Universidad de Valladolid

ESCUELA DE INGENIERÍAS INDUSTRIALES

Máster en Ingeniería Industrial

MASTER EN INGENIERÍA INDUSTRIAL

ESCUELA DE INGENIERÍAS INDUSTRIALES

UNIVERSIDAD DE VALLADOLID

TRABAJO FIN DE MÁSTER

Desarrollo de un vehículo autónomo a escala con capacidad de navegación y seguimiento de marcas viales

Autor: D. Fernando Eduardo Cuenca Rodríguez-Monsalve
Tutor: D. Eduardo Zalama Casanova

Valladolid, Septiembre de 2020

RESUMEN

En la actualidad, los fabricantes de automóviles desarrollan cada vez más sistemas que aumentan la capacidad de conducción autónoma del vehículo y reducen las acciones que debe realizar el conductor. El objetivo final de estos avances es lograr que los coches conduzcan sin la necesidad de que haya nadie dirigiéndolos.

Este proyecto tiene como objetivo poner en funcionamiento un modelo de coche a escala con los actuadores y sensores necesarios para conducir de forma autónoma. De esta forma la Escuela de Ingenierías Industriales dispondrá en sus laboratorios de vehículos que podrán ser utilizados para realizar pruebas y estudios en este campo. Además de la puesta en marcha de uno de estos coches, el proyecto también pretende ser una guía rápida para crear otros o para arreglarlos en caso de ser necesario.

Estos coches están dirigidos por una Odroid y un Arduino que utilizan el *framework* ROS para comunicarse con los sensores y actuadores. El lenguaje de programación de estos dispositivos es C++.

Palabras clave: coche autónomo, ROS, openCV, Arduino, C++

ABSTRACT

Currently, vehicle manufacturers are growing their investment in developing systems that increase de autonomous capacities of cars which reduces the amount of actions that drivers have to execute to perform their tasks. The final goal of this advances is to achieve driverless vehicles.

This project's goal is the creation of a scaled car with the actuators and sensors needed to drive in an autonomous way. In this way, the *Escuela de Ingenierías Industriales* will have in its laboratories vehicles that can be used to develop projects or realize studies in this field. This project also has the purpose of being an easy guide to create more cars or in the case there is the need of solving problems.

These cars are controlled with an Odroid and an Arduino that use ROS to communicate with the sensors and actuators. The programming language used for these devices is C++.

Keywords: autonomous car, ROS, OpenCV, Arduino, C++

AGRADECIMIENTOS

Quiero agradecer a mi familia y amigos por todo el apoyo que me han proporcionado a lo largo de estos años. Muchas gracias por haber estado ahí siempre que os he necesitado.

También quiero agradecer a Eduardo Zalama por la oportunidad de realizar este proyecto y por la ayuda que me ha proporcionado a lo largo de éste.

ÍNDICE

FIGURAS

1) INTRODUCCIÓN Y OBJETIVOS

1.1. EL COCHE AUTÓNOMO

1.1.1. ¿POR QUÉ DESARROLLAR EL COCHE AUTÓNOMO?

Los coches autónomos son el futuro de la conducción. Estos vehículos van a permitir a sus usuarios utilizar el tiempo de conducción en realizar otras actividades como pueden ser leer, trabajar o dormir. Pero la utilidad de esta tecnología no se limita a la optimización del tiempo, también va a permitir reducir los accidentes de tráfico. Hay que tener en cuenta que el 90% de estos accidentes son debidos al factor humano [1] el cual desaparecerá con estos coches. Además, el coche autónomo tendrá una labor social al permitir el acceso a la conducción a personas que actualmente no pueden hacerlo (personas con movilidad reducida, con problemas de visión ...).

1.1.2. DESARROLLO DEL COCHE AUTÓNOMO

El nivel de desarrollo del coche autónomo se puede dividir en 6 niveles [2]:

Nivel 0: Coches que no disponen de ninguna ayuda electrónica para la conducción.

Nivel 1: Coches con automatismos como la alerta de salida de carril. Permiten al conductor una conducción más cómoda.

Nivel 2: Coche con control de movimiento tanto longitudinal como lateral. No cuenta con detección y respuesta frente a objetos de forma completa

Nivel 3: Gran nivel de autonomía, es decir, el coche puede decidir cuándo cambiar de carril, adelantar, etc. Aun así, el usuario tiene que estar preparado para intervenir si el sistema lo solicita.

Nivel 4: No es necesario que el usuario esté preparado para intervenir. Sin embargo, en ciertas condiciones el coche no podrá funcionar de forma autónoma y el usuario tendrá que conducirlo de forma "manual".

Nivel 5: Automatización plena, lo que implica un vehículo sin pedales ni volante.

Los coches comerciales más avanzados a día de hoy se encuentran en el nivel 3 aunque ciertas compañías como *Tesla* afirman que podrían vender coches con un nivel 4 de autonomía antes del final de 2020.

1.2. CONTEXTO DEL PROYECTO

Como se ha mencionado en el apartado anterior, el coche autónomo es una de las tecnologías más importantes que se están desarrollando actualmente en el mundo de la automoción. Es por ello por lo que la Escuela de Ingenierías Industriales de la Universidad de Valladolid, que tiene un fuerte vínculo con el campo del automóvil (Renault, Michelin ...), desea aumentar sus investigaciones en este campo y por lo que en 2017 participó en la competición del "*SEAT autonomous driving challenge*" en la cual el equipo de AMUVAG (asociación de robótica de la UVa) logró el primer puesto [3].

En esta competición el equipo de la universidad de Valladolid tuvo que programar un coche a escala 1:10 para que condujese de forma autónoma en tres pruebas distintas de dificultad ascendente:

- Primera prueba: Conducir en una carretera sin otros vehículos u obstáculos y sin tener en cuenta las señales de tráfico.
- Segunda prueba: Conducir en una carretera con otros obstáculos y sin tener en cuenta las señales de tráfico.
- Tercera prueba: Conducir en una carretera con otros obstáculos, teniendo en cuenta las señales de tráfico y teniendo que realizar una maniobra de aparcamiento.

Para la realización de estas pruebas, la empresa organizadora entregó a cada equipo participante un prototipo de coche autónomo para que lo programasen y realizasen las pruebas. Al acabar este evento, los equipos tuvieron que devolver el prototipo con el que habían participado.

1.3. OBJETIVOS DEL PROYECTO

El objetivo de este proyecto es el de desarrollar un prototipo propio de coche autónomo que permitirá un mayor conocimiento de su estructura, programación y funcionamiento, lo que redundará en la participación de futuras competiciones.

El diseño y programación del coche está inspirado en los "*AutoNOMOS Model cars*" desarrollados por *Freie Universität Berlin* para fines educativos y de cuya página web [4] se ha descargado el sistema operativo del ordenador que va a bordo del vehículo.

Este trabajo pretende ser también una guía rápida y sencilla en caso de querer montar más modelos como este.

Para lograr esto se ha desglosado el proyecto en los siguientes subobjetivos:

- **Diseño hardware del vehículo:**

  Este subobjetivo busca identificar los diferentes componentes necesarios para construir un coche autónomo, las adaptaciones físicas que se deben realizar para obtener el comportamiento deseado y cómo deben conectarse entre ellos.

- **Aprendizaje y manejo de herramientas para la programación. Programación ROS:**

  Con este objetivo se pretende aprender el funcionamiento de los programas y herramientas software para trabajar en el coche autónomo. Se hace especial inciso en ROS (Robotic Operating System) al ser la herramienta con la que más se va a trabajar.

- **Control de alto nivel Odroid. Instalación y programación de módulos ROS:**

  Con este objetivo se desea realizar una guía que muestre los pasos a realizar para instalar en el ordenador personal y en la Odroid (ordenador que va montado en el coche) los programas necesarios para el desarrollo del proyecto. Además, se quiere explicar también los pasos a realizar para verificar que los sensores utilizados son compatibles con las bibliotecas ROS y que funcionan de forma correcta.

- **Control de bajo nivel Arduino:**

  El objetivo es programar el microcontrolador Arduino (el cual se comunica con la Odroid y es el encargado de accionar los actuadores) y realizar la calibración de los actuadores.

- **Resultados experimentales**

  Si se completan todos los subobjetivos anteriores entonces todos los componentes funcionan correctamente. Por tanto el siguiente objetivo es comprobar la idoneidad del trabajo para realizar un coche autónomo. Para ello se pretende realizar la programación de algunas de las maniobras del "*SEAT autonomous driving challenge*" y estudiar los resultados obtenidos.

1.4. CONTENIDO DE LA MEMORIA

Para la realización de esta memoria se ha utilizado una estructura que corresponde con la de los subobjetivos. El esquema utilizado es el siguiente (excluyendo el primer capítulo):

- **Capítulo segundo**

  En este capítulo se especifican todos los dispositivos que se han utilizado para construir el coche, así como la forma en la que se han conectado y acondicionado para su correcto funcionamiento.

- **Capítulo tercero**

  Se realiza una breve descripción del software utilizado para el proyecto: C++, Arduino y Linux. Se hace una descripción más exhaustiva de la principal herramienta utilizada en el trabajo: ROS.

- **Capítulo cuarto**

  Este capítulo es una guía en la cual se describe paso a paso los comandos a introducir en el ordenador y la Odroid para realizar lo siguiente:

  - Instalar ROS en el ordenador.
  - Instalar el sistema operativo de la Odroid.
  - Comprobar el funcionamiento del lidar.
  - Comprobar el funcionamiento de la cámara.

- **Capítulo quinto**

  Breve guía de como instalar el software necesario para trabajar con el Arduino. Explicación de las diferentes funciones utilizadas en el código del Arduino y calibración de los componentes conectados a éste.

- **Capítulo sexto**

  Estudio de los programas de conducción autónoma y explicación de los ensayos realizados para su verificación.

- **Capítulo séptimo**

  Resumen final de los resultados del proyecto y planteamiento de posibles futuras líneas de desarrollo.

- **Capítulo octavo**

  Bibliografía con las fuentes consultadas para realizar este proyecto.

- **Capítulo noveno**

  Anexos que incluyen el código de los programas utilizados para las pruebas de conducción autónoma.

2) DISEÑO DEL HARDWARE DEL VEHÍCULO

El primer paso para poder comenzar el desarrollo del coche autónomo ha sido escoger el hardware necesario para su funcionamiento. Para realizar estas decisiones se han tomado como referencia los componentes de los vehículos utilizados en el "*SEAT autonomous driving challenge*", especificados en la página web de la Universidad de Berlín [4]. Después se ha preguntado a los alumnos que participaron en la competición cuales habían utilizado y cuáles no.

2.1. LISTA DE COMPONENTES

2.1.1. COCHE DE RADIOCONTROL

Se ha utilizado un coche de radiocontrol a escala 1:10 como base del vehículo y se han retirado la carrocería y los dispositivos que no eran necesarios. Se ha conservado la base del coche sobre la que se apoyan el resto de componentes, el parachoques para protegerlo en caso de colisión, la suspensión, las ruedas y el sistema de correas y engranajes que transmite el giro del motor a las ruedas.

Sobre este armazón se ha colocado una plancha rectangular de plástico rígido transparente sobre el cual se han montado el resto de dispositivos, a excepción del motor y el servomotor que están montados en la propia base del coche.

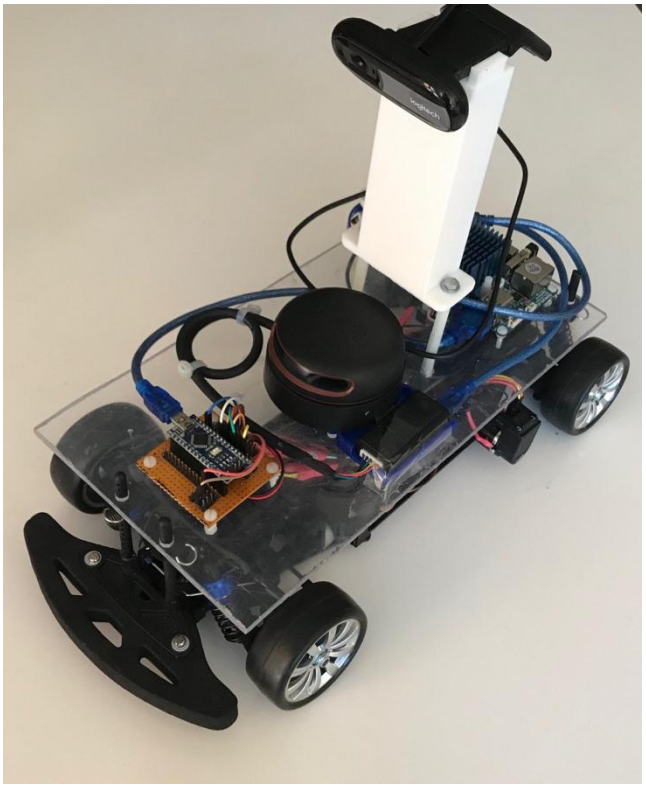

Ilustración 1: Vista superior del coche

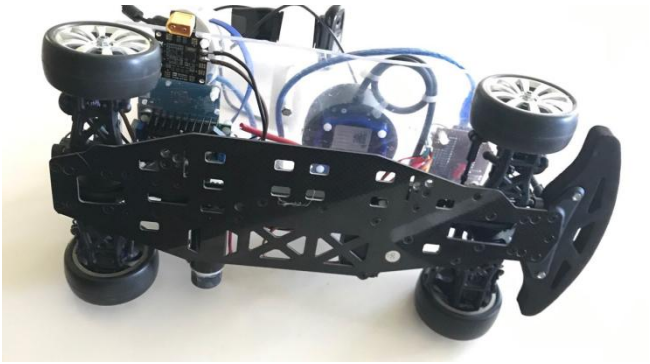

Ilustración 2: Vista inferior del coche

2.1.2. ODROID-XU4

Ilustración 3: Odroid-XU4

La Odroid-XU4 es un ordenador de tamaño y precio reducido (similar a una Raspberry Pi, aunque más potente). Funciona con el sistema operativo Linux y es el ordenador a bordo desde el cual se ejecutan los programas que controlan el coche. Dispone de 2 puertos USB 3.0 que se utilizan para conectarlo a los demás dispositivos.

2.1.3. ARDUINO NANO

Ilustración 4: Arduino nano

El Arduino nano es un microcontrolador que, en este caso, tiene como objetivo comunicarse con la Odroid para controlar el motor y el servomotor, además de transmitir la información del *encoder*.

2.1.4. RPLIDAR A2M8-R3

Ilustración 5: Rplidar A2M8-R3

Un lidar (Laser Imaging Detection And Ranging) es un dispositivo que permite determinar la distancia desde un emisor láser a un objeto o superficie utilizando un haz láser pulsado. La distancia al objeto se determina midiendo el tiempo de retraso entre la emisión del pulso y su detección a través de la señal reflejada.

El Rplidar A2M8-R3 es un lidar de bajo coste que proporciona una detección de 360º, con una frecuencia de rotación de 10 Hz y capaz de detectar obstáculos hasta a 16 metros de distancia.

Este Lidar permite al coche detectar obstáculos para así poder evitarlos.

2.1.5. CÁMARA Y MONTURA

El coche utilizado en el *"SEAT autonomous driving challenge"* utilizaba una cámara tipo *Kinect*. Este tipo de cámaras no solo obtienen una imagen RGB sino que además posee un sensor de profundidad que obtiene la distancia a la que se encuentran los objetos que observa. El equipo que participó en la competición comunicó que solo se utilizó la funcionalidad de la imagen ya que para obtener la distancia a la que se encuentran los objetos utilizaron solo el lidar.

Por las razones que se han mencionado anteriormente, al realizar este modelo, se ha decidido emplear una cámara RGB. La cámara escogida es una Logitech Webcam c170 que permite obtener video con una calidad de 640x480 pixeles.

Ilustración 6: Logitech webcam c170

Las imágenes obtenidas con esta cámara tienen dos objetivos: detectar las líneas del carril, lo que permitirá al coche permanecer entre ellas, y detectar señales como pueden ser las de semáforo.

Debido a que esta cámara no es de gran angular, para obtener en una misma imagen objetos cercanos (las líneas de la carretera a la altura del parachoques) y objetos lejanos (un semáforo) es necesario colocar a la cámara a mayor altura que el resto de los dispositivos.

Había que diseñar, por tanto, un soporte para colocar la cámara teniendo en cuenta que debía ser pequeño para interferir lo mínimo con el lidar, el cual no sería capaz de detectar objetos que estuviesen situados detrás del soporte.

Sabiendo que el lidar toma información en un plano horizontal, lo importante en el diseño del soporte era que a la altura a la cual obtiene los datos haya la mínima cantidad de interferencias. Finalmente se tomó como decisión colocar el soporte de la cámara sobre dos columnas. De esta forma se interfiere lo mínimo con el lidar como se observa en la Ilustración 7. En rojo aparecen los puntos en los cuales las columnas producen interferencia en el lidar. En verde se muestra toda la zona en la cual se producirían interferencias si se hubiese hecho el soporte macizo en lugar de con columnas.

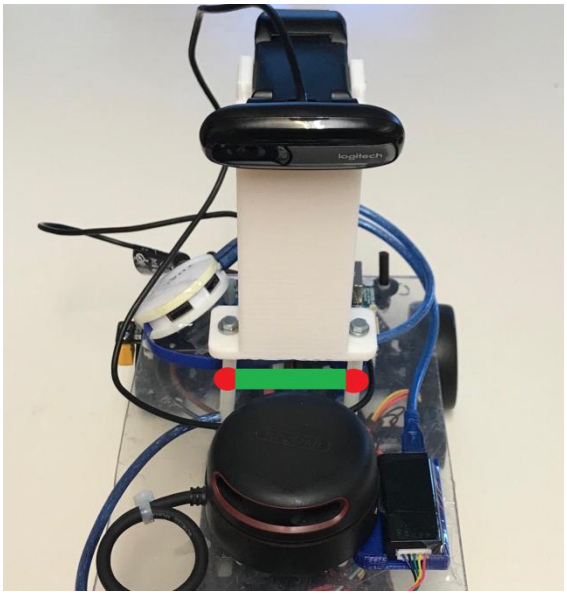

Ilustración 7: Interferencia soporte cámara

Sin embargo, la forma de las columnas es poco estable si se quiere lograr más altura.

Por lo tanto, había que diseñar una pieza para lograr la altura necesaria. Esta pieza debía de cumplir los siguientes requisitos:

- La forma de la parte inferior debía estar adaptada para poder ser atornillada a las columnas.
- La forma de la parte superior debía estar adaptada para que la cámara se pudiera encajar.
- Debía pesar poco para evitar que las columnas pandeasen con el peso.
- Debía ser rígida para evitar que la propia pieza pandease.

Ante estas premisas se consideró que la mejor forma de obtener esta pieza era mediante la impresión 3D ya que permite diseñar la pieza con la forma deseada. Además, el PLA (material que se suele utilizar en este tipo de fabricación) cumple las características de rigidez y ligereza.

El siguiente paso, por tanto, era diseñar la pieza. Para el diseño de la pieza se ha utilizado el programa FreeCAD.

Ilustración 8: Icono FreeCAD

FreeCAD es un programa de diseño asistido por computadora en tres dimensiones. A diferencia de CATIA (programa de CAD que se enseña en la asignatura de Dibujo Asistido por Ordenador en el grado de Tecnologías Industriales), su utilización es gratuita y es de código abierto.

La pieza diseñada que se puede observar en la Ilustración 9 consta de tres partes:

- La parte inferior de forma rectangular con las esquinas redondeadas y con dos agujeros por los cuales pasar los tornillos que se unen a las columnas sobre las cuales se apoya el soporte.
- La parte principal, que es el bloque que va a proporcionar altura a la cámara de una forma estable.
- La parte superior, que dispone de cuatro prismas rectangulares en los cuales se va a apoyar la cámara. Estos prismas sirven para encajar la cámara utilizando los trozos de plástico que sobresalen de su parte inferior como se observa en la Ilustración 10.

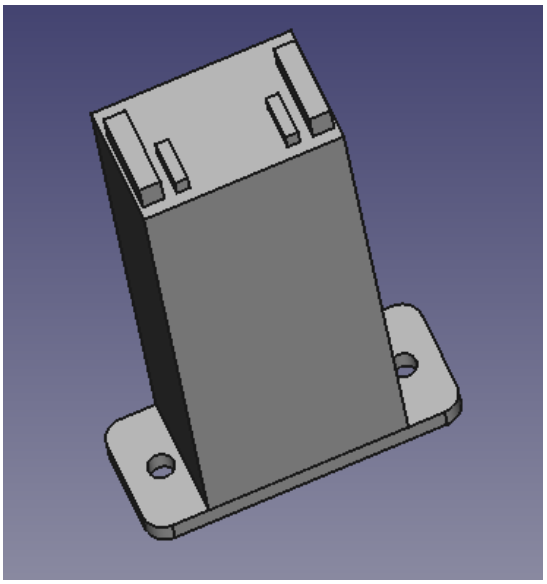

Ilustración 9: Diseño soporte en FreeCAD

Ilustración 10: Parte inferior de la cámara

Una vez realizado el diseño de la pieza se ha exportado al formato stl, formato que define la geometría de objetos 3D y que es utilizado por el software de control de las impresoras 3D.

Después de obtener la pieza en formato stl hay que introducirla en un programa que permita ajustar los parámetros de impresión 3D. El programa que ha sido utilizado para este propósito es Ultimaker Cura.

Ilustración 11: Logo de Ultimaker Cura

Lo primero que hay que introducir en este programa es la impresora 3D utilizada (en este caso una *Creality 3D Ender 5*) y después las características de impresión. Debido a que es una pieza sencilla, se han utilizado los parámetros que el programa clasifica como "*Standard Quality*". Siendo los parámetros más característicos:

- Altura de capa: 0,2 mm
- Relleno del 20% con patrón cúbico
- Velocidad de impresión: 80 mm/s

Además, es importante la posición en la que se coloca la pieza para su impresión como se observa en la Ilustración 12. A la hora de escoger la colocación, hay que evitar en la medida de lo posible que haya voladizos. Para esta pieza la colocación es muy intuitiva.

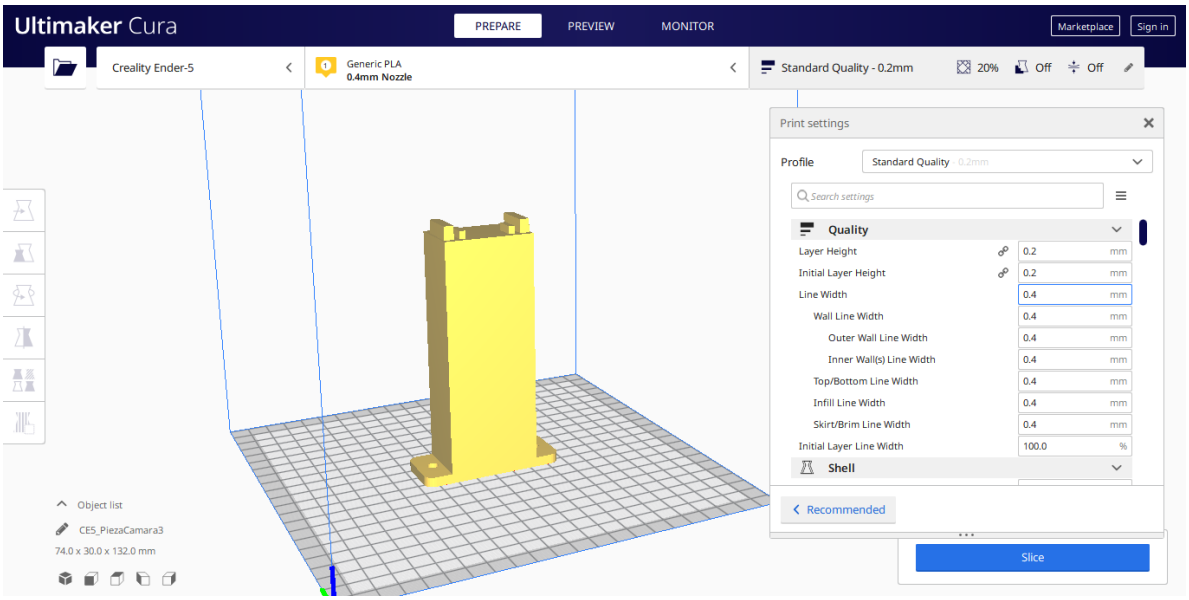

Ilustración 12: Soporte para imprimir en 3D

2.1.6. MOTOR CON *ENCODER*

El motor utilizado es un motor con escobillas *maxon DC motor*, que posee un *encoder* rotatorio incremental de dos canales.

El motor está conectado mediante una serie de engranajes y correas dentadas a las ruedas delanteras y traseras (todas giran a la misma velocidad).

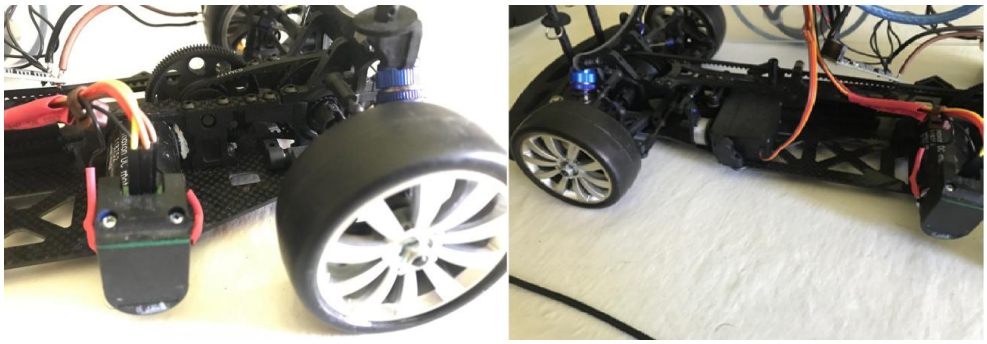

Ilustración 13: Motor con encoder y mecanismo de transmisión a las ruedas

El motor es el actuador que se encarga del movimiento lineal del coche, a través del cual acelera y frena. El *encoder* permite saber qué velocidad lleva el vehículo.

2.1.7. SERVOMOTOR XCITERC XLS-19S

Ilustración 14: Servomotor XLS-19s

El servomotor está unido a la dirección delantera del coche y es el que permite controlar la dirección de este.

2.1.8. *POLOLU MD07B*

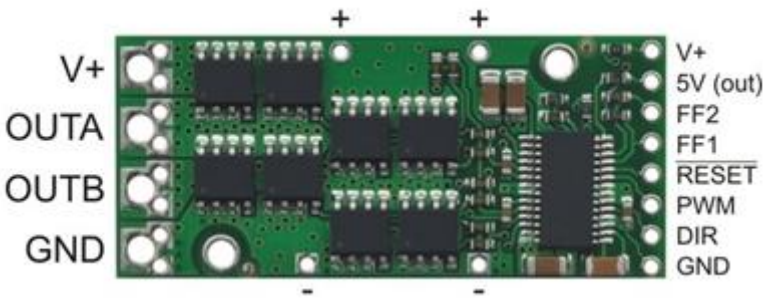

Ilustración 15: Pololu MD07b

El *pololu MD07b* es un controlador de motores que sirve de interfaz entre el Arduino, el motor y su alimentación. Este controlador es necesario ya que las señales de control del Arduino con las que controla la velocidad del motor funcionan con 5V, mientras que el motor trabaja con 12V. Además, permite controlar el sentido en el que gira el motor, lo que va a permitir que el coche se mueva hacia delante o hacia detrás.

2.1.9. PDB-XT60

Ilustración 16: PDB-XT60

El *PDB-XT60* es una placa de distribución de alimentación. Esta placa tiene un conector que permite acoplarla fácilmente a la batería LiPo que se utiliza para alimentar los componentes del coche. En esta placa es fácil soldar cables para así llevar la potencia a los distintos dispositivos.

2.1.10. BATERIA LIPO DE 3 CELDAS

Para la alimentación eléctrica del coche se utiliza una batería LiPo de 3 celdas (11,1 V) con conector XT60 para poder acoplarla a la *PDB-XT60*

2.1.11. *QSKJ DC-DC CONVERTER 10-32V TO 12-35V*

Ilustración 17: *QSKJ DC-DC Converter 10-32V to 12-35V*

Este dispositivo es un transformador que va a permitir transformar los 11,1V proporcionados por la batería LiPo en 12V para alimentar a la Odroid.

2.2. CONEXIONES

Una vez seleccionado todos los componentes hay que conectarlos entre ellos para que el coche funcione. En la Ilustración 18 se puede observar un esquema que muestra el cableado entre los distintos dispositivos (los colores utilizados para los cables del esquema son los colores de los cables que tiene el coche realmente).

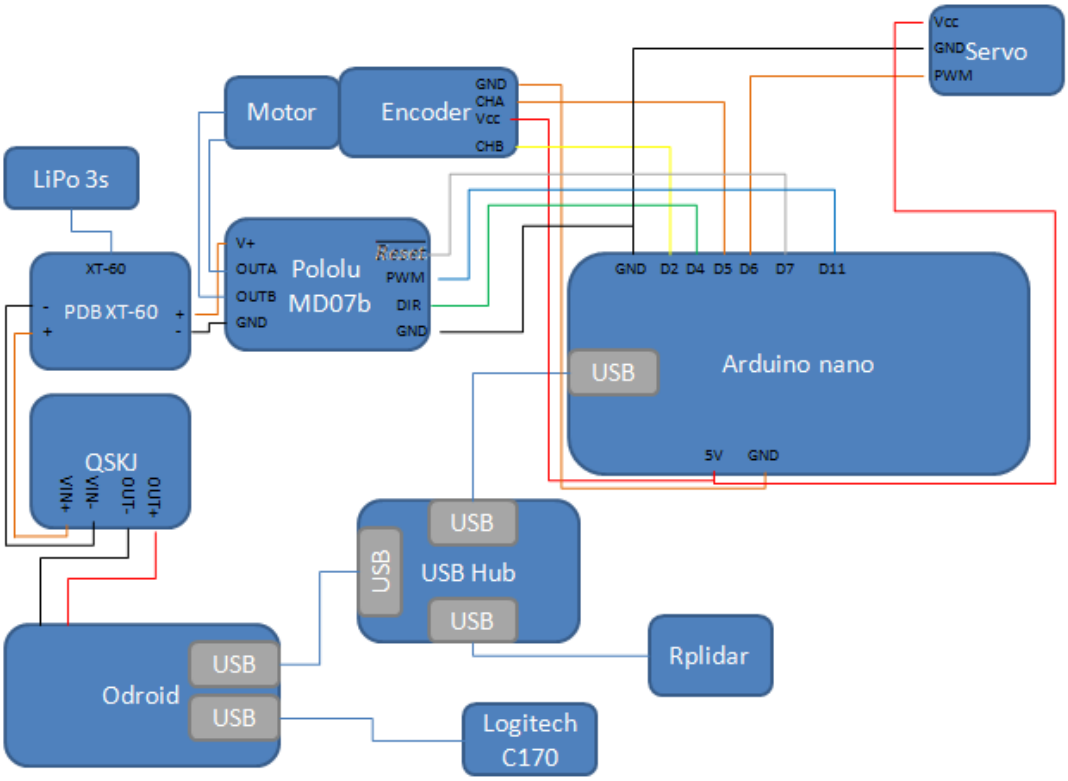

Ilustración 18: Esquema de conexiones

Para la realización de este esquema fue necesario un estudio detallado de los *datasheets* de los distintos componentes para asegurar que las conexiones eran posibles y se realizaban correctamente.

Este esquema no solo es útil para el montaje del coche sino también a lo largo de todo su ciclo de vida ya que en caso de que algún cable se desconecte (lo cual puede ocurrir durante el transporte del vehículo) permite encontrar rápidamente donde hay que conectarlo. Además, es necesario saber cómo están conectados los distintos *pins* del Arduino para su programación.

3) APRENDIZAJE Y MANEJO DE HERRAMIENTAS PARA LA PROGRAMACIÓN. PROGRAMACIÓN ROS.

Para la programación del coche autónomo hace falta tener conocimientos de Linux, C++, ROS y Arduino. En este capítulo se va a realizar una breve descripción de estas herramientas.

3.1. LINUX

Linux es un sistema operativo al igual que Windows o MacOS, con la particularidad de ser software libre (todo su código fuente puede ser utilizado, modificado y redistribuido libremente por cualquier persona, empresa o institución).

Ilustración 19: Símbolo de Linux

El sistema operativo instalado en la Odroid es Linux, más concretamente, la distribución Ubuntu 16.0.4 adaptada para la Odroid que se puede descargar de la página del proyecto del coche autónomo de la Universidad de Berlín [4].

Este sistema adaptado para el ordenador a bordo del coche tiene la particularidad de no tener una interfaz gráfica. Esto significa que para trabajar en la Odroid es necesario conectarse a ella desde otro ordenador y solo se la puede programar usando un terminal.

A pesar de que Linux es un sistema operativo muy conocido, sobre todo por desarrolladores, la mayoría de personas no están acostumbradas a trabajar con él y menos aún sin interfaz gráfica. Es por ello por lo que se recomienda familiarizarse con este sistema operativo antes de trabajar con el coche autónomo. La mejor manera para ello es realizar un tutorial como puede ser el de la universidad de Surrey [5].

Para la realización de este proyecto es necesario tener un ordenador con el cual conectarse a la Odroid. Es recomendable que este ordenador tenga también instalado una distribución de Ubuntu 16.0.4 (directamente o en una máquina virtual) ya que permitirá probar códigos en este antes de intentarlo en la Odroid. Al tener interfaz gráfica resultará más fácil realizar comprobaciones así. Aunque sea posible realizar el proyecto con un ordenador Windows este proyecto se ha realizado con una distribución Linux y toda la información está referida para trabajar con este sistema operativo.

3.2. C++

C++ es un lenguaje de programación de alto nivel cuya sintaxis es heredada de C. A diferencia de C es un programa orientado a objetos (POO). Se obtiene una primera aproximación a este lenguaje de programación en la asignatura de Fundamentos de Informática del primer curso de los grados de Ingeniería Industrial, siendo necesario para este trabajo estudiarlo en mayor profundidad.

C++ se utiliza para realizar los distintos programas que controlan el coche, como puede ser el programa encargado de que el coche siga las líneas de la carretera.

Una de las bibliotecas más importantes de C++ que se utiliza en este trabajo, sin tener en cuenta la de ROS de la cual se habla más adelante, es openCV (open Computer Vision).

Ilustración 20: Logo de openCV

OpenCV es una biblioteca libre de visión artificial desarrollada en C++ ampliamente utilizada en la robótica móvil y que va a permitir tratar las imágenes obtenidas por la cámara del coche.

3.3. ARDUINO

Arduino es una plataforma de creación de electrónica de código abierto, la cual está basada en hardware y software libre. Los dispositivos Arduino son microcontroladores económicos que permiten controlar actuadores y leer señales de sensores.

Ilustración 21: Símbolo Arduino

El lenguaje de programación que usa Arduino es una versión adaptada del lenguaje de programación C++. Gracias a su característica de software libre y a su gran comunidad es fácil encontrar en internet bibliotecas que facilitan la creación de código para sistemas complicados. Una de estas bibliotecas es la biblioteca ROS que va a permitir que el Arduino se comunique con la Odroid y se transmita la información que controla el coche.

El código se escribe en el ordenador en el Arduino IDE (que se puede descargar en la página web [6]) y después se sube a la placa del Arduino.

3.4. ROS

Una de las principales dificultades de este proyecto fue aprender a utilizar ROS [7](Robot Operating System).

Para comprender qué es ROS es necesario saber lo que es un *framework* en el campo de la programación y del software.

Un *framework* es una estructura tecnológica compuesta de módulos, métodos de programación y librerías que sirven de base para el desarrollo de software en un área concreta como puede ser la robótica.

Ventajas de utilizar un *framework* para programar:

- Evitar escribir código repetitivo
- Reducir el tiempo de desarrollo
- Aumentar la portabilidad de las aplicaciones programadas
- Realizar código más complejo que si partes de cero

Desventajas de utilizar un *framework* para programar:

- Tiempo necesario para aprender su funcionamiento
- Mayor consumo de recursos
- Aumenta la complejidad de programas simples

Conociendo la definición de *framework*, se puede describir ROS como un *framework* destinado a la creación y utilización de aplicaciones en el campo de la robótica.

ROS utiliza principalmente tres elementos para su funcionamiento:

- Paquetes
- Nodos
- Comunicación entre nodos

3.4.1. PAQUETES

El software en ROS está organizado en paquetes. Un paquete puede contener un nodo, una librería, conjunto de datos, o cualquier cosa que pueda constituir un módulo. Los paquetes pueden organizarse en pilas (stacks).

Los elementos que suelen componer un paquete son los siguientes:

- **CMakeLists.txt** Este archivo contiene las instrucciones que permiten la construcción del paquete, como puede ser: nombre del paquete, árbol de dependencias de otros paquetes, rutas de librerías, rutas de ficheros de mensajes o servicios a construir, ejecutables a generar, etc.
- **package.xml** Al igual que el fichero anterior sirve para definir dependencias del paquete, de compilación, versión, etc. Es muy similar a CMakeList.txt pero tiene un carácter más informativo.
- **CodigoFuente.cpp** Suelen estar localizados en una carpeta con nombre src. Definen las aplicaciones que componen al paquete. Pueden estar escritos en el lenguaje de programación C++ (.cpp) o en Python (.py).

- **Directorio srv** Los ficheros que definen los servicios, en caso de haberlos, se encuentran localizados en esta carpeta.

- **Directorio msg** Al igual que con el anterior directorio aquí se encuentran los ficheros que definen la estructura de los mensajes.

Esta estructura de los paquetes permite construir los ejecutables de forma sencilla con la herramienta *catkin*.

Esta herramienta incluida en ROS y que a su vez se trata de un paquete, es la aplicación encargada de construir paquetes generando los ejecutables

Los principales paquetes que se van a utilizar para este proyecto son los siguientes:

- **Rplidar_ros** [8]: Este paquete permite leer la información del *Rplidar* y la transforma en mensajes *ROSLaserScan*.

- **Rosserial** [9]: permite la comunicación en serie con el Arduino nano. Este nodo gestiona la publicación y suscripción a los mensajes del Arduino.

- **Usb_cam** [10]: Hace de interfaz con las cameras USB y publicas las imágenes como *sensor_msgs::Image* (formato de imágenes que usa ROS)

3.4.2. NODOS

Los nodos son los distintos programas de ROS que se ejecutaran para realizar diferentes funciones y se pueden comunicar entre ellos.

En el coche autónomo, por ejemplo, hay un nodo que se encarga de controlar el motor con el que se mueve el vehículo, otro que se encarga de recibir las imágenes de la cámara y otro que analiza esta información para determinar si hay un semáforo y se comunica con el nodo que controla el motor para frenar en caso de que esté en rojo.

El nodo más importante de ROS y que hay que inicializar antes que cualquier otro nodo es el nodo Máster. Este nodo es el núcleo de ROS y es necesario para que funcionen el resto de nodos.

Las funciones principales del nodo máster son las siguientes:

- Registrar que nodos se están ejecutando

- Permitir la comunicación entre los nodos a través de servicios y mensajes

3.4.3. COMUNICACIÓN ENTRE NODOS

Una de las mayores ventajas que ROS ofrece a los proyectos es la posibilidad de ejecutar múltiples nodos a la vez. Pero para que esto nos permita crear sistemas complejos es necesario que estos nodos puedan comunicarse entre ellos.

ROS tiene dos formas diferentes de comunicación entre los nodos que mantienen una estructura común lo que permite la compatibilidad entre distintos paquetes.

Las principales formas comunicación son:

- Por *topics*
- Por servicios

3.4.3.1. COMUNICACIÓN POR *TOPICS*

La mejor manera de ver cómo funcionan los *topics* es compararlos con un tablón de anuncios en el que algunos nodos podrán dejar mensajes y otros podrán leerlos.

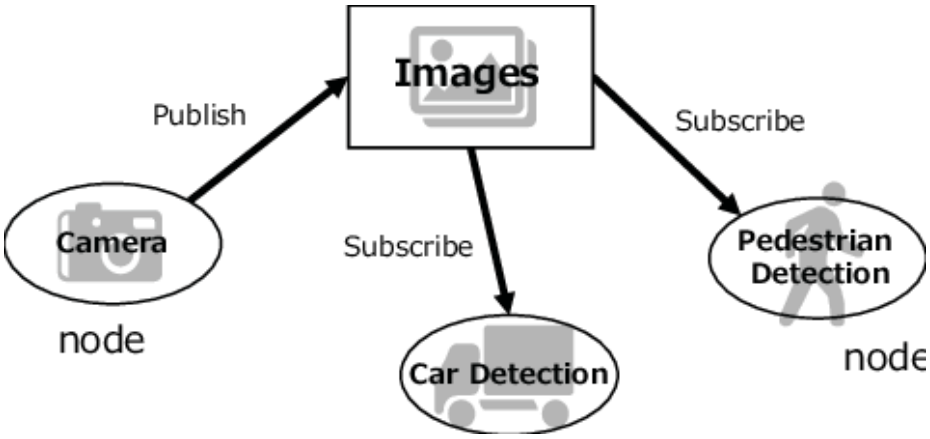

Ilustración 22: Esquema de funcionamiento de los *topics*

En la comunicación por *topics* existen los nodos pueden tener dos roles: suscriptores o publicadores

El nodo publicador es aquel que creará el *topic* para compartir información con el resto de *topics* mediante mensajes. Estos mensajes son visibles por los nodos suscritos al *topic*. Normalmente los mensajes se publican con una determinada periodicidad.

El nodo suscriptor es aquel nodo que necesita la información publicada por un nodo publicador para funcionar. Para ello tendrá que suscribirse con anterioridad al *topic* correspondiente y entonces podrá leer los mensajes publicados.

Los nodos publicadores no controlan quien se suscribe a los *topics* por lo que este tipo de comunicación es unidireccional (asíncrona).

Para permitir la modularidad y compatibilidad con la que funciona ROS es necesario que los mensajes estén bien definidos. Los mensajes tienen una estructura sencilla que puede contener varios campos y queda definida en un fichero de mensaje .msg

Ejemplos de mensajes sencillos que se utilizan en ROS son:

- int8: un entero de 8bits
- twist: dos vectores de 3 componentes(ejes x,y,z) que contienen la componente linear y angular de un objeto

Los mensajes pueden llegar a ser mucho más complejos y específicos. Un ejemplo son los mensajes tipo *Image* utilizados para almacenar imágenes obtenidas por cámaras.

3.4.3.2. COMUNICACIÓN POR SERVICIOS

Como se ha descrito, los mensajes vía *topic* son unidireccionales. Esto puede ser un problema ya que el publicador no sabe si un suscriptor concreto ha recibido el mensaje.

En algunos casos puede que sea necesario saber si otro nodo ha recibido el mensaje. Es por esto que en ROS existe también la comunicación por servicios.

Los servicios permiten la comunicación bidireccional entre nodos.

La estructura de los servicios se divide en cuatro bloques:

- **Servidor:** Es el nodo que va a suministrar los servicios a los nodos clientes
- **Cliente:** Los nodos clientes usan los servicios del nodo servidor y pueden transmitir y recibir información al servidor
- **Variable de servicio:** Es la variable que comparten el cliente y el servidor con la que comparten información.

  Posee dos campos: *request* y *response*.

  El campo *request* es rellenado por el cliente y transmitido al servidor mientras que el campo response es rellenado por el servidor y lo transmite como respuesta al cliente.

- **Acción asociada al servicio:** En el nodo del servidor se ejecutará una acción asociada al servicio que podrá utilizar la información de la sección *request* rellenada por el cliente.

  Una vez realizada la acción el servidor puede rellenar el campo *response* de la variable compartida que podrá ser leído por el cliente.

4) CONTROL DE ALTO NIVEL EN ODROID. INSTALACIÓN Y PROGRAMACIÓN DE MODULOS ROS

4.1. INSTALAR ROS EN EL ORDENADOR

ROS tiene un gran número de distribuciones pero como la distribución instalada en la Odroid es *kinetic* hay que instalar la misma versión en el ordenador con distribución Ubuntu que se utiliza en el proyecto [11].

Para ello hay que seguir los siguientes pasos:

1. En el menú de búsqueda del ordenador (se accede pulsando el botón de Windows en el teclado) escribir *Software & Updates* y clicar en el botón que aparece.

2. Seleccionar las casillas de *multiverse*, *universe* y *restricted* (configurar Ubuntu para que acepte este tipo de repositorios).

3. Abrir el terminal y escribir (permite al ordenador acceder a los archivos de *packages.ros.org*):

   ```
   sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu
   $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
   ```

4. Después hay que escribir también:

   ```
   sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-
   key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
   ```

5. Hay que asegurar que la versión Debian esta actualizada:

   ```
   sudo apt-get update
   ```

6. Finalmente se instala ROS kinetic en el ordenador:

   ```
   sudo apt-get install ros-kinetic-desktop-full
   ```

4.2. PREPARACIÓN DE LA ODROID

4.2.1. INSTALAR EL SISTEMA OPERATIVO DE LA ODROID

Al igual que en las Raspberry PI toda la información se guarda en una tarjeta microSD en las Odroid la información se guarda en una tarjeta eMMC (de al menos 16GB de memoria) donde habrá que instalar el sistema operativo.

Este sistema operativo se encuentra para descargar en la página de la universidad de Berlín [4]. Existen varias versiones pero en este proyecto se va a trabajar con la 3.1 al ser la más reciente. Esta versión contiene Ubuntu 16.04 y ROS *kinetic*.

Una vez descargado el fichero que contiene el sistema operativo hay que instalar en el ordenador personal el paquete *libterm-readkey-perl* que es el que va a permitir instalar la eMMC. Este paquete se puede descargar escribiendo en la consola Linux (para abrir la consola Linux usar Ctrl+Alt+T):

```
sudo apt-get install libterm-readkey-perl
```

Una vez hecho esto hay que conectar la tarjeta eMMC al ordenador, para ello se usa un adaptador SD a eMMC

Ilustración 23: Adaptador SD a eMMC

Se extrae la carpeta comprimida que se ha descargado y se navega hasta ella en el terminal:

```
cd linux-image-for-odroid-xu4-ubuntu16.04-kinetic
```

Después, con el comando *ls* buscamos la ubicación de la carta eMMC escribiendo:

```
ls /dev/sd*
```

Con este comando aparecerán todos los dispositivos SD conectados al ordenador, en caso de duda de cuál es el de la tarjeta eMMC es aconsejable poner el comando y mirar el listado que aparece, después retirar el adaptador del ordenador y volver a escribir el comando. Aparecerá la misma lista de antes pero faltara la tarjeta eMMC. El nombre será algo parecido a */dev/sdb.* Después se escribe en el terminal (sustituyendo */dev/sdb* por el nombre que haya aparecido en nuestro ordenador):

```
sudo ./install.pl /dev/sdb
```

La instalación dura bastante tiempo. Cuando este acabando se pide al usuario que escoja una de las configuraciones propuestas como se observa en la Ilustración 24.

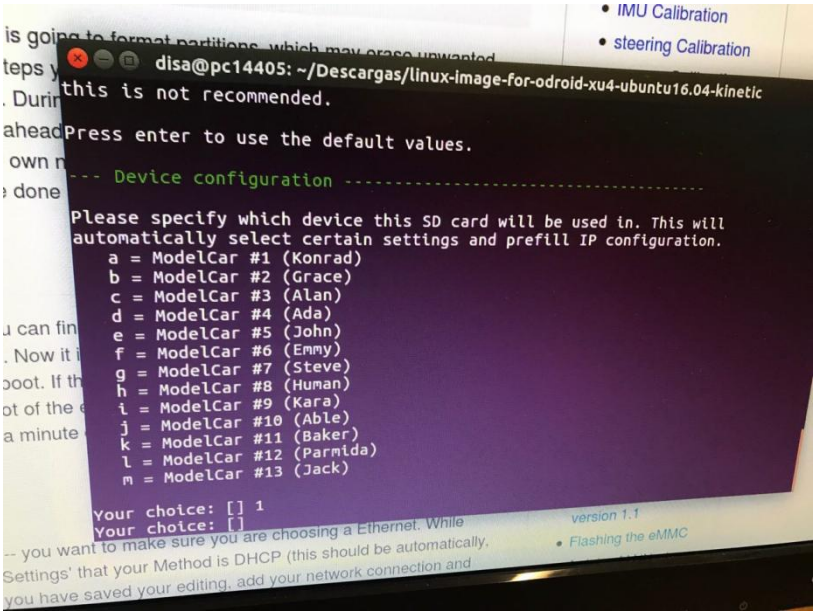

Ilustración 24: Escoger configuración Odroid

Para escoger una configuración hay que escribir la letra deseada, en el caso de este proyecto se escogió la configuración "a". Una vez escogida aparecen en pantalla los parámetros de configuración de la Odroid como su dirección IP y la contraseña como se observa en la Ilustración 25:

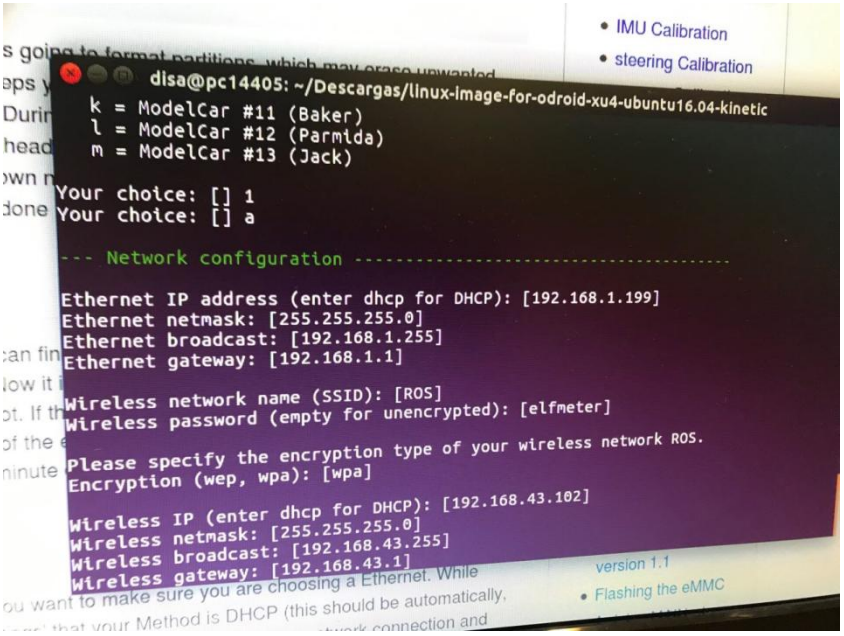

Ilustración 25: Parámetros de la configuración

Es muy importante conservar estos parámetros de configuración ya que son necesarios para trabajar con la Odroid.

Una vez finalizada la instalación ya se puede retirar la tarjeta del ordenador y colocarla en la Odroid. Después de colocarla en la Odroid es muy importante colocar el interruptor de la Odroid en la posición eMMC como se observa en la Ilustración 26.

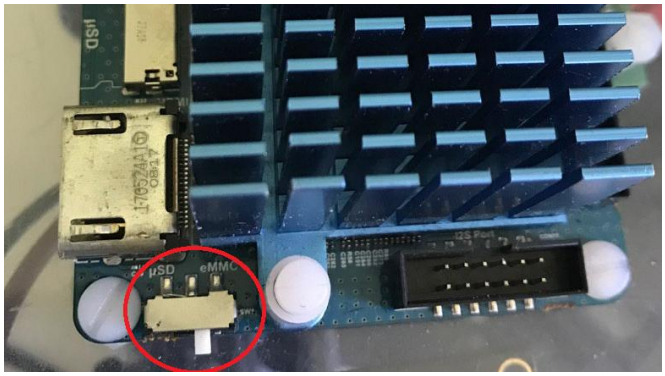

Ilustración 26: Interruptor eMMC

Una vez realizado esto, ya se puede alimentar la Odroid. Si la luz azul que se encuentra al lado de la conexión de la alimentación parpadea es buena señal. La primera vez que encendamos la Odroid requerirá inicializarse lo que puede llevar un par de minutos.

Una vez esperado el par de minutos hay que conectar la Odroid al ordenador utilizando un cable Ethernet. Después de conectar el cable en el ordenador hay que realizar los siguientes pasos:

1. Ir a "*Network connections*" y clicar en el botón "*Add*".
2. Seleccionar como tipo de conexión "Ethernet" y clicar en el botón "*Create*".
3. En la pestaña Ethernet en la lista de "*Device*" escoger la de la Odroid.

En la pestaña *IPv4 Settings* asegurarse que el método "*Automatic*" (DHCP) esta seleccionado.

En el menú de conexiones escoger la nueva conexión creada.

Para asegurar que la configuración de la conexión se ha realizado con éxito hay que comprobar el ping entre los componentes. Para ello escribir en el terminal el comando siguiente (el número a escribir es el que aparece en el *Ethernet IP address* que se observa en la Ilustración 25):

```
ping 192.168.1.199 -c 1
```

En caso que la conexión este bien configurada aparecerá un mensaje como el de la Ilustración 27.

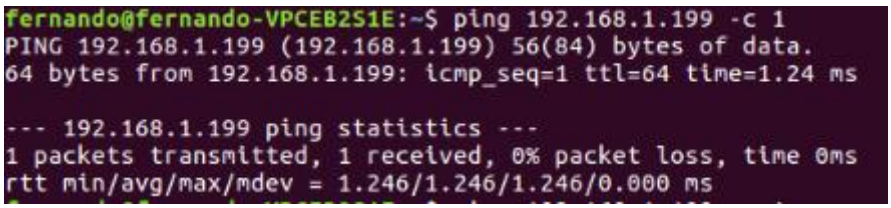

Ilustración 27: Ping correcto Odroid

En caso de haber un problema de conexión el mensaje será como el de la Ilustración 28.

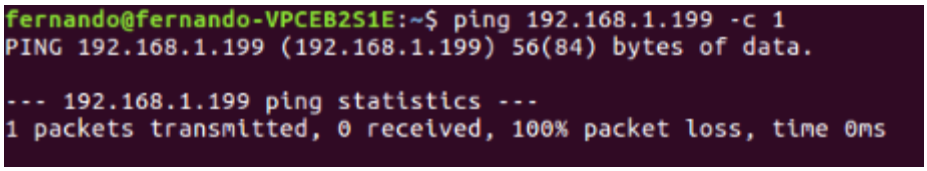

Ilustración 28: Ping incorrecto Odroid

4.2.2. ESTABLECER LA COMUNICACIÓN VÍA ROS ENTRE EL ORDENADOR Y LA ODROID

Para comprobar la comunicación hay que tener el ordenador conectado a la Odroid mediante un cable Ethernet. Se escribe en el terminal del ordenador lo siguiente (192.168.1.199 es la dirección IP de la Odroid):

```
ssh root@192.168.1.199
```

Una vez conectado hay que introducir la contraseña de la Odroid. La contraseña es *elfmeter* (ver apartado [*Wireless password*] de la Ilustración 25. Si se logra conectar se observa que lo que aparece ahora en la ventana de comandos es una línea del estilo *root@<pc-name>* (siendo <*pc-name*> el nombre de la Odroid, en el caso de la configuración 1 el nombre es Konrad). Si se logra conectar con éxito entonces hay que cerrar esta ventana del terminal.

Con esta acción lo que se logra es poder controlar la Odroid, los comandos que se lancen en esta terminal no se ejecutaran en el ordenador si no que se ejecutan en la Odroid. Esta va a ser la forma en la que se trabaja con ella y que va a permitir programarla.

El ordenador se va a conectar al *roscore* que se ejecuta en la Odroid. Para ello hay que cambiar la variable *ROS_HOSTNAME* a la dirección IP del ordenador y la variable *ROS_MASTER_URI* a la dirección IP de la Odroid. Si no cambiásemos estas variables los nodos de ROS ejecutados en el ordenador no se conectarían al *roscore* de la Odroid ya que

intentarían conectarse al *roscore* del ordenador y por tanto no se podría lograr la comunicación entre los dos dispositivos.

Para obtener la dirección IP del ordenador hay que escribir en el terminal:

```
ifconfig
```

La dirección aparece en el apartado enp4s0 en la sección *inet addr* y tiene la forma: 192.168.1.### (en mi caso era 192.168.1.105). Una vez se conoce la dirección IP del ordenador se escribe en el terminal las dos siguientes líneas:

```
export ROS_MASTER_URI=http://192.168.1.199:11311
export ROS_IP=192.168.1.105
```

Esto es necesario hacerlo en cada uno de los terminales que se creen en los que se quiera comunicar con la Odroid. Para evitar esto es aconsejable añadir estas dos líneas al fichero *~/.bashrc*. Para ello hay que escribir en el terminal:

```
vi ~/.bashrc
```

Se abrirá un fichero de texto y hay que añadir al final de este documento las dos líneas que se escribieron en el terminal.

A partir de este momento los nodos de ROS creados en el ordenador se conectan automáticamente al *roscore* de la Odroid. Si en algún momento se quiere probar un programa de ROS en el ordenador sin usar la Odroid habría que escribir en cada terminal:

```
export ROS_HOSTNAME=localhost
export ROS_MASTER_URI=http://localhost:11311
```

De forma parecida, si se quiere evitar tener que escribir la contraseña de la Odroid cada vez que se conecte a ella entonces hay que escribir en el terminal la siguiente línea:

```
ls ~/.ssh/id_rsa.pub || ssh-keygen
```

Se presiona la tecla *enter* varias veces hasta que dejen de salir mensajes y después se escribe en el terminal:

```
ssh-copy-id root@192.168.1.199
```

Si todo se ha configurado correctamente entonces debería ser posible ver desde el ordenador los *topics* creados por la Odroid y también publicar información del ordenador a *topics* que la Odroid puede leer.

Para verificar esto hay que abrir 6 terminales que denominaremos OT1, OT2, OT3 (Odroid Terminal) y CT1, CT2, CT3 (*Computer* Terminal).

En OT1, OT2, OT3 escribimos:

```
ssh root@192.168.1.199
```

Debido a que el sistema descargado para la Odroid viene con un fichero *Launch* que se lanza cuando se enciende ya hay ciertos nodos que se ejecutan en cuanto se alimenta la Odroid. Una forma de ver esto es darse cuenta que el lidar empieza a girar cuando se alimenta la Odroid. El script que permite esto es el *autostart.sh* que se encuentra en el fichero *root*. Este script se puede modificar si se quiere añadir otros programas que se ejecuten al inicializar la Odroid.

En OT1 se puede observar los nodos inicializados por *autostart.sh* escribiendo:

```
rosnode list
```

Si el terminal muestra un error porque no es capaz de comunicarse con el máster entonces en OT1 escribir:

```
roscore
```

Una vez hecho esto escribir en OT2 lo siguiente:

```
rostopic pub -r 1 /first std_msgs/String "data: 'hello'"
```

Con este comando la Odroid publica en el *topic first* con un ratio de 1 hercio el mensaje de tipo *String "hello"*.

Y en CT1 escribir lo siguiente:

```
rostopic echo /first
```

Con este comando el ordenador se subscribe al *topic first.* Si todo funciona bien entonces en CT1 deberían verse los mensajes creados en OT2 (en este caso los mensajes de "*hello*")

Para comprobar la comunicación en la otra dirección escribir en CT2:

```
rostopic pub -r 1 /second std_msgs/String "data: 'bye'"
```

Y en OT3 escribir:

```
rostopic echo /second
```

Si todo funciona correctamente entonces en OT3 se leerán los mensajes creados en CT2 ("*bye*").

Se puede obtener una representación visual de cómo están conectados estos nodos y *topics* escribiendo en CT3 lo siguiente:

```
rqt_graph
```

En la Ilustración 29 se observan los dos topics (*first* y *second*) y como en ambos hay un subscriptor y un publicador.

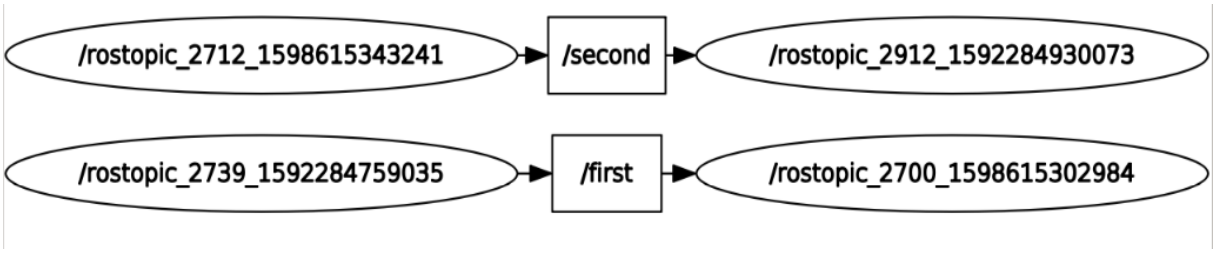

Ilustración 29: Ejemplo rqt_graph

Con esto queda comprobado que la comunicación entre el ordenador y la Odroid funciona.

4.2.3. PREPARACIÓN DEL LIDAR

Para trabajar con el Lidar en ROS se utiliza la biblioteca *rplidar_ros*. Esto nos va a permitir detectar la distancia a la que se encuentran los objetos que rodean al coche autónomo. Hay que tener en cuenta que también detectara aquellas partes del coche que se encuentren a la misma altura que el Lidar y estos puntos por tanto deberían ser ignorados.

Para verificar el comportamiento del Lidar hay que cerrar todos los terminales y abrir tres nuevos que llamaremos OT1, OT2 y CT1.

En OT1 y OT2 hay que escribir:

```
ssh root@192.168.1.199
```

En OT1 detenemos todos los procesos de ROS que se estén ejecutando en la Odroid escribiendo:

```
pkill ros
```

Se debe observar como el Lidar deja de girar. Después en OT1 hay que escribir:

```
roscore
```

En OT2 se escribe:

```
roslaunch rplidar_ros rplidar.launch
```

En CT1 hay que escribir:

```
rostopic list
```

Si todo funciona correctamente en la lista debería aparecer el *topic /scan* que es donde aparecen los datos obtenidos por el lidar. En este *topic* aparecen los puntos obtenidos por el Lidar y se podría ver la información escribiendo *rostopic echo /scan* sin embargo esto para nosotros no nos daría una idea de que está ocurriendo. Lo mejor es ver la información en una interfaz gráfica. Para eso se va a utilizar Rviz.

En CT1 se escribe:

```
rviz
```

Con esto se abre la interfaz de Rviz. Para observar los datos obtenidos por el Lidar hay que realizar los siguientes pasos:

En G*lobal Options -> Fixed frame* escribir *laser*

Clicar en el botón de abajo a la izquierda "*Add"* clicar en la pestaña "*By topic"* y escoger */LaserScan*

Después de realizar estos pasos deberían aparecer en pantalla (ver Ilustración 30) unos puntos rojos que son los objetos que rodean al Lidar.

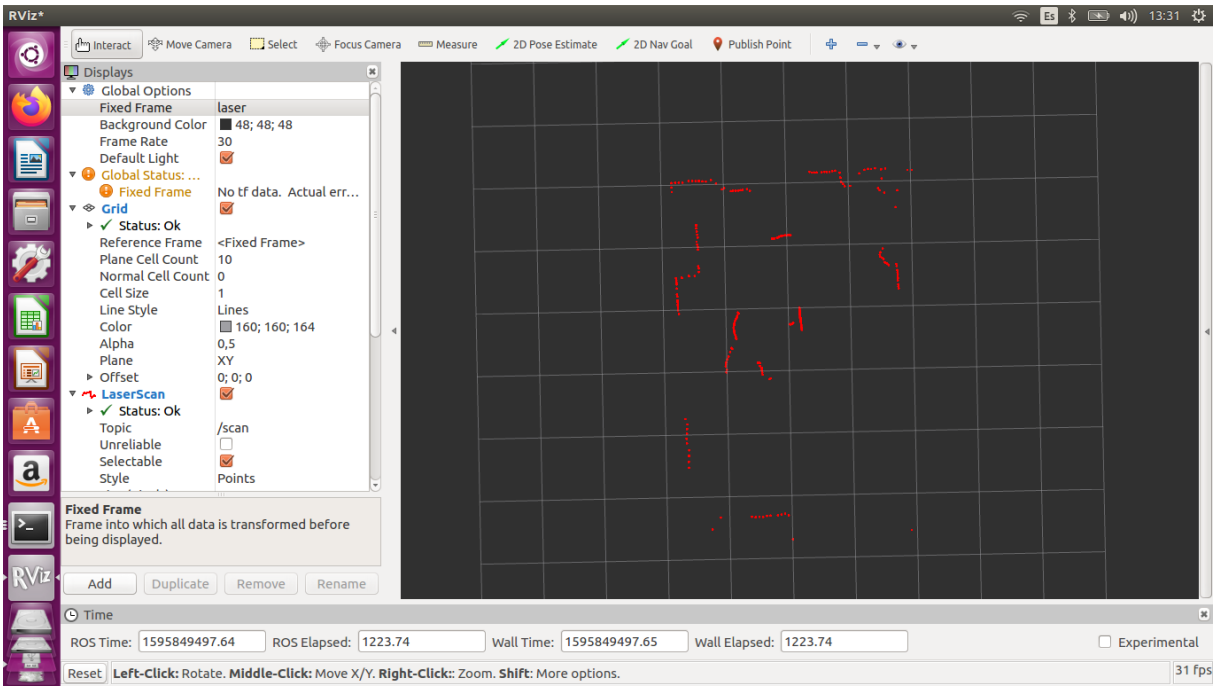

Ilustración 30: Mapa de puntos Lidar

4.2.4. PREPARACIÓN DE LA CÁMARA USB

Para verificar el buen funcionamiento de la cámara el proceso es similar al de la verificación del Lidar. Para ello hay que cerrar todos los terminales y abrir tres nuevos que llamaremos OT1, OT2 y CT1.

En OT1 y OT2 hay que escribir:

```
ssh root@192.168.1.199
```

En OT1 detenemos todos los procesos de ROS que se estén ejecutando en la Odroid:

```
pkill ros
```

Ahora hay que determinar en qué puerto se encuentra la cámara. Para ello se escribe en OT1:

```
ls -ltrh /dev/video*
```

Aparece una lista con varios puertos. Para saber cuál es el de la cámara lo mejor es desconectar la cámara de la Odroid y volver a ejecutar el comando. El puerto debería ser de la forma */dev/videoX*

Una vez encontrado se inicia el *roscore* en OT1. En OT2 se inicia la cámara escribiendo:

```
rosrun usb_cam usb_cam_node _video_device:=/dev/video0
_pixel_format:=mjpeg _image_width:=640 _image_height:=480
```

Aparecerán los siguientes mensajes de error que se pueden ignorar:

```
[swscaler @ 0x14a5f0] No accelerated colorspace conversion found from
yuv422p to rgb24
[swscaler @ 0x14a5f0] deprecated pixel format used, make sure you did
set range correctly
```

En CT1 se observa que los *topics* con la información de las imágenes se está transmitiendo escribiendo:

```
rostopic list
```

Deberían aparecer varios *topics* precedidos por */usb_cam*. Para ver las imágenes de la cámara del *topic /usb_cam/image_raw* hay que escribir en CT1:

```
rosrun image_view image_view image:=/usb_cam/image_raw
```

Tras ejecutar este comando se debería poder ver en pantalla lo que está viendo la cámara

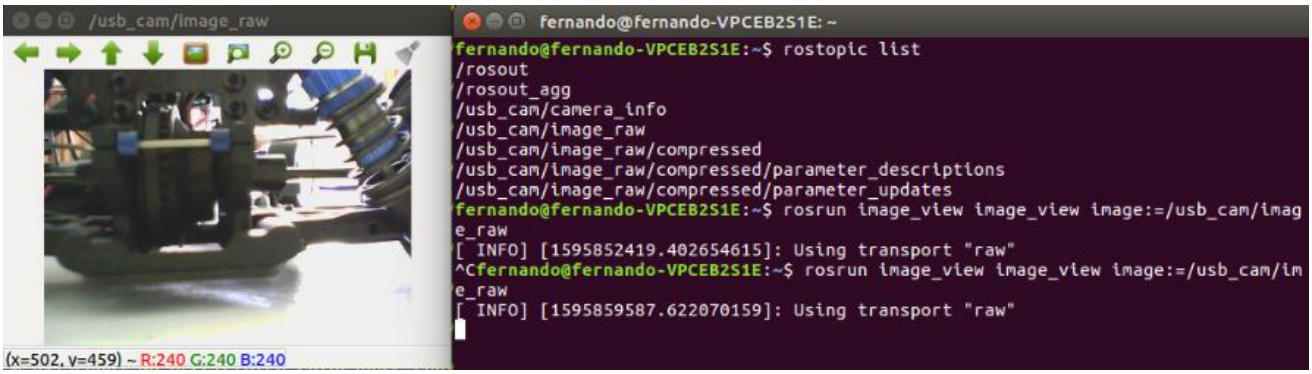

Ilustración 31: Vista de la cámara USB

5) CONTROL DE BAJO NIVEL ARDUINO

5.1. INSTALACIÓN DEL IDE DE ARDUINO

El primer paso para preparar el Arduino y los componentes conectados a él (*encoder*, servomotor y motor) es descargar el IDE de Arduino.

El IDE se puede descargar de la propia página de Arduino [6].

La carpeta comprimida descargada hay que extraerla. Una vez hecho esto, hay que instalarlo usando el script *install.sh.* Para ello hay que cerrar todos los terminales y abrir uno nuevo. En este terminal escribir:

```
cd ~/Downloads
tar xf arduino-1.8.13-linux64.tar.xz -C ~/
cd ~/arduino-1.8.13
./install.sh
```

Donde está escrito "1.8.13" hay que sustituirlo por la versión de Arduino que se haya descargado (ver el nombre de la carpeta descargada).

Después de esto hay que descargar las carpetas con las bibliotecas de la página de la universidad de Berlín escribiendo:

```
git clone https://github.com/AutoModelCar/auto_arduino_nano.git
cd auto_arduino_nano
git checkout version-3-rosserial
```

Hay que añadir las bibliotecas descargadas (MPU6050, Adafruit_NeoPixel, I2Cdev, Rosserial_Arduino_Library) usando el gestor de bibliotecas del IDE de Arduino.

El código utilizado en el Arduino aparece en el anexo 1. Hay que pegar este código en el IDE del Arduino y compilarlo dando al botón de *verificar.*

Después de compilar el código hay que conectar el Arduino al ordenador. Para poder subir el código hay que dar permisos al puerto USB al que está conectado. Para ello en el IDE del Arduino *Tools->Serial Port* veremos el puerto que tendrá la forma */dev/ttyUSBX*. Se abre un terminal y se escribe:

```
sudo chmod 777 /dev/ttyUSB0
```

Con esto daremos derechos de escritura y lectura al puerto USB. Esta acción de dar derechos al puerto habrá que realizarla también en la Odroid.

Ahora se puede subir el código al Arduino clicando en el botón de "*subir*.

5.2. CÓDIGO DEL ARDUINO

El código comienza declarando las bibliotecas y variables que se van a usar.

Aquellas variables que acaban en "_PIN" y que se han declarado usando *#define* son para declarar los pines a los que están conectados los distintos componentes. Se puede comprobar que coinciden con los del esquema de la Ilustración 18. En caso de haber modificado las conexiones respecto a los de este esquema habría que modificar el código respectivamente.

La biblioteca *<avr/pgmspace.h>* va a permitir guardar información en la memoria flash en vez de la SRAM con el objetivo de optimizar el programa

La biblioteca *<Servo.h>* permite controlar fácilmente servomotores.

La biblioteca *<Ros.h>* permite la comunicación utilizando ROS. Esta biblioteca va a permitir crear publicadores y suscriptores ROS con los que el Arduino se comunicara con la Odroid. Es necesario incluir también los tipos de mensaje ROS que se van a transmitir e inicializar el nodeHandle.

El Arduino va a tener dos publicadores y dos suscriptores:

- El publicador *pubTwist* va a publicar al *topic twist* la velocidad a la que está avanzando el coche. El mensaje es tipo *twist*.
- El publicador *pubTicks* va a publicar al *topic ticks* el numero de *ticks* del *encoder* que hay entre dos tomas de datos. El mensaje es tipo UInt16.
- El suscriptor *steeringCommand* se va a suscribir al *topic steering*. En este *topic* la Odroid publicara cuanto quiere que gire el servomotor. El mensaje tiene que ser tipo Float32.
- El suscriptor *speedCommand* se va a suscribir al *topic speed*. En este *topic* la Odroid publicará la velocidad a la que quiere que gire el motor. El mensaje tiene que ser tipo UInt16.

StartTimer2

La función *StartTimer2* va a ajustar los parámetros del *timer2* [11] interno del Arduino.

Un *timer* es un módulo interno de un microcontrolador que permite crear una señal periódica con una frecuencia que el usuario puede determinar. Una de sus principales funciones es contar con la frecuencia a la que se le ha configurado.

El Arduino nano posee tres *timers*: timer0, timer1, timer2. Sin embargo hay ciertas funciones que hacen uso de estos *timers*. Por ejemplo la biblioteca servo hace uso del timer0 y la función delay() hace uso del timer1. Si se modifica uno de esos *timers* se modificaría también el comportamiento de esas bibliotecas o funciones. Es por esto que el *timer* que se modifica es el 2 ya que en el resto del programa no se hace uso de funciones o bibliotecas que necesiten este *timer* para su funcionamiento.

El *timer2* es de 8 bits, por tanto puede contar hasta 255. Si se configura el *timer* para que trabaje a 100Hz, es decir, con un periodo de 10 ms contaría de 0 a 255 en un tiempo de 255*10ms = 2,55s. Al llegar a este punto el Arduino genera una interrupción que puede ejecutar una función.

Los *timers* funcionan de forma independiente al procesador lo que permite que otros procesos se ejecuten hasta que el *timer* avise de que ha finalizado la cuenta.

Hay que tener en cuenta que los *timers* tienen ciertos pines asignados y en el caso del *timer2* sus pines asignados son el 3 y el 11. El PWM del motor se va a controlar con *timer2* y es por esto que se conecta al pin 11. Es muy importante tener esto en cuenta ya que el PWM del motor solo se podrá conectar al pin 3 y el 11, no valen el resto de pines.

En este caso al estar conectado al pin 11 se le declara como salida.

La variable TIFR2 determina el valor en el que se encuentra el contador (de 0 a 255). Se iguala a 0 para inicializarla.

La línea "TIMSK2 = TIMSK2 | 0x01" habilita las interrupciones cuando se produce el desbordamiento, es decir, se genera una interrupción cuando el contador llega a 255.

La línea "TCCR2B = (TCCR2B & 0b11111000) | 0x02" va a provocar que el *timer2* funcione a 7812Hz que es la velocidad que se quiere que tenga el controlador PWM del motor.

Esta función se va a ejecutar durante la inicialización (*setup*) del Arduino.

ISR

ISR es un tipo de función especial, es una interrupción. Esta función se ejecuta cuando se cumple una condición. En ese momento se deja de ejecutar el programa principal del Arduino y se ejecuta la función ISR. Una vez ejecutada la función ISR, el programa principal sigue ejecutándose desde el punto en el cual se vio interrumpido por ISR.

En este caso la función ISR va a ejecutarse cuando el contador del *timer2* se desborde y lo que va a hacer esta función es volver a poner a 0 la variable TIFR2 y aumentar en 1 el valor de la variable T10vs2. La variable T10vs2 va a contar las veces que el *timer2* sufre desbordamiento.

Encoder

Para entender cómo funciona esta función primero hay que entender el funcionamiento de un *encoder*, concretamente un *encoder* rotativo incremental.

Un *encoder* rotativo incremental es un dispositivo que va a proporcionar un pulso cada vez que gira un determinado ángulo. El ángulo viene determinado por el número de pulsos por vuelta que proporciona el *encoder*, si este número es de 1024 entonces se sabe que un pulso implica un giro de 360/1024=0,35º. Es bastante habitual que dispongan de dos salidas, estas salidas se encuentran desplazadas 90º eléctricos y el uso de las dos permite aumentar la precisión del *encoder* y determina el sentido de giro (horario o antihorario).

En este caso puesto que no se busca una gran precisión y el sentido de giro lo va determinar el motor, que se comporta de forma conocida, solo se va a utilizar una de las salidas del *encoder*.

Se quiere que la función *encoder* se ejecute cada vez que se produce un pulso del *encoder*, para ello en la inicialización (*setup*) está la siguiente línea:

```
attachInterrupt(digitalPinToInterrupt(ENCODER_PIN), encoder, RISING);
```

La función *encoder* va a determinar el tiempo que ha transcurrido entre dos pulsos del *encoder*, lo que va a permitir obtener la velocidad a la que se mueve el coche.

Para ello lo primero que hace es determinar si el pulso que se acaba de recibir es el primero que emite el coche desde que está parado, si es el caso entonces se ignora y se cambia el estado de la variable *first_rising.* Esto es necesario ya que se va a obtener el tiempo entre dos pulsos, si el coche estaba parado entonces el tiempo obtenido no daría información sobre la velocidad del coche sino del tiempo que este ha estado parado.

En el caso de que no sea el primer pulso se obtiene entonces el número de veces que el *timer2* ha contado el paso del tiempo, para ello se multiplica por 255 el número de veces que se ha desbordado (recordemos que el contador desborda cuando llega a 255) y se le suma el valor en el que se encuentra el contador. Este valor se divide entre 10 para un mejor funcionamiento del programa.

Una vez obtenido este valor se vuelven a poner los contadores a 0 para obtener el tiempo hasta el siguiente pulso.

onSteeringCommand

Esta función está suscrita al *topic steering*. Cuando recibe un valor, que está comprendido entre -20 y 20, va a girar el servomotor en función del valor obtenido. El valor recibido determina el ángulo con el que giran las ruedas, los ángulos negativos implican que el coche gira a la izquierda y los ángulos positivos implican que el coche gira a la derecha.

Debido a la construcción del coche el servomotor no puede girar todo lo que debería poder ya que las ruedas chocan con otras partes del vehículo. Es por esto que hay que calibrarlo para determinar hasta que ángulo puede girar, si se solicita al servomotor girar más allá de sus restricciones físicas se podría llegar a dañarlo.

Para este coche los valores obtenidos se observan en la línea

```
servo_pw = map(cmd_msg.data, 20, -20, 635, 1465);
```

Lo que hace esta línea es transformar el dato del *topic* de valor entre 20 y -20 a un valor entre 635 y 1465. La obtención de los valores entre 635 y 1465 se obtienen realizando los siguientes pasos:

Se cierran todos los terminales del ordenador y se abren tres nuevos OT1, OT2 y OT3

Se conectan los tres terminales a la Odroid con el comando

```
ssh root@192.168.1.199
```

En OT1 se paran todos los comandos ROS que se estén ejecutando (pkill ros) y se inicia el roscore

En OT2 se conecta al Arduino escribiendo el siguiente comando:

```
rosrun rosserial_python serial.node.py _port:=/dev/ttyArduino
_baud:=500000
```

En OT3 se publican valores al *topic /steering* usando el siguiente comando:

```
rostopic pub /steering std_msgs/Float32 "data: X"
```

Donde X va a ser el valor deseado entre -20 y 20. Se aconseja empezar escribiendo un valor de 0 ya que este valor debería ser el que hace que las ruedas no estén giradas. Una vez hecho esto se va reduciendo el valor en pequeños pasos (de 5 en 5 por ejemplo) lo que va a ir girando las ruedas poco a poco. Va a haber un cierto punto en el cual las ruedas ya no pueden girar más porque se encuentran bloqueadas, incluso es posible que el servomotor empiece a hacer ruido. En ese momento hay que volver a dejar el servomotor en el último punto en el cual no estaba bloqueado y apuntar el valor.

Ejemplo: se pone un valor de 10 y el servomotor gira sin problema, al escribir un valor de 15 el servomotor pasa a estar bloqueado. Entonces hay que volver a poner el valor de 10 para evitar forzar el servomotor y apuntar el valor.

La función `map` hace una interpolación lineal por tanto el valor que se introduce al servomotor es el siguiente:

$$Valor\ servomotor = \frac{1465 - 635}{40} * (X + 20) + 635 \quad (1)$$

Por tanto, para lograr que al introducir un 20 se llegue al valor límite del servomotor hay que sustituir el valor de 1465 escrito en la función `map` por el Valor servomotor que se obtiene para el *X* que da el límite. En el caso del ejemplo sería:

$$Valor\ servomotor = \frac{1465 - 635}{40} * 30 + 635 = 1258 \quad (2)$$

Y habría que sustituir la línea del código de Arduino por:

```
servo_pw = map(cmd_msg.data, -20, 20, 635, 1258);
```

El valor inferior de la interpolación (635) se obtiene de forma parecida. Una vez obtenido los dos valores hay que modificar la línea del código en el Arduino IDE, compilarlo y volver a subirlo al Arduino. Después se vuelve a realizar la prueba de calibración, si al enviar -20 o 20 se observa que el servomotor no está forzado entonces se ha realizado correctamente.

onSpeedCommand

La función *onSpeedCommand* está suscrita al *topic speed* gracias al cual la Odroid envía al Arduino la velocidad a la que debe funcionar. El valor enviado por la Odroid debe ser un valor entre 0 y 1020.

Para comprender el funcionamiento de esta función hay que entender cómo funciona el Pololu MD07b y como está conectado al Arduino y al motor (ver la Ilustración 15 y la Ilustración 18).

El pin $\overline{RESET}$ si no está alimentado pone al Pololu en estado de bajo consumo. Es por eso por lo que el pin 7 que está conectado a este pin se encuentra siempre en estado HIGH.

El pin PWM es el que va a determinar la velocidad a la que gira al motor. Este pin que está relacionado con el timer2 funciona de forma diferente a los demás ya que lo que se va a hacer es controlar durante que porción del tiempo del timer2 el pin se encuentra en estado HIGH y en estado LOW. Si OCR2A vale 255 la salida será todo el tiempo HIGH, si OCR2A vale 0 la salida será todo el tiempo LOW (hay que recordar que el timer2 funciona a una determinada frecuencia y es capaz de contar hasta 255).

A mayor valor de OCR2A más rápido girara el motor. Si el valor recibido es superior a 0 entonces el coche ira hacia delante, si el valor recibido es inferior a 0 entonces el coche ira hacia detrás y si el valor es 0 el coche permanecerá parado.

Esto ocurre gracias a la tabla de verdad del Pololu MD07b que se observa a continuación:

PWM	DIR	OUTA	OUTB	MARCHA MOTOR
H	L	L	H	DELANTE
H	H	H	L	DETRAS
L	X	L	L	FRENADO

setup

Es la función que se ejecuta nada más iniciar el Arduino. Permite inicializar el resto de funciones y los modos de los *pins* utilizados.

loop

Es la función principal que se ejecuta en bucle mientras el Arduino está alimentado. Esta función va a ser la encargada de obtener la velocidad a la que se mueve el coche gracias al *Encoder*.

El publicador *pubTwist* va a publicar en el *topic twist* la velocidad de giro en rad/s a la que están girando las ruedas. Este valor puede deducir a partir de la variable *deltatime* obtenida gracias a la función *Encoder*.

Esta deducción se puede realizar de forma experimental o forma teórica:

Forma teórica

La variable *deltatime* almacena la cantidad de *ticks* del *timer2* dividida entre 10 que ocurren entre dos pulsos del *encoder*. Sabiendo que el *timer2* funciona con una frecuencia de 7812,5 Hz, es decir un periodo de $\frac{1}{7812,5} = 1,28 * 10^{-4} s$ y que el *encoder* posee 1024 divisiones se puede obtener el tiempo que el motor tarda en dar una vuelta con la formula siguiente:

$$T = \frac{s}{vuelta} = \frac{numero\ de\ ticks}{division\ encoder} * \frac{s}{tick} * \frac{division\ encoder}{vuelta}$$

Sabiendo que:

- El valor obtenido por *deltatime* es $\frac{numero\ de\ ticks}{division\ encoder}$ dividido entre 10
- El numero de $\frac{s}{ticks}$ es la inversa de la frecuencia, el periodo
- $\frac{division\ encoder}{vuelta}$ es el numero de divisiones del *encoder* (que vale 1024)

De aquí todos los parámetros son conocidos desde el principio salvo $\frac{numero\ de\ ticks}{division\ encoder}$ que obtiene el *encoder* a partir del Arduino.

Sin embargo, como el *encoder* está acoplado al motor y no a las ruedas este periodo de giro no da la información a la que giran. Para saberlo sería necesario conocer el ratio de giro entre el motor y las ruedas que puede obtenerse mirando los engranajes que los unen. Debido a que el acceso a estos engranajes era complicado sin tener que desmontar todo el coche decidí utilizar la forma experimental para obtener el valor que relaciona *deltatime* con la velocidad de giro de las ruedas.

Forma experimental

Para la forma experimental lo que se ha realizado es modificar la línea de código

```
twist_msg.linear.x = deltatime*0.006
```

Por la siguiente:

```
twist_msg.linear.x = deltatime
```

De esta forma el mensaje que se publicara al *topic twist* no es la velocidad de giro si no el valor de *deltatime*. Una vez modificado el código se sube al Arduino.

A continuación se realiza una marca fácilmente identificable en una de las ruedas del coche (con un bolígrafo por ejemplo).

Ilustración 32: Marca realizada en la rueda para calibrar el *encoder*

Antes de realizar el siguiente paso hay que asegurarse de que las ruedas del coche no están en contacto con el suelo, es recomendable apoyar el vehículo en una caja para que rueden en el aire.

Después se realizan los pasos para conectar la Odroid al Arduino que se realizaron para la calibración del *encoder* (apartado de la función *onSteeringCommand*) cambiando el comando que se escribía en OT3 por el siguiente:

```
rostopic pub speed std_msgs/Int16 "data:  100" --once
```

Con este comando las ruedas deberían empezar a girar lentamente. Se abre una cuarta consola CT1 y se escribe el siguiente comando:

```
rostopic  echo /twist
```

En el terminal aparece entonces el valor de twist que al haber modificado el código el valor que se observa es el de *deltatime*.

Una vez todo esto está preparado hay que coger un cronometro, empezar a calcular el tiempo cuando la marca realizada en la rueda pase por una determinada posición y parar cuando la marca vuelva a pasar por ese punto.

Con esto se tiene entonces el tiempo que tarda en dar una vuelta la rueda con el respectivo valor de *deltatime*. Es recomendable repetir este proceso varías veces ya que el valor de *deltatime* puede variar un poco de la misma forma que el tiempo obtenido por el cronometro. Una vez realizado el proceso varias veces, eliminar aquellos valores que se

diferencien mucho de los demás que seguramente sean culpa de un error de medida y realizar la media con los restantes.

Esto se ha realizado publicando un valor de 100 en el *topic /speed.* Realizar este proceso para otros valores también, hasta que las ruedas giren a una velocidad a partir de la cual no es posible de distinguir con claridad cuando la marca vuelve a pasar por el sitio observado.

En la tabla siguiente se observan los valores que obtuve durante la calibración del coche y con los que obtuve el valor de 0,006. Dividiendo el tiempo que tarda en dar una vuelta por deltaTime se obtiene el cociente por el cual hay que multiplicar deltaTime para saber el tiempo que tarda en girar la rueda.

Valor motor	Tiempo para dar una vuelta (s)	deltaTime	Tiempo/deltaTime
100	6,5	1000	0,0065
150	3,42	590	0,0058
200	2,4	400	0,006
250	1,84	300	0,0061
300	1,5	250	0,006

Con esto se obtiene el tiempo que tarda la rueda en dar una vuelta.

$$T = \frac{s}{vuelta} = deltatime * 0{,}006 \tag{3}$$

Si se quieren obtener los rad/s se obtienen con la siguiente ecuación:

$$\omega = \frac{rad}{s} = \frac{vuelta}{s} * \frac{rad}{vuelta}$$

$$\omega = \frac{2\pi}{T} \tag{4}$$

Si se quiere obtener la velocidad del coche hay que tener en cuenta que el diámetro D de las ruedas es de 6 cm:

$$v = \omega \frac{D}{2} \tag{5}$$

6) RESULTADOS EXPERIMENTALES

Una vez realizados todos los pasos anteriores, el coche autónomo ha quedado listo para ser programado y ser utilizado como réplica de los coches autónomos proporcionados por SEAT para el *"SEAT autonomous driving challenge"*.

En este capítulo se pretende mostrar cómo se han calibrado y programado los distintos programas de prueba para determinar que el coche realmente puede conducir de forma autónoma. Estos programas pretenden servir como plantilla para los próximos usuarios dando ejemplos de cómo hacer los programas que utilizan los diferentes sensores y actuadores.

6.1. CREACIÓN DE UN ESPACIO DE TRABAJO ROS

El primer paso que hay que realizar antes de empezar a programar en ROS es crear un espacio de trabajo ROS en la Odroid y en el ordenador. Es necesario crearlo también en el ordenador ya que la Odroid carece de interfaz gráfica y para programar los programas que usan la cámara es necesario poder observar las imágenes obtenidas y ver cómo están siendo manipuladas por el programa. Se van a describir los pasos necesarios para crear el espacio de trabajo en la Odroid que, en este caso, son los mismos que para crearlo en el ordenador.

El primer paso es conectarse con la Odroid escribiendo en el terminal

```
ssh root@192.168.1.199
```

Una vez conectado se crea el espacio de trabajo el cual se va a almacenar en la carpeta *catkin_ws* . Para ello se escribe en el terminal lo siguiente:

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/
catkin_make
source devel/setup.bash
```

Para poder utilizar los programas guardados en este espacio de trabajo utilizando el comando *rosrun* es necesario modificar el archivo *.bashrc*. Para ello se escribe en el terminal:

```
cd
nano .bashrc
```

Se abre un editor de texto y se añade la siguiente línea al final del fichero:

```
source ~/catkin_ws/devel/setup.bash
```

6.2. CREACIÓN DEL PAQUETE *SIGUE_LINEAS*

Una vez creado el espacio de trabajo ya se pueden crear los paquetes con los que se va a trabajar. En este caso se crea el paquete que se va a encargar de dirigir a la cámara para que el coche siga las líneas de la carretera. Este paquete va a tener varias dependencias, las cuales se incluyen desde su creación para que resulte más fácil. Se pueden añadir nuevas dependencias más tarde modificando el fichero *CMakeLists.txt* y *package.xml*, por lo que no es necesario incluir todas desde el comienzo.

Para crear el paquete *sigue_lineas* se escribe lo siguiente en el terminal

```
cd ~/catkin_ws/src
```

```
catkin_create_pkg sigue_lineas camera_info_manager cv_bridge
image_transport message_runtime roscpp rospy sensor_msgs std_msgs
geometry_msgs
```

Estas dependencias son las que permiten a este paquete obtener información de la cámara y poder trabajar así con las imágenes recibidas. Las dependencias *std_msgs* y *geometry_msgs* son las que permiten enviar y recibir mensajes al Arduino para poder controlar el motor y el servomotor.

Debido a que este paquete va a utilizar la biblioteca OpenCV para tratar las imágenes es necesario añadir también el paquete de OpenCV. Para ello se escribe en el terminal:

```
cd ~/catkin_ws/src/sigue_lineas
nano CMakeLists.txt
```

Este comando abre con el editor de texto *nano* el documento CMakeLists.txt. A este documento hay que añadirle la siguiente línea:

```
find_package(OpenCV)
```

Además, hay que modificar la sección "*include_directories*" para que incluya también OpenCV. El texto debería quedar de la siguiente forma:

```
include_directories(
# include
  ${catkin_INCLUDE_DIRS}
  ${OpenCV_INCLUDE_DIRS}
)
```

Este paquete contiene dos programas: *camera_publisher.cpp* y *camera_sigue_lineas.cpp* es por ello que al final de *CMakeLists.txt* hay que indicarlo escribiendo:

```
add_executable(camera_publisher src/camera_publisher.cpp)
target_link_libraries(camera_publisher ${catkin_LIBRARIES}
${OpenCV_LIBRARIES})


add_executable(camera_sigue_lineas src/camera_sigue_lineas.cpp)
target_link_libraries(camera_sigue_lineas ${catkin_LIBRARIES}
${OpenCV_LIBRARIE$
```

6.2.1. PROGRAMA *CAMERA_PUBLISHER*

El programa *camera_publisher* que se puede leer en el Anexo 2 es el encargado de obtener las imágenes de la cámara y publicarlas en forma de *topic*.

Este programa crea el nodo *image_publisher* el cual pública en el *topic camera/image* las imágenes obtenidas por la cámara. Las imágenes de la cámara son obtenidas en el programa gracias a la biblioteca OpenCV, sin embargo, no se pueden publicar estas imágenes en ROS directamente. Es necesario utilizar la biblioteca *cv_bridge* para poder publicar las imágenes en un *topic*.

6.2.2. PROGRAMA *CAMERA_SIGUE_LINEAS*

El programa *camera_sigue_lineas* que se puede leer en el Anexo 3 es el encargado de procesar las imágenes obtenidas por la cámara para determinar cómo debe girar el coche para mantenerse dentro del carril.

Para realizar estas acciones el programa se subscribe al *topic camera/image* en el cual el programa *camera_publisher* publica la información de la cámara. Para controlar el servomotor, el programa publica en el *topic steering* al cual está subscrito el Arduino.

La función *imageCallback* es la encargada de procesar la imagen y determinar el ángulo necesario del servomotor. Lo primero que realiza es transformar la información del *topic camera/image* en una matriz BGR (blue green red) que puede ser utilizada por OpenCV que se puede observar en la Ilustración 33.

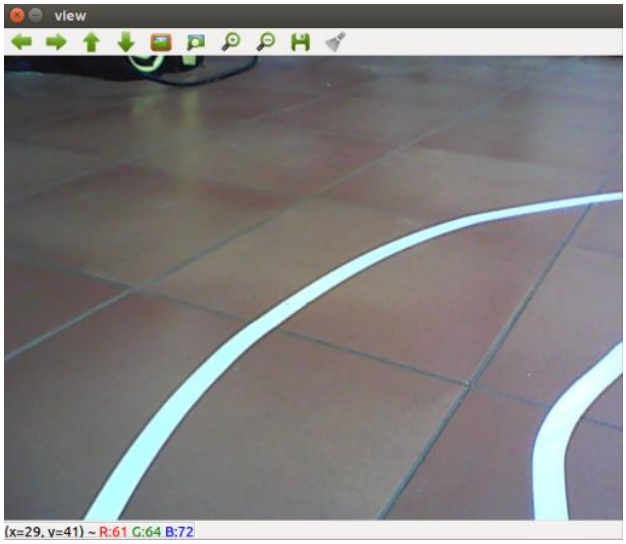

Ilustración 33: Imagen original obtenida por la cámara USB

La imagen obtenida se pasa a una matriz en escala de grises utilizando la función *cvtColor* (ver Ilustración 34). En este tipo de escala los pixeles solo tienen un valor a diferencia de las matrices BGR en las cuales los pixeles están compuestos de tres valores diferentes.

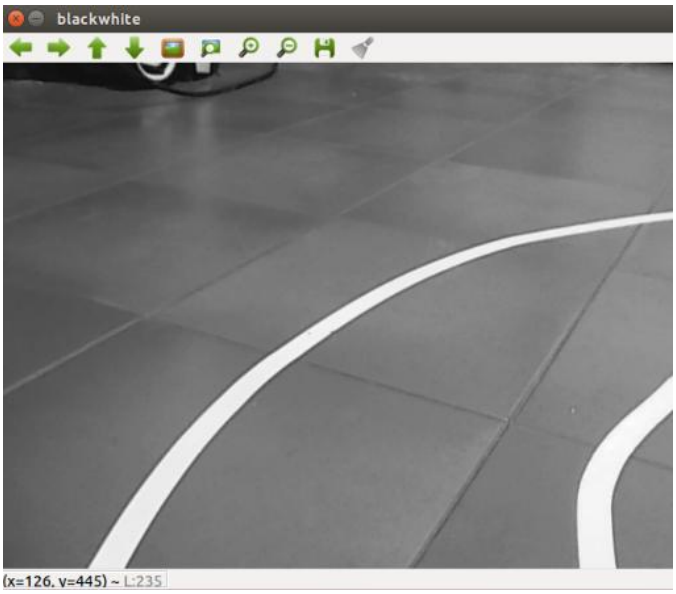

Ilustración 34: Imagen convertida a escala de grises

Este cambio se realiza para poder aplicar la función *threshold* a la imagen. La función *threshold* tiene como objetivo transformar la imagen en escala de grises en una imagen binarizada con solo blancos y negros. En esta escala, los pixeles blancos tienen el valor máximo (255) y los pixeles negros tienen el valor mínimo (0).

La función *threshold* tiene 5 modos distintos de funcionamiento, como se puede observar en la página de OpenCV [12]. En este caso se va a utilizar el método *threshold binary inverted*. Este modo lo que hace es que los pixeles con un valor superior al valor límite seleccionado por el usuario pasan a tener un valor de 0 mientras que los pixeles que tienen un valor inferior pasan a tener un valor máximo, también definido por el usuario. Esta función se representa con la siguiente expresión:

- Si *ValorPixel > Límite* entonces *ValorPixel = 0*
- Si *ValorPixel <= Límite* entonces *ValorPixel=ValorMáx*

El objetivo de obtener esta imagen en blanco y negro es que las líneas que delimitan el carril de la carretera tengan un color distinto al del resto de la imagen para así poder obtener fácilmente dónde se encuentran. Esto se puede observar en la Ilustración 35.

El siguiente paso, por tanto, es determinar cuál es el valor límite a seleccionar. Para ello hay que tener en cuenta que las líneas de la carretera son blancas y por tanto tendrán valores de 255. Sin embargo, debido a las condiciones de iluminación de donde se estén realizando las pruebas este valor no va a ser exactamente de 255. Por tanto, la mejor manera de obtener este valor es obtener la imagen en escala de grises de la carretera y observar cual es el valor de los pixeles. Hay que tener cuidado ya que si se escoge un valor límite muy alto podría haber ciertas zonas de las líneas menos iluminadas que tengan un valor inferior y no se reconocerían como tales. Por otro lado, un valor límite muy pequeño tendría el efecto contrario provocando que el programa interprete que ciertos pixeles forman parte de las líneas de la carretera cuando no es así, como se puede observar en la Ilustración 36.

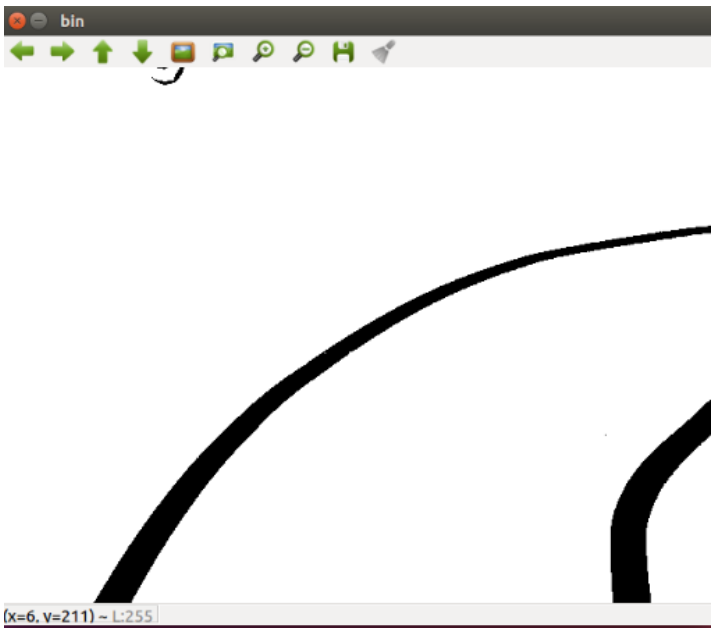

Ilustración 35: Imagen binarizada. Líneas de la carretera en negro

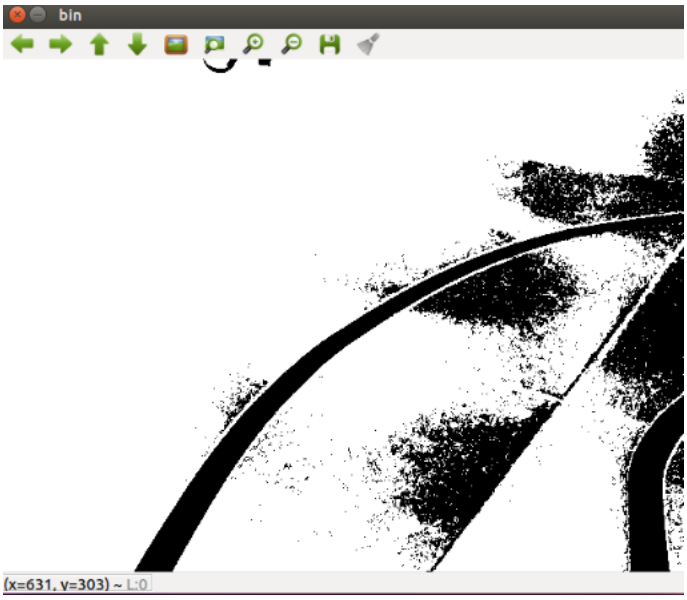

Ilustración 36: Imagen binarizada con un valor límite incorrecto

Una vez obtenidas las imágenes binarizadas que permiten separar los pixeles de las líneas de la carretera de los demás, falta el algoritmo para determinar la dirección del coche.

El primer objetivo, por tanto, es determinar qué dirección tiene la carretera. Para ello, se toman dos filas de la matriz de la imagen binarizada y se obtiene el "centro de gravedad" de las marcas viales. Este "centro de gravedad" se obtiene sumando la posición en el eje horizontal de los pixeles de las líneas de la carretera y dividiendo este sumatorio por el número estos pixeles que hay en esa fila. Este cálculo se puede expresar con la siguiente fórmula:

$$CdG = \frac{\sum_{i=0}^{c} x_i}{255 * n} \tag{6}$$

Siendo:

- c: el número de columnas de la matriz
- x: el valor del pixel numero i de la fila seleccionada. Valiendo 255 aquellos que corresponden con una línea de la carretera y 0 los demás.
- *n*: el número de pixeles de la fila seleccionada correspondientes a líneas de la carretera

Utilizando esta fórmula se obtiene el punto central del carril en la fila escogida.

Se repite este proceso con otra fila de la matriz obteniéndose así el punto central del carril en dos zonas diferentes. Se puede observar la obtención de estos centros de gravedad en la Ilustración 37.

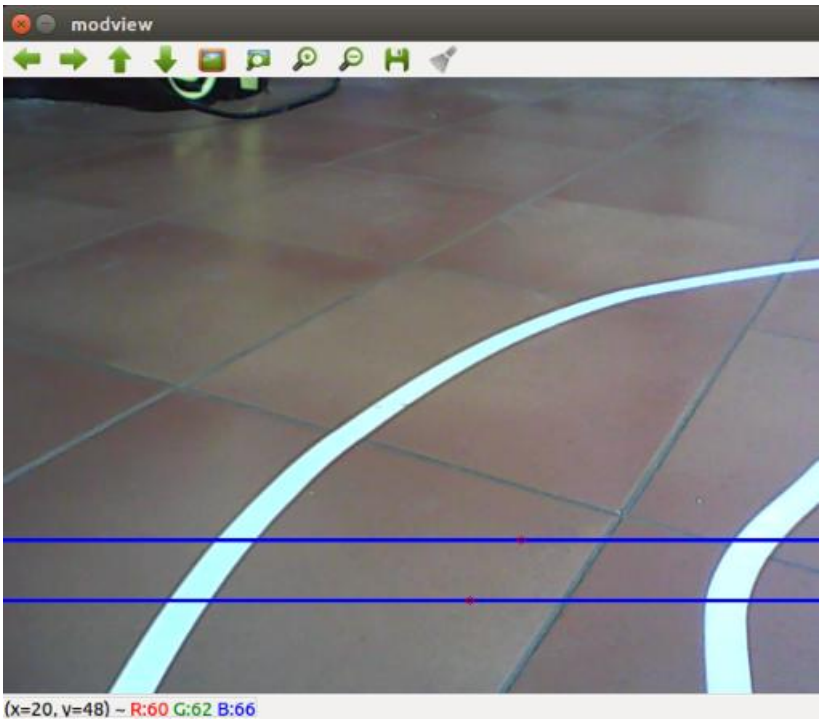

Ilustración 37: Centro de gravedad de las líneas de la carretera

Uniendo estos dos puntos se crea una recta que determina la dirección de la carretera. Para determinar la dirección que debe tomar el servomotor, se usa un controlador proporcional que toma como parámetros la distancia del coche a la recta y el ángulo de la recta. La variable de la distancia es la que va a permitir al coche ir centrado en la carretera mientras que la variable del ángulo es la que le va a permitir ir con la misma dirección que la carretera.

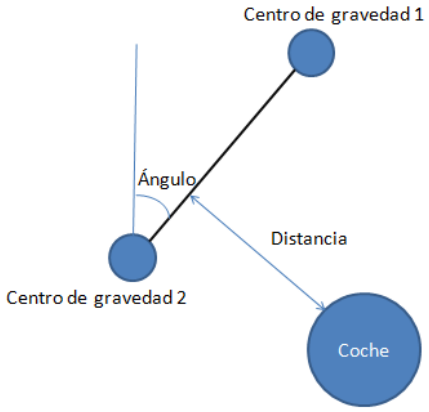

Ilustración 38: Distancia y ángulo del coche a la dirección de la carretera

Para obtener los parámetros del ángulo y la distancia hay que tener en cuenta que ya se conoce la posición de los dos centros de gravedad y que se considera que el coche se encuentra en la columna central de la matriz en la fila inferior.

Por tanto, hay que utilizar las distancias verticales y horizontales de los centros de gravedad respecto del coche como se pueden observar en la Ilustración 39.

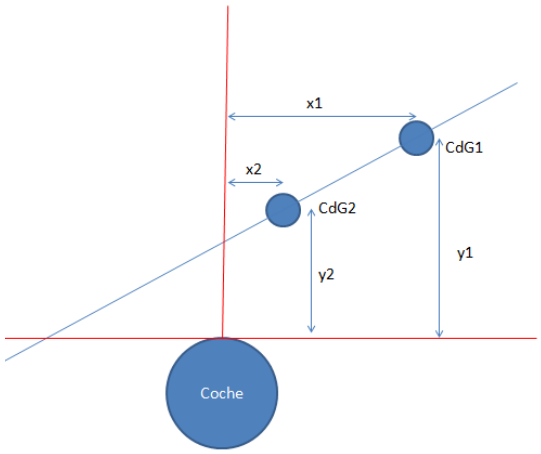

Ilustración 39: Distancias de los centros de gravedad al coche

Aplicando trigonometría se obtiene la distancia y el ángulo usando las siguiente fórmulas:

$$angulo = \text{atan}\left(\frac{x_1 - x_2}{y_1 - y_2}\right) \quad (7)$$

$$distancia = (x_2 - \tan(angulo) * y_2) * \cos(ang) \quad (8)$$

Una vez se tienen estos parámetros se obtiene el ángulo que hay que enviar al servomotor utilizando la siguiente fórmula:

$$servomotor = K_a * angulo + K_d * distancia \quad (9)$$

Siendo K_a y K_d coeficientes obtenidos empíricamente hasta seguir correctamente la carretera.

6.2.3. VERIFICACIÓN DEL PAQUETE *SIGUE_LINEAS*

Los códigos del paquete *sigue_lineas* hay que pegarlos dentro de la carpeta *src* que se encuentra en la carpeta *sigue_lineas*. Una vez hecho esto hay que compilar el código, para ello se escribe en el terminal:

```
cd ~/catkin_ws
catkin_make
```

Una vez realizado esto, el coche está listo para realizar el seguimiento de las líneas. Para ello lo primero es realizar una carretera con alguna curva para comprobar que sigue correctamente las líneas. Es aconsejable utilizar una superficie oscura y con unas condiciones de iluminación en las cuales la luz no refleje demasiado en el suelo. Después, utilizando cinta aislante blanca se crean las líneas que delimitan la carretera.

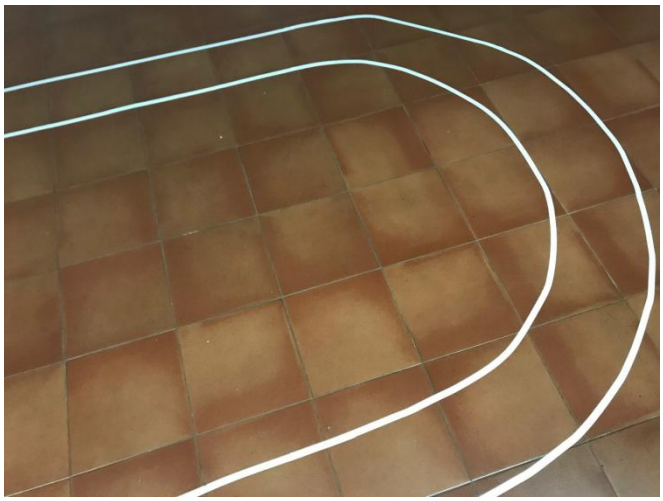

Ilustración 40: Ejemplo de carretera de pruebas

El programa *sigue_lineas* requiere de 5 nodos diferentes para funcionar:

- roscore
- rosserial
- camera_publisher
- camera_sigue_lineas
- speed

Lanzar estos nodos de forma normal requeriría trabajar con 5 terminales diferentes lo cual no es óptimo. Es por lo que que en ROS existen los ficheros *launch.* Los ficheros *launch* son una herramienta de ROS que permite ejecutar múltiples nodos escribiendo un solo comando en un terminal. Esto permite ahorrar tiempo y facilita el trabajo al usar solo un terminal. Para crear este tipo de ficheros es necesario crear una carpeta denominada *"launch"* dentro del paquete *sigue_lineas.* El fichero *launch* utilizado para este proceso se ha guardado como *camera.launch y* se encuentra en el Anexo 4. Para ejecutarlo hay que escribir el siguiente comando en el terminal:

```
roslaunch sigue_lineas camera.launch
```

Al ejecutar el fichero *launch* aparece en la pantalla del ordenador la imagen obtenida por la cámara y todas sus transformaciones. Hay que utilizar esta información para calibrar la función *threshold* del programa *camera_sigue_lineas* y ajustar la iluminación de la sala. Para ello se observa la pantalla *blackwhite* (Ilustración 34) y pasando el ratón por encima de la imagen aparece cual es el valor de los pixeles en escala de grises. Hay que anotar cuales son los valores entre los cuales varían los pixeles de las líneas de la carretera y entre cuales varían el resto. Para que el programa pueda funcionar no puede haber valores que coincidan entre ambos intervalos.

En el caso de la sala utilizada para realizar las pruebas existen dos posibilidades de iluminación en las cuales los valores de los pixeles corresponden a los que se observan en la Ilustración 41.

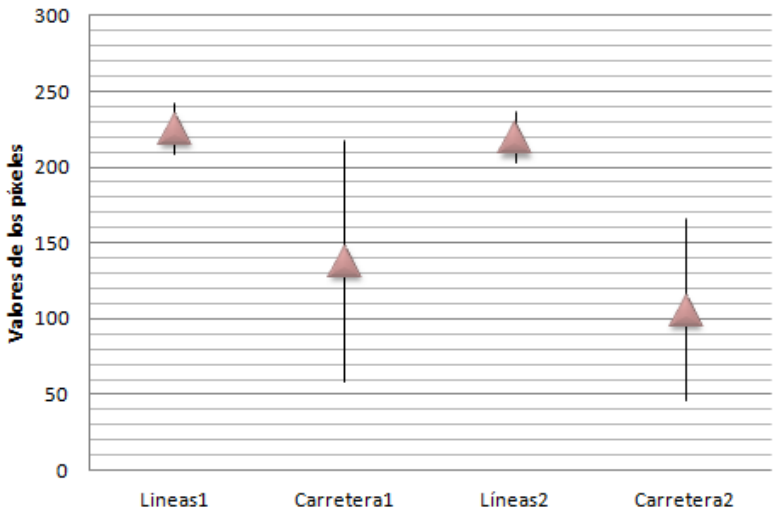

Ilustración 41: Valores de los pixeles en función de la iluminación

Se puede observar como en el caso de la iluminación 1 los valores de los pixeles se cruzan por lo que no se puede utilizar, sin embargo en el caso de la iluminación 2 los valores de los pixeles se encuentran bien diferenciados. Con estos resultados quedo expuesto que la iluminación a utilizar es la número 2 y se puso como límite superior de la función *threshold* 200 ya que todos los píxeles de las líneas tienen un valor superior a este número mientras que el resto de píxeles tienen un valor inferior. Una vez seleccionado el valor de *threshold* se vuelve a ejecutar el programa y se observa la pantalla *bin*, si en la imagen obtenida solo se ve en negro las marcas viales (como en la Ilustración 35) entonces se ha calibrado correctamente.

El siguiente paso es calibrar K_d y K_a. Para ello hay que tener en cuenta lo siguiente:

- K_a afecta al seguimiento de las curvas, un valor demasiado pequeño provocara que el coche no gire lo suficiente mientras que un valor demasiado grande provocara que el coche gire demasiado.

- K_d afecta al centrado del coche en la carretera, un valor demasiado grande provocará que el coche vaya haciendo un movimiento con forma de "S" intentando centrarse mientras que con un valor demasiado el coche no lograra centrarse en la carretera si en algún momento se descentra.

Para realizar la calibración se va a ejecutar el fichero *launch*. Si en algún momento es necesario frenar el coche (debido a que vaya a colisionar por ejemplo) hay que escribir en otro terminal lo siguiente:

```
rostopic pub speed std_msgs/Int16 "data:  0" --once
```

A continuación, se presentan las iteraciones que se han realizado para calibrar estos valores con las observaciones de cada iteración:

Número de iteración	Valor K_a	Valor K_d	Comentarios
1	1	0	El coche gira en exceso
2	0,8	0	El coche no gira lo suficiente
3	0,9	0	El coche sigue la dirección de la carretera pero

			no va centrado
4	0,9	0,1	El coche se sale de la carretera rápidamente
5	0,9	0,01	El coche se sale de la carretera
6	0,9	0,001	El coche tiene un funcionamiento similar al del caso 3
7	0,9	0,005	El coche sigue la carretera de forma centrada

Para entender mejor el funcionamiento de los programas y como están conectados los distintos topics se puede abrir un nuevo terminal y escribir lo siguiente:

```
rqt_graph
```

Al realizarlo se obtiene la Ilustración 42. En esta se puede observar como el nodo *image_publisher* (nodo del programa *camera_publisher)* publica el *topic /camera* al cual está suscrito el nodo *image_listener* (nodo del programa *camera_sigue_lineas*). El *topic camera* contiene las imágenes obtenidas por la cámara. El nodo *image_listener* a su vez publica en el *topic steering* la información que va a utilizar el Arduino (*serial node)* para controlar el servomotor.

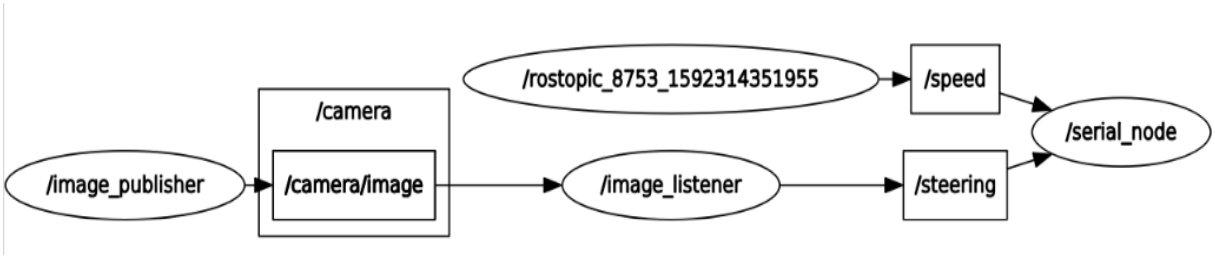

Ilustración 42: Rqt_graph de sigue_lineas

6.3. CREACIÓN DEL PAQUETE *EVITA_OBSTACULOS*

Después de haber creado el paquete *sigue_lineas* se ha creado un paquete que hace uso del lidar con el objetivo de simular el rebasamiento de un obstáculo o el adelantamiento de un coche De esta forma queda documentado como utilizar en un programa los dos sensores más importantes del vehículo autónomo (la cámara y el lidar) y sería posible superar dos de las tres pruebas del *"SEAT autonomous driving challenge"* (que aparecen descritas en el apartado 1.2).

El paquete se crea de la misma forma que se ha creado el paquete *sigue_lineas* teniendo en cuenta que en este caso las dependencias son solo las siguientes:

```
roscpp rospy sensor_msgs std_msgs geometry_msgs
```

6.3.1. CALIBRACIÓN DEL LIDAR

Puesto que se va a utilizar el Lidar para este paquete, es necesario entender cómo funciona. Para ello se añade el programa *client.cpp* (ver Anexo 5) al paquete y se ejecuta junto al *rplidar.launch.*

El programa *client.cpp* se suscribe al *topic /scan* el cual contiene la información transmitida por el *lidar.* Una vez hecho esto muestra por el terminal la información contenida del *topic*

(ver Ilustración 43) que viene a ser: número de puntos obtenidos en una vuelta del lidar, ángulo de cada uno de los puntos y su valor de distancia.

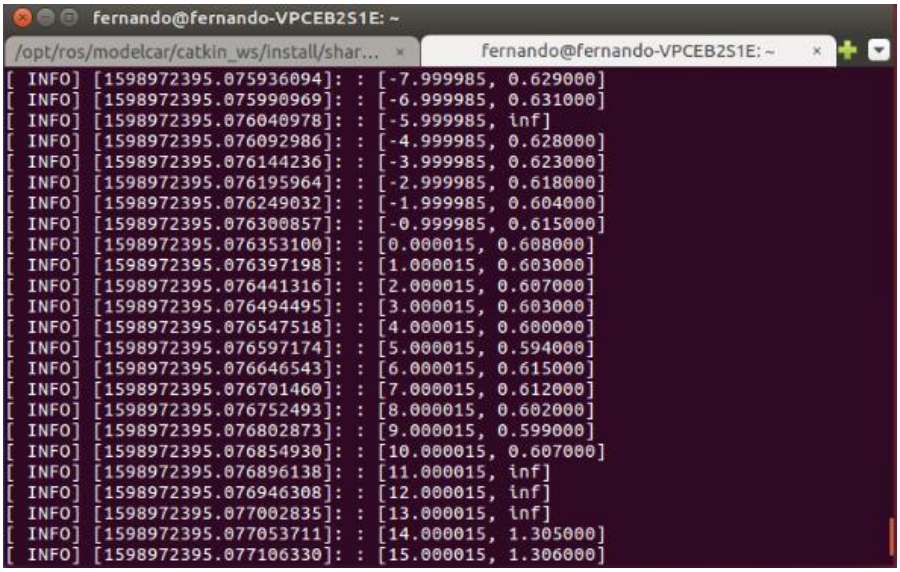

Ilustración 43: Terminal del programa *client.cpp*

Lo primero que se observa es que obtiene 360 puntos, uno por grado de giro y que la matriz con la información empieza por el ángulo de -180º y acaba en 180º. El objetivo de la calibración es determinar a que corresponden esos ángulos en el vehículo (parte frontal, trasera ...). Para ello se coloca un objeto próximo al Lidar y se mueve hasta observar en el terminal que valores están cambiando. Durante la calibración se determinó que para este modelo de lidar el punto de 0º (que es el punto 180 de la matriz de datos) es el punto opuesto al punto de entrada del cable de alimentación del Lidar.

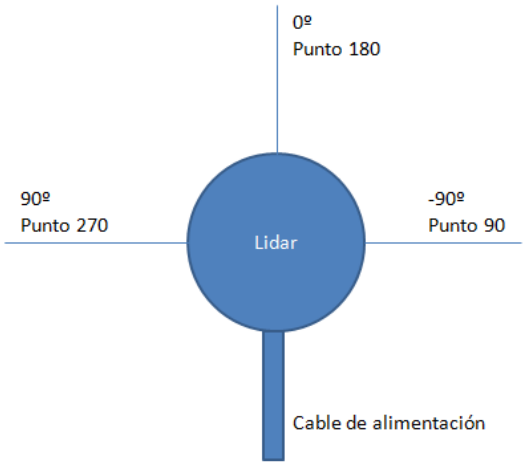

Ilustración 44: Ángulos del Lidar

Por la forma en la que está montado el Lidar en el coche utilizado para este proyecto el ángulo de 90º corresponde con la parte delantera del coche y el de 0º con la parte derecha. Puesto que en el programa se va a trabajar con la matriz de datos es importante tener en cuenta entonces que la información de un punto que se encuentra a Xº se encuentra en la posición X+180 de la matriz.

El siguiente paso de la calibración es determinar a que corresponden los valores de la matriz de datos. Para ello se colocó un objeto a 1 metro del lidar (hay que medir la distancia desde

el lidar hasta el objeto, no desde el coche hasta el objeto) y se obtuvo un valor de 1, después se colocó el objeto a 70 centímetros y se obtuvo un valor de 0,7. Por tanto se pudo deducir que el lidar utilizado ya está calibrado y proporciona la información de la distancia en metros.

6.3.2. PROGRAMA *EVITA*

Una vez el lidar calibrado se procedió a realizar el programa *evita.cpp* (ver anexo 6)que será el encargado de que el coche evite los obstáculos.

Para la realización del programa se ha desglosado el proceso de adelantamiento en 6 maniobras diferentes:

- Maniobra 1: El coche avanza recto por la carretera hasta que el obstáculo que debe adelantar entra en su zona de detección la cual está definida en el programa por el ángulo que comprende y la distancia que alcanza. Esta zona de detección abarca la parte delantera derecha del coche.

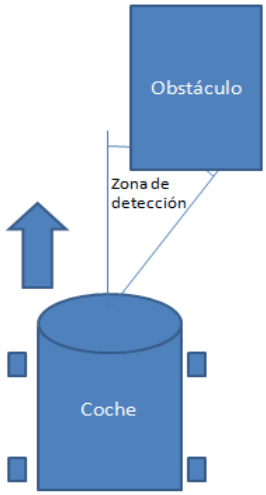

Ilustración 45: Maniobra 1 del coche

- Maniobra 2: Al detectar el obstáculo, el coche procede a girar a la izquierda hasta que el obstáculo sale de su zona de detección (diferente a la de la maniobra 1). Esta zona de detección abarca la parte delantera derecha del coche. Esto implica que el coche ya ha girado lo suficiente como para rebasar el obstáculo. Se realiza una temporización de la duración de esta maniobra.

- Maniobra 3: En el momento en el que el obstáculo sale de la zona de detección del coche este procede a girar a la derecha durante el mismo tiempo que el temporizado por la maniobra 2. De esta forma el coche queda paralelo a su posición antes de la maniobra de adelantamiento situado más a la izquierda.

- Maniobra 4: El coche avanza recto hasta que no detecta obstáculos en su zona de detección (diferente a las dos anteriores). Esta zona de detección abarca la parte delantera derecha del coche. De esta forma el coche avanza paralelo al obstáculo hasta que detecta que ya lo ha adelantado.

- Maniobra 5: Tras adelantar al obstáculo el coche procede a girar a la derecha una cantidad de tiempo igual a la temporizada en la maniobra 2.

- Maniobra 6: Una vez acabada la maniobra 5 el coche procede a girar a la izquierda una cantidad de tiempo igual a la temporizada en la maniobra 2.

Tras acabar estas 6 maniobras el coche se encuentra situado en la misma dirección que antes de comenzar el adelantamiento pero situado al otro lado del obstáculo.

En la Ilustración 46 se observa el diagrama de flujo del adelantamiento.

6.3.3. VERIFICACIÓN DEL PAQUETE *EVITA_OBSTACULOS*

Una vez escrito el programa hay que verificar su funcionamiento. Para eso se coloca delante del coche algún objeto (una caja de zapatos por ejemplo) que represente el obstáculo que debe adelantar. Al igual que para el paquete *sigue_lineas* se hace un fichero *launch* que en este caso está compuesto por *rplidar.launch,* el programa *evita.cpp* y una publicación al topic */speed* para que el coche avance.

Esta verificación no solo sirve para comprobar que el programa realiza las acciones deseadas sino también para calibrar las distintas zonas de detección en las cuales hay que modificar la distancia y el ángulo (se tomaran como ángulos los que se observan en la Ilustración 44 teniendo en cuenta la posición relativa que tiene el lidar frente al coche tal y como se describe en el apartado 6.3.1).

La zona de detección 1 determina cuando el coche percibe el obstáculo e inicia la operación de adelantar. Para esta zona lo más importante es el parámetro de la distancia puesto que el del ángulo vale cualquier rango que comprenda la zona delantera del coche, en este caso se ha escogido entre 45º y 90º. A continuación, se exponen los distintos valores de distancia con los que se ha iterado:

Número iteración	Distancia (en metros)	Valoración
1	1	El coche gira demasiado pronto. No se ajusta a un adelantamiento realista.
2	0,5	El coche empieza el giro demasiado tarde y golpea contra el obstáculo.
3	0,65	El coche logra iniciar el giro a una distancia razonable y logra realizar el inicio del giro sin golpear al obstáculo.

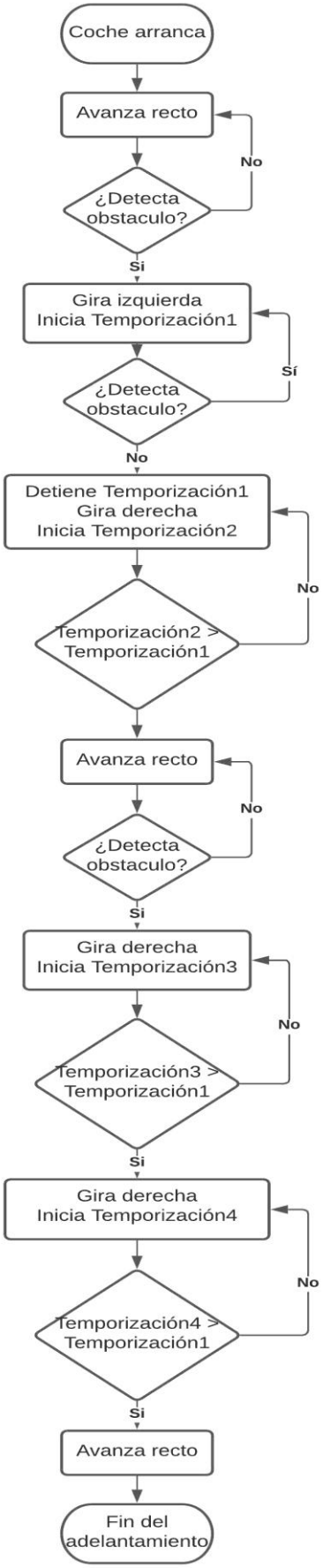

Ilustración 46: Diagrama de flujo adelantamiento

La zona de detección 2 determina a partir de qué momento el coche procede a girar en el otro sentido para ponerse paralelo al obstáculo. Empieza a girar en el momento en el que no detecta obstáculo en esa zona, en este caso la distancia no tiene importancia puesto que el coche se encontrara cerca del objeto. Valores con los que se ha iterado:

Numero de iteración	Rango del ángulo	Comentarios
1	0º - 90º	El coche tarda demasiado en iniciar la maniobra alejándose demasiado del obstáculo.
2	45º - 90º	El coche adelanta de forma correcta
3	60º - 90º	El coche inicia demasiado pronto el giro a la derecha y golpea contra el obstáculo.

Con la iteración 2 el coche realizaba la maniobra correctamente, sin embargo, se quiso apurar más la maniobra con la iteración 3. Finalmente se guardaron los valores de la iteración 2.

Finalmente queda por calibrar la zona de detección 3 que determina a partir de qué momento se considera que el coche ha rebasado el obstáculo y puede por tanto empezar a girar para recuperar su posición en el carril derecho. Los valores con los que se ha iterado son los siguientes:

Numero de iteración	Rango del ángulo	Comentarios
1	45º - 90º	El coche intenta la maniobra demasiado pronto y se golpea contra el obstáculo.
2	0º - 90º	El coche adelanta de forma correcta.

Tras calibrar las tres zonas de detección el coche logra realizar la maniobra de adelantamiento de forma correcta sin golpear el obstáculo.

6.4. CREACIÓN DEL PAQUETE *APARCAMIENTO*

Tras haber creado el paquete *evita_obstaculos* se procedió a crear el programa *aparcamiento.cpp* (ver anexo 7) en el paquete *aparcamiento*.. Las dependencias del paquete son las siguientes:

```
roscpp rospy sensor_msgs std_msgs geometry_msgs
```

6.4.1. PROGRAMA *APARCAMIENTO*

El objetivo de este programa es que el coche realice un aparcamiento en paralelo entre dos coches. Este programa es similar al utilizado para evitar obstáculos. Al igual que en ese programa, se ha dividido la maniobra en distintos movimientos que se pueden observar con las flechas de colores en la Ilustración 47.

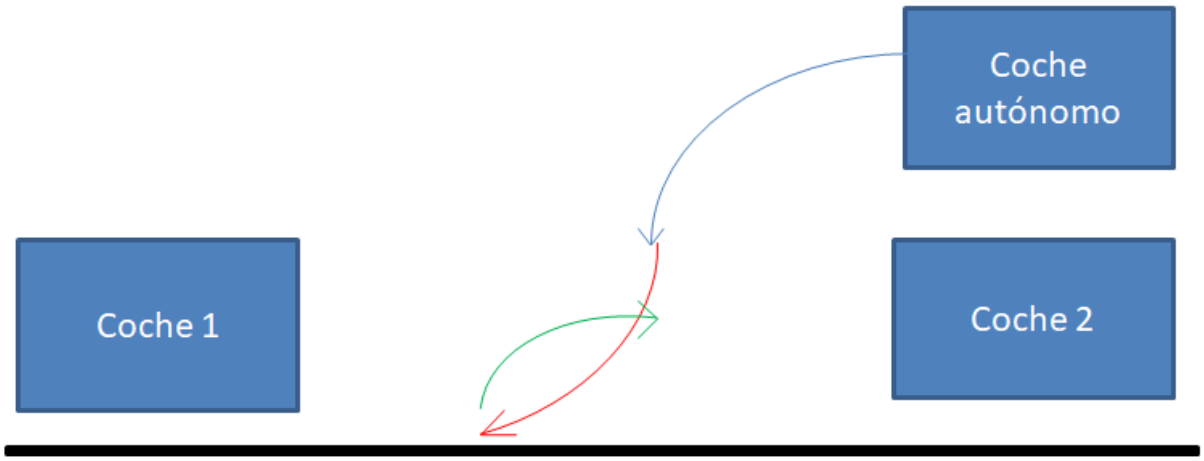

Ilustración 47: Maniobra de aparcamiento

- En la primera maniobra, que se encuentra representada en azul, el coche se mueve hacia detrás girando las ruedas hacia la derecha.
- En la segunda maniobra, representada en rojo, el coche se mueve hacia detrás girando las ruedas hacia la izquierda.
- En la última maniobra, representada en verde, el coche se mueve hacia delante girando las ruedas hacia la derecha.

El coche debe realizar estas maniobras sin sobrepasar la línea negra que representa el borde de la carretera.

Estas maniobras buscan reproducir el comportamiento que tiene habitualmente un conductor a la hora de aparcar en paralelo. Al acabar estas dos maniobras el coche autónomo debería encontrarse entre el coche1 y el coche2 y estar alineado con ellos.

6.4.2. VERIFICACIÓN DEL PAQUETE *APARCAMIENTO*

Después de realizar el programa hay que ajustar los parámetros para que el aparcamiento se realice correctamente.

Para realizar las pruebas se colocan dos objetos que representen al *coche1* y al *coche2* (para este trabajo se han utilizado dos cajas de zapato) y se disponen de forma que quede un espacio entre ellos que será donde el coche tendrá que aparcar. A continuación, se coloca el coche autónomo en paralelo a uno de estos obstáculos de forma que la parte trasera sea la parte más cercana del vehículo al aparcamiento.

Estas maniobras disponen de tres parámetros: tiempo girando, ángulo de giro y velocidad de desplazamiento. Al ser una maniobra que debe realizarse en poco espacio se utilizan los ángulos de giro más altos permitidos por el servomotor. Se escoge una velocidad suficientemente lenta para que el coche maniobre sin riesgo y lo suficientemente rápida para no tardar demasiado en ejecutar los movimiento. Finalmente el valor escogido es de 300 cuando avanza hacia delante y -300 cuando retrocede.

De esta forma quedan fijadas dos de las tres variables. Al ser un proceso secuencial, se ha ajustado el tiempo de cada maniobra en el mismo orden en el que se ejecutan en el programa.

No hay una fórmula exacta para determinar hasta donde debe llegar cada una de las tres maniobras, hay que aplicar el sentido común y fijarse en que el coche se encuentra aparcado correctamente tras realizarlas.

Para la primera maniobra se probaron los siguientes tiempos:

Iteración	Tiempo maniobra 1	Observaciones
1	4 segundos	El coche no gira suficiente. Cuando realice la siguiente maniobra no va a quedar bien situado.
2	6 segundos	El coche gira demasiado. Cuando realice la siguiente maniobra va a superar el límite de la carretera.
3	5 segundos	El coche queda bien situado para las siguientes maniobras.

Para la segunda maniobra se trabajó con los siguientes tiempos

Iteración	Tiempo maniobra 2	Observaciones
1	4 segundos	El coche gira demasiado quedando mal situado.
2	3 segundos	El coche queda en una buena situación para iniciar la siguiente maniobra.

Al finalizar la última maniobra, el coche debe quedar bien aparcado. Se trabajo con los siguientes tiempos:

Iteración	Tiempo maniobra 3	Observaciones
1	1 segundo	El coche no gira lo suficiente como para quedar bien alineado.
2	2 segundos	El coche queda bien situado y alineado con los otros dos.

6.5. CREACIÓN DEL PAQUETE SEMAFORO

El último de los paquetes realizados es el *semaforo.* Este paquete al utilizar la cámara tiene las mismas dependencias que el paquete *sigue_lineas*

6.5.1. PROGRAMA *SEMAFORO*

El objetivo de este programa es que el coche sea capaz de detectar la luz de los semáforos deteniéndose en caso de que se encuentren en rojo.

Para la realización de este programa (ver anexo 8) se han realizado las siguientes hipótesis:

- Los semáforos siempre se sitúan en el lado izquierdo de la carretera.

  Esta hipótesis permite reducir la carga computacional del programa ya que solo es necesario realizar el estudio de la parte izquierda de la imagen en la búsqueda de la luz del semáforo.

- Los semáforos son los únicos objetos rojos con los que el coche se va a cruzar.

  Por la forma en la que se realiza el programa si hubiese otros objetos rojos el coche podría confundirlos con las luces del semáforo.

El programa se suscribe al *topic camera/image* (al igual que el programa *sigue_lineas),* el cual contiene las imágenes obtenidas por la cámara en formato *bgr (blue green red)*. Si el semáforo está en rojo el valor *red* de los pixeles obtenidos será muy alto, sin embargo, está no es condición suficiente para detectar el rojo ya que el color blanco de las líneas de la carretera también tiene un valor *red* elevado. Por tanto para determinar si realmente los pixeles observados son rojos no solo es necesario que su valor *red* sea alto si no que además tienen que tener un valor reducido de *blue* y de *green*.

Teniendo todo esto en cuenta, las acciones que realiza el programa son las siguientes:

- Obtiene el valor *red* de los pixeles situados en la parte izquierda de la imagen.
- Obtiene el valor *blue* y *green* de aquellos pixeles con un valor *red* elevado.
- Si ningún pixel cumple los criterios para ser considerado rojo entonces el coche sigue avanzando (o arranca en caso de que el coche estuviese parado por culpa de haber detectado previamente que el semáforo estaba en rojo).
- Si algún pixel es considerado rojo entonces se para el motor del vehículo.

6.5.2. VERIFICACIÓN DEL PAQUETE *SEMAFORO*

Para verificar el funcionamiento del paquete es necesario disponer de un dispositivo que actué como semáforo. Aunque la idea más intuitiva pueda ser realizarlo con unos LEDs esta no es buena idea ya que cuando la cámara observa directamente los LEDs se satura y los interpreta como blancos. Para este trabajo se ha utilizado un móvil en el cual se muestran imágenes rojas o verdes por la pantalla utilizando un valor poco elevado de luminosidad (para evitar saturar la imagen de la cámara).

Para este paquete los únicos valores que hace falta calibrar son los valores de *red, green* y *blue* utilizados para determinar si un pixel es de color rojo. Estos valores se obtienen fácilmente a través de la pantalla de visualización de imágenes de la cámara como se observa en la Ilustración 31.

7) CONCLUSIONES

7.1. LOGROS TÉCNICOS

Como se ha descrito en el capítulo 1, para el desarrollo de este trabajo se definieron varios subobjetivos. En este apartado se van a describir los logros técnicos obtenidos correspondientes a estos objetivos:

- **Diseño hardware del vehículo**

Este era el primer objetivo del proyecto ya que era necesario conocer el hardware que se iba a utilizar para poder trabajar con él. El proyecto buscaba realizar un coche a escala con los componentes necesarios para que pueda conducir de forma autónoma.

De esta forma se ha desarrollado un vehículo funcional que únicamente necesita una cámara y un lidar para obtener la información de su entorno.

- **Control de alto nivel Odroid. Instalación y programación del módulo ROS**

El siguiente objetivo era acondicionar el ordenador montado en el coche para que funcione correctamente. Se ha instalado en la Odroid todos los programas y bibliotecas ROS necesarios y se ha verificado su capacidad para comunicarse y obtener información del lidar y de la cámara.

Este proceso ha quedado documentado de forma exhaustiva en forma de guía asegurando que el trabajo sea fácilmente reproducible en caso de ser necesario.

- **Control de bajo nivel Arduino**

El programa realizado para el Arduino le permite recibir y enviar ordenes correctamente a la Odroid y controlar los actuadores del coche. Además se ha calibrado el servomotor para obtener un control adecuado del mismo y el encoder para conocer la velocidad del coche.

- **Resultados experimentales**

Este último subobjetivo pretendía confirmar la validez del modelo como coche autónomo. Para ello se ha realizado con éxito cuatro programas que permiten al coche seguir las líneas de la carretera, adelantar a un obstáculo que se encuentre en el carril ,realizar una maniobra de aparcamiento y detectar la luz de un semáforo.

- **Realización de un coche autónomo**

Este era el objetivo principal del proyecto y habiendo cumplido todos los subobjetivos que lo componen se verifica la creación de un modelo a escala capaz de conducir de forma autónoma y de participar en próximas competiciones.

7.2. CONOCIMIENTOS ADQUIRIDOS

Los logros realizados durante el proyecto no son solo técnicos ya que la realización de este proyecto también tiene un carácter pedagógico. La realización de este proyecto permite a su realizador adquirir las siguientes competencias:

- Trabajar en un entorno Linux sin interfaz gráfica.

- Profundizar en el lenguaje de programación C++.
- Aprendizaje de la herramienta ROS.
- Manipulación de imágenes obtenidas con una cámara.
- Tratamiento de datos obtenidos con un Lidar.
- Impresión de modelos 3D.

7.3. POSIBLES LÍNEAS DE DESARROLLO

En este apartado se incluyen posibles mejoras que pueden realizarse en el futuro:

- Afrontar nuevos retos que puedan ser incluidos en las próximas competiciones como puede ser el adelantamiento de objetos móviles o el reconocimiento de otros tipos de marcas viales horizontales o verticales.
- Realizar un circuito completo y realista a escala en el que pueda funcionar este prototipo. Este circuito abre la posibilidad de desarrollar nuevos algoritmos y estrategias de conducción autónoma. Al estar en unas condiciones realistas este desarrollo puede ser el paso previo a la aplicación de los algoritmos en un vehículo real simplificando el proceso y aumentando la seguridad.

8) BIBLIOGRAFÍA

[1] Dirección General de Tráfico (2015). "CUESTIONES DE SEGURIDAD VIAL, CONDUCCIÓN EFICIENTE, MEDIO AMBIENTE Y CONTAMINACIÓN" . http://www.dgt.es/Galerias/seguridad-vial/formacion-vial/cursos-para-profesores-y-directores-de-autoescuelas/XVIII-Curso-de-Profesores/Seguridad-Vial.pdf p.17

[2] Toyota (2018). "NIVELES DE CONDUCCIÓN AUTÓNOMA". https://www.toyota.es/world-of-toyota/articles-news-events/niveles-de-conduccion-autonoma

[3] SEAT (16 de Noviembre 2017). "Alumnos de la Universidad de Valladolid ganan la primera edición del SEAT Autonomous Driving Challenge". https://www.seat-mediacenter.es/newspage/allnews/company/2017/Alumnos-de-la-Universidad-de-Valladolid-ganan-la-primera-edicion-del-SEAT-Autonomous-Driving-Challenge.html

[4] Freie Universität Berlin (2018). "AutoNOMOS Model". https://github.com/AutoModelCar/AutoModelCarWiki/wiki

[5] Michael Stonebank (2001). "UNIX Tutorial" http://www.ee.surrey.ac.uk/Teaching/Unix/unix1.html

[6] Página oficial de Arduino (En línea). https://www.arduino.cc/en/main/software

[7] Página oficial de ROS (En línea). https://www.ros.org/

[8] Página ROS rplidar (En línea). https://wiki.ros.org/rplidar

[9] Página ROS rosserial (En línea). http://wiki.ros.org/rosserial

[10] Página ROS usb_cam (En línea). http://wiki.ros.org/usb_cam

[11] Página ROS Kinetic (En línea). http://wiki.ros.org/kinetic/Installation/Ubuntu

[12] Rubén E-Marmolejo (2017). "Arduino Timer - Interrupciones con el Timer2" https://hetpro-store.com/TUTORIALES/arduino-timer/

[13] Página OpenCV (2019) "Basic thresholding operations" https://docs.opencv.org/2.4/doc/tutorials/imgproc/threshold/threshold.html

9) ANEXOS

9.1. Anexo 1: Código del Arduino

```
#include <avr/pgmspace.h>

#include <Servo.h>
#define MAX_DELTATIME 20000
#define DIR_PIN 4
#define SERVO_PIN 6

#define ENCODER_PIN 2

#define RESET_PIN 7
//#define TEST_COMMUNICATION_LATENCY


#include <ros.h>
#include <std_msgs/Bool.h>
#include <std_msgs/Int8.h>
#include <std_msgs/Int16.h>
#include <std_msgs/UInt8.h>
#include <std_msgs/UInt16.h>
#include <std_msgs/Float32.h>
#include <std_msgs/String.h>
#include <geometry_msgs/Twist.h>

const char STEERING_TOPIC[]  PROGMEM  = { "steering" };
const char SPEED_TOPIC[]  PROGMEM  = { "speed" };
const char TWIST_TOPIC[]  PROGMEM  = { "twist" };
const char TICKS_TOPIC[]  PROGMEM  = { "ticks" };

ros::NodeHandle nh;

std_msgs::Float32 steering_msg;
std_msgs::UInt16 ticks_msg;
geometry_msgs::Twist twist_msg;

ros::Publisher pubTwist(FCAST(TWIST_TOPIC), &twist_msg);
ros::Publisher pubTicks(FCAST(TICKS_TOPIC), &ticks_msg);

void onSteeringCommand(const std_msgs::Float32 &cmd_msg);
void onSpeedCommand(const std_msgs::Int16 &cmd_msg);

ros::Subscriber<std_msgs::Float32>
steeringCommand(FCAST(STEERING_TOPIC),
onSteeringCommand);
ros::Subscriber<std_msgs::Int16>
speedCommand(FCAST(SPEED_TOPIC), onSpeedCommand);

#ifdef TEST_COMMUNICATION_LATENCY
#include <std_msgs/Bool.h>
std_msgs::Bool resp_msg;

ros::Publisher pubResponse("resp", &resp_msg);
void onLatency(const std_msgs::Bool &cmd_msg) {
  resp_msg.data = true;
  pubResponse.publish(&resp_msg);
}
ros::Subscriber<std_msgs::Bool> requestCommand("req",
onLatency);
#endif


Servo myservo; // se crea el objeto para controlar el servomotor
int servo_pw = 1500;   // variable utilizada para controlar la
posición del servo
int last_pw = 0;
bool servo_initialized = false;
volatile unsigned long T1Ovs2;
volatile int16_t encoder_counter;
volatile uint16_t ticks_counter;
volatile int16_t last_encoder_counter;
volatile unsigned long deltatime = 0;
volatile boolean first_rising = true;
int8_t direction_motor = 1;

void StartTimer2(void) {
  pinMode(11, OUTPUT);
  TCNT2 = 0;
  TIFR2 = 0x00;
  TIMSK2 = TIMSK2 | 0x01;
  TCCR2A = _BV(COM2A1) | _BV(COM2B1) | _BV(WGM21) |
_BV(WGM20);
  TCCR2B = _BV(CS22);
  TCCR2B = (TCCR2B & 0b11111000) | 0x02;//62500HZ=
16MHz/1024/2=7812 Timer works 7812 Hz
  sei();
}

ISR(TIMER2_OVF_vect) {
  TIFR2 = 0x00;
  T1Ovs2++;//Aumenta el contador de "overflow"
}

void encoder() {
  cli();
  if (!first_rising) {
    deltatime = T1Ovs2 * 25 + T1Ovs2 * 5 / 10 + TCNT2 / 10;//
prevent overflow of integer number!
  }

  T1Ovs2 = 0;       //Inicialización del contador de "overflow"
  TCNT2 = 0;
  first_rising = false;
  encoder_counter++;
  ticks_counter++;
  sei();
}


void onSteeringCommand(const std_msgs::Float32 &cmd_msg)
{
  if ((cmd_msg.data <= 20) && (cmd_msg.data >= -20)) {
    // scale it to use it with the servo (value between 0 and
180)
    servo_pw = map(cmd_msg.data, 20, -20, 635, 1465);

    if (last_pw!=servo_pw) {
      myservo.writeMicroseconds(servo_pw);
    }

    if (!servo_initialized) {
      // attaches the servo on pin 6 to the servo object
      myservo.attach(SERVO_PIN);
      servo_initialized = true;
    }

    last_pw = servo_pw;
  }
}

void onSpeedCommand(const std_msgs::Int16 &cmd_msg) {
  int16_t motor_val = cmd_msg.data / 4;

  if (abs(motor_val) > 255) {
    return;
  }

  uint8_t servo_val = (uint8_t) abs(motor_val);

  // si se modifica la velocidad con el parámetro 0 se mantiene
la antigua dirección y solo se modifica el valor de "val"
  if (motor_val < 0) {
    digitalWrite(DIR_PIN, HIGH);
```

```
    direction_motor = -1;
  } else if (motor_val > 0) {
    digitalWrite(DIR_PIN, LOW);
    direction_motor = 1;
  }

  if (servo_val < 15) {
    servo_val = 15;
  }

  OCR2A = servo_val;
}

void setup() {
  nh.getHardware()->setBaud(500000);
  nh.initNode();
  nh.advertise(pubTwist);
  nh.advertise(pubTicks);

  nh.subscribe(steeringCommand);
  nh.subscribe(speedCommand);
#ifdef TEST_COMMUNICATION_LATENCY
  nh.subscribe(requestCommand);
  nh.advertise(pubResponse);
#endif
  pinMode(RESET_PIN,OUTPUT);
  digitalWrite(RESET_PIN,HIGH);

  pinMode(ENCODER_PIN, INPUT);
  pinMode(DIR_PIN, OUTPUT);
  pinMode(ENCODER_PIN, INPUT_PULLUP);
  digitalWrite(ENCODER_PIN, HIGH);          //pull up
  StartTimer2();
  attachInterrupt(digitalPinToInterrupt(ENCODER_PIN),
encoder, RISING);
}

void loop() {

        if (last_encoder_counter == encoder_counter) {
          // no se obtiene nada así que se comprueba si el
coche se ha parado
          if (T1Ovs2 * 25 + T1Ovs2 * 5 / 10 + TCNT2 / 10 >
MAX_DELTATIME) {
            twist_msg.linear.x = 0.0;
            twist_msg.linear.y = 0.0;
            twist_msg.linear.z = 0.0;

            first_rising = true;

            pubTwist.publish(&twist_msg);

            deltatime = 0;
          }
        }
          // we did receive data from the motor
        else {
          if (deltatime != 0) {
            twist_msg.linear.x = deltatime*0.006;
          } else {
            twist_msg.linear.x = 0.0;
          }

          twist_msg.linear.x = twist_msg.linear.x *
direction_motor;
          twist_msg.linear.y = 0.0;
          twist_msg.linear.z = 0.0;
          pubTwist.publish(&twist_msg);
        }
        last_encoder_counter = encoder_counter;

        ticks_msg.data = ticks_counter;
        ticks_counter = 0;
        pubTicks.publish(&ticks_msg);
        nh.spinOnce();
        delay(10);
```

9.2. Anexo 2: Programa camera_publisher

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <opencv2/highgui/highgui.hpp>
#include <cv_bridge/cv_bridge.h>
#include <sstream> // for converting the command line
parameter to integer

int main(int argc, char** argv)
{

  ros::init(argc, argv, "image_publisher");
  ros::NodeHandle nh;
  image_transport::ImageTransport it(nh);
  image_transport::Publisher pub =
it.advertise("camera/image", 1);
  cv::VideoCapture cap(0);
  // Se comprueba que el dispositivo de video puede ser abierto
  if(!cap.isOpened()) return 1;
  cv::Mat frame;
  sensor_msgs::ImagePtr msg;

  ros::Rate loop_rate(5);
  while (nh.ok()) {
    cap >> frame;
    // Se comprueba si la imagen obtenida tiene contenido o esta
vacía
    if(!frame.empty()) {
      msg = cv_bridge::CvImage(std_msgs::Header(), "bgr8",
frame).toImageMsg();
      pub.publish(msg);
      cv::waitKey(1);
    }

    ros::spinOnce();
    loop_rate.sleep();
  }
```

9.3. Anexo 3: Programa camera_sigue_lineas

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <opencv2/highgui/highgui.hpp>
#include <cv_bridge/cv_bridge.h>
//#include <geometry_msgs/Twist.h>
#include <std_msgs/Float32.h>

#define FILA_SUP 0.75  //Fila superior a analizar, en % sobre el
total
#define FILA_INF 0.85  //Fila inferior a analizar, en % sobre el
total

using namespace std;
using namespace cv;

ros::Publisher pubSteering;

int calcCdg(uchar* ptrFila, int fila, int nc, Mat& colorFrame)
{
   float suma=0;
   float total=0;
   float cdg=0;
   for (int i=0; i<nc; i++)
   {
      suma += i*(255-ptrFila[i]);
      total += 255-ptrFila[i];
      circle(colorFrame, Point(i, fila), 1, Scalar(255, 0, 0));
   }
   if(total>0)
      cdg=suma/total;
   else
      cdg=-1;

   return cdg;
}

void imageCallback(const sensor_msgs::ImageConstPtr& msg)
{
 std_msgs::Float32 ang_msg;
try
 {
   //cv::imshow("view", cv_bridge::toCvShare(msg, "bgr8")-
>image);

   //cv::waitKey(30);
   Mat colorFrame=cv_bridge::toCvShare(msg, "bgr8")->image;
   Scalar color(0,0,255);

   cv::imshow("view",colorFrame);
   cv::waitKey(30);

   int nf=colorFrame.rows;
   int nc=colorFrame.cols;

   Mat grisFrame;
   cvtColor(colorFrame, grisFrame, COLOR_BGR2GRAY);
//transforma a niveles de gris

   cv::imshow("blackwhite", grisFrame);
   cv::waitKey(30);

   Mat binFrame;
   // threshold(grisFrame, binFrame, 0, 255,
THRESH_BINARY_INV | CV_THRESH_OTSU);
   threshold(grisFrame, binFrame, 190, 255, 1);
   cv::imshow("bin", binFrame);
   cv::waitKey(30);

   uchar* ptrFila;
   float cdg_sup, cdg_inf;
   int filaSup= (int) (FILA_SUP * nf);
   ptrFila= binFrame.ptr<uchar>(filaSup); //apunta a la filaInf
   cdg_sup = calcCdg(ptrFila, filaSup, nc, colorFrame);



   int filaInf= (int) (FILA_INF * nf);
   ptrFila= binFrame.ptr<uchar>(filaInf); //apunta a la filaInf
   cdg_inf = calcCdg(ptrFila, filaInf, nc, colorFrame);

   if(cdg_sup> -1) circle(colorFrame, Point(cdg_sup, filaSup), 3,
color);
   if(cdg_inf> -1) circle(colorFrame, Point(cdg_inf, filaInf), 3,
color);

   cv::imshow("modview", colorFrame);
   cv::waitKey(30);


   float x1, x2, y1, y2, A, B, C, dist, ang, Kd, Ka;

   if( cdg_sup> -1 && cdg_inf> -1 && filaSup!=filaInf )
{
      //Traslada al punto central de la ultima linea.
      y2 = nf-filaSup;
      y1 = nf-filaInf;
      x2 = cdg_sup-nc/2.;
      x1 = cdg_inf-nc/2.;
      ROS_INFO_STREAM( "(x1, y1)= ( " << x1 << " , " << y1 << " )"
);
      ROS_INFO_STREAM( "(x2, y2)= ( " << x2 << " , " << y2 << " )"
);

      //Recta; distancia y  angulo
      A= y1-y2;
      B= x2-x1;
      ang =atan( (x2-x1)/(y2-y1) );

      C = tan(ang)*y1;
      dist = (x1-C)*cos(ang);
   }
   else
   {
      dist= -1;
      ang = -1;

   }

   ROS_INFO_STREAM("Distancia: " << dist << " Angulo: " << ang
<< " rad  (" << ang*180./3.141592 << " grad)");
   Kd = 0.005;
   Ka = 0.9;
   ang_msg.data = (Ka*ang+Kd*dist)*180/3.14;
   if(ang_msg.data < -20)
   {
    ang_msg.data = -20;
   }
   else if(ang_msg.data > 20)
   {
    ang_msg.data = 20;
   }
   pubSteering.publish(ang_msg);
   ROS_INFO_STREAM("Angulo: " << ang_msg.data << " ");
 }
 catch (cv_bridge::Exception& e)
 {
   ROS_ERROR("No se ha podido convertir de '%s' a 'bgr8'.",
msg->encoding.c_str());
 }
```

```
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "image_listener");
  ros::NodeHandle nh;

  cv::namedWindow("view");
  cv::startWindowThread();
  cv::namedWindow("blackwhite");
  cv::startWindowThread();
  cv::namedWindow("bin");
  cv::startWindowThread();
  cv::namedWindow("modview");
  cv::startWindowThread();
  image_transport::ImageTransport it(nh);
  image_transport::Subscriber sub =
it.subscribe("camera/image", 1, imageCallback);

  pubSteering =
nh.advertise<std_msgs::Float32>("steering",1000);

  std_msgs::Float32 ang_msg;

  ros::spin();
  cv::destroyWindow("view");
  cv::destroyWindow("blackwhite");
  cv::destroyWindow("bin");
  cv::destroyWindow("modview");
}
```

9.4. Anexo 4: ROSlaunch camera.launch

```
<launch>

    <node
        pkg="sigue_lineas"
        type="camera_publisher"
        name="camera_publisher"
    />

    <node
        pkg="sigue_lineas"
        type="camera_sigue_lineas"
        name="camera_sigue_lineas"
    />


    <node   pkg="rosserial_python"  type="serial_node.py"
name="serial_no$
    <param name="port"     type="string"
value="/dev/ttyArduino"/>
    <param name="baud"     type="int"     value="500000"/>
    </node>

    <node
    pkg="rostopic"
    type="rostopic"
    name="pub"
    args="pub /speed std_msgs/Int16 500 --once"
    />
</launch>
```

9.5. Anexo 5: client.cpp

```
#include "ros/ros.h"
#include "sensor_msgs/LaserScan.h"

#define RAD2DEG(x) ((x)*180./M_PI)

void scanCallback(const sensor_msgs::LaserScan::ConstPtr&
scan)
{
   int count = scan->scan_time / scan->time_increment;
   ROS_INFO("I heard a laser scan %s[%d]:", scan-
>header.frame_id.c_str(), count);
   ROS_INFO("angle_range, %f, %f", RAD2DEG(scan-
>angle_min), RAD2DEG(scan->angle_max));

   for(int i = 0; i < count; i++) {
      float degree = RAD2DEG(scan->angle_min + scan-
>angle_increment * i);
      ROS_INFO(": [%f, %f]", degree, scan->ranges[i]);
   }
}

int main(int argc, char **argv)
{
   ros::init(argc, argv, "rplidar_node_client");
   ros::NodeHandle n;

   ros::Subscriber sub =
n.subscribe<sensor_msgs::LaserScan>("/scan", 1000,
scanCallback);

   ros::spin();

   return 0;
```

9.6. Anexo 6: evita.cpp

```
#include "ros/ros.h"
#include "sensor_msgs/LaserScan.h"
#include <std_msgs/Float32.h>
#include <ctime>
#include <sys/time.h>

#define RAD2DEG(x) ((x)*180./M_PI)

ros::Publisher pubSteering;

int maniobra=0;
struct timeval tCGiro,tFGiro;
unsigned long tGirando;

void scanCallback(const sensor_msgs::LaserScan::ConstPtr&
scan)
{
  std_msgs::Float32 ang_msg;
  int count = scan->scan_time / scan->time_increment;
  int contadorObstaculo=0;
  ROS_INFO("Maniobra %d", maniobra);
  if(maniobra == 0){
            for(int i = 225; i < 270; i++) {
                  //float degree = RAD2DEG(scan-
>angle_min + scan->angle_increment * i);
                  if(scan->ranges[i]<0.65){
                        contadorObstaculo += 1;
                  }
            }
            if (contadorObstaculo >= 3){
                  maniobra=1;
                  ang_msg.data=-20;
                  pubSteering.publish(ang_msg);
                  gettimeofday(&tCGiro, NULL);
            }
  }
  else if(maniobra == 1){
            for(int i = 225; i < 270; i++) {
        if(scan->ranges[i]<0.65){
            contadorObstaculo += 1;
        }
    }
    if (contadorObstaculo == 0){
        maniobra = 2;
        ang_msg.data = 20;
                  pubSteering.publish(ang_msg);
        gettimeofday(&tFGiro, NULL);
                  tGirando = (tFGiro.tv_sec - tCGiro.tv_sec)
* 1000000 + tFGiro.tv_usec - tCGiro.tv_usec;
                  gettimeofday(&tCGiro, NULL);
            }
  }
  else if(maniobra == 2){
            gettimeofday(&tFGiro, NULL);
            if ((tFGiro.tv_sec - tCGiro.tv_sec) * 1000000 +
tFGiro.tv_usec - tCGiro.tv_usec >= tGirando){
                  maniobra = 3;
                  ang_msg.data = 0;
                  pubSteering.publish(ang_msg);
            }
  }

  else if(maniobra == 3){
    for(int i = 180; i < 270; i++) {
        if(scan->ranges[i]<0.65){
            contadorObstaculo += 1;
        }
    }
    if (contadorObstaculo == 0){
        maniobra = 4;
        ang_msg.data = 20;
        pubSteering.publish(ang_msg);
                  gettimeofday(&tCGiro, NULL);
    }
  }

  else if(maniobra == 4){
    gettimeofday(&tFGiro, NULL);
    if ((tFGiro.tv_sec - tCGiro.tv_sec) * 1000000 +
tFGiro.tv_usec - tCGiro.tv_usec >= tGirando){
            maniobra = 5;
            ang_msg.data = -20;
            pubSteering.publish(ang_msg);
                  gettimeofday(&tCGiro, NULL);
    }
  }

  else if(maniobra == 5){
    gettimeofday(&tFGiro, NULL);
    if ((tFGiro.tv_sec - tCGiro.tv_sec) * 1000000 +
tFGiro.tv_usec - tCGiro.tv_usec >= tGirando){
            maniobra = 0;
            ang_msg.data = 0;
            pubSteering.publish(ang_msg);
    }
  }


}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "rplidar_node_client");
  ros::NodeHandle nh;

  ros::Subscriber sub =
nh.subscribe<sensor_msgs::LaserScan>("/scan", 1000,
scanCallback);
  pubSteering =
nh.advertise<std_msgs::Float32>("steering",1000);
  ros::spin();

  return 0;
}
```

9.7. Anexo 7: aparcamiento.cpp

```
#include "ros/ros.h"
#include "sensor_msgs/LaserScan.h"
#include <std_msgs/Float32.h>
#include <std_msgs/Int16.h>

#include <ctime>
#include <sys/time.h>

ros::Publisher pubSteering;
ros::Publisher pubSpeed;


std_msgs::Float32 ang_msg;
std_msgs::Int16 speed_msg;

struct timeval tCGiro,tFGiro;

void move(int time,  int angle, int speed){
		speed_msg.data = speed;
	pubSpeed.publish(speed_msg);
		ang_msg.data = angle;
		pubSteering.publish(ang_msg);
		gettimeofday(&tCGiro, NULL);
		gettimeofday(&tFGiro, NULL);
		while ((tFGiro.tv_sec - tCGiro.tv_sec) * 1000000 +
tFGiro.tv_usec - tCGiro.tv_usec < time * 1000000){
			gettimeofday(&tFGiro, NULL);
		}
}

int maniobra = 0;

int main(int argc, char **argv)
{
  ros::init(argc, argv, "aparcamiento_node");
  ros::NodeHandle nh;

  pubSteering =
nh.advertise<std_msgs::Float32>("steering",1000);
  pubSpeed = nh.advertise<std_msgs::Int16>("speed",1000);

  ros::Rate loop_rate(10);

  while(ros::ok()){
			if(maniobra == 0){

				ROS_INFO("primera
maniobra");
				move(5,20,-400);
		ROS_INFO("segunda maniobra");
				move(3,-20,-400);
		ROS_INFO("tercera maniobra");
				move(2,20,400);
		ROS_INFO("parar");
				move(0,0,0);
				maniobra = 1;
			}
		ros::spinOnce();
		loop_rate.sleep();
  }


  //return 0;
}
```

9.8. Anexo 8: semaforo.cpp

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <opencv2/highgui/highgui.hpp>
#include <cv_bridge/cv_bridge.h>
#include <std_msgs/Int16.h>

using namespace std;
using namespace cv;

ros::Publisher pubSpeed;

void imageCallback(const sensor_msgs::ImageConstPtr& msg)
{
 std_msgs::Int16 speed_msg;
try
 {
   Mat colorFrame=cv_bridge::toCvShare(msg, "bgr8")->image;

   cv::imshow("view",colorFrame);
   cv::waitKey(30);

   int nf=colorFrame.rows;
   int nc=190;   //Se observa solo unas columnas de la imagen
colorFrame.cols;

   //Vec3b pixel = colorFrame.at<Vector3b>(Point(0,0));
   uint8_t* pixelPtr = (uint8_t*)colorFrame.data;
   int cn = colorFrame.channels();
   Scalar_<uint8_t> bgrPixel;

   int rojo = 0;

   for(int i = 0; i < nf; i++)
   {
            for(int j = 0; j < nc; j++)
            {
                     if(pixelPtr[i*190*cn + j*cn + 2] > 200) //Se
observa si valor del rojo alto
                     {
                              if(pixelPtr[i*190*cn + j*cn] <
200 && pixelPtr[i*190*cn + j*cn + 1] < 200) //Azul y verde
bajos. Así sabemos que no es Blanco
                              {
                                       rojo = 1;
                                       i = nf;
                                       j = nc;
                              }
                     }
            }
   }
   speed_msg.data = (1-rojo)*400;
   pubSpeed.publish(speed_msg);
   ROS_INFO_STREAM("Valor rojo: " << rojo << " ");
 }
 catch (cv_bridge::Exception& e)
 {
   ROS_ERROR("Could not convert from '%s' to 'bgr8'.", msg-
>encoding.c_str());
 }
}

int main(int argc, char **argv)
{
 ros::init(argc, argv, "semaforo_listener");
 ros::NodeHandle nh;

 cv::namedWindow("view");
 cv::startWindowThread();

 image_transport::ImageTransport it(nh);
 image_transport::Subscriber sub =
it.subscribe("camera/image", 1, imageCallback);

 pubSpeed = nh.advertise<std_msgs::Int16>("speed",1000);

 ros::spin();
 cv::destroyWindow("view");

}
```