle.append((corr_angle[i-1] - corr_angle[i]))

    elif (i == (len(corr_angle) - 1))
      self.CarAngle.append((corr_angle[i] - corr_angle[1]))
```

La cuarta acción que se realiza en la inicialización, calcula y guarda en una matriz las distancias entre puntos con el fin, a posteriori, de saber cuándo cambiar el ángulo de giro. Las instrucciones programadas se agrupan en la función *PointsDistance* y calculan las hipotenusas respecto a dos puntos de referencia consecutivos. Con la instrucción *PointDist.append(distance)* se añade la distancia calculada en la matriz *PointDist*.

```
def PointsDistance(self):
# ---------------------- Local variables ------------------------ #
   x_distance = 0.0
   y_distance = 0.0
   distance = 0.0

# --------------------------- Code ------------------------------ #
   # Calculate distance between two points
   for i in range(self.WaypointsNum):
      if (i != (self.WaypointsNum - 1)):
        x_distance = abs(self.Waypoints[i][0] - self.Waypoints[i+1][0])
        y_distance = abs(self.Waypoints[i][1] - self.Waypoints[i+1][1])

       else:
         x_distance = abs(self.Waypoints[i][0] - self.Waypoints[0][0])
         y_distance = abs(self.Waypoints[i][1] - self.Waypoints[0][1])

         distance = math.sqrt((x_distance ** 2) + (y_distance ** 2))

         # Save the distance in a variable
         self.PointDist.append(distance)
```

Finalmente, en la inicialización se inicializa el nodo y se le indican los tópicos a los que se tiene que suscribir y publicar.

Por otro lado, en la función *Execute*, que incluye la clase *CurveDetector*, es donde se determina la localización del vehículo en el circuito, el ángulo del servomotor, la velocidad del motor DC y el número de vueltas dado al circuito, en cada instante de tiempo.

Para facilitar la programación, se da por hecho que el coche va a comenzar entre el punto de referencia 1 y 2. Así, a la hora de encontrar los ángulos de giro del servomotor, solo habrá que buscar dentro de la matriz *PointDist* de forma ordenada y aplicarlo al servomotor durante la primera mitad de la distancia entre los dos puntos en los que se encuentra el vehículo. De este modo, se evita que el vehículo esté constantemente girando y se salga del circuito. Destacar que al ángulo de la matriz se suma o resta a 90º, puesto que siempre será el ángulo anterior.

Antes de exponer el código de este paso, se ejemplifica para una mayor compresión. Si el coche se encuentra en el punto de referencia 1, el ángulo del trayecto 1 y 2 se encontrará en la posición 0 de la matriz *PointDist*. Se supone que dicho ángulo resulta ser negativo, con lo cual indica que se deberá girar a la izquierda. Por lo tanto, como los ángulos correspondientes al giro izquierdo del servomotor van del 91 al 135, habrá que hacer el valor absoluto de dicho ángulo para poder sumárselos a 90º, posición inicial del servomotor.

```
# Select angle
self.Angle = self.CarAngle[self.AngleCount]

# -------------------- CAR DIRECTION ---------------------- #
# Correct to the middle of the waypoint distance
if (car_dist > (self.PointDist[self.Counter]/2)):
   if (self.Angle >= 0):
      self.Direction = "right"
      self.Angle = 90 - self.Angle

    else:
      self.Direction = "right"
      self.Angle = 90 + abs(self.Angle)

else:
   self.Angle = 90
   self.Direction = "straight"
```

Cuando el vehículo está próximo al siguiente punto de referencia, se vuelve a acceder a la matriz *PointDist* para coger el siguiente ángulo de giro y realizar los pasos explicado previamente. Ahora bien, para cambiar el ángulo de giro a tiempo, se calcula la distancia entre el vehículo y el siguiente punto de referencia con la función *CarDistance* y se compara un valor determinado experimentalmente. Si la distancia es menor o igual al valor, se cogerá el siguiente ángulo y distancia entre los dos siguientes puntos y se actuará sobre el sobre servomotor por tal de volver a corregir la dirección.

```
# ---------------------- Local variables ------------------------ #
margin = 0.2     # Margin to change the waypoint

# Calculates the distance between the car and the current waypoint
car_dist = self.CarDistance(self.Counter)
```

```
# ---------------------- Change reference ----------------------- #
# Change the waypoints reference
if ((car_dist <= margin) and (self.Counter < (self.WaypointsNum - 1))):
    self.Counter += 1

elif ((car_dist <= margin) and (self.Counter == (self.WaypointsNum -
1))):
    self.Counter = 0
```

La velocidad se incrementa o decrementa en función del ángulo del servomotor puesto que, si el ángulo se encuentra en los extremos, el coche necesitará de más par para moverse.

```
# -------------------- CAR SPEED ---------------------- #
if ((self.Angle >= 80) and (self.Angle < 100)):
    self.Speed = 16
elif (((self.Angle >= 70) and (self.Angle < 80)) or ((self.Angle >=
100) and (self.Angle < 110))):
    self.Speed = 18
elif (((self.Angle >= 60) and (self.Angle < 70)) or ((self.Angle >=
110) and (self.Angle < 120))):
    self.Speed = 20
else:

     self.Spedd = 22
```

Por último, dentro de la función de *Execute*, se actualiza la información al tópico *curve_parameters* si alguno de los mensajes ha cambiado. La función que se utiliza es *SendInfo,* y cumple con el mismo objetivo que la que se incluye en el nodo *ps4_controller*.

La función *Execute* se ejecutará continuamente hasta que el vehículo realice el número de vueltas impuesto o se haga un reinicio del programa mediante el mando inalámbrico.

Si el coche llega al último punto de referencia y ha realizado el número de vueltas que se le ha indicado, el vehículo se parará y el programa preguntará si se desea hacer un reinicio para comenzar de nuevo. Si se pulsa la tecla “y” del teclado del ordenador externo, se reiniciará.

```
# ------------------------ CAR MOVEMENT ------------------------ #
# Stop the car
if (self.LapsCount == self.Laps):
    self.Movement = "stop"
    self.Direction = "straight"
    self.Speed = 0

    value = input("Do you want to reset the program:\n")
    print(f'You entered {value}')
    if (value == "y"):
        self.Counter = 0
        self.LapsCount = 0
        self.AngleCount = 0
    else:

         rospy.on_shutdown(myhook)
```

Por otro lado, cuando se detecte, a través del tópico *controller_parameters*, que se ha pulsado el botón representado con un cuadrado, se realizará el reinicio del movimiento autónomo siempre y cuando en la terminal se confirme la operación, pulsando la tecla "y" del teclado.

```
def ps4_controller(self,data):

        self.operating_mode = data.operating_mode

        # These statements prevent the square button bounce
        if self.flag_previous_square_press != data.flag_square_press:
            self.flag_previous_square_press = data.flag_square_press

            if (data.operating_mode == "automatic"):
                value = input("Do you want to reset the program:\n")
                print(f'You entered {value}')
                if (value == "y"):
                    self.Counter = 0
                    self.LapsCount = 0
                    self.AngleCount = 0
```

6.4.8. Control autónomo. Nodo autonomous_control

El nodo *autonomous_control* es un *publisher* y un *subscriber* al mismo tiempo. Recibe mensajes a través de los tópicos *hedge_rcv_bin* y *controller_state* y envía mensajes a través del tópico *autonomous_parameters*.

El objetivo de este nodo es determinar el ángulo de corrección del servomotor cuando el vehículo se ha desviado de la trayectoria calculada a partir de los puntos de referencia guardados.

La implementación del código se realiza mediante una máquina de estados finitos. Dicha máquina está compuesta por una serie de estados que calculan las potenciales combinaciones de las variables internas del programa y es capaz de transicionar de un estado a otro en función de los resultados obtenidos en el actual estado y, en algunos casos, del estado anterior.

El diagrama de estados de este código se muestra a continuación, en la **figura 6.23**. En él se representan los posibles estados con círculos y las transiciones con flechas. Se observan 11 posibles estados y las transiciones entre ellos en función del resultado obtenido en el estado actual y, además, en el último estado, se requiere conocer cuál ha sido el estado anterior.

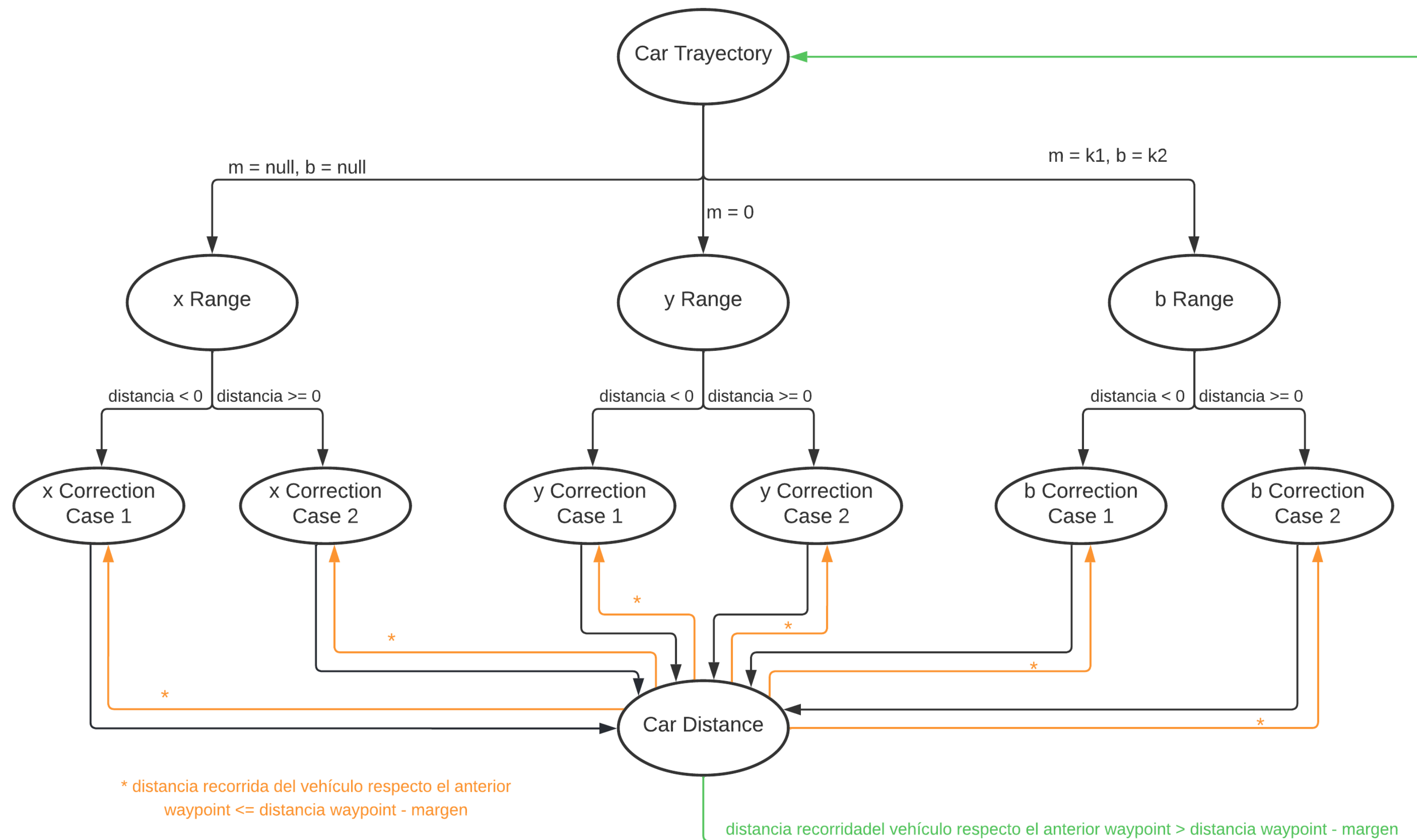


Figura 6.23. Máquina de estados finitas implementada en el nodo *autonomous_control* (Fuente: propia).

En el primer estado, el *Car Trayectory*, se determina la trayectoria que debe de seguir el vehículo a partir de los dos puntos de referencia entre los cuales se ubica. Para tal efecto, se encuentra la ecuación característica de la recta que contiene los dos puntos, planteando un sistema de ecuaciones lineales cuyas incógnitas son la pendiente y el término independiente de la recta. Mediante la interpretación de los resultados, se pueden dar tres posibles casos para describir la trayectoria actual:

- **Caso 1.** La trayectoria será perpendicular al eje OY cuando no se haya ninguna solución.
- **Caso 2.** La trayectoria será perpendicular al eje OX cuando la pendiente de la recta sea igual a cero.
- **Caso 3.** La trayectoria presentará una cierta inclinación cuando la pendiente sea distinta de cero.

A partir de la trayectoria parametrizada, se transiciona a uno de los tres siguientes estados. De este modo, se cambiará al estado *x Range* cuando se dé el caso 1, al *y Range* cuando se dé el caso 2 y al *b Range* cuando se dé el caso 3. El propósito de estos estados es determinar el margen izquierdo y derecho de la recta calculada, que delimitan la superficie por la cual el vehículo circulará correctamente. Por tanto, el coche corregirá el ángulo del servomotor cuando se ubique fuera del área delimitada por los dos márgenes.

El cálculo de los márgenes se realiza en función del sentido que lleva el coche a la hora de recorrer el circuito. Así, en cada uno los estados se obtienen los márgenes según sí el vehículo circula en sentido horario o antihorario, dando lugar a seis nuevos posibles casos que se ilustran en las siguientes figuras. La **figura 6.24** muestra los dos casos obtenidos en el estado *x Range*, la **figura 6.25** los del estado *y Range* y la **figura 6.26** los del *b Range*.

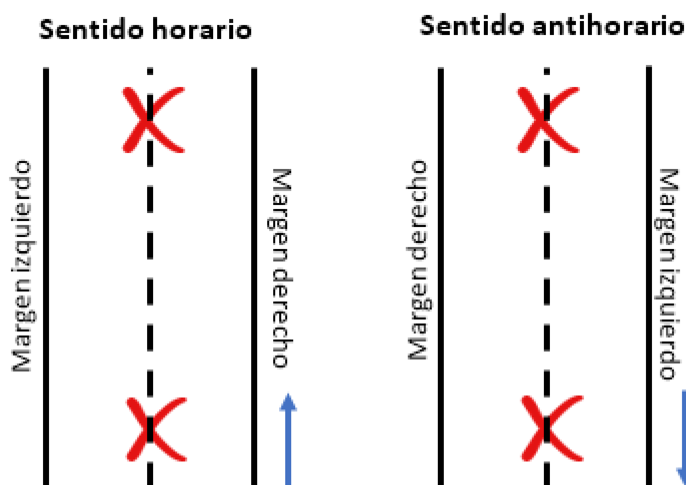


Figura 6.24. Márgenes calculados en el caso del estado *x Range*. Las cruces rojas representan los puntos de referencias, la flecha azul indica el sentido de giro, la línea discontinua negra es la recta calculada y las líneas continuas son los márgenes (Fuente: propia).

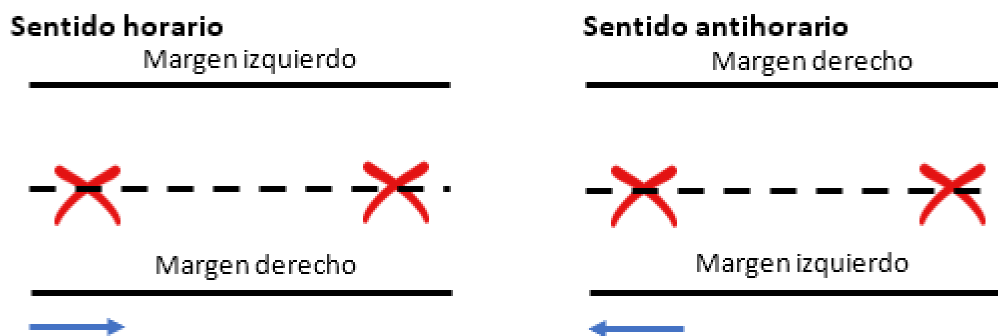


Figura 6.25. Márgenes calculados en el caso del estado *y Range*. Las cruces rojas representan los puntos de referencias, la flecha azul indica el sentido de giro, la línea discontinua negra es la recta calculada y las líneas continuas son los márgenes (Fuente: propia).

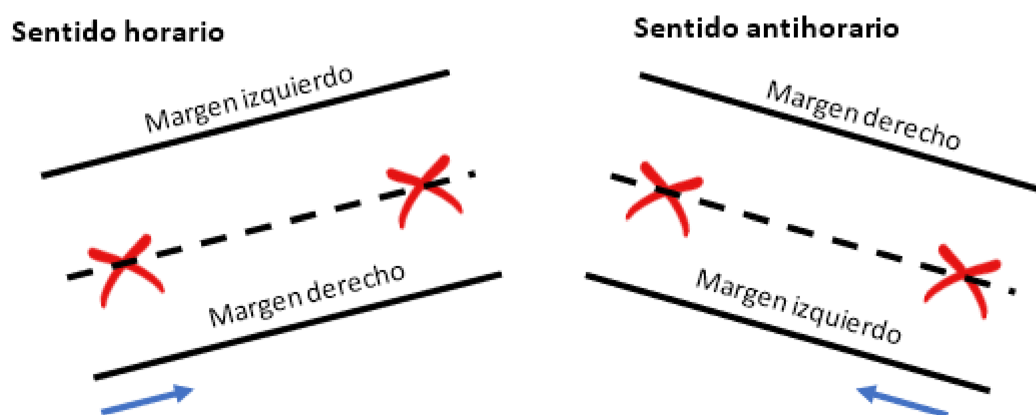


Figura 6.26. Márgenes calculados en el caso del estado *b Range*. Las cruces rojas representan los puntos de referencias, la flecha azul indica el sentido de giro, la línea discontinua negra es la recta calculada y las líneas continuas son los márgenes (Fuente: propia).

Los siguientes seis estados, corrigen la trayectoria del vehículo cuando se ha salido del área delimitada por los márgenes calculados con anterioridad. Para ello, se compara la localización actual del vehículo con los márgenes calculados, de tal forma que si el vehículo se desvía por el lado izquierdo o derecho se aplica una corrección al servomotor para que cambie de dirección.

La transición al último estado se hará después de comprobar la trayectoria calculada con la ubicación actual del vehículo. El estado *Car Distance*, determina si el vehículo se encuentra entre los dos puntos de referencia donde se ha parametrizado la recta. Si el vehículo sigue en la trayectoria calculada, se volverá al estado anterior con el fin de corregir la posición del servomotor cuando se desvía. Por el contrario, si el vehículo ya no se encuentra entre estos dos puntos, se volverá al primer estado para calcular la nueva trayectoria.

Por último, esta máquina de estados siempre estará en bucle hasta que se llegue al último punto de referencia y se haya completado el número de vueltas indicado. Adicionalmente, mediante el mando inalámbrico, se podrá hacer un reinicio del programa o una parada de emergencia cuando se requiera.

A diferencia de los programas vistos hasta ahora, la organización de este código se realiza con un conjunto de clases encargadas de realizar los cálculos de cada estado, las transiciones entre estados y el bucle de la máquina de estados. Además, se añade una clase más que contiene la declaración de los estados y transiciones, la inicialización de ROS y donde se da lugar el intercambio de mensajes mediante los tópicos.

Cada estado se programa en clases diferentes, que incluyen siempre las mismas funciones:

- ***__init__***. En esta función se programan las instrucciones que se van a realizar, solamente, en la primera vez que se utilice la clase.
- ***Enter***. Sirve para guardar en el estado actual los valores de las variables comunes a todos los estados.
- ***Execute***. Es donde se programa la función del estado y se establece la condición de cambio de estado.
- ***Exit***. Conjunto de instrucciones que se realizan tras salir del estado. Normalmente, se utiliza para cambiar el valor de las variables comunes a todos los estados con la finalidad de conseguir que el bucle de la máquina de estados quede bien implementado.

A continuación, se muestra un ejemplo de las clases que contiene los estados. La clase mostrada a continuación contiene el estado *x Range*. En la inicialización de ésta se declara la variable que se va a utilizar solamente en esta clase. En la función *Enter* se guarda el margen que se desea dejar para delimitar la superficie por la cual no se aplica la corrección al servomotor y, se calcula la distancia entre puntos de referencia para determinar el sentido de giro. En la función *Execute*, se determina el margen izquierdo y derecho en función de si el coche recorre el circuito en sentido horario o antihorario. Por último, en la función *Exit* se pone el *flag ChangeWaypoint* a 0 con el propósito de indicar que ahora mismo no es necesario cambiar de puntos de referencia.

```
class xRange(State):
    def __init__(self,FSM):
        State.__init__(self,FSM)
        self.Distance = 0.0
    def Enter(self):
        State.EnterMargin(self)
        self.Distance = State.Waypoint_y1 - State.Waypoint_y0
    def Execute(self):
        if (self.Distance >= 0):
            State.LeftMargin  = State.Waypoint_x1 - self.Margin
            State.RightMargin = State.Waypoint_x1 + self.Margin
            self.FSM.ToTransition("toxCorrection_Up") # CHANGE STATE
        else:
            State.LeftMargin  = State.Waypoint_x1 + self.Margin
            State.RightMargin = State.Waypoint_x1 - self.Margin
            self.FSM.ToTransition("toxCorrection_Down") # CHANGE STATE
    def Exit(self):
        State.ChangeWaypointFlag = 0
```

Otra clase importante es la clase *FSM* donde se programan un conjunto de funciones con la finalidad de declarar en la clase principal, que en este caso es *CarNavigation*, los nombres de los diferentes estados y transiciones y de definir el estado inicial. Además, se encarga de realizar las transiciones a los estados y del bucle de la máquina de estados.

```
class FSM(object):
    def __init__(self,character):
        PublishingMSG.__init__(self)
        self.states = {}
        self.transitions = {}
        self.curState = None
        self.prevState = None
        self.trans = None
        self.flagStart = 1
        self.i = 0

    def AddTransition(self,transName,transition):
        self.transitions[transName] = transition

    def AddState(self,stateName,state):
        self.states[stateName] = state

    def SetState(self,stateName):
        self.prevState = self.curState
        self.curState = self.states[stateName]

    def ToTransition(self,toTrans):
        self.trans = self.transitions[toTrans]

    def ExecuteAndChangeState(self):
        if(self.trans):
            self.curState.Exit()
            self.SetState(self.trans.toState)
            self.curState.Enter()
            self.curState.Execute()
            self.trans = None
        self.curState.Execute()

    def Execute(self):
        if self.flagStart == 1:
            self.flagStart = 0
            self.curState.Enter()
            PublishingMSG.publish_messages(self)
            self.curState.Execute()
            while (self.i < 3):
                self.ExecuteAndChangeState()
                self.i += 1
            self.i = 0
        elif (State.ChangeWaypointFlag == 0):
            while(self.i<2):
                self.ExecuteAndChangeState()
        elif (State.ChangeWaypointFlag == 1):
            while(self.i<3):
                self.ExecuteAndChangeState()
                self.i += 1
            self.i = 0
```

En la clase *CarNavigation*, se declaran los estados y transiciones con la ayuda de las funciones creadas en la clase anterior, se inicializa ROS y se programan dos funciones que recogen la información de los tópicos *hedge_pos_ang* y *controller_state*. Con la función *car_coordinates* se guarda la localización del vehículo, mientras que con la función *ps4_controller* se mira si se ha pulsado el botón con el símbolo de un cuadrado para realizar el reinicio del programa.

```
class CarNavigation():
    # GLOBAL VARIABLES
    Current_x = 0.0
    Current_y = 0.0
    Flag_Reset = 0
    Operating_Mode = "manual"
    Laps = 0

    def __init__(self, laps):
        self.Flag_Start = 1
        CarNavigation.Laps = laps

    # --------------------- FSM Initialization ---------------------- #
        self.FSM = FSM(self)
         # STATES
        self.FSM.AddState("CarTrayectory",CarTrayectory(self.FSM))
        self.FSM.AddState("xRange",xRange(self.FSM))
        self.FSM.AddState("xCorrection_Up",xCorrection_Up(self.FSM))
       self.FSM.AddState("xCorrection_Down",xCorrection_Down(self.FSM))
        self.FSM.AddState("yRange",yRange(self.FSM))
        self.FSM.AddState("yCorrection_Up",yCorrection_Up(self.FSM))
       self.FSM.AddState("yCorrection_Down",yCorrection_Down(self.FSM))
        self.FSM.AddState("bRange",bRange(self.FSM))
        self.FSM.AddState("bCorrection_Up",bCorrection_Up(self.FSM))
       self.FSM.AddState("bCorrection_Down",bCorrection_Down(self.FSM))
        self.FSM.AddState("CalDistance",CalDistance(self.FSM))

        # TRANSITIONS
  self.FSM.AddTransition("toCarTrayectory",Transition("CarTrayectory"))
        self.FSM.AddTransition("toxRange",Transition("xRange"))
self.FSM.AddTransition("toxCorrection_Up",Transition("xCorrection_Up"))
self.FSM.AddTransition("toxCorrection_Down",Transition("xCorrection_Dow
n"))
        self.FSM.AddTransition("toyRange",Transition("yRange"))
self.FSM.AddTransition("toyCorrection_Up",Transition("yCorrection_Up"))
self.FSM.AddTransition("toyCorrection_Down",Transition("yCorrection_Dow
n"))
        self.FSM.AddTransition("tobRange",Transition("bRange"))
self.FSM.AddTransition("tobCorrection_Up",Transition("bCorrection_Up"))
self.FSM.AddTransition("tobCorrection_Down",Transition("bCorrection_Dow
n"))
   self.FSM.AddTransition("toCalDistance",Transition("CalDistance"))

        # FISRT STATE
        self.FSM.SetState("CarTrayectory")
```

```
    # --------------------- ROS Initialization ---------------------- #

        # SUBSCRIBER
        rospy.init_node('autonomous_control', anonymous=True)
        rospy.Subscriber('hedge_pos_ang', hedge_pos_ang,
self.car_coordinates,queue_size=1,buff_size=2**24)
        rospy.Subscriber("controller_state", controller_state,
self.ps4_controller,queue_size=1,buff_size=2**24)
        rospy.loginfo("Publisher Node Started, now publishing
messages")

    def car_coordinates(self,data):
        CarNavigation.Current_y = data.y_m
        CarNavigation.Current_x = data.x_m

    def ps4_controller(self,data):
        CarNavigation.Operating_Mode = data.operating_mode

        # These statements prevent the square button bounce
        if (self.flag_previous_square_press != data.flag_square_press):
            self.flag_previous_square_press = data.flag_square_press

            CarNavigation.Flag_Reset = 1
```

Por último, la publicación de los ángulos de corrección por el tópico *autonomous_parameters* se programa en la función *SendInfo*, que es la misma función que contiene los códigos visto en los anteriores nodos.

6.4.9. Control del vehículo. Nodo car_control

El nodo *car_control* es el encargado de actuar sobre el motor DC y servomotor en base a la información recibida por los tópicos *ps4_states, curve_parameters* y *autonomous_parameters*.

Cuando se elige el modo manual, pulsando el botón con el símbolo de un triángulo en el mando inalámbrico, este nodo actuará sobre el vehículo con, tan solo, los mensajes publicados en el tópico *ps4_controller*. Sin embargo, cuando se cambie al modo automático tras pulsar el botón representado con una cruz, el coche se moverá a partir de los mensajes recibidos por el tópico *curves_parameters* y *autonomous_parameters*. Además, en este modo, estará atento al tópico *ps4_controller* para saber si hay que realizar un cambio de modo y si hay que realizar una parada de emergencia.

En el código de este nodo se utiliza la librería *pigpiod* para facilitar la generación de la señal PWM que controla el servomotor y el ESC. En concreto, se emplea la función *set_PWM_frequency* y *set_servo_pulsewidth* con el propósito de establecer en los pines de control del variador y del servomotor la frecuencia de la señal con el ancho de pulso adecuado para conseguir el ángulo de giro y velocidad deseadas.

La implementación del código se realiza en una única clase llamada *Car_Control* donde se agrupan un conjunto de funciones encargadas de la inicialización de ROS y de la señal PWM, de la recepción de los mensajes recibido a través de los diferentes tópicos a los que se suscribe el nodo y del control del servomotor y del ESC en base a los mensajes recibidos.

Al utilizar por primera vez la clase, se ejecuta la función *__init__* que inicializa las variables globales y, mediante la función *setup_servo_motor* se establece la señal PWM en los pines 33 y 35, donde se conecta el pin de control del ESC y del servomotor, tal y como se observa en el esquemático del **apartado 4.4**. Adicionalmente, se inicializa el nodo y se le indica los tópicos a los que se debe suscribir.

Resaltar que, por primera vez, dentro de la función de inicialización se utiliza la función *on_shutdown* proporcionado por ROS. Esta instrucción ejecutará la función que se encuentra dentro de sus paréntesis antes de que el nodo deje de funcionar. En este caso, se llevará a cabo la función *shutdown* con la se pretende parar el motor y dejar el servomotor en la posición de 90º con el objetivo de no perder el control del vehículo una vez parado el nodo.

```
def __init__(self,ESC,SERVO):
        # GPIO
        self.ESC = ESC
        self.SERVO = SERVO

        # Controller variables
        self.operating_mode = "manual"
        self.Flag_Emergency_Stop =   0 # Indicate the emergency stop

        # Controller and Autonomous control variables
        self.Direction = "stop"
        self.Movement = "straight"
        self.Angle = 90  # Servo middle position
        self.Speed = 16  # Minimum speed

        # Autonomous angles
        self.Trayectory_Angle = 90 # Servo middle position
        self.Angle_Correction = 0  # Car shunt

        # Servo and motor initialization
        self.setup_servo_motor()
        print("Car Setup Done")

        # car_control node initialitzation
        rospy.init_node('car_control', anonymous=True)
        rospy.Subscriber('controller_state', controller_state,
self.manual_control)
        rospy.Subscriber('autonomous_parameters',
autonomous_parameters,
self.autonomous_control,queue_size=1,buff_size=2**24)
rospy.Subscriber('curve_parameters',curve_parameters,self.curve_detecto
r,queue_size=1,buff_size=2**24)
        rospy.loginfo("Car Control Node Started")

        rospy.on_shutdown(self.shutdown)
```

En la función *setup_servo_motor* se habilitan los pines de la *Raspberry Pi 4 Model B* con la línea de código *GPIO.setmode(GPIO.BCM)*, se declaran que los pines 33 y 35 se van a utilizar como salida y se inicializa las señales PWM con una frecuencia de 50 Hz para el ESC y de 300 Hz para el servomotor, y un ancho de pulso de 1,5 ms con el que se consigue que el motor esté parado y el ángulo del servomotor sea de 90º.

```
def setup_servo_motor(self):
        GPIO.setmode(GPIO.BCM)  # Enable pins

        # Initialization ESC and Servo GPIO
        PI.set_mode(self.ESC, pigpio.OUTPUT)
        PI.set_mode(self.SERVO, pigpio.OUTPUT)

        # Set frequency
        PI.set_PWM_frequency(self.ESC, 50)     # ESC PWM 50 Hz
        PI.set_PWM_frequency(self.SERVO, 300) # SERVO PWM 300 Hz

        # Start PWM
        PI.set_servo_pulsewidth(self.ESC, 1500)  # 1,5 ms - Motor brake
        PI.set_servo_pulsewidth(self.SERVO, 1500)# 1,5 ms - Servo 90°
```

La función *manual_control* guarda los mensajes que se envían por el tópico *controller_state*. En particular, almacena siempre en variables globales los valores de los mensajes *operating_mode* y *flag_emergency_stop* y los valores de los mensajes *direction*, *movement*, *speed* y *angle* cuando se elige el modo manual. Con los valores almacenados, posteriormente se actuará sobre la señal de control del servomotor y el ESC.

```
def manual_control(self,data):
        self.Operating_Mode = data.operating_mode
        self.Flag_Emergency_Stop = data.flag_emergency_stop

        if (self.Operating_Mode == "manual"): # Manual mode
            self.Direction = data.direction
            self.Movement = data.movement
            self.Speed = data.speed
            self.Trayectory_Angle = data.angle
```

Por otro lado, cuando el control del vehículo pasa a ser automático, los mensajes que se guardan son los recibidos a través de los tópicos *curve_parameters* y *autonomous_parameters*. Mediante la función *curve_detector* se guardan los mensajes del tópico *curve_parameters* y con la función *autonomous_control* los del *autonomous_control*.

Destacar que, las variables que se emplean para almacenar los mensajes relacionados con el movimiento y la velocidad del vehículo son las misma que las utilizadas en la función *manual_control*. Sin embargo, se emplean dos nuevas variables con la finalidad de determinar el ángulo del servomotor posteriormente. Así, la primera variable es *Trayectory_Angle* y se encuentra en la función *curve_detector* y la segundo es *Angle_correction* que se encuentra en la función *autonomous_control*.

```
def curve_detector(self,data):
        if (self.Operating_Mode == "automatic"):
            self.Movement = data.movement
            self.Trayectory_Angle = data.angle
            self.Speed = data.speed
            self.Direction = data.direction
```

En la función *autonomous_control*, además de guardar los mensajes del tópico relacionado, se satura el ángulo de corrección para que siempre esté comprendido entre el rango (-5,-2)U(2,5). Esta saturación se ha ajustado de forma experimental con el propósito de conseguir que el vehículo no se salga del circuito. Adicionalmente, se indica que cuando el ángulo de corrección que reciba sea 90º, este pasará a ser cero debido a que no deberá de corregir la trayectoria que está siguiendo.

```
def autonomous_control(self,data):
       if (self.Operating_Mode == "automatic"):
            self.Angle_Correction = data.angle

            # Saturation of the correction angle
            if (self.Angle_Correction > 5):
                self.Angle_Correction = 5
            elif (((self.Angle_Correction < 2) and
(self.Angle_Correction > -2)) or (self.Angle_Correction == 90)):
                  self.Angle_Correction = 0
            else:
                self.Angle_Correction = -5
```

Finalmente, con la función *car_control* se satura la velocidad y los ángulos del servomotor con la intención de no dañar a los actuadores, se actúa sobre las dos señales PWM en función de los mensajes recibidos y se muestra por la terminal el estado del vehículo.

El vehículo siempre se encontrará en movimiento a no ser que se haya activado la parada de emergencia o mediante los tópicos se reciba que debe parar porque el vehículo ha completado el número de vueltas indicado.

El ángulo del servomotor se determina a partir de la suma de las variables *Trayectory_angle* y *Angle_Correction* y la velocidad y dirección a través de las variables *Speed* y *Direction*.

```
def car_control(self):
        # Car Movement
        if (self.Flag_Emergency_Stop == 0):

            # Motor saturation. Speed - 0 to 100 %
            if (self.Speed > 100):
                self.Speed = 100
            elif (self.Speed < 10):
                self.Speed = 10

            # Sum the curve angle with the angle correction
            self.Angle = self.Trayectory_Angle + self.Angle_Correction
            # Servo saturation. Angle - 45 to 135°
            if (self.Angle > 135):   # Max left
                self.Angle = 135
```

```
            elif (self.Angle < 45): # Max right
                self.Angle = 45

            # Determinate the direction of the car
            if (self.Angle == 90):
                self.Direction = "straight"
            elif (self.Angle > 90):
                self.Direction = "left"
            elif (self.Angle < 90):
                self.Direction = "right"

            # Movement: forward, back or stop
            if (self.Movement == "forward"):
                self.moveF(self.Speed)

            elif (self.Movement == "back"):
                self.moveB(self.Speed)

            elif (self.Movement == "stop"):
                self.stop()

            # Direction: right,left or straight
            if ((self.Direction == "right") or (self.Direction ==
"left")):
                self.changeDirection(self.Angle)

            elif (self.Direction == "straight"):
                self.goStraight()

            # Shows the status of the car through the terminal
            print("Operting Mode: ", self.Operating_Mode)
            print("Car status: ", self.Movement,", ", self.Direction)
            print("Speed: ", self.Speed, ", Angle: ", self.Angle)
            print()

        # Stops the car when the emergency PS4 button is pressed
        elif (self.Flag_Emergency_Stop == 1):
            self.stop()
            self.goStraight()
            print("Emergency Stop Done")
```

Las funciones *moveF*, *moveB*, *stop*, *changeDirection* y *goStraight* que aparecen dentro de la función *car_control* sirven para realizar la conversión de la velocidad y los ángulos al ancho de pulso correspondiente de cada señal PWM generada. Como todas estas funciones tienen el mismo propósito, a continuación, se describe solamente una de ellas.

La conversión de los ángulos al ancho de pulso que determina la posición del servomotor, se realiza mediante la función *changeDirection* que tienen como argumento el ángulo que se desea convertir. Como se ha mencionado en el **apartado 4.3.4**, con un ancho de pulso de 1 ms se consigue posicionar el servomotor en 0º mientras que un ancho de pulso de 2 ms, se posiciona a 180º. Por lo tanto, para hacer girar el servomotor a un ángulo determinado se realiza el siguiente factor de conversión.

$$ancho\ de\ pulso\ [\mu s] = \frac{(2000 - 1000)[\mu s] * ángulo[º]}{180º} + 1000\ \mu s \qquad \textbf{(Ec. 6.1)}$$

Así, la función quedaría de la siguiente forma aplicando la **ecuación 6.1**.

```
def changeDirection(self,angle):
        # min duty  - 1000 us
        # max duty  - 2000 us
        # min angle - 0°
        # max angle - 180°
        duty = ((2000-1000)*angle)/180 + 1000
        PI.set_servo_pulsewidth(self.SERVO,duty)
```

7. Funcionamiento y resultados

En este apartado, se determina la exactitud y precisión del sistema de navegación y se redactan los resultados obtenidos con el algoritmo de control desarrollado.

7.1. Balizas ultrasónicas

Antes de mostrar los datos y resultados, es necesario tener claros los conceptos de exactitud y precisión. La exactitud es la cualidad de un sensor de obtener una lectura próxima al verdadero valor, mientras que la precisión es la cualidad de obtener la mínima dispersión entre diferentes medidas, es decir, es la cualidad de conseguir lecturas muy próximas unas de otras [34].

Con el fin de obtener la exactitud y precisión de las balizas, primero, se miden con un metro las distancias reales de los catorce puntos de referencia respecto al origen de coordenadas establecido en el **apartado 6.4.1** (**tabla 7.1**) y se guardan las distancias de cada punto de referencia medidas con las balizas, es decir, se anotan las coordenadas de la baliza móvil en cada punto de referencia que se obtienen con el sistema de navegación (**tabla 7.2**). Cabe destacar que en la **tabla 7.2** se recogen tres lecturas de cada posición con el objetivo de obtener más información a la hora de analizar los datos.

Tabla 7.1. Distancias reales respecto al origen de coordenadas de los puntos de referencia (Fuente: propia).

Número waypoint	REAL	
	x (m)	**y** (m)
1	2,30	2,95
2	2,34	2,40
3	2,77	1,66
4	2,77	0,89
5	2,40	0,55
6	1,93	0,30
7	1,15	0,30
8	0,59	0,51
9	0,37	0,89
10	0,37	1,62
11	0,74	2,40
12	0,69	2,90
13	1,16	3,48
14	1,76	3,48

Tabla 7.2. Distancias obtenidas con las balizas ultrasónicas (Fuente: propia).

Número	LECTURA 1		LECTURA 2		LECTURA 3		MEDIA	
waypoint	x (m)	y (m)	x (m)	y (m)	x (m)	y (m)	x (m)	y (m)
1	2,35	2,97	2,35	2,98	2,34	2,96	2,35	2,97
2	2,36	2,40	2,37	2,40	2,36	2,41	2,36	2,40
3	2,80	1,61	2,80	1,63	2,81	1,62	2,80	1,62
4	2,84	0,84	2,83	0,83	2,82	0,86	2,83	0,84
5	2,37	0,51	2,39	0,52	2,40	0,51	2,39	0,51
6	1,96	0,25	1,96	0,26	1,92	0,24	1,95	0,25
7	1,16	0,28	1,15	0,25	1,14	0,27	1,15	0,27
8	0,64	0,47	0,60	0,48	0,58	0,52	0,61	0,49
9	0,35	0,85	0,38	0,87	0,37	0,86	0,37	0,86
10	0,33	1,57	0,32	1,58	0,34	1,57	0,33	1,57
11	0,71	2,42	0,71	2,41	0,73	2,40	0,72	2,41
12	0,66	2,90	0,67	2,91	0,67	2,88	0,67	2,90
13	1,15	3,52	1,16	3,50	1,14	3,49	1,15	3,50
14	1,76	3,48	1,76	3,48	1,75	3,50	1,76	3,49

7.1.1. Exactitud de las balizas

A partir de las coordenadas de los puntos de referencia mediadas con dos métodos diferentes (**tabla 7.1** medidas reales y **tabla 7.2** medidas obtenidas por las balizas) se calcula el error relativo y absoluto del sistema de navegación ultrasónico con el fin de obtener la exactitud de las medidas. En la **tabla 7.3** se recogen los errores calculados.

La fórmula empleada para el cálculo del error absoluto corresponde a la **ecuación 7.1** y la del cálculo del error relativo corresponde a la **ecuación 7.2**. El error relativo se multiplica por 100 para obtener el tanto por ciento.

$$\mathcal{E}_a = |X_i - \bar{X}| \quad \textbf{(Ec. 7.1)}$$

$$\mathcal{E}_r = \frac{\mathcal{E}_a}{\bar{X}} * 100 \quad \textbf{(Ec. 7.2)}$$

Donde:

- $\mathcal{E}_a$: error absoluto
- $\bar{X}$: valor real de la medida
- X_i: valor obtenido en la medición
- $\mathcal{E}_r$: error relativo

Tabla 7.3. Errores relativos y absolutos del sistema de navegación para interiores (Fuente: propia).

Número	ERROR ABSOLUTO		ERROR RELATIVO	
waypoint	x (m)	y (m)	x (%)	y (%)
1	0,05	0,02	2,17	0,68
2	0,02	0,00	0,85	0,00
3	0,03	0,04	1,08	2,41
4	0,06	0,05	2,17	5,62
5	0,01	0,04	0,42	7,27
6	0,02	0,05	1,04	16,67
7	0,00	0,03	0,00	10,00
8	0,02	0,02	3,39	3,92
9	0,00	0,03	0,00	3,37
10	0,04	0,05	10,81	3,09
11	0,02	0,01	2,70	0,42
12	0,02	0,00	2,90	0,00
13	0,01	0,02	0,86	0,57
14	0,00	0,01	0,00	0,29

Para analizar los resultados, se realiza la **tabla 7.4** que muestra el promedio de los errores absolutos y relativos.

Tabla 7.4. Promedio de errores absolutos y relativos (Fuente: propia).

PROMEDIO DE ERRORES	
ABSOLUTO X (m)	0,02
ABSOLUTO Y (m)	0,03
RELATIVO X (%)	2,03
RELATIVO Y (%)	3,88

Como se puede observar en la **tabla 7.4**, el promedio de los errores absolutos es de 0,02 m en el eje X y de 0,03 m en el eje Y, y el promedio de los errores relativos es de 2,03 % en el eje X y de 3,88 % en el eje Y. Por tanto, se puede concluir que presenta una elevada exactitud. Sin embargo, en la **tabla 7.3** se observa que, en algunos puntos de referencia, como es el caso del 6 y el 9, el error se dispara. Esto puede ser debido a que en ciertas zonas las balizas estacionarias no son capaces de filtrar adecuadamente el ruido por los objetos que hay alrededor que producen rebotes y originan ecos en la señal ultrasónica emita por las balizas.

7.1.2. Precisión de las balizas

Como se ha definido anteriormente, la precisión es la cualidad de obtener la mínima dispersión entre diferentes medidas. Así que, para determinar la precisión se calcula el error relativo y absoluto de cada lectura obtenida con respecto a la media (**tabla 7.5**). Recordar que se han realizado tres lecturas recogidas en la **tabla 7.2**. De este modo, se podrá comprobar si la precisión que se obtiene es la misma que asegura el fabricante, que es de ±2 cm.

Tabla 7.5. Errores relativos y absolutos del sistema de navegación para interiores respecto a cada lectura realizada y a la media de las tres (Fuente: propia).

RESPECTO LA MEDIA Y LA LECTURA 1				RESPECTO LA MEDIA Y LA LECTURA 2				RESPECTO LA MEDIA Y LA LECTURA 3			
ERROR ABSOLUTO		ERROR RELATIVO		ERROR ABSOLUTO		ERROR RELATIVO		ERROR ABSOLUTO		ERROR RELATIVO	
x (m)	y (m)	x (%)	y (%)	x (m)	y (m)	x (%)	y (%)	x (m)	y (m)	x (%)	y (%)
0,00	0,00	0,00	0,00	0,00	0,01	0,00	0,34	0,01	0,01	0,43	0,34
0,00	0,00	0,00	0,00	0,01	0,00	0,42	0,00	0,00	0,01	0,00	0,42
0,00	0,01	0,00	0,62	0,00	0,01	0,00	0,62	0,01	0,00	0,36	0,00
0,01	0,00	0,35	0,00	0,00	0,01	0,00	1,19	0,01	0,02	0,35	2,38
0,02	0,00	0,84	0,00	0,00	0,01	0,00	1,96	0,01	0,00	0,42	0,00
0,01	0,00	0,51	0,00	0,01	0,01	0,51	4,00	0,03	0,01	1,54	4,00
0,01	0,01	0,87	3,70	0,00	0,02	0,00	7,41	0,01	0,00	0,87	0,00
0,03	0,02	4,92	4,08	0,01	0,01	1,64	2,04	0,03	0,03	4,92	6,12
0,02	0,01	5,41	1,16	0,01	0,01	2,70	1,16	0,00	0,00	0,00	0,00
0,00	0,00	0,00	0,00	0,01	0,01	3,03	0,64	0,01	0,00	3,03	0,00
0,01	0,01	1,39	0,41	0,01	0,00	1,39	0,00	0,01	0,01	1,39	0,41
0,01	0,00	1,49	0,00	0,00	0,01	0,00	0,34	0,00	0,02	0,00	0,69
0,00	0,02	0,00	0,57	0,01	0,00	0,87	0,00	0,01	0,01	0,87	0,29
0,00	0,01	0,00	0,29	0,00	0,01	0,00	0,29	0,01	0,01	0,57	0,29

Para analizar los resultados obtenidos, se realiza la **tabla 7.6** en la que se recoge la media de errores absolutos y relativos.

Tabla 7.6. Promedio de errores absolutos y relativos (Fuente: propia).

PROMEDIO DE ERRORES	
ABSOLUTO X (m)	0,01
ABSOLUTO Y (m)	0,01
RELATIVO X (%)	0,98
RELATIVO Y (%)	1,09

Observando la tabla anterior, se puede deducir que se obtiene un error relativo bajo con un error absoluto por debajo de la precisión marcada por el fabricante de ±2 cm.

Con los resultados obtenidos se concluye que a pesar de que los errores respecto a las distancias reales son elevados, la precisión es correcta, por debajo de lo que marca el fabricante. Por lo tanto, se considera que se ha logrado implementar adecuadamente el sistema de navegación ultrasónico.

7.2. Algoritmo de control

Respecto al algoritmo de control implementado, se determina que se desenvuelve correctamente, calculando la trayectoria a seguir y actuando sobre las señales PWM del servomotor y del ESC con el fin de hacer girar y/o cambiar la velocidad del vehículo, en función de su ubicación en el circuito. Además, es capaz de corregir la posición del servomotor cuando se desvía de la ruta calculada.

Adicionalmente, el algoritmo desarrollado es capaz de controlar vehículo adecuadamente en cualquier circuito siempre y cuando se guarden previamente unos puntos de referencia que indiquen el inicio o final de una recta o una curva, y las coordenadas de las balizas estacionarias sean positivas.

Los resultados obtenidos en referencia al movimiento autónomo son que el vehículo es capaz de recorrer el circuito del laboratorio A5.4 de la EEBE de la UPC.

Por otro lado, el vehículo reacciona según lo programado cuando se controla de forma manual. El código del control del mando inalámbrico Bluetooth determina la velocidad y dirección del vehículo según el estado de los joysticks y botones, enviando los datos generados al procesador a bordo, quien procesa la información recibida y actúa en base a ella sobre el servomotor y el controlador del motor.

8. Propuestas de mejora

El vehículo autónomo ideal es capaz de llegar a su destino calculando y siguiendo la trayectoria más óptima, respetando las señales y normas de tráfico y ejecutando un algoritmo de control con la finalidad de tomar decisiones propias en base a la información extraída de diversos sensores. Adicionalmente, debe saber adaptarse a cualquier imprevisto que ocurra durante el viaje como, por ejemplo, desviarse de su ruta si encuentra algún obstáculo en medio de la carretera o frenar si se detecta que se va a producir una colisión.

Aunque aún queda un largo camino que recorrer hasta llegar al nivel 5 de conducción autónoma (que consisten en un vehículo capaz de desplazarse de un punto a otro y tomar decisiones propias en base al entorno que le rodea, tal y como se definió en el **apartado 3.1** de este proyecto), a continuación se proponen una serie de posibles mejoras en el ámbito de la navegación autónoma, campo en el que se ha centrado este proyecto. Además, se exponen un conjunto de propuestas que se podrían implementar en un futuro con el objetivo de lograr el vehículo completamente autónomo.

8.1. NVIDIA Jetson

En este proyecto se utiliza un ordenador externo con el objetivo de liberar de carga al micrordenador a bordo del vehículo debido a que éste presenta ciertas limitaciones de computación. Con el uso del ordenador externo, se consigue aumentar la velocidad de respuesta haciendo que el vehículo se anticipe a su movimiento y pueda reaccionar a su debido tiempo.

Una posible mejora, sería utilizar un procesador a bordo del vehículo más potente que el actual, de tal forma que no haría falta depender de un ordenador externo. Por lo tanto, la solución sería incorporar en el vehículo una Jetson, nombre comercial de una serie de *Single Board Computer* (Ordenador de Placa Única) creados por NVIDIA y pensados para utilizarse en el mundo de la robótica [35].

Figura 8.1. Placa NVIDIA Jetson Nano (Fuente: [35]).

Esta serie de placas comercializadas por NVIDIA son muy parecidas físicamente a las Raspberry Pi. Sin embargo, la Jetson presenta capacidades muy superiores en aplicaciones relacionadas con la visión por computador, la navegación de robots e inteligencia artificial. La gran diferencia es que es incorporar una mejor CPU y GPU que permiten procesar a altas velocidades pesados algoritmos de control. Por esta razón, sería una buena opción incorporarla en futuros trabajos, aunque su precio de venta se encuentre por encima de los 200 euros.

8.2. LIDAR

Actualmente, el vehículo es capaz de desplazarse de un punto a otro estableciendo por sí solo una ruta a partir de unos puntos de referencia guardados con anterioridad. Sin embargo, con los sensores que incorpora no es capaz de tomar decisiones si surge algún imprevisto en la trayectoria. Por ello, una posible mejora sería la incorporación de un LIDAR.

LIDAR es el acrónimo de *Light Detection and Ranging* o *Laser Imaging Detection and Ranging*, que se traduce como sistema de medición y detección de objetos mediante láser. Consiste en un emisor de rayos láser infrarrojos que al impactar sobre objetos se reflejan, volviendo al dispositivo y siendo captados por una lente receptora. Aplican, también, la técnica de Time of Flight con el objetivo de detectar objetos en tiempo real y de determinar la distancia entre éste y los objetos con una elevada precisión y resolución [36].

Figura 8.2. Muestra de un LIDAR (Fuente: [36]).

El sensor gira constantemente sobre sí mismo para capturar más datos sobre los objetos que le rodea. De esta manera, se obtiene una nube de puntos que al procesarla con un computador se consigue una imagen tridimensional en tiempo real del entorno.

Mediante ROS se facilita la tarea del procesado de la información, ya que incorpora una serie de herramientas capaces de interpretar los datos obtenidos. Éstas muestran las distancias a las que se encuentran los objetos y realizan un mapeado en 3D del entono. En la siguiente figura se muestra un ejemplo de la utilización de estas herramientas.

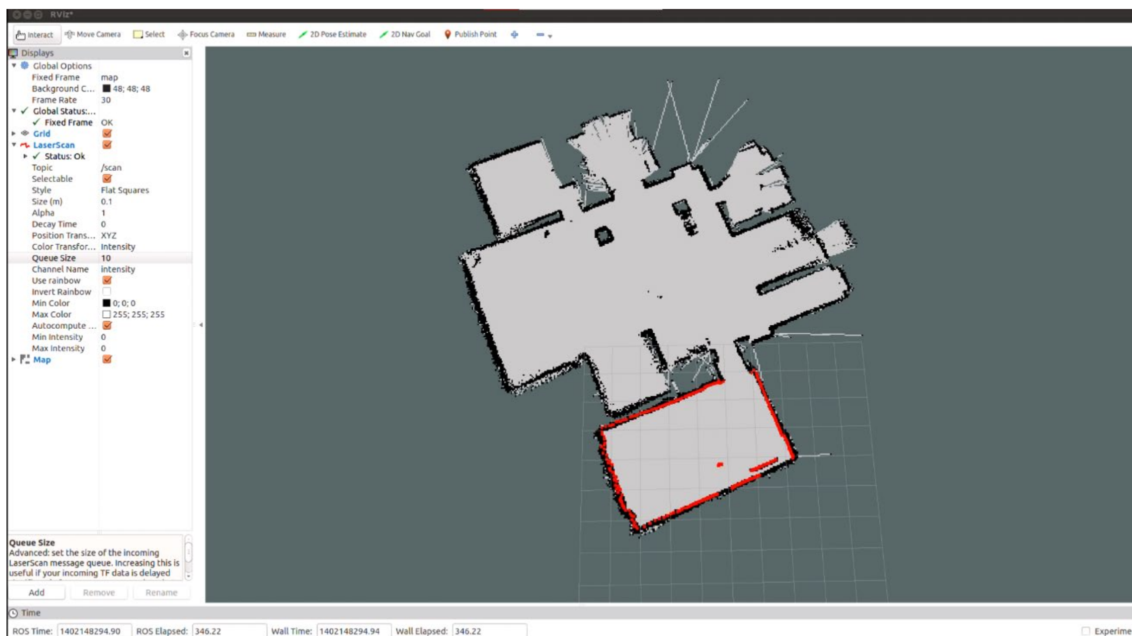


Figura 8.3. Mapeado resultante tras aplicar herramientas de ROS y utilizar un LIDAR. (Fuente: [37]).

8.3. SLAM

Alcanzar la solución del problema de posicionamiento en interiores cuando surgen imprevistos, no es suficiente con el uso del LIDAR y las diferentes herramientas de interpretación de resultados. Esto se debe a que tan solo se crea un mapa en función de los objetos encontrados en su entorno, y no se recalcula la trayectoria en función de la información procesada. Por esta razón, es necesario aplicar algoritmos matemáticos con el objetivo de desviar al vehículo de su ruta.

La técnica más utilizada hoy en día que cumple con este propósito, se conoce con el acrónimo de SLAM (*Simultaneous Location And Mapping*). El SLAM permite la elaboración de mapas de entornos desconocidos por un robot al mismo tiempo que estima su trayectoria en tiempo real [38].

Al igual que ROS proporciona herramientas para la interpretación de los datos obtenido a partir del LIDAR, también incluye programas con la intención de implementar el SLAM de forma sencilla.

8.4. Adición de funcionalidades

En este subapartado se proponen tres posibles mejores a implementar en un futuro con el propósito de lograr el nivel 5 de conducción autónoma, en el que el conductor pasa a ser un pasajero más, tal y como se definió en el **apartado 3.1** de este proyecto.

8.4.1. Detección de señales de tráfico

La incorporación de una cámara al proyecto daría la posibilidad de detectar señales de tráfico mediante técnicas de *Machine Learning* y *Deep Learning*. Así, se podrían respetar las normas de tráfico actuando autónomamente sobre los actuadores del vehículo.

Estas técnicas de reconocimiento son unas disciplinas del campo de la inteligencia artificial que, a través de la ejecución de un conjunto de algoritmos, dotan a los computadores de la capacidad de encontrar patrones en datos masivos, elaborar predicciones, razonar y sacar sus propias conclusiones [39].

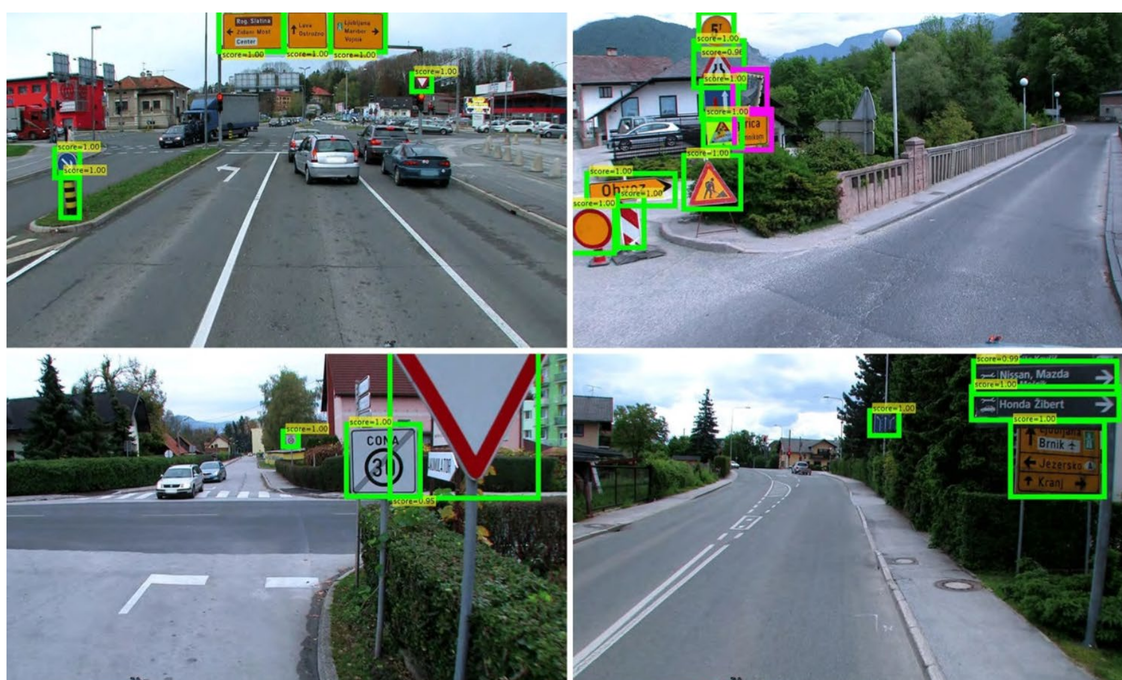


Figura 8.4. Ejemplo de reconocimiento de señales de tráfico en diferentes situaciones (Fuente: [40]).

8.4.2. Detección del estado de carga de la batería

Otra posible propuesta de mejora, sería monitorizar el estado de carga de la batería en cada instante de tiempo (SoC o *State of Charge* en inglés). Cuando se emplean baterías es muy importante vigilar continuamente que sus celdas trabajen dentro del *Safe Operating Area*. Esta área se define como las

condiciones de tensión, corriente y temperatura en las que se puede esperar que las celdas funcionen correctamente. Si las celdas no trabajan regularmente dentro de unos parámetros prestablecidos, según los elementos químicos con los que se han fabricado, podrían llegar a ser peligrosas estimulando reacciones químicas indeseadas que podrían provocar inflamaciones en las celdas o incluso, en casos extremo, explosiones.

En este caso, como se ha utilizado una batería de LiPo, es importante asegurar que sus celdas nunca bajen de los 2,8 V. En relación con la corriente, no se debe superar el límite indicado por el fabricante. Adicionalmente, la temperatura siempre debe de estar por debajo de los 60º C.

Así pues, para controlar estos parámetros es aconsejable añadir un sensor de tensión, de corriente y de temperatura.

8.4.3. Aparcamiento autónomo

A día de hoy, en el mercado automovilístico existen vehículos dotados con la capacidad de aparcar autónomamente en unas ciertas condiciones. Por lo tanto, otra posible funcionalidad que se podría incluir sería que el coche se aparcara por sí solo.

La acción de aparcar se podría dar cuando el usuario se lo indique o bien cuando se detecte que la batería esté a punto de agotarse.

9. Análisis del impacto ambiental

El impacto ambiental es la alteración del medio ambiente, causada por la actividad del ser humano [41]. A día de hoy, el cambio climático es uno de los temas más preocupantes en la sociedad, ya que la contaminación está modificando drásticamente el medio ambiente. La responsabilidad de los ciudadanos es fundamental para llevar a cabo acciones sostenibles que contribuyan a no dañar el medio ambiente. Por ello, se considera oportuno dedicar un apartado exclusivamente al análisis del impacto ambiental de este proyecto.

En relación con los productos eléctricos y electrónicos, estos presentan una vida útil reducida y al dejarlos de utilizar pueden ocasionar problemas de contaminación si no se les realiza un adecuado tratamiento. Adicionalmente, a la hora de su fabricación, se deben acatar una serie de reglamentos sobre la restricción de sustancias peligrosas.

Durante la elección de los diferentes aparatos electrónicos incluidos en este proyecto, se han considerado aquellos que cumplen con los requisitos establecidos por la Directiva 2011/65/EU, conocida con el nombre de RoHS (*Restriction of Hazardous Substances*).

La Directiva RoHS entró en vigor en junio del 2011 en la Unión Europa con el propósito de restringir el uso de seis sustancias peligrosas en aparatos eléctricos y electrónicos. Dichas sustancias son el cadmio (Cd), el mercurio (Hg), el plomo (Pb), el cromo hexavalente (CrVI), los polibromobifenilos (PBB) y los polibromodifeniléteres (PBDE). Las concentraciones máximas admitidas según la enmienda 2005/618/CE son de 0.1% del peso del material homogéneo para el mercurio, plomo, cromo VI, PBB y PBDE y de 0.01% para el cadmio [42].

Figura 9.1. Logotipo de RoHS (Fuente: [43]).

Por otro lado, siempre se opta por la reparación de los aparatos eléctricos y electrónicos, con el fin de alargar su vida útil y no malgastar nuevo material. Además, hay que considerar que este proyecto seguirá utilizándose tras la finalización de este trabajo. Esto se debe a que se empleará en la docencia y posiblemente más alumnos proseguirán con el desarrollo del mismo en futuros proyectos.

Por último, en la **tabla 9.1** se recogen los diversos elementos empleados con los materiales que se han utilizado durante su fabricación, mientras que en la **tabla 9.2** se destaca el impacto ambiental de los materiales principales y se exponen su método de reciclaje.

Tabla 9.1. Materiales de los dispositivos utilizados (Fuente: propia).

COMPONENTE	MATERIALES
Vehículo D12 Kei Truck	Plástico, policarbonato
Raspberry Pi 4 Model B, motor DC, variador, servomotor y reguladores, mando inalámbrico, balizas	Cobre, fibra de vidreo/Vidreo epoxi FR4 y en pequeñas cantidades: cerámica, aluminio, carbono, níquel, estaño, polímeros
Batería de Lipo	Polímero de litio y aluminio
Cables	Cobre y plástico para el aislante

Tabla 9.2. Impacto ambiental y reciclado de cada material (Fuente: propia).

MATERIAL	IMPACTO AMBIENTAL	RECLICADO
Aluminio	Su producción genera grandes cantidades de dióxido de carbono, que contribuyen negativamente al efecto invernadero.	Material 100% reciclable. Al reciclarlo se ahorra el 90% de la energía que se necesita para producirlo
Cobre	En concentraciones elevadas, es tóxico para los seres humanos y nocivo para las plantas y animales.	Material 100% reciclable. Si se realiza esta acción supone un 85% de ahorro de energía.
Polímeros de plástico	Su lenta descomposición y su composición química generan problemas ambientales, perjudicando la vida terrestre y la de los océanos. Además, durante su fabricación y descomposición se liberar toxinas perjudiciales para el entorno y la salud de cualquier organismo.	Su reciclado es difícil, aunque se puede reaprovechar para el elaborado de madera plástica, fibra textil y botellas

Polímero de litio	Provoca intoxicaciones, estados de confusión, en grandes cantidades puede incluso causar la muerte si se ingiere. Es inflamable y explosivo y presenta una lenta biodegradación.	Su reutilización es difícil puesto que a lo largo del tiempo pierde sus propiedades químicas, disminuyendo su vida útil.
Vidrio	A pesar de que se requieren de grandes cantidades de energía para su producción y su descomposición es lenta, es un material respetuoso con el medioambiente	Material 100% reciclable que requiere de un 26 % menos de la energía de su fabricación.

Conclusiones

A lo largo de este proyecto se ha desarrollado un algoritmo de control que permite a un vehículo a escala recorrer el circuito del laboratorio A5.4 de la Escola d'Enginyeria de Barcelona Est (Campus Diagonal-Besòs de la UPC). Adicionalmente, se ha implementado la comunicación externa con la finalidad de realizar los cálculos en un ordenador y de enviar, tan sólo, la velocidad y dirección pertinentes a la *Raspberry Pi 4 Model B*, procesador a bordo del vehículo, que actúa sobre el servomotor y el controlador del motor del vehículo en función de la información recibida.

Cabe destacar que el algoritmo permite desenvolverse adecuadamente ante cualquier circuito siempre y cuando las balizas estacionarias están colocadas en el primer cuadrante, es decir, que sus coordenadas sean positivas y, también, permite controlar el vehículo de forma manual mediante un mando inalámbrico Bluetooth.

La elección de ROS ha facilitado la implementación de los códigos programados y de la comunicación externa, puesto que su función es organizar diferentes procesos posibilitando la transmisión de información entre ellos mediante el uso de nodos y tópicos. Por otro lado, la utilización de ROS ha aportado cierta profesionalidad al trabajo, ya que este entorno está presente en la mayoría de proyectos relacionados con el mundo de la robótica.

El reto más grande al que se ha tenido que afrontar ha sido la implementación del sistema de navegación. Ha sido difícil encontrar una idónea ubicación de las balizas estacionarias, con el fin de disminuir los ruidos y ecos en las señales ultrasónicas y configurar los parámetros de las balizas para lograr una buena precisión en la localización del vehículo, sin perjudicar a la latencia de envío de datos hacia el ordenador.

Asimismo, se han planteado diferentes propuestas con el objetivo de conseguir un vehículo completamente autónomo, capaz de conducir por cualquier entorno, respetando las normas de tráfico y haciendo frente a los imprevistos que puedan surgir a lo largo de su ruta.

La proyección a futuro de este trabajo se centra en la docencia, dando la opción de utilizarlo en la optativa de Implementación de Sistemas de Control Automáticos (ISCA), de cuarto curso del Grado en Ingeniería Electrónica Industrial y Automática de la EEBE, o de ampliarlo en futuros trabajos de investigación teniendo en cuenta las propuestas de mejoras analizadas.

Respecto a la propuesta de trabajo, se considera que ha sido muy interesante y se agradece la oportunidad dada de desarrollarla debido a que en la carrera no se enseña ROS y ha sido la única forma de motivarme para explorar a fondo este entorno tan famoso en el mundo de la robótica. Durante el transcurso de este proyecto, se han consolidado las bases del sistema operativo de ROS y se ha

profundizado sobre sus conocimientos. Además, se han aumentado las habilidades de programación con el lenguaje de programación de alto nivel de Python.

En conclusión, se han alcanzado con creces los objetivos iniciales planteados, consiguiendo una correcta implementación de los diferentes sistemas, elementos y componentes utilizados con el fin de lograr la navegación autónoma del vehículo. Adicionalmente, se ha adquirido la suficiente experiencia para analizar el proyecto de manera objetiva y proponer posibles mejoras futuras. Paralelamente, se ha conseguido plasmar satisfactoriamente en este proyecto mi evolución a lo largo de los cuatro años que dura el Grado de Ingeniería Electrónica Industrial y Automática y las nuevas habilidades y competencias en a ROS y Python, adquiridas tras el transcurso de éste.

Planificación temporal

A continuación, se presenta el diagrama de Gantt correspondiente a este proyecto, donde se representan las actividades realizadas con sus fechas de inicio y de finalización y con las horas dedicadas.

Este diagrama se ha elaborado teniendo en mente la metodología establecida en el **apartado 2.3**. Las actividades se han dividido en pequeños objetivos cortos que tienen que conseguirse en un corto plazo de tiempo con la finalidad de aumentar la satisfacción y conseguir un mes de pruebas para mejorar los resultados. De este modo, las tareas se han agrupado en seis grupos que se han diferenciado con colores diferentes.

Por otro lado, las fechas son orientativas, ya que la metodología del tipo ágil permite flexibilidad y el tiempo de aprendizaje y estudio de Linux y ROS no se incluye puesto que se ha ido adquiriendo a lo largo del proyecto conforme se iba implementando los códigos.

Diagrama de Gantt

TAREA	INICIO	FIN	HORAS
Puesta a punto			
Instalación de Ubuntu	14-2-22	20-2-22	6 h
Instalación de ROS	14-2-22	20-2-22	2 h
Hardware			
Elección y prueba de los componentes	21-2-22	27-2-22	65 h
Diseño PCB	28-2-22	6-3-22	10 h
Montaje del vehículo y balizas	28-2-22	6-3-22	6 h
Sistema de navegación ultrasónico			
Aprendizaje del software y hardware	14-2-22	27-3-22	42 h
Estudio del mejor posicionamiento	14-3-22	27-3-22	40 h
Implementación del sistema en el laboratorio	28-3-22	3-4-22	12 h
Software			
Programación del control vehículo (control del driver del motor y del servomotor)	21-2-22	13-3-22	16 h
Programación del control manual	14-3-22	27-3-22	16 h
Implementación de la comuniación con ROS	28-3-22	3-4-22	10 h
Implementación del sistema de navegación en ROS	28-3-22	3-4-22	20 h
Programación del control autónomo	4-4-22	31-4-22	140 h
Implementación de los códigos en ROS	21-23-22	31-4-22	100 h
Puesta en marcha			
Pruebas y mejoras de códigos	2-5-22	5-6-22	100 h
Presentación del proyecto			
Redacción de la memoria	25-4-22	12-6-22	125 h
Preparación de la defensa	13-6-22	19-6-22	40 h

SEMANA 1 · SEMANA 2 · SEMANA 3 · SEMANA 4 · SEMANA 5 · SEMANA 6 · SEMANA 7 · SEMANA 8 · SEMANA 9 · SEMANA 10 · SEMANA 11 · SEMANA 12 · SEMANA 13 · SEMANA 14 · SEMANA 15 · SEMANA 16 · SEMANA 17 · SEMANA 18

Presupuesto

Seguidamente, se detalla el estudio económico de este proyecto, incluyendo la mano de obra y los componentes y materiales utilizados.

Mano de obra

Se estima el presupuesto de la mano de obra con una aproximación de las horas trabajadas y el sueldo estimado de una ingeniera junior.

La realización del trabajo de final de grado en la EEBE le corresponde un total de 24 crédito ECTS, equivaliendo cada crédito a 25 horas de dedicación, lo que da lugar a 600 horas trabajadas. A la hora de la realización del proyecto, se han invertido unas 750 horas, tal y como se recoge en la planificación temporal de este proyecto.

Además, se considera que el sueldo medio de una ingeniera junior es de 14 euros la hora [44], por lo que se calcula que el sueldo total es de 10.500 euros.

Tabla 0.1. Coste de ingeniería, impuestos incluidos (Fuente: propia).

HORAS TRABAJADAS	PRECIO (€/h)	TOTAL (€)
750	14	**10.500**

Componentes y materiales

En la siguiente tabla se enumeran los diversos componentes y materiales empleados para la implementación del proyecto con sus respectivos precios. Además, al final se detalla el presupuesto total invertido.

Tabla 0.2. Coste total de material y componentes utilizados, impuestos incluidos (Fuente: propia).

COMPONENTE/MATERIAL	CANTIDAD	PRECIO UD. (€)	PRECIO (€)
Vehículo D12 Kei Truck	1	77,78	77,78
Raspberry Pi 4 Model B	1	94,30	94,30
Tarjeta micro SD 16 GB	1	11,99	11,99
Motor DC	1	6,99	6,99
Variador TAS-202	1	16,29	16,29
Servomotor PDI-1181MG	1	4,49	4,49
Batería LiPo de 11,1 V y 5200 mAh	1	33,60	33,60

Convertidor JZK 24/12 V a 5 V 5 A	1	6,99	6,99
Regulador S13V30F5	1	19,95	19,95
Mando inalámbrico DUALSHOCK 4	1	59,99	59,99
Starter Set Super-NIA-3D (incluye 5 balizas y 1 modem)	1	499,00	499,00
Placa de topos	1	1,99	1,99
Interruptor	1	0,50	0,50
LED	1	0,05	0,05
Resistencia de 470 Ω	1	0,05	0,05
Borne para PCB	3	0,10	0,30
Pieza 20 pines macho de 2.54 mm	1	0,10	0,10
Pieza 20 pines hembra de 2.54 mm	1	0,10	0,10
Conector XT60 hembra y macho	2	0,79	1,58
Adaptador GPIO cable de 5 pines	1	0,99	0,99
Cable USB C	1	6,99	6,99
Tablero de madera MDF 30x60x1 cm	1	4,09	4,09
Listón de abeto sin cepillar 25x25mm x 2,4M	2	3,39	6,78
Escuadra plana con ángulo de 28x40x12 mm	4	0,51	2,04
Pack 30 tornillos para madera de 12 mm	1	2,79	2,79
Cinta de doble cara	1	7,19	7,19
Rollo de velcro de 5 m	1	5,49	5,49
TOTAL (€)			**872,40**

Total

El presupuesto de mano de obra ha sido de 10.500 €, mientras que gasto de componentes y materiales es de 872,40 €. Por tanto, el presupuesto total es **11.372,40 €**, tal y como muestra la siguiente tabla.

Tabla 0.3. Presupuesto total del proyecto, impuestos incluidos (Fuente: propia).

MANO DE OBRA	10.500 €
COMPONENTES Y MATERIALES	872,40 €
TOTAL	**11.372,40 €**

Se debe destacar que el primer prototipo siempre sale más caro debido a que se le suman el coste de las horas de trabajo de investigación, las pruebas realizadas y el coste de implementación. A medida que se vayan fabricando más unidades el precio del producto bajará.

Bibliografía

[1] Martínez, Jose. CLEM (2020). "*Navegación SLAM: robots que construyen mapas*". Disponible en: https://clem.es/noticia/navegacion-slam-robots-que-construyen-mapas-514 (Consulta: 4 de abril del 2022)

[2] Garrido Sotomayor, Sandra. IeBS (2021). "*Las metodologías ágiles más utilizadas y sus ventajas dentro de la empresa*". Disponible en: https://www.iebschool.com/blog/que-son-metodologias-agiles-agile-scrum/ (Consulta: 4 de abril del 2022)

[3] Páez, Gabriel. Economipedia (2021). "*Vehículo autónomo*". Disponible en: https://economipedia.com/definiciones/vehiculo-autonomo.html (Consulta: 6 de abril del 2022)

[4] Adeva, Roberto. Adsl Zone (2020). "*Todo lo que debes saber sobre el coche autónomo*". Disponible en: https://www.adslzone.net/reportajes/movilidad/coche-autonomo/ (Consulta: 6 de abril del 2022)

[5] Raspberry Pi (2020). "*¿Qué es la Raspberry Pi?*" Disponible en: https://raspberrypi.cl/que-es-raspberry/ (Consulta: 11 de abril del 2022).

[6] Raspberry Pi (2016). "*Raspberry Pi 3 Model B +".* Disponible en: https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/ (Consulta: 11 de abril del 2022)

[7] Raspberry Pi (2019). "*Raspberry Pi 4 Model B".* Disponible en: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/ (Consulta: 11 de abril del 2022)

[8] Marvelmind robotics. (2021). "*Marvelmind Indoor Navigation System Operating manual".* (Consulta: 28 de febrero del 2022)

[9] Terabee (2020). "*Time-of-Flight principle*". Disponible en: https://www.terabee.com/time-of-flight-principle/ (Consulta: 14 de abril del 2022)

[10] Silen and sistem (2021). "*¿Cuál es la velocidad del sonido?*". Disponible en: https://www.terabee.com/time-of-flight-principle/ (Consulta: 14 de abril del 2022)

[11] Sextantes (2020). "*Cómo funciona un GPS, trilateración*". Disponible en: https://www.sextantes.com/como-funciona-un-gps/ (Consulta: 23 de mayo del 2022)

[12] García, Eva María. ResearchGate (2017). “*Técnicas de Localización en Redes Inalámbricas de Sensores”.* Disponible en: https://www.researchgate.net/profile/Eva-Garcia-15 (Consulta: 23 de mayo del 2022).

[13] El Corte Inglés (2016). “*Mando inalámbrico Dualshock*”. Disponible en: https://www.elcorteingles.es/videojuegos/A20341004-mando-inalambrico-dualshock-negro-v2-ps4/ (Consulta: 14 de abril del 2022)

[14] Llamas, Luis (2016). “*Tipos de motores rotativos para proyectos de Arduino*”. Disponible en: https://www.luisllamas.es/tipos-motores-rotativos-proyectos-arduino/ (Consulta: 14 de abril del 2022)

[15] Llamas, Luis (2016). “*Controlar motores de corriente continua con Arduino y L298N*”. Disponible en: https://www.luisllamas.es/arduino-motor-corriente-continua-l298n/ (Consulta: 14 de abril del 2022)

[16] Mercado Libre. “*Jx Servo PDI-1181mg*”. Disponible en: https://articulo.mercadolibre.com.mx/MLM-1379058947-jx-servo-pdi-1181mg-35kg-gear-servo-digital-para-_JM (Consulta: 14 de abril del 2022)

[17] 12v24v products. “*Análisis de las 14 mejores baterías LiPo 12 V*”. Disponible en: https://www.12v24vproducts.org/es/bateria-lipo-12v-5000mah (Consulta: 15 de abril del 2022)

[18] 12v24v products. “*Los mejores 20 convertidores DC-DC 12V*”. Disponible en: https://www.12v24vproducts.org/es/convertidor-dc-dc-12v (Consulta: 15 de abril del 2022)

[19] Pololu. “*5 V, 3 A Step-Up/Step-Down Voltage Regulator S13V30F5*”. Disponible en: https://www.pololu.com/product/4082 (Consulta: 23 de mayo del 2022)

[20] Canonical (2019). “*Ubuntu logo*”. Disponible en: https://design.ubuntu.com/brand/ubuntu-logo/ (Consulta: 20 de abril del 2022)

[21] Ortego Delgado, Daniel (2017). “*Qué es ROS (Robot Operating System)*”. Disponible en: https://openwebinars.net/blog/que-es-ros/ (Consulta: 20 de abril del 2022)

[22] Arimetrics. “*Qué es Framework*”. Disponible en: https://www.arimetrics.com/glosario-digital/framework (Consulta: 20 de abril del 2022)

[23] ROS (2020). “*ROS Documentation*”. Disponible en: http://wiki.ros.org/ (Consulta: 3 de mayo del 2022)

[24] ROS (2020). "*Ros Noetic Ninjemys*". Disponible en: http://wiki.ros.org/noetic (Consulta: 3 de mayo del 2022)

[25] Generation ROBOTS (2022). "*ROS – Robot Operating System*". Disponible en: https://www.generationrobots.com/blog/en/ros-robot-operating-system-2/ (Consulta: 3 de mayo del 2022)

[26] ROS (2020). "*ROS Filesystem Concepts*". Disponible en: http://wiki.ros.org/msg (Consulta: 3 de mayo del 2022)

[27] Arimetrics. "*Qué es URI*". Disponible en: https://www.arimetrics.com/glosario-digital/uri (Consulta: 3 de mayo del 2022)

[28] Velasco, Rubén. Soft Zone (2022). "Aprende a dominar la Terminal de Linux como un profesional". Disponible en: https://www.softzone.es/linux/tutoriales/terminal-linux/ (Consulta: 5 de mayo del 2022)

[29] Logos-Marcas (2022). "*Python Logo*". Disponible en: https://logos-marcas.com/python-logo/ (Consulta: 5 de mayo del 2022)

[30] Colaborador de TechTarget. Computer Weekly (2021). "*Programación orientada a objetos, OOP*". Disponible en: https://www.computerweekly.com/es/definicion/Programacion-orientada-a-objetos-OOP (Consulta: 5 de mayo del 2022)

[31] Frontline Hobbies. "*WPL D12 1/10 RC RWD Drift Truck RTR Silver".* Disponible en: https://www.frontlinehobbies.com.au/wpl-d12-1-10-rc-rwd-kei-drift-truck-kit (Consulta: 6 de mayo del 2022)

[32] Compu Hoy (2022). "*¿Qué es el archivo Bashrc en Linux?*" Disponible en: https://www.compuhoy.com/que-es-el-archivo-bashrc-en-linux/ (Consulta: 6 de mayo del 2022)

[33] Python (2022). "*Clases*" Disponible en: https://docs.python.org/es/3/tutorial/classes.html (Consulta: 11 de junio del 2022)

[34] Universidad de Tarapacá. "*Tecnología de sensores*" Disponible en: http://www.eudim.uta.cl/ (Consulta: 4 de junio del 2022)

[35] Roca, Josep. Hard Zone (2021). "*¿Es NVIDIA Jetson una alternativa válida o mejor a Raspberry Pi?*" Disponible en: https://hardzone.es/reportajes/que-es/nvidia-jetson/ (Consulta: 6 de mayo del 2022)

[36] Ibáñez, Pablo. Motor Pasión (2017). "*Qué es un LIDAR, y cómo funciona el sensor más caro de los coches autónomos*". Disponible en: https://www.motorpasion.com/tecnologia (Consulta: 16 de mayo del 2022)

[37] Shikai Chen. Youtube (2014). "*SLAM base don RPLIDAR and ROS Hector Mapping*". Disponible en: https://www.youtube.com/watch?app=desktop&v=pCF7P7u8pDk (Consulta: 16 de mayo del 2022)

[38] López Torres, Patricia. Trabajo Final de Máster (2016). "*Análisl de algoritmos para localización y mapeo simultáneo de objetos*". Disponible en: https://idus.us.es/bitstream/handle/11441/54393/TFM_PatriciaL%C3%B3pezTorres.pdf?sequence=1&isAllowed=y (Consulta: 17 de mayo del 2022)

[39] Iberdrola (2020). "*Descubre los principales beneficios del Machine Learning*". Disponible en: https://www.iberdrola.com/innovacion/machine-learning-aprendizaje (Consulta: 17 de mayo del 2022)

[40] Domen Tabernik. Visual Cognitive Systems Laboratory (2017). "*Traffic-sign* detection". Disponible en: https://www.vicos.si/research/traffic-sign-detection/ (Consulta: 17 de mayo del 2022)

[41] Responsabilidad Social Empresarial y Sustentabilidad (2022). "*Impacto Ambiental: Qué es, definición, tipos, causas, medición y ejemplo*". Disponible en: https://responsabilidadsocial.net/impacto-ambiental-que-es-definicion-tipos-causas-medicion-y-ejemplo (Consulta: 28 de mayo del 2022)

[42] SGS. "*Bienes de consumo y venta minorista: ROHS*". Disponible en: https://www.sgs.es/es-es/consumer-goods-retail/electrical-and-electronics-total-solution-services/audio-video-and-household-appliances/rohs (Consulta: 29 de mayo del 2022)

[43] NEFAB. "*¿Qué es ROHS?".* Disponible en: https://www.nefab.com/es/mexico/politicas/sustentabilidad/rohs/que-es/appliances/rohs (Consulta: 29 de mayo del 2022)

[44] Glassdoor. "*Sueldos para Ingeniero Junior".* Disponible en: https://www.glassdoor.com.mx/Sueldos/madrid-ingeniero-junior-sueldo (Consulta: 12 de mayo del 2022)

Anexo A: Puesta en funcionamiento del sistema de navegación de interiores

A1. Introducción

El presente anexo es un manual de instrucciones cuyo propósito es guiar al usuario en la puesta en funcionamiento del *Super-NIA-3D*, set de inicio de la empresa *Marvelmind robotics* empleado con la finalidad de desarrollar un sistema de navegación en interiores.

El *Super-NIA-3D* está compuesto por cinco balizas (*o beacons*, en inglés) ultrasónicas que permiten localizar inalámbricamente robots, vehículos o personas dentro de un edificio. Además, incorpora un módem que recoge y envía al PC la información captada por el conjunto de dispositivos incluidos en el set.

Por otro lado, la empresa *Marvelmind robotics* dispone de una aplicación para configurar y ajustar el sistema de navegación. En ésta, también, se muestra la posición de las diferentes balizas y permite establecer una ruta por la cual el vehículo se desplazará autónomamente.

El conjunto destaca por la exactitud en el posicionamiento de las balizas debido a que proporciona una precisión de hasta ±2 cm. Adicionalmente, presenta un batería de 1000 mAh con la que se garantiza una autonomía de dos días a un par de meses, según el modo de funcionamiento en el que trabaje el sistema.

A1.1. Objeto

La intención de este anexo es facilitar y agilizar al usuario la puesta en marcha del sistema de navegación de interior *Super-NIA-3D*.

En concreto, el objetivo es orientar al usuario en:

- La instalación, tanto en Windows como en Linux, del programa *Dashboard*.
- La actualización a la última versión de software de los diferentes dispositivos.
- La configuración del sistema de navegación.
- La instalación de las balizas y el módem para obtener en el programa *Dashboard* la información generada.

A2. Puesta en funcionamiento del Super-NIA-3D

En este apartado, se enumeran y se muestran el conjunto de componentes que forman el set *Super-NIA-3D* y se detallan una serie de requisitos que se han de cumplir para garantizar el correcto funcionamiento del sistema. Seguidamente, se explican paso a paso las instrucciones para actualizar los diferentes dispositivos a la última versión de software, instalar el programa *Dashboard* y configurar el sistema de navegación.

A2.1. Material y requisitos del sistema de navegación

El set de inicio *Super-NIA-3D* de la empresa *Marvelmind robotics* incluye:

- 4 x *Super-Beacons* estacionarios
- 1 x *Super-Beacon* móvil
- 1 x Módem v5.1

En la **figura A1** se observa el material que contiene el kit. En la esquina inferior derecha se encuentra el módem y el resto son los cinco *Super-Beacons*.

Figura A1. Componentes del set de inicio *Super-NIA-3D*: módem y *Super-Beacons* (Fuente: [8]).

Destacar que, para garantizar un correcto funcionamiento del sistema de navegación se han de cumplir los siguientes requisitos:

- La distancia mínima entre balizas debe ser 3 m y no debe superar los 30 m.
- La distancia entre el módem y las balizas debe ser como mínimo de 1 m.
- Para la localización de objetos en un plano en 3D (X, Y, Z): hay que asegurar una línea de visión sin obstrucciones entre la baliza móvil y como mínimo tres balizas estacionarias.
- Para la localización de objetos en un plano en 2D (X, Y): hay que asegurar una línea de visión sin obstrucciones entre la baliza móvil y como mínimo dos balizas estacionarias.

A2.2. Instalación del programa Dashboard

A continuación, se describen los pasos para instalar el programa *Dashboard* en Windows y en Linux.

- **Instalación en Linux**

1. Descargar el paquete comprimido, que contine la última versión de software de los diferentes dispositivos y el fichero .exe para instalar la aplicación, de la página web oficial de la empresa *Marvelmind robotics* (https://marvelmind.com/download/). En la siguiente imagen se muestra marcado en rojo el archivo que se debe descargar.

Figura A2. Paquete a descargar que contiene el instalador del programa y la última versión de software del módem y de los *Super-Beacons* (Fuente: propia).

2. Descomprimir el zip que se ha descargado e ir a la dirección: *marvelmind_SW_2022_02_06\Dashboard\windows*

3. Ejecutar uno de los dos ficheros .exe según el tipo del sistema operativo del computador donde se realiza la instalación y seguir los pasos que se van indicando.
   En la siguiente figura se observan los pasos realizados una vez ejecutado el fichero .exe.

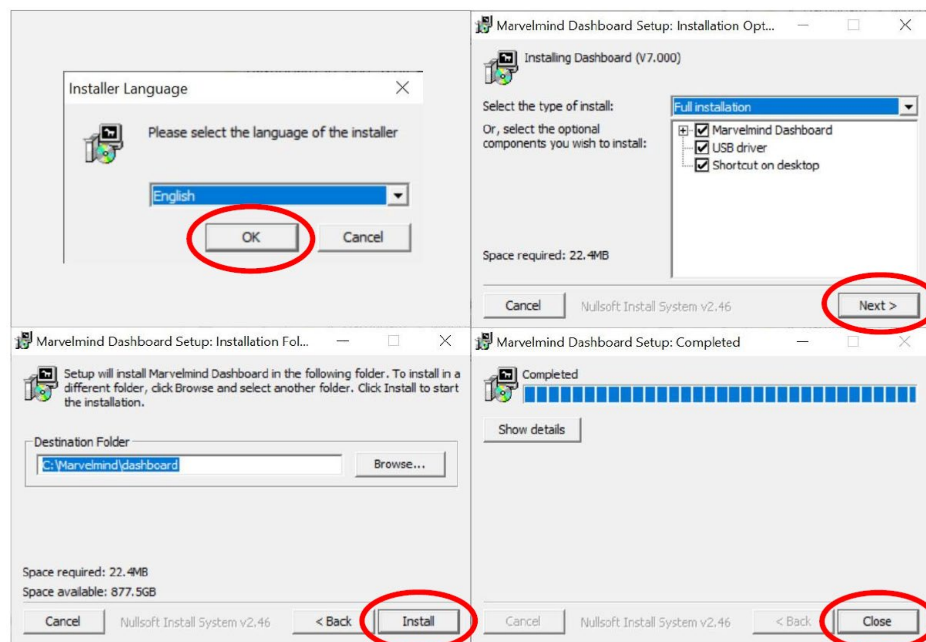


Figura A3. Pasos para la instalación del programa *Dashboard* (Fuente: propia).

- **Instalación en Linux**

Las siguientes instrucciones sirven para instalar el *Dashboard* en cualquier distribución de Linux, aunque solo se ha probado que funcione en Ubuntu 20.04.3 LTS.

1. Descargar el paquete comprimido, que contine la última versión de software de los diferentes dispositivos y el instalador del programa, de la página web oficial de la empresa *Marvelmind robotics* (https://marvelmind.com/download/). En la **figura A2** se muestra el paquete que hay que descargar.
2. Descomprimir el zip que se ha descargado y copiar las carpetas *Dashboard* y *API* al directorio que se quiera utilizar para el programa.
3. En este paso hay tres opciones posibles:
   - Si el programa se desea instalar en una plataforma de hardware ARM de 32 bits, habrá que ir a la dirección dentro de la carpeta previamente descomprimida:

     marvelmind_SW_2022_02_06/Dashboard/Linux/arm32
   - Si el programa se desea instalar en una plataforma de hardware ARM de 64 bits, habrá que ir a la dirección dentro de la carpeta previamente descomprimida:

     marvelmind_SW_2022_02_06/Dashboard/Linux/arm64
   - Si el programa se desea instalar en una plataforma de hardware x86, habrá que ir a la siguiente dirección dentro de la carpeta previamente descomprimida:

     marvelmind_SW_2022_02_06/Dashboard/Linux/x86
4. Abrir un terminal en el directorio que toque según el paso 3. Para ello, hacer clic derecho dentro de la carpeta y seleccionar la opción abrir en un terminal.
5. En la terminal escribir los siguientes comandos.

```
sudo cp libdashapi.so/usr/local/lib
sudo ldconfig
```

6. Presionar la tecla Enter.
7. En este paso hay dos posibles caminos:
   - Si la plataforma de hardware es ARM escribir el siguiente comando y seguidamente presionar la tecla Enter.

     ```
     sudo chmod 0777 ./dashboard_arm
     ```
   - Si la plataforma de hardware es x86 escribir el siguiente comando y seguidamente presionar la tecla Enter.

     ```
     sudo chmod 0777 ./dashboard_x86
     ```

8. Tras realizar los sietes paso anteriores, el programa ya estará listo para su uso. Cada vez que se desee utilizar el *Dashboard* habrá que ir al directorio pertinente según el paso 3 y hacer doble clic en el archivo *Dashboard_x86*. En la siguiente figura se muestra en rojo el archivo que hay que abrir para ejecutar el programa.

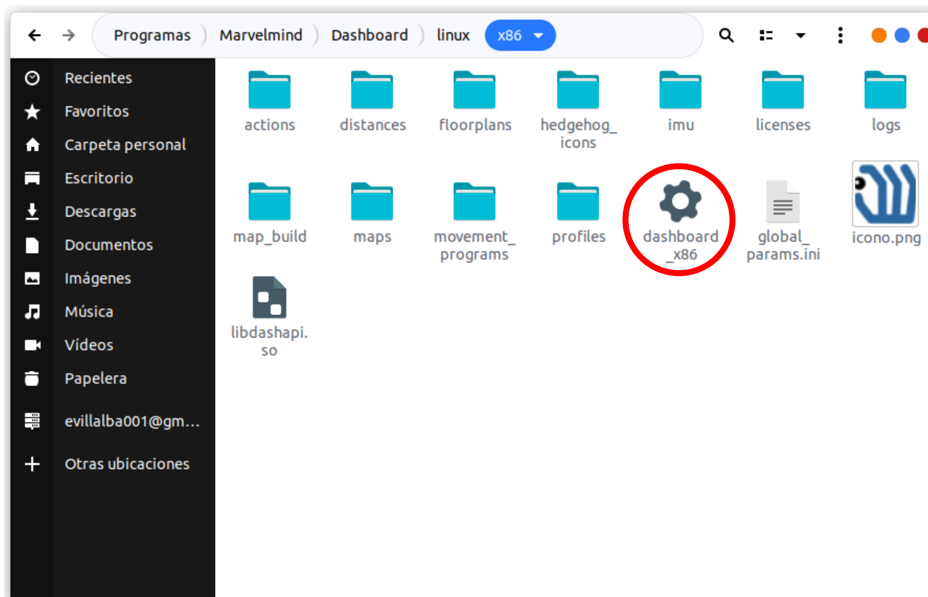


Figura A4. Archivo con el que se ejecuta el programa *Dashboard* (Fuente: propia).

A2.3. Actualización del software vía USB

Seguidamente, se explican los pasos para actualizar vía USB el software de los cinco *Super-Beacons* y el módem. Para poder llevar a cabo las siguientes instrucciones es necesario tener instalado el programa *Dashboard*.

1. Cargar los cinco *Super-beacons* usando un cable USB. La carga completa dura entre 1 y 2 horas.
2. Encender los cinco *Super-beacons*. Con la ayuda de un destornillador pequeño, conmutar cuidadosamente el segundo interruptor al estado *KE* dejando el primero en la posición en la que se encuentra, tal y como indica la figura 2.4.

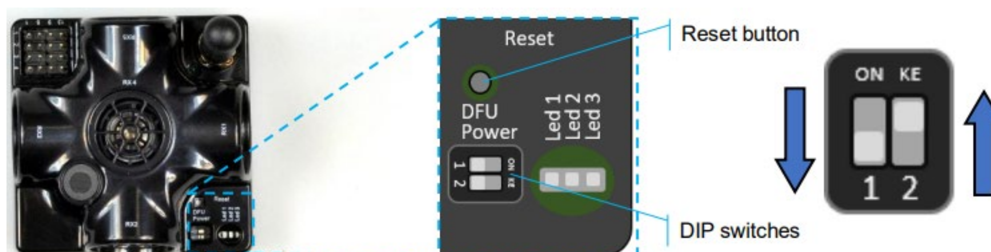


Figura A5. Encendido del *Super-beacon* (Fuente: [8]).

Para saber si se ha encendido correctamente, comprobar que el LED 3 parpadea tras transcurrir un breve instante de tiempo.

3. Abrir el programa *Dashboard*, instalado previamente.
4. Conectar el módem al PC con un cable USB.
5. En la barra de herramientas del programa ir a *Firmware* → *Upload firmware*.

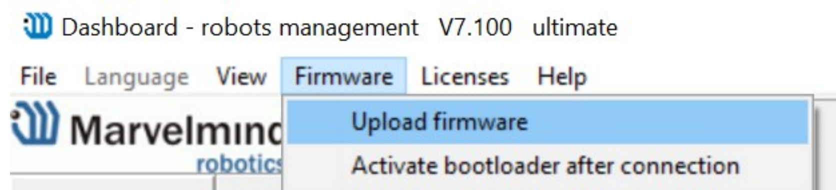


Figura A6. Ilustración del paso 5 y 13 (Fuente: propia).

6. En la pestaña que se ha abierto, clicar sobre el botón *Open file*.
7. Elegir el paquete descomprimido que se ha descargado en el paso 1 del apartado instalación del *Dashboard*. Concretamente, seleccionar el fichero .HEX ubicado en la siguiente dirección:

marvelmind_SW_2022_02_06\Software_nia\modem_hw51_nia

Una vez seleccionado clicar sobre el botón Abrir.

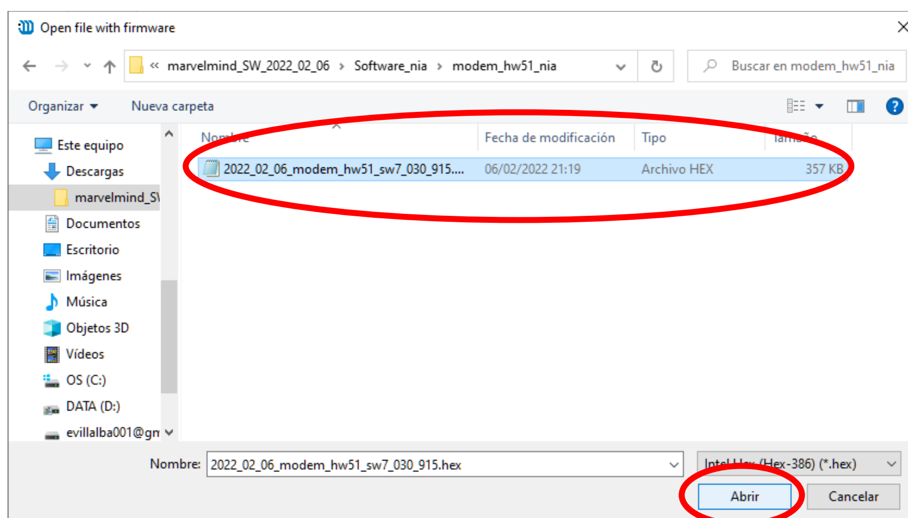


Figura A7. Selección del fichero .HEX del módem (Fuente: propia).

8. Clicar *Next.*
9. Esperar hasta que el módem se haya actualizado y clicar sobre el botón *OK.*
10. Pulsar el botón *Reset* del módem.
11. Desenchufar el módem del ordenador.
12. Conectar con el cable USB uno de los cinco *beacons*.
13. En la barra de herramientas del programa ir a *Firmware* → *Upload firmware*, **figura A6**.

14. Elegir el paquete descomprimido que se ha descargado en el paso 1 del apartado instalación del *Dashboard*. Concretamente, seleccionar el fichero .HEX ubicado en la dirección:

    marvelmind_SW_2022_02_06\Software_nia\super_beacon_nia

Una vez seleccionado clicar sobre el botón Abrir.

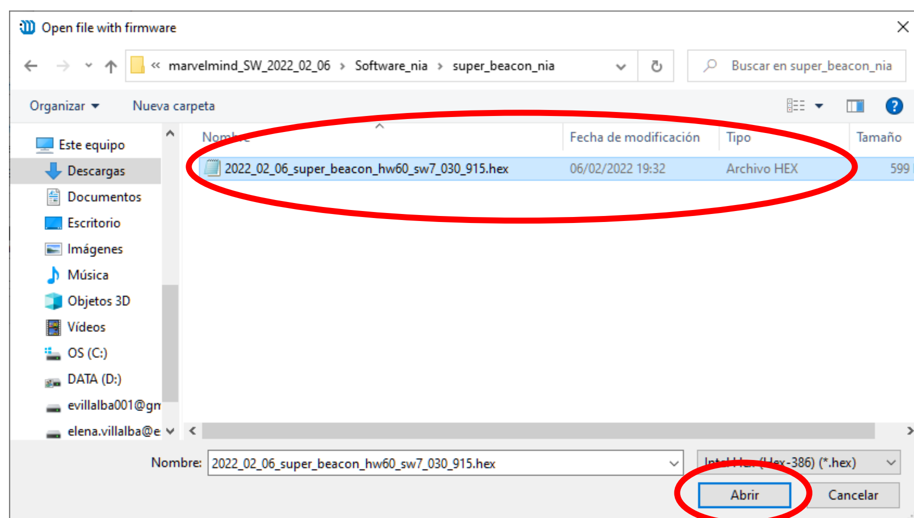


Figura A8. Selección del fichero .HEX del *beacon* (Fuente: propia).

15. Clicar *Next.*
16. Esperar hasta que el *beacon* se haya actualizado.
17. Pulsar el botón *Reset* del *beacon*.
18. Desconectar el *beacon* del ordenador y conectar el siguiente. Al conectar el siguiente, el programa automáticamente actualizará el beacon conectado.
19. Repetir los pasos 16, 17 y 18 cuatro veces más para actualizar los beacons restantes.
20. Una vez actualizados todos los *beacons* clicar sobre el botón *OK*.

A2.4. Configuración del sistema

Una vez actualizado el software de los diferentes dispositivos se procede a configurar y activar el sistema. Las siguientes instrucciones se deberán llevar a cabo para cada *beacon* a excepción del paso 5, ya que solo es necesario para el *beacon* móvil. En cambio, para configurar el módem solo hará falta seguir el paso 1 y 2.

1. Mientras el *beacon* o el módem están conectado al *Dashboard* mediante un cable USB, clicar sobre el botón *Default*, ubicado en la esquina inferior derecha del programa. De esta manera, se cargará la configuración por defecto.

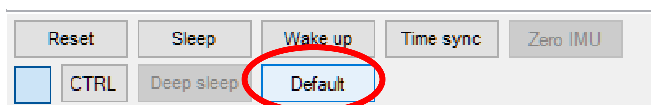


Figura A9. Botón *Default* ubicado en la esquina inferior derecha del programa *Dashboard* (Fuente: propia).

2. Comprobar que la configuración de la radio del módem sea la misma que la de cada *beacon*. En la siguiente figura se muestran los diferentes parámetros que hay que revisar.

CPU ID	Copy to clipboard	072449
Firmware version		V7.030 Super-Beacon-2
Power save functions		enabled / active
Hedgehog mode		disabled
Supply voltage, V (3.5..4.2)		3.97
Time from reset, h:m:s		00:05:44 / 15:49:18 / 0
RSSI from modem, dBm		-66
RSSI to modem, dBm		n/a
Profile		General
Radio frequency band		"915 MHz"
Carrier frequency, MHz		919.0
Radio channel		0
Device address (1..254)		4
Height, m (-320.000..320.000)		0.000
Measured temperature, °C		23
Ultrasonic frequency, Hz (100..65000)		31000
Desired speed, % (0..100)		n/a

Figura A10. Configuración de la radio (Fuente: propia).

3. Anotar la dirección de cada *beacon* para usarla posteriormente o cambiarla por la dirección deseada para más comodidad. Destacar que la dirección puede adquirir un número del 1 al 254. En la siguiente imagen se indica el lugar donde se establece la dirección del dispositivo.

CPU ID	Copy to clipboard	072449
Firmware version		V7.030 Super-Beacon-2
Power save functions		enabled / active
Hedgehog mode		disabled
Supply voltage, V (3.5..4.2)		3.97
Time from reset, h:m:s		00:05:44 / 15:49:18 / 0
RSSI from modem, dBm		-66
RSSI to modem, dBm		n/a
Profile		General
Radio frequency band		"915 MHz"
Carrier frequency, MHz		919.0
Radio channel		0
Device address (1..254)		4
Height, m (-320.000..320.000)		0.000
Measured temperature, °C		23
Ultrasonic frequency, Hz (100..65000)		31000
Desired speed, % (0..100)		n/a

Figura A11. Configuración de la dirección del beacon (Fuente: propia).

4. Configurar la frecuencia de ultrasonido para los cinco *beacons*. Para ello, hay que fijarse en cada una de las etiquetas de los *beacons*, ubicadas en sus bases, e introducir en el programa la frecuencia correspondiente de cada *beacon*. En la siguiente imagen se indica donde se configura la frecuencia de los dispositivos.

CPU ID	Copy to clipboard	072449
Firmware version		V7.030 Super-Beacon-2
Power save functions		enabled / active
Hedgehog mode		disabled
Supply voltage, V (3.5..4.2)		3.97
Time from reset, h:m:s		00:05:44 / 15:49:18 / 0
RSSI from modem, dBm		-66
RSSI to modem, dBm		n/a
Profile		General
Radio frequency band		"915 MHz"
Carrier frequency, MHz		919.0
Radio channel		0
Device address (1..254)		4
Height, m (-320.000..320.000)		0.000
Measured temperature, °C		23
Ultrasonic frequency, Hz (100..65000)		31000
Desired speed, % (0..100)		n/a

Figura A12. Configuración de la frecuencia del ultrasonido (Fuente: propia).

5. Configurar un *beacon* como móvil. Cuando se esté configurando el *beacon* que estará en el objeto a rastrear, pulsar sobre la opción *Hedgehog mode*.

CPU ID	Copy to clipboard	08514D
Firmware version		V7.030 Super-Beacon-2
Power save functions		enabled / active
Hedgehog mode		enabled
Supply voltage, V (3.5..4.2)		4.09

Figura A13. Configuración del beacon como móvil (Fuente: propia).

6. Tras realizar la configuración, pulsar sobre el botón *Reset* del *beacon*.

Después de actualizar los diferentes dispositivos a la última versión de software y configurar el sistema, el módem y los *beacons* estarán listos para utilizarse.

A2.5. Uso del sistema

En este apartado se explica la manera óptima de posicionar los *beacons* para obtener un mejor resultado y cómo utilizar el programa *Dashboard* para visualizar la trayectoria y la posición del objeto que se desea rastrear.

1. Colocar los *beacons* estacionarios en diferentes sitios de la habitación. La localización óptima para proporcionar una buena cobertura ultrasónica sería en las paredes posicionándolos en una posición vertical. Recordar encender los cinco *beacons* tal y como se indica en el paso 2 del **apartado A2.3** y no obstaculizar la señal colocando objetos en medio de los *beacons*.
2. Conectar el módem mediante un cable USB al ordenador.
3. Abrir el programa *Dashboard*. En la parte inferior izquierda, debe aparecer un rectángulo en verde, indicando que el módem está conectado, tal y como se muestra en la **figura A14**.

Figura A14. Comprobación del conexionado del módem conectado (Fuente: propia).

4. Despertar los *beacons* estacionarios haciendo doble clic en el panel del *Dashboard*.

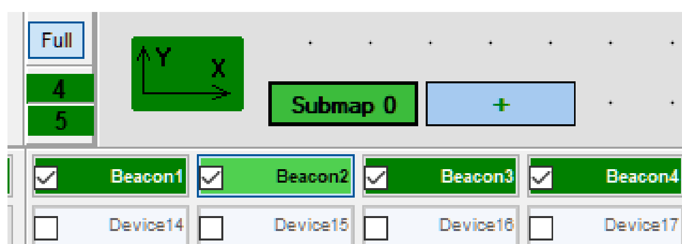


Figura A15. Panel *Dashboard* que indica el modo de funcionamiento de los *beacons* (Fuente: propia).

Volviendo a hacer doble clic sobre el dispositivo en el panel del *Dashboard*, el modo de funcionamiento del *beacon* pasará a *sleep mode*. En este modo, el dispositivo no actualizará su posición ni tampoco enviará ondas ultrasónicas para detectar el *beacon* móvil. Adicionalmente, al cambiar de estado se debe esperar unos diez segundos para que el dispositivo pueda reaccionar adecuadamente.

Otro punto a tener en cuenta es que, si el módem no está conectado al ordenador, los *beacons* pasaran al modo *sleep* automáticamente después de transcurrir aproximadamente un minuto.

Destacar que, si el dispositivo aparece en verde en el panel *Dashboard* indica que está en el modo *wake up*, es decir, el *beacon* estará despierto y listo para funcionar. En cambio, si el dispositivo aparece en blanco se encontrará en *sleep mode*.

5. Si se quiere utilizar el mapa en 2D, se debe anotar la altura en la que está situada cada *beacon* en el parámetro que se indica en la siguiente figura. Además, si los *beacons* estacionarios se encuentran a diferentes alturas y se quiere construir un mapa en 3D es recomendable apuntar las alturas para una mayor precisión.

Height, m (-320.000..320.000)	0.000
Measured temperature, °C	23
Ultrasonic frequency, Hz (100..65000)	31000
Desired speed, % (0..100)	n/a

Figura A16. Configuración de la altura en se encuentra los *beacons* (Fuente: propia).

6. Comprobar las distancias medidas por los *beacons* estacionarios. Para ello, observar si en la tabla de distancias, situada en la esquina izquierda del programa, aparecen las celdas en color blanco. Si son de amarillo y/o rojo, indica que la distancia entre *beacons* no se está midiendo correctamente. Para solucionar el problema, intentar reposicionar los *beacons* y asegurar que no hay obstáculos en medio.

P01	1	2	3	4
1		0.178	0.529	0.512
2	0.178		0.515	0.420
3	0.529	0.515		0.231
4	0.512	0.420	0.231	
5				

Figura A17. Tabla de posiciones (Fuente: propia).

7. Congelar el mapa clicando sobre la opción *freeze submap* cuando en la tabla aparezcan todas las posiciones en blanco. De esta manera, los beacons estáticos pararán de medir la posición relativa y estarán preparados para medir la distancia del *beacon* móvil.

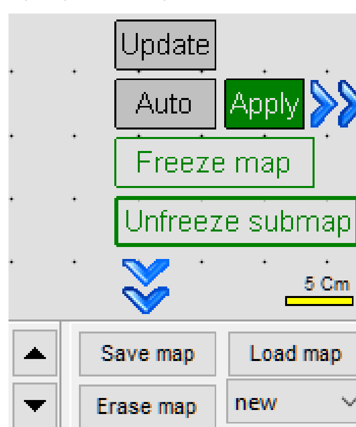


Figura A18. Botón para congelar el mapa y el submapa (Fuente: propia).

8. Encender y despertar el *beacon* móvil siguiendo los mismos pasos que se han realizado para encender el *beacon* estacionario.

Anexo B: Códigos

B1. Nodo ps4_controller

```
1   #!/usr/bin/env
2
3   #-----------#
4   # LIBRARIES #
5   #-----------#
6   # ROS libraires
7   import rospy
8   from car.msg import controller_state
9
10  # PS4 controller library
11  from pyPS4Controller.controller import Controller
12
13  #------------------#
14  # CLASS DEFINITION #
15  #------------------#
16  class MyController(Controller):

17      # ------------------------------------------------------------------ #
18      # Name: __init__                                                     #
19      # Description: setup PS4 controller and topics variables and         #
20      # declaration of the variables which are used throughout the code    #
21      # Parameters:                                                        #
22      # Result:                                                            #
23      # ------------------------------------------------------------------ #
24      def __init__(self, **kwargs):
25          Controller.__init__(self, **kwargs) # Controller initialization
26
27          # Variables definition with its initial status
28          msg.operating_mode = "manual"
29          msg.movement = "stop"
30          msg.direction = "straight"
31          msg.speed = 0     # Stop
32          msg.angle = 90    # Middle position
33          msg.flag_emergency_stop = 0
34          msg.flag_square_press = 0
35          pub.publish(msg) # Send to the topic the initial messages state
36
37          # Variable to count the number of times the R1 and L1 buttons
38          # have been pressed to set the motor speed
39          self.speed_counter = 0
40
41          # Variable to send information
42          self.prevMovement = ""
43          self.prevDirection = ""
44          self.prevSpeed = 10
45          self.prevAngle =  0
46
47
48
49
50
51
52
53
54
55
```

```
56      # --------------------------------------------------------------------- #
57      # Name: on_square_press                                                 #
58      # Description: when the triangle button is pressed, the car is          #
59      # controlled in manual mode                                             #
60      # Parameters:                                                           #
61      # Result:                                                               #
62      # --------------------------------------------------------------------- #
63      def on_square_press(self):    # Triangle button
64          msg.operating_mode = "manual"
65          msg.flag_emergency_stop = 0
66          pub.publish(msg)
67
68      # --------------------------------------------------------------------- #
69      # Name: on_circle_press                                                 #
70      # Description: when the X button is pressed, the car is controlled in #
71      # automatic mode                                                        #
72      # Parameters:                                                           #
73      # Result:                                                               #
74      # --------------------------------------------------------------------- #
75      def on_circle_press(self):    # X button
76          msg.operating_mode = "automatic"
77          msg.flag_emergency_stop = 0
78          pub.publish(msg)
79
80      # --------------------------------------------------------------------- #
81      # Name: on_triangle_press                                               #
82      # Description: when the circle button is pressed, the car stopped       #
83      # Parameters:                                                           #
84      # Result:                                                               #
85      # --------------------------------------------------------------------- #
86      def on_triangle_press(self): # Circle button
87          msg.flag_emergency_stop = 1
88          pub.publish(msg)
89
90      # --------------------------------------------------------------------- #
91      # Name: on_x_press                                                      #
92      # Description: when the square button is pressed, a flag is raised      #
93      # Parameters:                                                           #
94      # Result:                                                               #
95      # --------------------------------------------------------------------- #
96      def on_x_press(self): # Square button
97          msg.flag_square_press = 1
98          pub.publish(msg)
99
100     # --------------------------------------------------------------------- #
101     # Name: on_x_release                                                    #
102     # Description: when the square button is release, the square flag       #
103     # returns to normal state                                               #
104     # Parameters:                                                           #
105     # Result:                                                               #
106     # --------------------------------------------------------------------- #
107     def on_x_release(self): # Square button
108         msg.flag_square_press = 0
109         pub.publish(msg)
110
111
112
113
114
115
116
117
118
119
```

```
120     # ------------------------------------------------------------------ #
121     # Name: on_L3_up                                                     #
122     # Description: when the left joystick is moved up, the car moves     #
123     # forward                                                            #
124     # Parameters: - value: joystick potentiometer value                  #
125     # Result:                                                            #
126     # ------------------------------------------------------------------ #
127     def on_L3_up(self,value): # Left joystick
128         if value < -10000:
129             msg.movement = "forward"
130         else:
131             msg.movement = "stop"
132         self.SendInfo()
133
134     # ------------------------------------------------------------------ #
135     # Name: on_L3_down                                                   #
136     # Description: when the left joystick is moved down, the car moves   #
137     # backwards                                                          #
138     # Parameters: - value: joystick potentiometer value                  #
139     # Result:                                                            #
140     # ------------------------------------------------------------------ #
141     def on_L3_down(self,value): # Left joystick
142         if value > 10000:
143             msg.movement = "back"
144         else:
145             msg.movement = "stop"
146         self.SendInfo()
147
148     # ------------------------------------------------------------------ #
149     # Name: on_L3_y_at_rest                                              #
150     # Description: when the y axis of the left joystick is at rest, the  #
151     # car stopped                                                        #
152     # Parameters:                                                        #
153     # Result:                                                            #
154     # ------------------------------------------------------------------ #
155     def on_L3_y_at_rest(self): # Left joystick
156         msg.movement = "stop"
157         self.SendInfo()
158
159     # ------------------------------------------------------------------ #
160     # Name: on_L2_press                                                  #
161     # Description: if the right joystick is moved to the right, the car  #
162     # turn to the right; if it is moved to the left the car turn to the  #
163     # left and if it is at rest, the car goes straight                   #
164     # Parameters: - value: joystick potentiometer value                  #
165     # Result:                                                            #
166     # ------------------------------------------------------------------ #
167     def on_L2_press(self, value): # Right joystick
168         if value <= -8000:
169             msg.direction = "left"
170         elif value >= 8000:
171             msg.direction = "right"
172         else:
173             msg.direction = "straight"
174
175         # Angle selection based on right joystick value
176         msg.angle = self.select_angle(value)
177         self.SendInfo()
178
179
180
181
182
183
```

```
184     # ------------------------------------------------------------------- #
185     # Name: on_R1_press                                                   #
186     # Description: when the R1 button is pressed, the car speed increases #
187     # Parameters:                                                         #
188     # Result:                                                             #
189     # ------------------------------------------------------------------- #
190     def on_R1_press(self): # R1 Button - Increase speed
191         self.speed_counter += 1
192         msg.speed = self.select_speed()
193         self.SendInfo()
194
195     # ------------------------------------------------------------------- #
196     # Name: on_L1_press                                                   #
197     # Description: when the L1 button is pressed, the car speed decreases #
198     # Parameters:                                                         #
199     # Result:                                                             #
200     # ------------------------------------------------------------------- #
201     def on_L1_press(self): # L1 Button - Decrease speed
202         self.speed_counter -= 1
203         msg.speed = self.select_speed()
204         self.SendInfo()
205
206     # ------------------------------------------------------------------- #
207     # Name: select_angle                                                  #
208     # Description: selects the angle based on the right joystick value   #
209     # Parameters: - joystick_value: right joystick value                  #
210     # Result: servo angle. Range: 45 to 135. 45: max right angle.         #
211     # 135: max left angle. 90: middle position                            #
212     # ------------------------------------------------------------------- #
213     def select_angle(self,joystick_value):
214         # ---------------------- Turn to the left ----------------------- #
215         if joystick_value <= -8000 and joystick_value > -16000:
216             return 105
217         elif joystick_value <= -16000 and joystick_value > -24000:
218             return 120
219         elif joystick_value <= -24000:
220             return 135 # Max value
221         # ---------------------- Turn to the right ---------------------- #
222         elif joystick_value >= 8000 and joystick_value < 16000:
223             return 75
224         elif joystick_value >= 16000 and joystick_value < 24000:
225             return 60
226         elif joystick_value >= 24000:
227             return 45 # Max value
228         # ------------------------ Go straight ------------------------- #
229         else:
230             return 90 # Middle position
231
232     # ------------------------------------------------------------------- #
233     # Name: select_speed                                                  #
234     # Description: selects the speed based on the number of times the L1  #
235     # and R1 button are pressed                                           #
236     # Parameters:                                                         #
237     # Result: motor speed. Range: 10 to 100                               #
238     # ------------------------------------------------------------------- #
239     def select_speed(self):
240         if self.speed_counter <= 1:
241             self.speed_counter = 1
242             return 16
243         elif self.speed_counter== 2:
244             return 18
245         elif self.speed_counter == 3:
246             return 20
247         elif self.speed_counter== 4:
```

```
248             return 25
249         elif self.speed_counter == 5:
250             return 30
251         elif self.speed_counter == 6:
252             return 40
253         elif self.speed_counter == 7:
254             return 50
255         elif self.speed_counter == 8:
256             return 60
257         elif self.speed_counter == 9:
258             return 70
259         elif self.speed_counter == 10:
260             return 80
261         elif self.speed_counter >= 11:
262             self.speed_counter = 11
263             return 90
264
265     # ---------------------------------------------------------------- #
266     # Name: SendInfo                                                   #
267     # Description: update the topics with new messages                 #
268     # Parameters:                                                      #
269     # Result:                                                          #
270     # ---------------------------------------------------------------- #
271     def SendInfo(self):
272         if ((self.prevMovement != msg.movement) or (self.prevDirection !=
    msg.direction) or (self.prevSpeed != msg.speed) or (self.prevAngle !=
    msg.angle)):
273             self.prevMovement = msg.movement
274             self.prevDirection = msg.direction
275             self.prevAngle =  msg.angle
276             self.prevSpeed =  msg.speed
277             pub.publish(msg)
278
279 # Initialization of topic, node and messages
280 pub = rospy.Publisher('controller_state',controller_state,queue_size=10)
281 rospy.init_node('ps4_controller', anonymous=True)
282 rospy.loginfo("PS4 Controller Node Started")
283 msg = controller_state()
284
285 # Initialization of the PS4 controller
286 controller = MyController(interface="/dev/input/js0",
    connecting_using_ds4drv=False)
287 controller.listen()
```

B2. Nodo hedge_rcv_bin

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <stdbool.h>
4   #include <semaphore.h>
5   #include <fcntl.h>
6   #include "ros/ros.h"
7   #include "std_msgs/String.h"
8   #include "marvelmind_nav/hedge_pos.h"
9   #include "marvelmind_nav/hedge_pos_a.h"
10  #include "marvelmind_nav/hedge_pos_ang.h"
11  #include "marvelmind_nav/beacon_pos_a.h"
12  #include "marvelmind_nav/hedge_imu_raw.h"
13  #include "marvelmind_nav/hedge_imu_fusion.h"
14  #include "marvelmind_nav/beacon_distance.h"
15  #include "marvelmind_nav/hedge_telemetry.h"
16  #include "marvelmind_nav/hedge_quality.h"
17  #include "marvelmind_nav/marvelmind_waypoint.h"
18  extern "C"
19  {
20  #include "marvelmind_nav/marvelmind_hedge.h"
21  }
22
23  #include <sstream>
24
25  #define ROS_NODE_NAME "hedge_rcv_bin"
26  #define HEDGE_POSITION_TOPIC_NAME "hedge_pos"
27  #define HEDGE_POSITION_ADDRESSED_TOPIC_NAME "hedge_pos_a"
28  #define HEDGE_POSITION_WITH_ANGLE_TOPIC_NAME "hedge_pos_ang"
29  #define BEACONS_POSITION_ADDRESSED_TOPIC_NAME "beacons_pos_a"
30  #define HEDGE_IMU_RAW_TOPIC_NAME "hedge_imu_raw"
31  #define HEDGE_IMU_FUSION_TOPIC_NAME "hedge_imu_fusion"
32  #define BEACON_RAW_DISTANCE_TOPIC_NAME "beacon_raw_distance"
33  #define HEDGE_TELEMETRY_TOPIC_NAME "hedge_telemetry"
34  #define HEDGE_QUALITY_TOPIC_NAME "hedge_quality"
35  #define MARVELMIND_WAYPOINT_TOPIC_NAME "marvelmind_waypoint"
36
37  struct MarvelmindHedge * hedge= NULL;
38
39  static uint32_t hedge_timestamp_prev= 0;
40  marvelmind_nav::hedge_pos hedge_pos_noaddress_msg;// hedge coordinates
    message (old version without address) for publishing to ROS topic
41  marvelmind_nav::hedge_pos_a hedge_pos_msg;// hedge coordinates message for
    publishing to ROS topic
42  marvelmind_nav::hedge_pos_ang hedge_pos_ang_msg;// hedge coordinates and
    angle message for publishing to ROS topic
43  marvelmind_nav::beacon_pos_a beacon_pos_msg;// stationary beacon
    coordinates message for publishing to ROS topic
44  marvelmind_nav::hedge_imu_raw hedge_imu_raw_msg;// raw IMU data message for
    publishing to ROS topic
45  marvelmind_nav::hedge_imu_fusion hedge_imu_fusion_msg;// IMU fusion data
    message for publishing to ROS topic
46  marvelmind_nav::beacon_distance beacon_raw_distance_msg;// Raw distance
    message for publishing to ROS topic
47  marvelmind_nav::hedge_telemetry hedge_telemetry_msg;// Telemetry message
    for publishing to ROS topic
48  marvelmind_nav::hedge_quality hedge_quality_msg;// Quality message for
    publishing to ROS topic
49  marvelmind_nav::marvelmind_waypoint marvelmind_waypoint_msg;// Waypoint
    message for publishing to ROS topic
50
51
```

```
52  static sem_t *sem;
53  struct timespec ts;
54
55  ////////////////////////////////////////////////////////////////////////
56
57  void semCallback()
58  {
59      sem_post(sem);
60  }
61
62  static int hedgeReceivePrepare(int argc, char **argv)
63  {
64       // get port name from command line arguments (if specified)
65      const char * ttyFileName;
66      uint32_t baudRate;
67      if (argc>=2) ttyFileName=argv[1];
68        else ttyFileName=DEFAULT_TTY_FILENAME;
69      if (argc>=3) baudRate= atoi(argv[2]);
70        else baudRate=DEFAULT_TTY_BAUDRATE;
71
72      // Init
73      hedge=createMarvelmindHedge ();
74      if (hedge==NULL)
75      {
76          ROS_INFO ("Error: Unable to create MarvelmindHedge");
77          return -1;
78      }
79      hedge->ttyFileName=ttyFileName;
80      hedge->baudRate= baudRate;
81      hedge->verbose=true; // show errors and warnings
82      hedge->anyInputPacketCallback= semCallback;
83      startMarvelmindHedge (hedge);
84      return 0;
85  }
86
87  static bool hedgeReceiveCheck(void)
88  {
89    if (hedge->haveNewValues_)
90      {
91          struct PositionValue position;
92          getPositionFromMarvelmindHedge (hedge, &position);
93
94          hedge_pos_msg.address= position.address;
95          hedge_pos_ang_msg.address= position.address;
96
97          hedge_pos_msg.flags= position.flags;
98          hedge_pos_noaddress_msg.flags= position.flags;
99          hedge_pos_ang_msg.flags= position.flags;
100         if (hedge_pos_msg.flags&(1<<1))// flag of timestamp format
101           {
102             hedge_pos_msg.timestamp_ms= position.timestamp;// msec
103             hedge_pos_noaddress_msg.timestamp_ms= position.timestamp;
104           }
105          else
106           {
107             hedge_pos_msg.timestamp_ms= position.timestamp*15.625;// alpha-
    cycles ==> msec
108             hedge_pos_noaddress_msg.timestamp_ms=
    position.timestamp*15.625;
109           }
110         hedge_pos_ang_msg.timestamp_ms= position.timestamp;
111
112         hedge_pos_msg.x_m= position.x/1000.0;
113         hedge_pos_msg.y_m= position.y/1000.0;
```

```
114         hedge_pos_msg.z_m= position.z/1000.0;
115
116         hedge_pos_noaddress_msg.x_m= position.x/1000.0;
117         hedge_pos_noaddress_msg.y_m= position.y/1000.0;
118         hedge_pos_noaddress_msg.z_m= position.z/1000.0;
119
120         hedge_pos_ang_msg.x_m= position.x/1000.0;
121         hedge_pos_ang_msg.y_m= position.y/1000.0;
122         hedge_pos_ang_msg.z_m= position.z/1000.0;
123
124         hedge_pos_ang_msg.angle= position.angle;
125
126         hedge->haveNewValues_=false;
127
128         return true;
129     }
130    return false;
131 }
132
133 static bool beaconReceiveCheck(void)
134 {
135   uint8_t i;
136   struct StationaryBeaconsPositions positions;
137   struct StationaryBeaconPosition *bp= NULL;
138   bool foundUpd= false;
139   uint8_t n;
140
141   getStationaryBeaconsPositionsFromMarvelmindHedge (hedge, &positions);
142   n= positions.numBeacons;
143   if (n == 0)
144     return false;
145
146   for(i=0;i<n;i++)
147   {
148       bp= &positions.beacons[i];
149       if (bp->updatedForMsg)
150       {
151           clearStationaryBeaconUpdatedFlag(hedge, bp->address);
152           foundUpd= true;
153           break;
154       }
155   }
156   if (!foundUpd)
157     return false;
158   if (bp == NULL)
159     return false;
160
161   beacon_pos_msg.address= bp->address;
162   beacon_pos_msg.x_m= bp->x/1000.0;
163   beacon_pos_msg.y_m= bp->y/1000.0;
164   beacon_pos_msg.z_m= bp->z/1000.0;
165
166   return true;
167 }
168
169 static bool hedgeIMURawReceiveCheck(void)
170 {
171   if (!hedge->rawIMU.updated)
172      return false;
173
174   hedge_imu_raw_msg.acc_x= hedge->rawIMU.acc_x;
175   hedge_imu_raw_msg.acc_y= hedge->rawIMU.acc_y;
176   hedge_imu_raw_msg.acc_z= hedge->rawIMU.acc_z;
177
```

```
178   hedge_imu_raw_msg.gyro_x= hedge->rawIMU.gyro_x;
179   hedge_imu_raw_msg.gyro_y= hedge->rawIMU.gyro_y;
180   hedge_imu_raw_msg.gyro_z= hedge->rawIMU.gyro_z;
181
182   hedge_imu_raw_msg.compass_x= hedge->rawIMU.compass_x;
183   hedge_imu_raw_msg.compass_y= hedge->rawIMU.compass_y;
184   hedge_imu_raw_msg.compass_z= hedge->rawIMU.compass_z;
185
186   hedge_imu_raw_msg.timestamp_ms= hedge->rawIMU.timestamp;
187
188   hedge->rawIMU.updated= false;
189
190   return true;
191 }
192
193 static bool hedgeIMUFusionReceiveCheck(void)
194 {
195   if (!hedge->fusionIMU.updated)
196      return false;
197
198   hedge_imu_fusion_msg.x_m= hedge->fusionIMU.x/1000.0;
199   hedge_imu_fusion_msg.y_m= hedge->fusionIMU.y/1000.0;
200   hedge_imu_fusion_msg.z_m= hedge->fusionIMU.z/1000.0;
201
202   hedge_imu_fusion_msg.qw= hedge->fusionIMU.qw/10000.0;
203   hedge_imu_fusion_msg.qx= hedge->fusionIMU.qx/10000.0;
204   hedge_imu_fusion_msg.qy= hedge->fusionIMU.qy/10000.0;
205   hedge_imu_fusion_msg.qz= hedge->fusionIMU.qz/10000.0;
206
207   hedge_imu_fusion_msg.vx= hedge->fusionIMU.vx/1000.0;
208   hedge_imu_fusion_msg.vy= hedge->fusionIMU.vy/1000.0;
209   hedge_imu_fusion_msg.vz= hedge->fusionIMU.vz/1000.0;
210
211   hedge_imu_fusion_msg.ax= hedge->fusionIMU.ax/1000.0;
212   hedge_imu_fusion_msg.ay= hedge->fusionIMU.ay/1000.0;
213   hedge_imu_fusion_msg.az= hedge->fusionIMU.az/1000.0;
214
215   hedge_imu_fusion_msg.timestamp_ms= hedge->fusionIMU.timestamp;
216
217   hedge->fusionIMU.updated= false;
218
219   return true;
220 }
221
222 static void getRawDistance(uint8_t index)
223 {
224   beacon_raw_distance_msg.address_hedge= hedge->rawDistances.address_hedge;
225   beacon_raw_distance_msg.address_beacon= hedge-
    >rawDistances.distances[index].address_beacon;
226   beacon_raw_distance_msg.distance_m= hedge-
    >rawDistances.distances[index].distance/1000.0;
227 }
228
229 static bool hedgeTelemetryUpdateCheck(void)
230 {
231     if (!hedge->telemetry.updated)
232         return false;
233
234     hedge_telemetry_msg.battery_voltage= hedge->telemetry.vbat_mv/1000.0;
235     hedge_telemetry_msg.rssi_dbm= hedge->telemetry.rssi_dbm;
236
237     hedge->telemetry.updated= false;
238     return true;
239 }
```

```
240
241 static bool hedgeQualityUpdateCheck(void)
242 {
243     if (!hedge->quality.updated)
244         return false;
245
246     hedge_quality_msg.address= hedge->quality.address;
247     hedge_quality_msg.quality_percents= hedge->quality.quality_per;
248
249     hedge->quality.updated= false;
250     return true;
251 }
252
253 static bool marvelmindWaypointUpdateCheck(void)
254 {uint8_t i,n;
255  uint8_t nUpdated;
256
257     if (!hedge->waypoints.updated)
258         return false;
259
260     nUpdated= 0;
261     n= hedge->waypoints.numItems;
262     for(i=0;i<n;i++)
263     {
264         if (!hedge->waypoints.items[i].updated)
265           continue;
266
267         nUpdated++;
268         if (nUpdated == 1)
269         {
270           marvelmind_waypoint_msg.total_items= n;
271           marvelmind_waypoint_msg.item_index= i;
272
273           marvelmind_waypoint_msg.movement_type= hedge-
>waypoints.items[i].movementType;
274           marvelmind_waypoint_msg.param1= hedge->waypoints.items[i].param1;
275           marvelmind_waypoint_msg.param2= hedge->waypoints.items[i].param2;
276           marvelmind_waypoint_msg.param3= hedge->waypoints.items[i].param3;
277
278           hedge->waypoints.items[i].updated= false;
279         }
280     }
281
282     if (nUpdated==1)
283     {
284         hedge->waypoints.updated= false;
285     }
286     return (nUpdated>0);
287 }
288
289 /**
290  * Node for Marvelmind hedgehog binary streaming data processing
291  */
292 int main(int argc, char **argv)
293 {uint8_t beaconReadIterations;
294   // initialize ROS node
295   ros::init(argc, argv, ROS_NODE_NAME);
296
297   sem = sem_open(DATA_INPUT_SEMAPHORE, O_CREAT, 0777, 0);
298
299   // prepare hedgehog data receiver module
300   hedgeReceivePrepare(argc, argv);
301
302
```

```
303 // ROS node reference
304   ros::NodeHandle n;
305
306   // Register topics for puplishing messages
307   ros::Publisher hedge_pos_ang_publisher =
    n.advertise<marvelmind_nav::hedge_pos_ang>(HEDGE_POSITION_WITH_ANGLE_TOPIC_
    NAME, 1000);
308   ros::Publisher hedge_pos_publisher =
    n.advertise<marvelmind_nav::hedge_pos_a>(HEDGE_POSITION_ADDRESSED_TOPIC_NAM
    E, 1000);
309   ros::Publisher hedge_pos_noaddress_publisher =
    n.advertise<marvelmind_nav::hedge_pos>(HEDGE_POSITION_TOPIC_NAME, 1000);
310
311   ros::Publisher beacons_pos_publisher =
    n.advertise<marvelmind_nav::beacon_pos_a>(BEACONS_POSITION_ADDRESSED_TOPIC_
    NAME, 1000);
312
313   ros::Publisher hedge_imu_raw_publisher =
    n.advertise<marvelmind_nav::hedge_imu_raw>(HEDGE_IMU_RAW_TOPIC_NAME, 1000);
314   ros::Publisher hedge_imu_fusion_publisher =
    n.advertise<marvelmind_nav::hedge_imu_fusion>(HEDGE_IMU_FUSION_TOPIC_NAME,
    1000);
315
316   ros::Publisher beacon_distance_publisher =
    n.advertise<marvelmind_nav::beacon_distance>(BEACON_RAW_DISTANCE_TOPIC_NAME
    , 1000);
317
318   ros::Publisher hedge_telemetry_publisher =
    n.advertise<marvelmind_nav::hedge_telemetry>(HEDGE_TELEMETRY_TOPIC_NAME,
    1000);
319   ros::Publisher hedge_quality_publisher =
    n.advertise<marvelmind_nav::hedge_quality>(HEDGE_QUALITY_TOPIC_NAME, 1000);
320
321   ros::Publisher marvelmind_waypoint_publisher =
    n.advertise<marvelmind_nav::marvelmind_waypoint>(MARVELMIND_WAYPOINT_TOPIC_
    NAME, 1000);
322
323   // 200 Hz
324   ros::Rate loop_rate(200);
325
326   // default values for position message
327   hedge_pos_ang_msg.address= 0;
328   hedge_pos_ang_msg.timestamp_ms = 0;
329   hedge_pos_ang_msg.x_m = 0.0;
330   hedge_pos_ang_msg.y_m = 0.0;
331   hedge_pos_ang_msg.z_m = 0.0;
332   hedge_pos_ang_msg.flags = (1<<0);// 'data not available' flag
333   hedge_pos_ang_msg.angle= 0.0;
334
335   hedge_pos_msg.address= 0;
336   hedge_pos_msg.timestamp_ms = 0;
337   hedge_pos_msg.x_m = 0.0;
338   hedge_pos_msg.y_m = 0.0;
339   hedge_pos_msg.z_m = 0.0;
340   hedge_pos_msg.flags = (1<<0);// 'data not available' flag
341
342   hedge_pos_noaddress_msg.timestamp_ms = 0;
343   hedge_pos_noaddress_msg.x_m = 0.0;
344   hedge_pos_noaddress_msg.y_m = 0.0;
345   hedge_pos_noaddress_msg.z_m = 0.0;
346   hedge_pos_noaddress_msg.flags = (1<<0);// 'data not available' flag
347
348   beacon_pos_msg.address= 0;
349   beacon_pos_msg.x_m = 0.0;
```

```
350   beacon_pos_msg.y_m = 0.0;
351   beacon_pos_msg.z_m = 0.0;
352
353
354   while (ros::ok())
355   {
356     if (hedge->terminationRequired)
357       {
358         break;
359       }
360
361     if (clock_gettime(CLOCK_REALTIME, &ts) == -1)
362      {
363        ROS_INFO("clock_gettime");
364        return -1;
365      }
366     ts.tv_sec += 2;
367     sem_timedwait(sem,&ts);
368
369     if (hedgeReceiveCheck())
370      {// hedgehog data received
371        ROS_INFO("Address: %d, timestamp: %d, %d, X=%.3f  Y= %.3f  Z=%.3f
    Angle: %.1f  flags=%d",
372                 (int) hedge_pos_ang_msg.address,
373                 (int) hedge_pos_ang_msg.timestamp_ms,
374                 (int) (hedge_pos_ang_msg.timestamp_ms -
    hedge_timestamp_prev),
375                 (float) hedge_pos_ang_msg.x_m, (float)
    hedge_pos_ang_msg.y_m, (float) hedge_pos_ang_msg.z_m,
376                 (float) hedge_pos_ang_msg.angle,
377                 (int) hedge_pos_msg.flags);
378        hedge_pos_ang_publisher.publish(hedge_pos_ang_msg);
379        hedge_pos_publisher.publish(hedge_pos_msg);
380        hedge_pos_noaddress_publisher.publish(hedge_pos_noaddress_msg);
381
382        hedge_timestamp_prev= hedge_pos_ang_msg.timestamp_ms;
383      }
384
385     beaconReadIterations= 0;
386     while(beaconReceiveCheck())
387      {// stationary beacons data received
388        ROS_INFO("Stationary beacon: Address: %d, X=%.3f  Y= %.3f  Z=%.3f",
389                 (int) beacon_pos_msg.address,
390                 (float) beacon_pos_msg.x_m, (float) beacon_pos_msg.y_m,
    (float) beacon_pos_msg.z_m);
391        beacons_pos_publisher.publish(beacon_pos_msg);
392
393        if ((beaconReadIterations++)>4)
394          break;
395      }
396
397     if (hedgeIMURawReceiveCheck())
398     {
399        ROS_INFO("Raw IMU: Timestamp: %08d, aX=%05d aY=%05d aZ=%05d
    gX=%05d gY=%05d gZ=%05d  cX=%05d cY=%05d cZ=%05d",
400                 (int) hedge_imu_raw_msg.timestamp_ms,
401                 (int) hedge_imu_raw_msg.acc_x, (int)
    hedge_imu_raw_msg.acc_y, (int) hedge_imu_raw_msg.acc_z,
402                 (int) hedge_imu_raw_msg.gyro_x, (int)
    hedge_imu_raw_msg.gyro_y, (int) hedge_imu_raw_msg.gyro_z,
403                 (int) hedge_imu_raw_msg.compass_x, (int)
    hedge_imu_raw_msg.compass_y, (int) hedge_imu_raw_msg.compass_z);
404        hedge_imu_raw_publisher.publish(hedge_imu_raw_msg);
405     }
```

```
406
407     if (hedgeIMUFusionReceiveCheck())
408     {
409         ROS_INFO("IMU fusion: Timestamp: %08d, X=%.3f  Y= %.3f  Z=%.3f
    q=%.3f,%.3f,%.3f,%.3f v=%.3f,%.3f,%.3f  a=%.3f,%.3f,%.3f",
410                 (int) hedge_imu_fusion_msg.timestamp_ms,
411                 (float) hedge_imu_fusion_msg.x_m, (float)
    hedge_imu_fusion_msg.y_m, (float) hedge_imu_fusion_msg.z_m,
412                 (float) hedge_imu_fusion_msg.qw, (float)
    hedge_imu_fusion_msg.qx, (float) hedge_imu_fusion_msg.qy, (float)
    hedge_imu_fusion_msg.qz,
413                 (float) hedge_imu_fusion_msg.vx, (float)
    hedge_imu_fusion_msg.vy, (float) hedge_imu_fusion_msg.vz,
414                 (float) hedge_imu_fusion_msg.ax, (float)
    hedge_imu_fusion_msg.ay, (float) hedge_imu_fusion_msg.az);
415         hedge_imu_fusion_publisher.publish(hedge_imu_fusion_msg);
416     }
417
418     if (hedge->rawDistances.updated)
419      {uint8_t i;
420
421         for(i=0;i<4;i++)
422         {
423           getRawDistance(i);
424           if (beacon_raw_distance_msg.address_beacon != 0)
425           {
426               ROS_INFO("Raw distance: %02d ==> %02d,  Distance= %.3f ",
427                 (int) beacon_raw_distance_msg.address_hedge,
428                 (int) beacon_raw_distance_msg.address_beacon,
429                 (float) beacon_raw_distance_msg.distance_m);
430               beacon_distance_publisher.publish(beacon_raw_distance_msg);
431           }
432         }
433         hedge->rawDistances.updated= false;
434      }
435
436     if (hedgeTelemetryUpdateCheck())
437      {
438          ROS_INFO("Vbat= %.3f V, RSSI= %02d ",
439                 (float) hedge_telemetry_msg.battery_voltage,
440                 (int) hedge_telemetry_msg.rssi_dbm);
441          hedge_telemetry_publisher.publish(hedge_telemetry_msg);
442      }
443
444     if (hedgeQualityUpdateCheck())
445      {
446          ROS_INFO("Quality: Address= %d,  Quality= %02d %% ",
447                 (int) hedge_quality_msg.address,
448                 (int) hedge_quality_msg.quality_percents);
449          hedge_quality_publisher.publish(hedge_quality_msg);
450      }
451
452     if (marvelmindWaypointUpdateCheck())
453      {
454          int n= marvelmind_waypoint_msg.item_index+1;
455          ROS_INFO("Waypoint %03d/%03d: Type= %03d,  Param1= %05d, Param2=
    %05d, Param3= %05d ",
456                 (int) n,
457                 (int) marvelmind_waypoint_msg.total_items,
    marvelmind_waypoint_msg.movement_type,
458                 marvelmind_waypoint_msg.param1,
    marvelmind_waypoint_msg.param2, marvelmind_waypoint_msg.param3);
459          marvelmind_waypoint_publisher.publish(marvelmind_waypoint_msg);
460      }
```

```
461
462       ros::spinOnce();
463
464       loop_rate.sleep();
465     }
466
467     // Exit
468     if (hedge != NULL)
469       {
470         stopMarvelmindHedge (hedge);
471         destroyMarvelmindHedge (hedge);
472       }
473
474      sem_close(sem);
475
476     return 0;
477 }
```

B3. Nodo save_waypoints

```
1    #!/usr/bin/env
2
3    #-----------#
4    # LIBRARIES #
5    #-----------#
6    import csv
7    import rospy
8    from std_msgs.msg import String
9    from marvelmind_nav.msg import hedge_pos_ang # Mobile beacon coordinates
10   from car.msg import controller_state         # PS4 controller state
11
12   #------------------#
13   # CLASS DEFINITION #
14   #------------------#
15   class Save_Waypoints():
16
17       # ------------------------------------------------------------------ #
18       # Name: __init__                                                     #
19       # Description: Initialization of node, topics and global variables.  #
20       # In addition, the csv file is opened and it is indicated that it is #
21       # going to be written in it                                          #
22       # Parameters:                                                        #
23       # Result:                                                            #
24       # ------------------------------------------------------------------ #
25       def __init__(self):
26
27           # Declaration of global variables
28           # Indicate the number of waypoints we want to save
29           self.num_waypoints = 13
30
31           # Indicate the current operating mode
32           self.operating_mode = "manual"
33
34           # Flag to indicate when we have to save the waypoint
35           self.flag_save_waypoint = 0
36
37           # Flag to indicate the previous state of square button
38           self.flag_previous_press = 0
39
40           # Open csv file where the waypoints will be saved
41           self.csv_file = open('waypoints','w', newline = '')
42           self.csv_write = csv.writer(self.csv_file)
43
44           # autonomous_control node initialitzation
45           rospy.init_node('save_waypoints', anonymous=True)
46           rospy.Subscriber("hedge_pos_ang", hedge_pos_ang,
     self.save_waypoints)
47           rospy.Subscriber("controller_state", controller_state,
     self.ps4_controller)
48           rospy.loginfo("Save Waypoints Node Started")
49
50
51
52
53
54
55
56
57
58
59
```

```
60      # --------------------------------------------------------------- #
61      # Name: ps4_controller                                            #
62      # Description: save the current operating mode and indicate if the    #
63      # square button has been pressed                                  #
64      # Parameters: - data: data received through the controller_state topic#
65      # Result:                                                         #
66      # --------------------------------------------------------------- #
67      def ps4_controller(self,data):
68
69          self.operating_mode = data.operating_mode
70
71          # These statements prevent the square button bounce
72          if self.flag_previous_press != data.flag_square_press:
73              self.flag_save_waypoint = data.flag_square_press
74              self.flag_previous_press = data.flag_square_press
75
76      # --------------------------------------------------------------- #
77      # Name: save_waypoints                                            #
78      # Description: save the waypoint in a csv file when the operating    #
79      # mode is manual and the square button is pressed                 #
80      # Parameters: - data: data received through the hedge_pos_ang topic  #
81      # Result:                                                         #
82      # --------------------------------------------------------------- #
83      def save_waypoints(self,data):
84
85          if self.operating_mode == "manual":
86
87              # Save waypoints under a certain condition
88              if self.num_waypoints >= 1 and self.flag_save_waypoint == 1:
89
90                 # The flag return to 0 to prevent saving the same
91                   coordinates
92                  self.flag_save_waypoint = 0
93                  self.num_waypoints -= 1
94
95                  # Save and print the waypoint it has saved
96                  self.csv_write.writerow([data.x_m, data.y_m])
97                  print("waypoint saved: ", data.x_m, data.y_m)
98
99              # Indicate that all waypoints have been save and close csv file
100             elif self.num_waypoints == 0:
101                 print("All waypoints are saved")
102                 self.csv_file.close()
103
104 #--------------#
105 # MAIN PROGRAM #
106 #--------------#
107 if __name__ == '__main__':
108     Saving_Waypoints = Save_Waypoints()
109     try:
110         rospy.spin() # Keeps python from exiting until this node is stopped
111
112     except rospy.ROSInitException:
113         pass
```

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola d'Enginyeria de Barcelona Est

B4. Nodo curves_detector

```
114 #!/usr/bin/env
115
116 #-----------#
117 # LIBRARIES #
118 #-----------#
119 # ROS libraries
120 import rospy
121 from std_msgs.msg import String
122 from car.msg import curve_parameters
123 from marvelmind_nav.msg import hedge_pos_ang # Mobile beacon coordinates
124 from car.msg import controller_state         # PS4 controller state
125
126 import csv
127 import math
128 import numpy as np
129 from time import sleep
130
131 #------------------#
132 # CLASS DEFINITION #
133 #------------------#
134 class CurveDetector():
135     # ------------------------------------------------------------------- #
136     # Name: __init__                                                      #
137     # Description: declaration of global variables, saving waypoints,     #
138     # calculation of car rotation angles,  initialization of ros node     #
139     # initialization and distance between waypoints                      #
140     # Parameters: - laps:  number of labs                                 #
141     # Result:                                                             #
142     # ------------------------------------------------------------------- #
143     def __init__(self,laps):
144
145         # ---------------------- Global variables ----------------------- #
146         self.Laps = laps
147         self.Waypoints = []  # Contain the coordinates
148         self.WaypointsNum = 0 # Numbers of waypoints
149
150         self.CarAngle = []
151         self.PointDist = []
152
153         self.Counter = 0
154         self.AngleCount = 0
155         self.LapsCount = 0
156
157         self.Movement = "forward"
158         self.Direction = "straight"
159         self.Speed = "16"
160         self.Angle = 90
161
162         # Previous valor of the variables
163         self.PrevMovement = ""
164         self.PrevDirection = ""
165         self.PrevSpeed = ""
166         self.PrevAngle = 0
167
168         #Variable for the ps4 controller
169         self.flag_previous_square_press = 2
170
171         # ---------------------- Local variables ------------------------ #
172         i = 0 # Indicate the row of the list
173
174
```

```
175         # ---------------------- Saving Waypoints ------------------------ #
176         # Open csv file where the waypoints are saved
177         csv_file = open('waypoints','r', newline = '')
178         csv_read = csv.reader(csv_file)
179
180         # Read and save all the waypoints and calculates the angles
181         for row in csv_read:
182             self.Waypoints.append(row) # Save rows
183
184             # x and y coordinates are converted from string to float
185             for j in range(2):
186                 self.Waypoints[i][j] = float(self.Waypoints[i][j])
187
188             i += 1 # Update de row number
189
190         # Close csv file
191         csv_file.close()
192
193         # Print all the waypoints
194         print("Waypoints")
195         print(self.Waypoints)
196
197         # Determinate the numbers of waypoints
198         self.WaypointsNum = len(self.Waypoints)
199
200         # Car is in the first waypoint
201         self.xCar = self.Waypoints[0][0]
202         self.yCar = self.Waypoints[0][1]
203
204         # -------------------- Angle Determination ---------------------- #
205         self.CalCarAngle() # Calculate car rotation angles
206         print()
207         print("Car Rotation Angle")
208         print(self.CarAngle)
209
210         # -------------------- Waypoints Distances ---------------------- #
211         self.PointsDistance()
212         print()
213         print("Distance between waypoints")
214         print(self.PointDist)
215         print()
216
217         # --------------------- ROS Initialization ---------------------- #
218         rospy.init_node('curve_detector', anonymous=True)
219         rospy.Subscriber('hedge_pos_ang', hedge_pos_ang,
    self.CarCoordinates,queue_size=1,buff_size=2**24)
220         rospy.Subscriber("controller_state", controller_state,
    self.ps4_controller,queue_size=1,buff_size=2**24)
221         self.PUB = rospy.Publisher('curve_parameters',curve_parameters,
    queue_size=1)
222         self.MSG = curve_parameters()
223         rospy.loginfo("Publisher Node Started, now publishing messages")
224         print()
225
226     # ------------------------------------------------------------------ #
227     # Name: CarCoordinates                                               #
228     # Description: save the current coordinates of the car               #
229     # Parameters: - data: data received through the hedge_pos_ang topic  #
230     # Result:                                                            #
231     # ------------------------------------------------------------------ #
232     def CarCoordinates(self,data):
233         self.xCar = data.x_m
234         self.yCar = data.y_m
235
```

```
236     # ------------------------------------------------------------------- #
237     # Name: ps4_controller                                                #
238     # Description: does a reset when the reset button is pressed and is   #
239     # in automatic mode                                                   #
240     # Parameters: - data: data received through the controller_state topic#
241     # Result:                                                             #
242     # ------------------------------------------------------------------- #
243     def ps4_controller(self,data):
244
245         self.operating_mode = data.operating_mode
246
247         # These statements prevent the square button bounce
248         if self.flag_previous_square_press != data.flag_square_press:
249             self.flag_previous_square_press = data.flag_square_press
250
251             if (data.operating_mode == "automatic"):
252                 value = input("Do you want to reset the program:\n")
253                 print(f'You entered {value}')
254                 if (value == "y"):
255                     self.Counter = 0
256                     self.LapsCount = 0
257                     self.AngleCount = 0
258
259     # ------------------------------------------------------------------- #
260     # Name: Execute                                                       #
261     # Description: get the current rotation angle and sends it to the     #
262     # car's onboard processor. Moreover, indicate de movement of the car  #
263     # Parameters:                                                         #
264     # Result:                                                             #
265     # ------------------------------------------------------------------- #
266     def Execute(self):
267         # ---------------------- Local variables ------------------------ #
268         margin = 0.2     # Margin to change the waypoint
269
270         # Calculates the distance between the car and the current waypoint
271         car_dist = self.CarDistance(self.Counter)
272         self.Angle = self.CarAngle[self.AngleCount]
273
274         # ------------------------ CAR MOVEMENT ------------------------- #
275         # Start condition, order the car to start moving
276         if ((self.Counter == 0) and (self.LapsCount == 0)):
277             self.Movement = "forward"
278             self.Speed = 16 # Min speed
279
280         # ------------------------ Laps counter ------------------------- #
281         if ((self.Counter == (self.WaypointsNum - 1)) and (self.LapsCount <
    self.Laps) and (car_dist <= margin)):
282             self.LapsCount += 1
283
284         # -------------------- CAR DIRECTION ---------------------- #
285         # Correct to the middle of the waypoint distance
286         if (car_dist > (self.PointDist[self.Counter]/2)):
287             if (self.Angle >= 0):
288                 self.Direction = "right"
289                 self.Angle = 90 - self.Angle
290
291             else:
292                 self.Direction = "right"
293                 self.Angle = 90 + abs(self.Angle)
294
295         else:
296             self.Angle = 90
297             self.Direction = "straight"
298
```

```
299         # -------------------- CAR SPEED ---------------------- #
300         if ((self.Angle >= 80) and (self.Angle < 100)):
301             self.Speed = 16
302         elif (((self.Angle >= 70) and (self.Angle < 80)) or ((self.Angle >= 
    100) and (self.Angle < 110))):
303             self.Speed = 18
304         elif (((self.Angle >= 60) and (self.Angle < 70)) or ((self.Angle >= 
    110) and (self.Angle < 120))):
305             self.Speed = 20
306         else:
307             self.Spedd = 22
308 
309         # ---------------------- Change reference ----------------------- #
310         # Change the waypoints reference
311         if ((car_dist <= margin) and (self.Counter < (self.WaypointsNum - 
    1))):
312             self.Counter += 1
313         elif ((car_dist <= margin) and (self.Counter == (self.WaypointsNum 
    - 1))):
314             self.Counter = 0
315 
316         # ---------------------- Change car angle ----------------------- #
317         if ((car_dist <= margin) and (self.AngleCount < 
    self.WaypointsNum)):
318             self.AngleCount += 1
319 
320         elif ((car_dist <= margin) and (self.AngleCount == 
    self.WaypointsNum)):
321             self.AngleCount = 1
322 
323         # ------------------------ CAR MOVEMENT ------------------------- #
324         # Stop the car
325         if (self.LapsCount == self.Laps):
326             self.Movement = "stop"
327             self.Direction = "straight"
328             self.Speed = 0
329             value = input("Do you want to reset the program:\n")
330             print(f'You entered {value}')
331             if (value == "y"):
332                 self.Counter = 0
333                 self.LapsCount = 0
334                 self.AngleCount = 0
335             else:
336                 rospy.on_shutdown(myhook)
337 
338         # Send the angle and direction to car
339         self.SendInfo()
340 
341     # ------------------------------------------------------------------- #
342     # Name: CalCarAngle                                                   #
343     # Description: calculates the angle between two consecutive lines and #
344     # stores it in an array                                               #
345     # Parameters:                                                         #
346     # Result:                                                             #
347     # ------------------------------------------------------------------- #
348     def CalCarAngle(self):
349         # ---------------------- Local variables ------------------------ #
350         V = np.zeros(2) # Vector of the line
351         cal_angle = 0
352         corr_angle = []
353         quadrant = 0
354         prev_quadrant = 0
355         angulo_excepcion = [0,0]
356 
```

```
357         for i in range(self.WaypointsNum + 1):
358             if (i == 1): # Start angle
359                 V[0] = self.Waypoints[i][0] - self.Waypoints[i-1][0]
360                 V[1] = self.Waypoints[i][1] - self.Waypoints[i-1][1]
361                 cal_angle = math.degrees(math.atan(V[0]/V[1]))
362                 corr_angle.append(cal_angle)
363                 self.CarAngle.append(corr_angle[i-1])
364
365             if (i >= 1):
366                 if (i < self.WaypointsNum):
367                     V[0] = self.Waypoints[i][0] - self.Waypoints[i-1][0]
368                     V[1] = self.Waypoints[i][1] - self.Waypoints[i-1][1]
369
370                 else:
371                     V[0] = self.Waypoints[0][0] - self.Waypoints[i-1][0]
372                     V[1] = self.Waypoints[0][1] - self.Waypoints[i-1][1]
373
374
375                 cal_angle = math.degrees(math.atan(V[1]/V[0]))
376
377                 if ((V[0] > 0) and (V[1] > 0)):   # First quadrant
378                     quadrant = 1
379                     corr_angle.append(cal_angle)
380                 elif ((V[0] < 0) and (V[1] > 0)): # Second quadrant
381                     quadrant = 2
382                     corr_angle.append(cal_angle + 180)
383                 elif ((V[0] < 0) and (V[1] < 0)): # Third quadrant
384                     quadrant = 3
385                     corr_angle.append(cal_angle + 180)
386                 else:                             # Fourth quadrant
387                     quadrant = 4
388                     corr_angle.append(cal_angle + 360)
389                     if prev_quadrant == 1:
390                         angulo_excepcion[0] = i
391                         angulo_excepcion[1] = abs(cal_angle)
392                 prev_quadrant = quadrant
393
394         for i in range (len(corr_angle)):
395             if ((i >= 2) and (angulo_excepcion[0] != i)):
396                 self.CarAngle.append((corr_angle[i-1] - corr_angle[i]))
397
398             elif ((i >= 2) and (angulo_excepcion[0] == i)):
399                 self.CarAngle.append((corr_angle[i-1] -
angulo_excepcion[1]))
400
401             if (i == (len(corr_angle) - 1)):
402                 self.CarAngle.append((corr_angle[i] - corr_angle[1]))
403
404     # ------------------------------------------------------------------- #
405     # Name: PointsDistance                                                #
406     # Description: calculates the distance of two consecutive points and #
407     # stores it in an array                                               #
408     # Parameters:                                                         #
409     # Result:                                                             #
410     # ------------------------------------------------------------------- #
411     def PointsDistance(self):
412         # ---------------------- Local variables ------------------------ #
413         x_distance = 0.0
414         y_distance = 0.0
415         distance = 0.0
416
417         for i in range(self.WaypointsNum):
418             if (i != (self.WaypointsNum - 1)):
```

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola d'Enginyeria de Barcelona Est

```
419                 x_distance = abs(self.Waypoints[i][0] -
    self.Waypoints[i+1][0])
420                 y_distance = abs(self.Waypoints[i][1] -
    self.Waypoints[i+1][1])
421
422             else:
423                 x_distance = abs(self.Waypoints[i][0] -
    self.Waypoints[0][0])
424                 y_distance = abs(self.Waypoints[i][1] -
    self.Waypoints[0][1])
425
426             distance = math.sqrt((x_distance ** 2) + (y_distance ** 2))
427             self.PointDist.append(distance)
428
429     # ------------------------------------------------------------------ #
430     # Name: CarDistance                                                  #
431     # Description: calculates the distance between the car and the       #
432     # current waypoint                                                   #
433     # Parameters: - i: number of the previous waypoint                   #
434     # Result:     - distance: distance in meters between the car and the #
435     #                   current waypoint                                 #
436     # ------------------------------------------------------------------ #
437     def CarDistance(self,i):
438         # ---------------------- Local variables ------------------------ #
439         x_distance = 0.0
440         y_distance = 0.0
441         distance = 0.0
442
443         if (i < (len(self.PointDist) - 1)):
444             x_distance = abs(self.Waypoints[i+1][0] - self.xCar)
445             y_distance = abs(self.Waypoints[i+1][1] - self.yCar)
446
447         else:
448             x_distance = abs(self.Waypoints[0][0] - self.xCar)
449             y_distance = abs(self.Waypoints[0][1] - self.yCar)
450
451         distance = math.sqrt((x_distance ** 2) + (y_distance ** 2))
452
453         #print("Car Distance: ", distance)
454
455         return distance
456
457     # ------------------------------------------------------------------ #
458     # Name: SendInfo                                                     #
459     # Description: update the topics with new messages                   #
460     # Parameters:                                                        #
461     # Result:                                                            #
462     # ------------------------------------------------------------------ #
463     def SendInfo(self):
464
465         if ((self.PrevMovement != self.Movement) or (self.PrevSpeed !=
    self.Speed) or (self.PrevAngle != self.Angle) or (self.PrevDirection !=
    self.Direction)):
466
467             self.MSG.movement = self.Movement
468             self.MSG.direction = self.Direction
469             self.MSG.speed = self.Speed
470             self.MSG.angle = self.Angle
471
472
473             self.PUB.publish(self.MSG)
474             print(self.LapsCount, self.Movement, self.Direction,
    self.Speed, self.Angle)
475             print()
```

```
476
477                 self.PrevMovement = self.Movement
478                 self.PrevDirection = self.Direction
479                 self.PrevSpeed = self.Speed
480                 self.PrevAngle = self.Angle
481
482 # ------------------------------------------------------------------- #
483 # Name: myhook                                                        #
484 # Description: print information when the node is in shutdown         #
485 # Parameters:                                                         #
486 # Result:                                                             #
487 # ------------------------------------------------------------------- #
488 def myhook():
489     print("Shutdown done")
490
491 #--------------#
492 # MAIN PROGRAM #
493 #--------------#
494 if __name__ == '__main__':
495     # The number between brackets indicate the number of laps
496     Car = CurveDetector(1)
497     rate = rospy.Rate(100) # 100 Hz
498     sleep(10) # Waits 10 second to enable the car control node
499
500     try:
501         while (not rospy.is_shutdown()):
502             #rospy.spin()
503             Car.Execute()
504             rate.sleep()
505
506     except rospy.ROSInitException:
507         print("Curves Detector Node Shut Down")
```

B5. Nodo autonomous_control

```
1   #!/usr/bin/env
2
3   #-----------#
4   # LIBRARIES #
5   #-----------#
6   # ROS libraries
7   import rospy
8   from car.msg import autonomous_parameters
9   from marvelmind_nav.msg import hedge_pos_ang # Mobile beacon coordinates
10  from car.msg import controller_state         # PS4 controller state
11
12  import csv
13  import math
14  from time import sleep
15
16  #------------------#
17  # CLASS DEFINITION #
18  #------------------#
19
20  # ------------------------------------------------------------------------ #
21  # Class name: LineEquation                                                 #
22  # Description: calculates the equation of the line with the actual and    #
23  # the next waypoints                                                       #
24  # Parameters: - x0: current waypoint x                                     #
25  #             - y0: current waypoint y                                     #
26  #             - x1: next waypoint x                                        #
27  #             - y1: next waypoint y                                        #
28  # Result: - Slope: m term of the equation of the line                      #
29  #         - IndTerm: independent term of the equation of the line          #
30  # ------------------------------------------------------------------------ #
31  class LineEquation(object):
32      def __init__(self,x0,y0,x1,y1):
33
34          # Parameters initialization
35          self.x0 = x0
36          self.y0 = y0
37          self.x1 = x1
38          self.y1 = y1
39
40          # Variables for the result
41          self.Slope = 0.0
42          self.IndTerm = 0.0
43
44      def Execute(self):
45          # mx0 + 1 = y0
46          # mx1 + 1 = y1
47
48          # Calculates determinant
49          det = (self.x0 * 1) - (1 * self.x1)
50
51          # System with solution
52          if det != 0:
53              self.Slope   = (self.y0 * 1 - 1 * self.y1) / det
54              self.IndTerm = (self.x0 * self.y1 - self.y0 * self.x1) / det
55
56          # System without solution
57          else:
58              self.Slope   = None
59              self.IndTerm = None
60
61
```

```
62  # ----------------------------------------------------------------------- #
63  # Class name: PublishingMSG                                               #
64  # Description: Publish message to the autonomous parameters' topics       #
65  # Parameters:                                                             #
66  # Result:                                                                 #
67  # ----------------------------------------------------------------------- #
68  class PublishingMSG(object):
69  
70      Angle = 90
71      PrevAngle = 0
72  
73      def __init__(self):
74          self.PUB = 
    rospy.Publisher('autonomous_parameters',autonomous_parameters,
    queue_size=1)
75          self.MSG = autonomous_parameters()
76  
77      def publish_messages(self):
78  
79          if (PublishingMSG.PrevAngle != PublishingMSG.Angle):
80  
81              # Update the topics messages
82              self.MSG.angle = PublishingMSG.Angle
83  
84              # Send the current messages
85              self.PUB.publish(self.MSG)
86  
87              # Save the previous messages
88              PublishingMSG.PrevAngle = PublishingMSG.Angle
89  
90  
91  #----------------------#
92  # FINITE STATE MACHINE #
93  #----------------------#
94  # ----------------------------- TRANSITIONS ----------------------------- #
95  # ----------------------------------------------------------------------- #
96  # Class name: Transition                                                  #
97  # Description: allows state change                                        #
98  # Parameters:                                                             #
99  # Result:                                                                 #
100 # ----------------------------------------------------------------------- #
101 class Transition(object):
102     def __init__(self,toState):
103         self.toState = toState
104 
105     def Execute(self):
106         print ("Transistioning...")
107 
108 
109 # ------------------------------- STATES -------------------------------- #
110 # ----------------------------------------------------------------------- #
111 # Class name: State                                                       #
112 # Description: The global variables to be used in the different states    #
113 # are defined                                                             #
114 # Parameters:                                                             #
115 # Result:                                                                 #
116 # ----------------------------------------------------------------------- #
117 class State(object):
118 
119     # Global Variables
120     Waypoint_x0 = 0.0
121     Waypoint_y0 = 0.0
122     Waypoint_x1 = 0.0
123     Waypoint_y1 = 0.0
```

```
124
125     EqLn_Slope = 0.0
126     EqLn_IndTerm = 0.0
127
128     LeftMargin = 0.0
129     RightMargin = 0.0
130
131     PrevState = ""
132     StopFlag = 0
133     ChangeWaypointFlag = 0
134
135     def __init__(self,FSM):
136         self.FSM = FSM
137         self.Margin = 0.0
138
139     def EnterMargin(self):
140         self.Margin  = 0.04 # Allowed error for the car trayectory
141
142     def EneterDistMargin(self):
143         self.DistMargin = 0.2 # Margin distance which the waypoint is
change
144
145     def Exit(self):
146         print("FSM shutting down")
147         start_input = input("press any key to continue the program")
148
149 # ----------------------------------------------------------------------- #
150 # Class name: CarTrayectory                                               #
151 # Description: Calculate the equation of the line that describes the path #
152 # that the car must follow.                                               #
153 # Parameters:                                                             #
154 # Result:                                                                 #
155 # ----------------------------------------------------------------------- #
156 class CarTrayectory(State, LineEquation):
157     def __init__(self,FSM):
158         State.__init__(self,FSM)
159
160         self.Waypoints = [] # List that will containt all the waypoints
161         self.WaypointsCounter = 1
162         self.CountLaps = 0
163         self.CurrentWaypoint = []
164         self.PrevWaypoint = []
165
166         # Open csv file where the waypoints are saved
167         csv_file = open('waypoints','r', newline = '')
168         csv_read = csv.reader(csv_file)
169
170         # Read and save all the waypoints
171         i = 0 # Indicate the row of the list
172         for row in csv_read:
173             # Save rows
174             self.Waypoints.append(row)
175             # x and y coordinates are converted from string to float
176             for j in range(2):
177                 self.Waypoints[i][j] = float(self.Waypoints[i][j])
178             i += 1 # Update de row number
179
180         # Close csv file
181         csv_file.close()
182
183         # Determinate the numbers of waypoints taking into account the 0
184         self.WaypointsNum = len(self.Waypoints) - 1
185
186
```

```
187     def Enter(self):
188         self.CurrentWaypoint = self.Waypoints[self.WaypointsCounter]
189         self.PrevWaypoint = self.Waypoints[self.WaypointsCounter - 1]
190
191         print("Waypoint Counter:  ",self.WaypointsCounter)
192         print("Previous Waypoint: ",self.PrevWaypoint)
193         print("Current Waypoint:  ",self.CurrentWaypoint)
194
195     def Execute(self):
196         print("Calculating car trayectory")
197
198         # Save the previous and current waypoints coordinates
199         State.Waypoint_x0 = self.PrevWaypoint[0]
200         State.Waypoint_y0 = self.PrevWaypoint[1]
201         State.Waypoint_x1 = self.CurrentWaypoint[0]
202         State.Waypoint_y1 = self.CurrentWaypoint[1]
203
204         # Calculate the equation of the line that describe the previous
205         # and current waypoints
    LineEquation.__init__(self,State.Waypoint_x0,State.Waypoint_y0,State.Waypoi
    nt_x1,State.Waypoint_y1)
206         LineEquation.Execute(self)
207
208         # Save in the global variable the slope and the independent term
209         # that it has been calculated with the object class
210         State.EqLn_Slope = self.Slope
211         State.EqLn_IndTerm = self.IndTerm
212
213         print("m = ",State.EqLn_Slope," b = ",State.EqLn_IndTerm)
214
215         # CHANGES STATE
216         # First case - EqLn parallel to y axis
217         if (( State.EqLn_Slope == None) and (State.EqLn_IndTerm == None)):
218             self.FSM.ToTransition("toxRange")
219
220         # Second case - EqLn parallel to x axis
221         elif (State.EqLn_Slope == 0):
222             self.FSM.ToTransition("toyRange")
223
224         # Third case - m = k1 and b = k2
225         else:
226             self.FSM.ToTransition("tobRange")
227
228     def Exit(self):
229         # Pick the next waypoints reference
230         if (self.WaypointsCounter < self.WaypointsNum):
231             self.WaypointsCounter += 1
232
233         # If there aren't more waypoints the FSM will shut down
234         elif (self.WaypointsCounter >= self.WaypointsNum):
235             self.CountLaps  +=1
236             self.WaypointsCounter = 1
237
238         if (self.CountLaps == CarNavigation.Laps):
239             # Reset of the program if you want
240             value = input("Do you want to reset the program:\n")
241             print(f'You entered {value}')
242             if (value == "y"):
243                 self.CountLaps = 0
244                 self.WaypointsCounter = 1
245             else:
246                 State.StopFlag = 1
247
248         print("Finished calculating car trayectory")
```

```
249 # ------------------------------------------------------------------- #
250 # Class name: xRange                                                   #
251 # Description: calculates the allowed deviation margins of the car when #
252 # it goes perpendicular to the y-axis                                  #
253 # Parameters:                                                          #
254 # Result:                                                              #
255 # ------------------------------------------------------------------- #
256 class xRange(State):
257     def __init__(self,FSM):
258         State.__init__(self,FSM)
259         self.Distance = 0.0
260
261     def Enter(self):
262         State.EnterMargin(self)
263         self.Distance = State.Waypoint_y1 - State.Waypoint_y0
264
265         print("Waypoints Distance = ", self.Distance)
266         print("I'm in xRange")
267
268     def Execute(self):
269         # CHANGE STATE
270         if (self.Distance >= 0):
271             State.LeftMargin  = State.Waypoint_x1 - self.Margin
272             State.RightMargin = State.Waypoint_x1 + self.Margin
273             self.FSM.ToTransition("toxCorrection_Up")
274
275         else:
276             State.LeftMargin  = State.Waypoint_x1 + self.Margin
277             State.RightMargin = State.Waypoint_x1 - self.Margin
278             self.FSM.ToTransition("toxCorrection_Down")
279
280         print("Left: ",State.LeftMargin,"Right: ",State.RightMargin)
281
282     def Exit(self):
283         State.ChangeWaypointFlag = 0
284
285 # ------------------------------------------------------------------- #
286 # Class name: xCorrection_Up                                           #
287 # Description: calculates the correction angle of the servomotor when the #
288 # car makes the circuit clockwise direction                            #
289 # Parameters:                                                          #
290 # Result:                                                              #
291 # ------------------------------------------------------------------- #
292 class xCorrection_Up(State,PublishingMSG):
293     def __init__(self,FSM):
294         State.__init__(self,FSM)
295         PublishingMSG.__init__(self)
296
297     def Enter(self):
298         print("Correcting car trayectory (UP)")
299         print("I'm in xCorrection_UP")
300
301     def Execute(self):
302         if CarNavigation.Current_x < State.LeftMargin:
303             print("Turn Right")
304             PublishingMSG.Angle = int(-50*CarNavigation.Current_x)
305
306         elif CarNavigation.Current_x > State.RightMargin:
307             print("Turn Left")
308             PublishingMSG.Angle = int(50*CarNavigation.Current_x)
309
310         else:
311             print ("Go straight")
312             PublishingMSG.Angle = 90
```

```
313
314         # Publishing messages
315         PublishingMSG.publish_messages(self)
316
317         # CHANGE STATE
318         self.FSM.ToTransition("toCalDistance")
319
320     def Exit(self):
321         State.PrevState = "xCorrection_Up"
322
323 # ----------------------------------------------------------------------- #
324 # Class name: xCorrection_Down                                            #
325 # Description: calculates the correction angle of the servomotor when the #
326 # car makes the circuit anticlockwise direction                           #
327 # Parameters:                                                             #
328 # Result:                                                                 #
329 # ----------------------------------------------------------------------- #
330 class xCorrection_Down(State,PublishingMSG):
331     def __init__(self,FSM):
332         State.__init__(self,FSM)
333         PublishingMSG.__init__(self)
334
335     def Enter(self):
336         print("I'm in xCorrection_DOWN")
337         print("Correcting car trayectory (DOWN)")
338
339     def Execute(self):
340         if CarNavigation.Current_x > State.LeftMargin:
341             print("Turn Right")
342             PublishingMSG.Angle =  int(-50*CarNavigation.Current_x)# Min
angle
343
344         elif CarNavigation.Current_x < State.RightMargin:
345             print("Turn Left")
346             PublishingMSG.Angle = int(50*CarNavigation.Current_x)# Min
angle
347
348         else:
349             print ("Go straight")
350             PublishingMSG.Angle = 90
351
352         # Publishing messages
353         PublishingMSG.publish_messages(self)
354
355         # CHANGE STATE
356         self.FSM.ToTransition("toCalDistance")
357
358     def Exit(self):
359         State.PrevState = "xCorrection_Down"
360
361 # ----------------------------------------------------------------------- #
362 # Class name: yRange                                                      #
363 # Description: calculates the allowed deviation margins of the car when   #
364 # it goes perpendicular to the x-axis                                     #
365 # Parameters:                                                             #
366 # Result:                                                                 #
367 # ----------------------------------------------------------------------- #
368 class yRange(State):
369
370     def __init__(self,FSM):
371         State.__init__(self,FSM)
372         self.Distance = 0.0
373
374
```

```
375     def Enter(self):
376         print("I am in y Range")
377         State.EnterMargin(self)
378         self.Distance = State.Waypoint_x1 - State.Waypoint_x0
379
380         print("Waypoints Distance = ", self.Distance)
381
382     def Execute(self):
383         # CHANGE STATE
384         if (self.Distance >= 0):
385             State.LeftMargin  = State.Waypoint_y1 + self.Margin
386             State.RightMargin = State.Waypoint_y1 - self.Margin
387             self.FSM.ToTransition("toyCorrection_Up")
388
389         else:
390             State.LeftMargin  = State.Waypoint_y1 - self.Margin
391             State.RightMargin = State.Waypoint_y1 + self.Margin
392             self.FSM.ToTransition("toyCorrection_Down")
393
394         print("Left: ",State.LeftMargin,"Right: ",State.RightMargin)
395
396     def Exit(self):
397         State.ChangeWaypointFlag = 0
398
399 # ----------------------------------------------------------------------- #
400 # Class name: yCorrection_Up                                              #
401 # Description: calculates the correction angle of the servomotor when the #
402 # car makes the circuit clockwise direction                               #
403 # Parameters:                                                             #
404 # Result:                                                                 #
405 # ----------------------------------------------------------------------- #
406 class yCorrection_Up(State,PublishingMSG):
407
408     def __init__(self,FSM):
409         State.__init__(self,FSM)
410         PublishingMSG.__init__(self)
411
412     def Enter(self):
413         print("I am in y Correction uP")
414         print("Correcting car trayectory (UP)")
415
416     def Execute(self):
417         if CarNavigation.Current_y > State.LeftMargin:
418             print("Turn Right")
419             PublishingMSG.Angle = int(-50*CarNavigation.Current_y)# Min
    angle
420
421         elif CarNavigation.Current_y < State.RightMargin:
422             print("Turn Left")
423             PublishingMSG.Angle = int(50*CarNavigation.Current_y) # Min
    angle
424
425         else:
426             print ("Go straight")
427             PublishingMSG.Angle = 90
428
429         # Publishing messages
430         PublishingMSG.publish_messages(self)
431
432         # CHANGE STATE
433         self.FSM.ToTransition("toCalDistance")
434
435     def Exit(self):
436         State.PrevState = "yCorrection_Up"
```

```
437 # ------------------------------------------------------------------------ #
438 # Class name: yCorrection_Down                                            #
439 # Description: calculates the correction angle of the servomotor when the #
440 # car makes the circuit anticlockwise direction                           #
441 # Parameters:                                                             #
442 # Result:                                                                 #
443 # ------------------------------------------------------------------------ #
444 class yCorrection_Down(State,PublishingMSG):
445 
446     def __init__(self,FSM):
447         State.__init__(self,FSM)
448         PublishingMSG.__init__(self)
449 
450     def Enter(self):
451         print("I am in y Correction DOWN")
452         print("Correcting car trayectory (DOWN)")
453 
454     def Execute(self):
455         if CarNavigation.Current_y < State.LeftMargin:
456             print("Turn Right")
457             PublishingMSG.Angle = int(-50*CarNavigation.Current_y) # Min
angle
458 
459         elif CarNavigation.Current_y > State.RightMargin:
460             print("Turn Left")
461             PublishingMSG.Angle = int(50*CarNavigation.Current_y)# Min
angle
462 
463         else:
464             print ("Go straight")
465             PublishingMSG.Angle = 90
466 
467         # Publishing messages
468         PublishingMSG.publish_messages(self)
469 
470         # CHANGE STATE
471         self.FSM.ToTransition("toCalDistance")
472 
473     def Exit(self):
474         State.PrevState = "yCorrection_Down"
475 
476 # ------------------------------------------------------------------------ #
477 # Class name: bRange                                                      #
478 # Description: calculates the allowed deviation margins of the car when   #
479 # it goes along a straight with a certain inclination                     #
480 # Parameters:                                                             #
481 # Result:                                                                 #
482 # ------------------------------------------------------------------------ #
483 class bRange(State):
484 
485     def __init__(self,FSM):
486         State.__init__(self,FSM)
487 
488     def Enter(self):
489         print("I am in b Range")
490 
491         State.EnterMargin(self)
492 
493         self.Distance = State.Waypoint_y1 - State.Waypoint_y0
494         print("Waypoints Distance = ", self.Distance)
495 
496     def Execute(self):
497         # CHANGE STATE
```

```
498         if ((State.EqLn_Slope > 0 and self.Distance > 0) or
    (State.EqLn_Slope < 0 and self.Distance < 0)):
499             State.LeftMargin  = State.EqLn_IndTerm + self.Margin
500             State.RightMargin = State.EqLn_IndTerm - self.Margin
501             self.FSM.ToTransition("tobCorrection_Up")
502
503         else:
504             State.LeftMargin  = State.EqLn_IndTerm - self.Margin
505             State.RightMargin = State.EqLn_IndTerm + self.Margin
506             self.FSM.ToTransition("tobCorrection_Down")
507
508     def Exit(self):
509         State.ChangeWaypointFlag = 0
510
511 # ------------------------------------------------------------------------ #
512 # Class name: bCorrection_Up                                               #
513 # Description: calculates the correction angle of the servomotor when the #
514 # car makes the circuit clockwise direction                                #
515 # Parameters:                                                              #
516 # Result:                                                                  #
517 # ------------------------------------------------------------------------ #
518 class bCorrection_Up(State,PublishingMSG):
519
520     def __init__(self,FSM):
521         State.__init__(self,FSM)
522         PublishingMSG.__init__(self)
523
524     def Enter(self):
525         print("I am in b Correction UP")
526         print("Correcting car trayectory (UP)")
527
528
529     def Execute(self):
530         if ((CarNavigation.Current_y - (State.EqLn_Slope *
    CarNavigation.Current_x)) > State.LeftMargin):
531             print("Turn Right")
532             PublishingMSG.Angle = int(-50*(CarNavigation.Current_y -
    (State.EqLn_Slope * CarNavigation.Current_x))) # Min angle
533
534         elif ((CarNavigation.Current_y - (State.EqLn_Slope *
    CarNavigation.Current_x)) < State.RightMargin):
535             print("Turn Left")
536             PublishingMSG.Angle = int(50*(CarNavigation.Current_y -
    (State.EqLn_Slope * CarNavigation.Current_x))) # Min angle
537
538         else:
539             print ("Go straight")
540             PublishingMSG.Angle = 90
541
542         # Publishing messages
543         PublishingMSG.publish_messages(self)
544
545         # CHANGE STATE
546         self.FSM.ToTransition("toCalDistance")
547
548     def Exit(self):
549         State.PrevState = "bCorrection_Up"
550
551
552
553
554
555
556
```

```
557 # ------------------------------------------------------------------- #
558 # Class name: bCorrection_Down                                        #
559 # Description: calculates the correction angle of the servomotor when the #
560 # car makes the circuit anticlockwise direction                       #
561 # Parameters:                                                         #
562 # Result:                                                             #
563 # ------------------------------------------------------------------- #
564 class bCorrection_Down(State,PublishingMSG):
565 
566     def __init__(self,FSM):
567         State.__init__(self,FSM)
568         PublishingMSG.__init__(self)
569 
570     def Enter(self):
571         print("I am in b Correction DOWN")
572         print("Correcting car trayectory (DOWN)")
573 
574     def Execute(self):
575         if ((CarNavigation.Current_y - (State.EqLn_Slope *
    CarNavigation.Current_x)) < State.LeftMargin):
576             print("Turn Right")
577             PublishingMSG.Angle = int(-50*(CarNavigation.Current_y -
    (State.EqLn_Slope * CarNavigation.Current_x))) # Min angle
578 
579         elif ((CarNavigation.Current_y - (State.EqLn_Slope *
    CarNavigation.Current_x)) > State.RightMargin):
580             print("Turn Left")
581             PublishingMSG.Angle = int(50*(CarNavigation.Current_y -
    (State.EqLn_Slope * CarNavigation.Current_x))) # Min angle
582 
583         else:
584             print ("Go straight")
585             PublishingMSG.Angle = 90
586 
587         # Publishing messages
588         PublishingMSG.publish_messages(self)
589 
590         # CHANGE STATE
591         self.FSM.ToTransition("toCalDistance")
592 
593     def Exit(self):
594         State.PrevState = "bCorrection_Down"
595 
596 # ------------------------------------------------------------------- #
597 # Class name: CalDistance                                             #
598 # Description: Determine if there has been a trajectory change        #
599 # Parameters:                                                         #
600 # Result:                                                             #
601 # ------------------------------------------------------------------- #
602 class CalDistance(State,PublishingMSG):
603     def __init__(self,FSM):
604         State.__init__(self,FSM)
605         PublishingMSG.__init__(self)
606 
607     def Enter(self):
608         print("Calculating distance between waypoints")
609         State.EneterDistMargin(self)
610 
611         if ((State.PrevState == "xCorrection_Up") or (State.PrevState ==
    "xCorrection_Down")):
612             self.Distance = abs(State.Waypoint_y1 - State.Waypoint_y0)
613             self.CurDistance = abs(CarNavigation.Current_y -
    State.Waypoint_y0)
614 
```

```
615         elif ((State.PrevState == "yCorrection_Up") or (State.PrevState ==
    "yCorrection_Down")):
616             self.Distance = abs(State.Waypoint_x1 - State.Waypoint_x0)
617             self.CurDistance = abs(CarNavigation.Current_x -
    State.Waypoint_x0)
618
619         else:
620             self.Distance = math.sqrt((State.Waypoint_x1 -
    State.Waypoint_x0) ** 2 + (State.Waypoint_y1 - State.Waypoint_y0) ** 2)
621             self.CurDistance = math.sqrt((CarNavigation.Current_x -
    State.Waypoint_x0) ** 2 + (CarNavigation.Current_y - State.Waypoint_y0) **
    2)
622
623         print("Current_x: ", CarNavigation.Current_x)
624         print("Current_y: ", CarNavigation.Current_y)
625         print("Distance between waypoints = ", self.Distance)
626         print("CurrentDistance = ", self.CurDistance)
627
628     def Execute(self):
629         # CHANGE STATE
630         if ((self.CurDistance >= (self.Distance - self.DistMargin)) and
    (State.StopFlag == 0)):
631             State.ChangeWaypointFlag = 1
632             print("ChangeFlag: ", State.ChangeWaypointFlag)
633             self.FSM.ToTransition("toCarTrayectory")
634
635         elif ((self.CurDistance < (self.Distance - self.DistMargin)) and
    (State.PrevState == "xCorrection_Up")):
636             self.FSM.ToTransition("toxCorrection_Up")
637
638         elif ((self.CurDistance < (self.Distance - self.DistMargin)) and
    (State.PrevState == "xCorrection_Down")):
639             self.FSM.ToTransition("toxCorrection_Down")
640
641         elif ((self.CurDistance < (self.Distance - self.DistMargin)) and
    (State.PrevState == "yCorrection_Up")):
642             self.FSM.ToTransition("toyCorrection_Up")
643
644         elif ((self.CurDistance < (self.Distance - self.DistMargin)) and
    (State.PrevState == "yCorrection_Down")):
645             self.FSM.ToTransition("toyCorrection_Down")
646
647         elif ((self.CurDistance < (self.Distance - self.DistMargin)) and
    (State.PrevState == "bCorrection_Up")):
648             self.FSM.ToTransition("tobCorrection_Up")
649
650         elif ((self.CurDistance < (self.Distance - self.DistMargin)) and
    (State.PrevState == "bCorrection_Down")):
651             self.FSM.ToTransition("tobCorrection_Down")
652
653         elif((self.CurDistance >= (self.Distance - self.DistMargin)) and
    (State.StopFlag == 1)):
654             print("FSM Shut Down")
655             print("Car stoped")
656             PublishingMSG.Angle = 0
657             PublishingMSG.publish_messages(self)
658
659     def Exit(self):
660         pass
661
662
663
664
665
```

```
666 # ------------------------ FINITE STATE MACHINES ------------------------ #
667 # ----------------------------------------------------------------------- #
668 # Class name: FSM                                                         #
669 # Description: implementation of the state machine                        #
670 # Parameters:                                                             #
671 # Result:                                                                 #
672 # ----------------------------------------------------------------------- #
673 class FSM(object):
674     def __init__(self,character):
675         PublishingMSG.__init__(self)
676         self.char = character
677         self.states = {}
678         self.transitions = {}
679         self.curState = None
680         self.prevState = None
681         self.trans = None
682 
683         self.flagStart = 1
684         self.i = 0
685     def AddTransition(self,transName,transition):
686         self.transitions[transName] = transition
687 
688     def AddState(self,stateName,state):
689         self.states[stateName] = state
690 
691     def SetState(self,stateName):
692         self.prevState = self.curState
693         self.curState = self.states[stateName]
694 
695     def ToTransition(self,toTrans):
696         self.trans = self.transitions[toTrans]
697 
698     def ExecuteAndChangeState(self):
699         if(self.trans):
700             self.curState.Exit()
701             #self.trans.Execute()
702             self.SetState(self.trans.toState)
703             self.curState.Enter()
704             self.curState.Execute()
705             self.trans = None
706 
707         self.curState.Execute()
708 
709     def Execute(self):
710         if self.flagStart == 1:
711             self.flagStart = 0
712             self.curState.Enter()
713             PublishingMSG.publish_messages(self)
714             self.curState.Execute()
715             while (self.i < 3):
716                 self.ExecuteAndChangeState()
717 
718                 # Program reset when the square button has been pressed
719                 if ((CarNavigation.Flag_Reset == 1) and
    (CarNavigation.Operating_Mode == "automatic")):
720                     CarNavigation.Flag_Reset = 0
721                     self.flagStart = 1
722                     i = 4
723 
724                 self.i += 1
725             self.i = 0
726 
727         elif (State.ChangeWaypointFlag == 0):
728             while(self.i<2):
```

```
729                 self.ExecuteAndChangeState()
730
731                 # Program reset when the square button has been pressed
732                 if ((CarNavigation.Flag_Reset == 1) and
    (CarNavigation.Operating_Mode == "automatic")):
733                     CarNavigation.Flag_Reset = 0
734                     self.flagStart = 1
735                     i = 4
736
737                 self.i += 1
738             self.i = 0
739
740         elif (State.ChangeWaypointFlag == 1):
741             while(self.i<3):
742                 self.ExecuteAndChangeState()
743
744                 # Program reset when the square button has been pressed
745                 if ((CarNavigation.Flag_Reset == 1) and
    (CarNavigation.Operating_Mode == "automatic")):
746                     CarNavigation.Flag_Reset = 0
747                     self.flagStart = 1
748                     State.ChangeWaypointFlag = 0
749                     i = 4
750
751                 self.i += 1
752             self.i = 0
753
754
755 # --------------------------- IMPLEMENTATION ---------------------------- #
756 # ----------------------------------------------------------------------- #
757 # Class name: CarNavigation                                               #
758 # Description: declaration of states and transitions and initialization   #
759 # of ros                                                                  #
760 # Parameters:                                                             #
761 # Result:                                                                 #
762 # ----------------------------------------------------------------------- #
763 class CarNavigation():
764     # GLOBAL VARIABLES
765     Current_x = 0.0
766     Current_y = 0.0
767     Flag_Reset = 0
768     Operating_Mode = "manual"
769     Laps = 0
770
771     def __init__(self, laps):
772
773         self.Flag_Start = 1
774         CarNavigation.Laps = laps
775
776         # --------------------- FSM Initialization ---------------------- #
777
778         self.FSM = FSM(self)
779          # STATES
780         self.FSM.AddState("CarTrayectory",CarTrayectory(self.FSM))
781         self.FSM.AddState("xRange",xRange(self.FSM))
782         self.FSM.AddState("xCorrection_Up",xCorrection_Up(self.FSM))
783         self.FSM.AddState("xCorrection_Down",xCorrection_Down(self.FSM))
784         self.FSM.AddState("yRange",yRange(self.FSM))
785         self.FSM.AddState("yCorrection_Up",yCorrection_Up(self.FSM))
786         self.FSM.AddState("yCorrection_Down",yCorrection_Down(self.FSM))
787         self.FSM.AddState("bRange",bRange(self.FSM))
788         self.FSM.AddState("bCorrection_Up",bCorrection_Up(self.FSM))
789         self.FSM.AddState("bCorrection_Down",bCorrection_Down(self.FSM))
790         self.FSM.AddState("CalDistance",CalDistance(self.FSM))
```

```
791         # TRANSITIONS
        self.FSM.AddTransition("toCarTrayectory",Transition("CarTrayectory"))
792     self.FSM.AddTransition("toxRange",Transition("xRange"))
793     self.FSM.AddTransition("toxCorrection_Up",Transition("xCorrection_Up"))
794 self.FSM.AddTransition("toxCorrection_Down",Transition("xCorrection_Down"))
795     self.FSM.AddTransition("toyRange",Transition("yRange"))
796     self.FSM.AddTransition("toyCorrection_Up",Transition("yCorrection_Up"))
797 self.FSM.AddTransition("toyCorrection_Down",Transition("yCorrection_Down"))
798     self.FSM.AddTransition("tobRange",Transition("bRange"))
799     self.FSM.AddTransition("tobCorrection_Up",Transition("bCorrection_Up"))
800 self.FSM.AddTransition("tobCorrection_Down",Transition("bCorrection_Down"))
801     self.FSM.AddTransition("toCalDistance",Transition("CalDistance"))
802
803         # FISRT STATE
804         self.FSM.SetState("CarTrayectory")
805
806         # --------------------- ROS Initialization ---------------------- #
807        rospy.init_node('autonomous_control', anonymous=True)
808         rospy.Subscriber('hedge_pos_ang', hedge_pos_ang,
    self.car_coordinates,queue_size=1,buff_size=2**24)
809         rospy.Subscriber("controller_state", controller_state,
    self.ps4_controller,queue_size=1,buff_size=2**24)
810         rospy.loginfo("Publisher Node Started, now publishing messages")
811
812     # --------------------- UPDATE CAR COORDINATES ---------------------- #
813     # ------------------------------------------------------------------ #
814     # Name: CarCoordinates                                                #
815     # Description: save the current coordinates of the car               #
816     # Parameters: - data: data received through the hedge_pos_ang topic   #
817     # Result:                                                             #
818     # ------------------------------------------------------------------ #
819     def car_coordinates(self,data):
820         CarNavigation.Current_y = data.y_m
821         CarNavigation.Current_x = data.x_m
822
823     # ------------------------------------------------------------------ #
824     # Name: ps4_controller                                                #
825     # Description: save the current operating mode and the square flag    #
826     # Parameters: - data: data received through the controller_state topic#
827     # Result:                                                             #
828     # ------------------------------------------------------------------ #
829     def ps4_controller(self,data):
830         CarNavigation.Operating_Mode = data.operating_mode
831
832         # These statements prevent the square button bounce
833         if (self.flag_previous_square_press != data.flag_square_press):
834             self.flag_previous_square_press = data.flag_square_press
835             CarNavigation.Flag_Reset = 1
836
837 #--------------#
838 # MAIN PROGRAM #
839 #--------------#
840 if __name__ == '__main__':
841     Car = CarNavigation(1) # Number of laps
842     rate = rospy.Rate(100) # 100 Hz
843     sleep(10)
844     try:
845         while (not rospy.is_shutdown()):
846             #rospy.spin()
847             Car.FSM.Execute()
848             rate.sleep()
849             # Execute FSM
850     except rospy.ROSInitException:
851         pass
```

B6. Nodo car_control

```
1   #!/usr/bin/env python
2
3   #-----------#
4   # LIBRARIES #
5   #-----------#
6   # Ros libraries
7   import rospy
8   from car.msg import controller_state
9   from car.msg import autonomous_parameters
10  from car.msg import curve_parameters
11
12  # GPIO library
13  import RPi.GPIO as GPIO
14  import pigpio
15
16  # Time library
17  from time import sleep
18
19  # pigpio library initializations
20  PI = pigpio.pi()
21
22  #------------------#
23  # CLASS DEFINITION #
24  #------------------#
25  class Car_Control():
26
27      # -------------------------------------------------------------------- #
28      # Name: __init__                                                       #
29      # Description: Servo and ESC pins assignment and car controller node   #
30      # initialization                                                       #
31      # Parameters: - ESC:   Control speed and direction of the motor        #
32      #             - SERVO: Servo control pin                               #
33      # Result:                                                              #
34      # -------------------------------------------------------------------- #
35      def __init__(self,ESC,SERVO):
36          # GPIO
37          self.ESC = ESC
38          self.SERVO = SERVO
39
40          # Controller variables
41          self.operating_mode = "manual"
42          self.Flag_Emergency_Stop = 0# Flag to indicate the emergency stop
43
44          # Controller and Autonomous control variables
45          self.Direction = "stop"
46          self.Movement = "straight"
47          self.Angle = 90  # Servo middle position
48          self.Speed = 16  # Minimum speed
49
50          # Autonomous angles
51          self.Trajectory_Angle = 90 # Servo middle position
52          self.Angle_Correction = 0  # Car shunt
53
54          # Servo and motor initialization
55          self.setup_servo_motor()
56          print("Car Setup Done")
57
58
59
60
61
```

```
62          # car_control node initialitzation
63          rospy.init_node('car_control', anonymous=True)
64          rospy.Subscriber('controller_state', controller_state,
    self.manual_control)
65          rospy.Subscriber('autonomous_parameters', autonomous_parameters,
    self.autonomous_control,queue_size=1,buff_size=2**24)
66
    rospy.Subscriber('curve_parameters',curve_parameters,self.curve_detector,qu
    eue_size=1,buff_size=2**24)
67          rospy.loginfo("Car Control Node Started")
68          rospy.on_shutdown(self.shutdown)
69
70      # ------------------------------------------------------------------ #
71      # Name: setup_servo_motor                                            #
72      # Description: Servo and ESC pins and PWM initialization             #
73      # Parameters:                                                        #
74      # Result:                                                            #
75      # ------------------------------------------------------------------ #
76      def setup_servo_motor(self):
77          GPIO.setmode(GPIO.BCM)  # Enable pins
78          GPIO.setwarnings(False) # Disable warnings
79
80          # Initialization ESC and Servo GPIO
81          PI.set_mode(self.ESC, pigpio.OUTPUT)
82          PI.set_mode(self.SERVO, pigpio.OUTPUT)
83
84          # Set frequency
85          PI.set_PWM_frequency(self.ESC, 50)     # ESC 50 Hz
86          PI.set_PWM_frequency(self.SERVO, 300) # SERVO 300 Hz
87
88          # Start PWM
89          PI.set_servo_pulsewidth(self.ESC, 1500)   # 1,5 ms - Motor brake
90          PI.set_servo_pulsewidth(self.SERVO, 1500) # 1,5 ms - Servo 90°
91
92      # ------------------------------------------------------------------ #
93      # Name: car_control                                                  #
94      # Description: control the servo and the ESC depend on the data      #
95      # received from the manual or automatic control                      #
96      # Parameters:                                                        #
97      # Result:                                                            #
98      # ------------------------------------------------------------------ #
99      def car_control(self):
100
101         # Car Movement
102         if (self.Flag_Emergency_Stop == 0):
103
104             # Motor saturation. Speed - 0 to 100 %
105             if (self.Speed > 100):
106                 self.Speed = 100
107             elif (self.Speed < 10):
108                 self.Speed = 10
109
110             # Sum the curve angle with the angle correction
111             self.Angle = self.Trayectory_Angle + self.Angle_Correction
112
113             # Servo saturation. Angle - 45 to 135°
114             if (self.Angle > 135):  # Max left
115                 self.Angle = 135
116             elif (self.Angle < 45): # Max right
117                 self.Angle = 45
118
119             # Determinate the direction of the car
120             if (self.Angle == 90):
121                 self.Direction = "straight"
```

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola d'Enginyeria de Barcelona Est

```
122                elif (self.Angle > 90):
123                    self.Direction = "left"
124                elif (self.Angle < 90):
125                    self.Direction = "right"
126
127                # Movement: forward, back or stop
128                if (self.Movement == "forward"):
129                    self.MoveF(self.Speed)
130
131                elif (self.Movement == "back"):
132                    self.MoveB(self.Speed)
133
134                elif (self.Movement == "stop"):
135                    self.stop()
136
137                # Direction: right,left or straight
138                if ((self.Direction == "right") or (self.Direction == "left")):
139                    self.changeDirection(self.Angle)
140
141                elif (self.Direction == "straight"):
142                    self.goStraight()
143
144                # Shows the status of the car through the terminal
145                print("Operting Mode: ", self.Operating_Mode)
146                print("Car status: ", self.Movement,", ", self.Direction)
147                print("Speed: ", self.Speed, ", Angle: ", self.Angle)
148                print()
149
150            # Stops the car when the emergency PS4 button is pressed
151            elif (self.Flag_Emergency_Stop == 1):
152                self.stop()
153                self.goStraight()
154                print("Emergency Stop Done")
155
156        # ------------------------------------------------------------------ #
157        # Name: manual_control                                               #
158        # Description: car control through PS4 controller                    #
159        # Parameters: - data: data received through the controller_state topic#
160        # Result:                                                            #
161        # ------------------------------------------------------------------ #
162        def manual_control(self,data):
163
164            self.Operating_Mode = data.operating_mode
165            self.Flag_Emergency_Stop = data.flag_emergency_stop
166
167            if (self.Operating_Mode == "manual"):
168                self.Direction = data.direction
169                self.Movement = data.movement
170                self.Speed = data.speed
171                Self.TrayectoryAngle = data.angle
172                self.Angle_Correction = 0
173
174            self.car_control()
175
176
177
178
179
180
181
182
183
184
185
```

```
186     # ------------------------------------------------------------------ #
187     # Name: autonomous_control                                           #
188     # Description: Received data through the autonomous control node     #
189     # Parameters: - data: data received through the autonomous_parameters #
190     #               topic                                                #
191     # Result:                                                            #
192     # ------------------------------------------------------------------ #
193     def autonomous_control(self,data):
194        if (self.Operating_Mode == "automatic"):
195             self.Angle_Correction = data.angle
196             # Saturation of the correction angle
197             if (self.Angle_Correction > 5):
198                 self.Angle_Correction = 5
199             elif (((self.Angle_Correction < 2) and (self.Angle_Correction >
    -2)) or (self.Angle_Correction == 90)):
200                   self.Angle_Correction = 0
201             else:
202                 self.Angle_Correction = -5
203
204     # ------------------------------------------------------------------ #
205     # Name: curve_detector                                               #
206     # Description: Received data through the curve detector node         #
207     # Parameters: - data: data received through the curve_parameters topic#
208     # Result:                                                            #
209     # ------------------------------------------------------------------ #
210     def curve_detector(self,data):
211         if (self.Operating_Mode == "automatic"):
212             self.Movement = data.movement
213             self.Trayectory_Angle = data.angle
214             self.Speed = data.speed
215             self.Direction = data.direction
216             self.car_control()
217
218     # ------------------------------------------------------------------ #
219     # Name: moveF                                                        #
220     # Description: converts the speed into an appropriate duty cycle in  #
221     # order to move the car forward                                      #
222     # Parameters: - speed: 0 to 100 %                                    #
223     # Result:                                                            #
224     # ------------------------------------------------------------------ #
225     def moveF(self,speed):
226         # min speed neede to start moveming - 10%
227         # max speed - 100%
228         # speed range 100 - 10 = 90
229         # min duty neede to start moveming - 1600 us
230         # max duty - 1900 us
231         duty = ((1900-1600)*(speed-10))/90 + 1600 # duty in us
232         PI.set_servo_pulsewidth(self.ESC,duty)     # set the ESC PWM duty
233
234     # ------------------------------------------------------------------ #
235     # Name: moveB                                                        #
236     # Description: converts the speed into an appropriate duty cycle in  #
237     # order to move the car back                                         #
238     # Parameters: - speed: 0 to 100 %                                    #
239     # Result:                                                            #
240     # ------------------------------------------------------------------ #
241     def moveB(self,speed):
242         # min speed neede to start moveming - 10%
243         # max speed - 100%
244         # speed range 100 - 10 = 90
245         # min duty neede to start moveming - 1450 us
246         # max duty - 1200 us
247         duty = ((1450-1200)*(100-speed))/90 + 1200 # duty in us
248         PI.set_servo_pulsewidth(self.ESC,duty)     # set the ESC PWM duty
```

```
249
250     # --------------------------------------------------------------------- #
251     # Name: stop                                                            #
252     # Description: stop the car                                             #
253     # Parameters:                                                           #
254     # Result:                                                               #
255     # --------------------------------------------------------------------- #
256     def stop(self):
257         PI.set_servo_pulsewidth(self.ESC,1500) # 1500 us - stop
258
259     # --------------------------------------------------------------------- #
260     # Name: changeDirection                                                 #
261     # Description: convert the angle into an appropriate duty cycle in     #
262     # order to change the car direction                                     #
263     # Parameters: - angle: servo angle 0 to 180°                            #
264     # Result:                                                               #
265     # --------------------------------------------------------------------- #
266     def changeDirection(self,angle):
267         # min duty - 1000 us
268         # max duty - 2000 us
269         duty = ((2000-1000)*angle)/180 + 1000
270         PI.set_servo_pulsewidth(self.SERVO,duty)
271
272     # --------------------------------------------------------------------- #
273     # Name: goStraight                                                      #
274     # Description: the car go straight                                      #
275     # Parameters:                                                           #
276     # Result:                                                               #
277     # --------------------------------------------------------------------- #
278     def goStraight(self):
279         PI.set_servo_pulsewidth(self.SERVO,1500) # 1500 us - 90°
280
281     # --------------------------------------------------------------------- #
282     # Name: shutdown                                                        #
283     # Description: Stop the motor, put the servo in the middle position    #
284     # and clean GPIO pins in order to be able to use them after            #
285     # closing the program                                                   #
286     # Parameters:                                                           #
287     # Result:                                                               #
288     # --------------------------------------------------------------------- #
289     def shutdown(self):
290         print("Car Control Node Shutting Down")
291         self.stop()
292         self.goStraight()
293
294 #--------------#
295 # MAIN PROGRAM #
296 #--------------#
297 if __name__ == '__main__':
298     Car = Car_Control(13,19)
299
300     try:
301         while not rospy.is_shutdown():
302             rospy.spin() # Keeps the program running
303
304     except rospy.ROSInitException:
305         pass
```