UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y DISEÑO INDUSTRIAL

Graduado en Ingeniería Electrónica Industrial y Automática

TRABAJO FIN DE GRADO

NAVEGACIÓN BASADA EN SEGUIMIENTO DE LÍNEAS MEDIANTE VISIÓN ARTIFICIAL PARA EL ROBOT ROBOGAIT SPORT

Autor: Lucas Gómez Velayos

Co-tutor:
David Álvarez Sánchez
Departamento de Ingeniería Eléctrica, Electrónica, Automática y Física Aplicada

Tutor:
Alberto Brunete González
Departamento de Ingeniería Eléctrica, Electrónica, Automática y Física Aplicada

Madrid, septiembre, 2024

ÍNDICE

Resumen

En este trabajo se ha profundizado en el diseño, selección de componentes, programación e integración de un sistema de detección de líneas y seguimiento de estas, adaptándose a la estructura, hardware y sistemas de control ya presentes en un prototipo de robot denominado ROBOGait SPORT. Se trata de un robot móvil de análisis de la marcha en humanos y, más concretamente, de la forma de correr de los atletas. Al comienzo de este trabajo, el sistema de GPS (Global Positioning System) planteado en primera instancia como forma de guiado del robot ya había sido desarrollado en otros proyectos, pero no ofrecía un buen funcionamiento. Por esta razón, en este trabajo se acomete un modo de guiado alternativo al que se había implementado en el prototipo original. Aprovechando que las pruebas con los sujetos de estudio del robot se realizan en pistas de atletismo, se planteó como método más eficaz utilizar la coloración diferenciada de las líneas que separan los carriles o calles de los corredores en el circuito, para delimitar y guiar también la ruta del robot mediante un sistema de visión artificial. Para este cometido, se ha utilizado la placa Raspberry Pi 5, la cámara Camera Module 3 de Raspberry y la biblioteca de código abierto de visión artificial OpenCV y de control de sensores fotográficos libcamera.

Abstract

In this work, the design process, part selection, programming and integration of a line detection and following system is thoroughly described, adapting to the structure, hardware and control systems already present in a prototype called ROBOGait SPORT. It is a mobile gait analysis robot for humans and, more specifically, it analyzes the way athletes run. At the start of this work, the GPS system proposed to guide the robot was already developed, but it did not function correctly. Because of this reason, in this an alternative guiding mode from the one implemented in the original prototype it's taken on. Taking advantage of the fact that tests with the subjects are done in athletic tracks, using the differentiated coloration of the lines that separate the runner's tracks in the circuit, to delimit and guide the route of the robot using a computer vision system was proposed as a more effective method. To do this, the Raspberry Pi 5 microcontroller, Raspberry's Camera Module 3 and the open source computer vision library OpenCV and the photographic sensor control library libcamera have been used.

Palabras clave:

Robótica, Visión Artificial, Seguimiento de líneas, Detección de bordes, OpenCV, ROS 2

Keywords:

Robotics, Computer Vision, Line following, Edge detection, OpenCV, ROS 2

Agradecimientos

A Pablo Quesada (técnico de laboratorio) por su paciencia, y por compartir sus conocimientos de redes y ayuda. Al profesor de la ETSIDI (Escuela Técnica Superior de Ingeniería y Diseño Industrial) Miguel Hernando por sus sugerencias y su tiempo, y, sobre todo, a Alberto Brunete, mi tutor del TFG (Trabajo Fin de Grado), y a David Álvarez, co-tutor, por su guía e inestimable ayuda y apoyo en las reuniones periódicas que hemos mantenido con el resto del equipo ROBOGait. A Carlos Ferreira, por ayudarme a entender el funcionamiento de sus mejoras en el robot con el que he trabajado para este proyecto.

A mi familia, en especial a mis padres, Pepe y Carmen, por apoyarme y alentarme siempre en mi camino de formación que, por ahora, culmina con este trabajo.
A Ana Cristina, a quien agradezco su apoyo incondicional, su amistad y el ánimo que me ha infundido. A mis compañeros y amigos de carrera por su compañerismo, los buenos momentos vividos y la simbiosis de conocimientos que hemos compartido, especialmente, a María y Manuel, quienes se embarcaron en sus propios TFG para el desarrollo de otros componentes de este proyecto al mismo tiempo que yo.

A los trabajadores y responsables del Centro Deportivo Municipal Estadio de Atletismo Vallehermoso de Madrid, quienes me facilitaron el uso de sus instalaciones para realizar la mayoría de las grabaciones empleadas para validar el sistema de visión artificial desarrollado en este trabajo.

Por último, a todos aquellos desarrolladores con los que me he topado que han publicado y compartido su código y lo han explicado para que pueda reutilizarse; de forma similar, a OpenCV y ROS 2, por mantener su código actualizado, libre y documentado para todo el mundo.

"Carpe Diem"

Horacio (65-8 a. C.)

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1| Introducción

1.1 Contexto

ROBOGait es el nombre de un proyecto de investigación desarrollado en la ETSIDI que lleva siendo supervisado desde hace años por un grupo de docentes de esta escuela bajo el que se encapsulan varios TFG, TFM (Trabajo Fin de Máster), tesis doctorales, prácticas, etc. realizados por distintos estudiantes a lo largo de varios cursos académicos. Cuando comencé este trabajo, el proyecto ROBOGAIT consistía esencialmente en dos prototipos de robot: por un lado, el denominado ROBOGait INDOOR, un primer prototipo de menor tamaño, que había sido concebido, como su propio nombre indica, para ser utilizado en interiores. El segundo prototipo, denominado ROBOGait SPORT, que es realmente el objeto de estudio en el que centra este trabajo, había sido pensado para ser utilizado en exteriores, sobre todo en pistas de atletismo, y utilizado para establecer un sistema que fuera capaz de funcionar a mayor velocidad, pensado para el análisis del movimiento en carrera de los atletas.

Fig. 1. Imagen del prototipo ROBOGait SPORT. Cortesía de ROBOGait, sacada de https://blogs.upm.es/robogait/prototypes/

En el último TFM realizado por el estudiante Carlos Ferreira sobre este mismo prototipo, ROBOGait SPORT, se describe la forma, funcionalidad y objetivos que se deberían desarrollar en dicho robot. Una de las funcionalidades que se consideró que debía ser mejorada se describe en el apartado final de la memoria de dicho TFM, titulado "Líneas Futuras", afirmando que: "... es necesario realizar pruebas del sistema de navegación diseñado. Una vez se disponga de sistemas de localización más precisos, se puede [sic.] empezar a realizar pruebas de navegación para comprobar el comportamiento de los algoritmos seleccionados" [1].

Además, y de forma extraoficial, en una conversación mantenida con el autor del TFM citado, este mencionó a los futuros investigadores del proyecto que hemos trabajado en este mismo prototipo, a lo largo del presente curso académico, que el sistema de GPS funcionaba correctamente en las pruebas que él había realizado en la terraza de la ETSIDI (la escuela en la que se llevaron a cabo los experimentos del trabajo). Sin embargo, cuando esas mismas pruebas se hacían en el exterior, no se obtenían resultados del todo satisfactorios. El autor teorizó sobre las causas, sin tener una respuesta clara a la incógnita de por qué se producía ese fallo en el funcionamiento del sistema de guiado del robot. Aunque la mejora de este sistema de guiado por GPS no es el tema central de este TFG, sí que es su causa; sin embargo, esta mejora sí será objeto de estudio central para otro TFG realizado en paralelo a este durante el presente curso académico 2023-24.

Debido al problema anteriormente apuntado, se planteó un nuevo objetivo: desarrollar un sistema alternativo de guiado del robot. Si el sistema de guiado basado en GPS anteriormente diseñado no pudiera mejorarse, o si no se observase que en las pruebas fuese lo suficientemente preciso, debido a múltiples variables por analizar, se tendría que plantear una nueva alternativa. En este caso, la propia experimentación en el uso para el que estaba diseñado el prototipo ayudó a encontrar esa solución alternativa: seguir las líneas blancas que separan los distintos carriles, o calles, de las pistas de atletismo donde se realizan esas pruebas de funcionamiento y guiado. La primera parte de este trabajo se ha centrado en idear y proyectar distintas formas de incorporar esta solución al prototipo actual. Tras la experimentación pertinente, la conclusión a la que se llegó en primera instancia consistió en que lo mejor para nuestro proyecto era desarrollar ese sistema de guiado alternativo en una nueva máquina o procesador que se comunicase con el resto de controladores, puesto que ya

se estaba utilizando el framework de robótica ROS 2 (Robot Operating System, ver apartado 3.1 Software a utilizar). Esta decisión y la selección de uso de la placa que utilizaremos, así como el hardware adicional y el software asociado, se discutirán con más detalle en los siguientes apartados.

1.2 Objetivos y alcance

Objetivo general:

- Diseñar e implementar en el prototipo ROBOGait SPORT ya existente un sistema de visión artificial que sea capaz de detectar y enviar información de las líneas, generalmente blancas, de un carril en una pista de atletismo.

Objetivos específicos:

- Familiarizarse con el entorno y diseñar una interfaz de desarrollo sencilla en el framework ROS 2 que se integre con el algoritmo de control existente en el prototipo, con la intención de facilitar su uso a los siguientes investigadores que vayan a trabajar sobre este.

- Elegir los componentes de hardware de sensórica y control más adecuados a los requerimientos del proyecto, y su montaje en el robot.

- Diseñar y desarrollar algoritmos capaces de separar y reconocer las líneas que separan los carriles de las pistas de atletismo, e integrar este en el sistema del prototipo ya existente (mediante ROS 2).

2| Estado de la cuestión

Al hacer una investigación previa de los recursos y soluciones más prominentes en el campo de detección de carriles o líneas, podemos observar que la amplia mayoría de los recursos encontrados, por no decir todos, tratan sobre carreteras o asfalto, y no sobre pistas de atletismo. Además, una gran mayoría de estos recursos, con alguna excepción, se centra en detectar líneas rectas en este tipo de terreno. Por ejemplo, en una de estas excepciones, a pesar de estudiar también carreteras y no sólo pistas de atletismo, se describe un algoritmo capaz de procesar curvas, aunque al final del texto se menciona (traducido): "El algoritmo ahora es principalmente para seguir/monitorizar la línea parada. El futuro trabajo es conseguir el trazado de línea variable" [2, Sec. 5]. Se presenta esta solución como válida en estas obras debido presuntamente a la poca curvatura que suelen presentar los carriles automovilísticos, al menos en autopista. Estos métodos podrían darnos algo de información sobre cómo separar las líneas de los carriles del fondo, así como algunos de los pasos del tratamiento de imagen, correcciones de perspectiva, etc.; sin embargo, esta especialización, o el hecho de asumir que las líneas son invariablemente rectas, hace inviable el uso o recreación de estos algoritmos de forma íntegra, siendo necesaria la adaptación y cambio de estas soluciones previas que hemos estudiado.

Otra característica destacable y que justifica una de las decisiones tomadas para este proyecto es la siguiente: casi toda la bibliografía usa la librería de visión artificial OpenCV, que se describirá con mayor detalle en el apartado 3.1. Esta librería sirve para la manipulación de vídeo y/o imágenes, e incluye implementaciones optimizadas de algoritmos muy extendidos en el campo de la visión artificial. Este consenso sobre el uso de OpenCV parece garantizar que se trata de un producto de confianza con el que se suele obtener un buen funcionamiento.
Además, la mayoría de las técnicas utilizadas en este trabajo concernientes a técnicas de visión artificial clásica, que es la finalidad principal de OpenCV, tienen precedentes en la literatura. Por ejemplo, es muy común el uso de la detección de bordes para la detección de líneas, específicamente el detector de bordes de Canny, como en [3][4], aunque también se han encontrado otros métodos similares, aunque distintos, como en [5]. En la Fig.10 podemos ver un ejemplo de aplicación del detector de Canny.
También existe buena cantidad de trabajos anteriores que presentan ejemplos de filtrado por colores y de uso de histogramas en blanco y negro o técnicas similares, como *sliding windows* [2][6][7][8] para la detección de la posición de las líneas. Estas técnicas sí son capaces de

detectar curvaturas, a diferencia de otras técnicas usadas en la literatura que detectan por diseño solo líneas rectas, como la transformación de líneas de Hough (Hough Line Transform). En la Fig. 2 podemos ver un ejemplo gráfico de la técnica *sliding windows*:

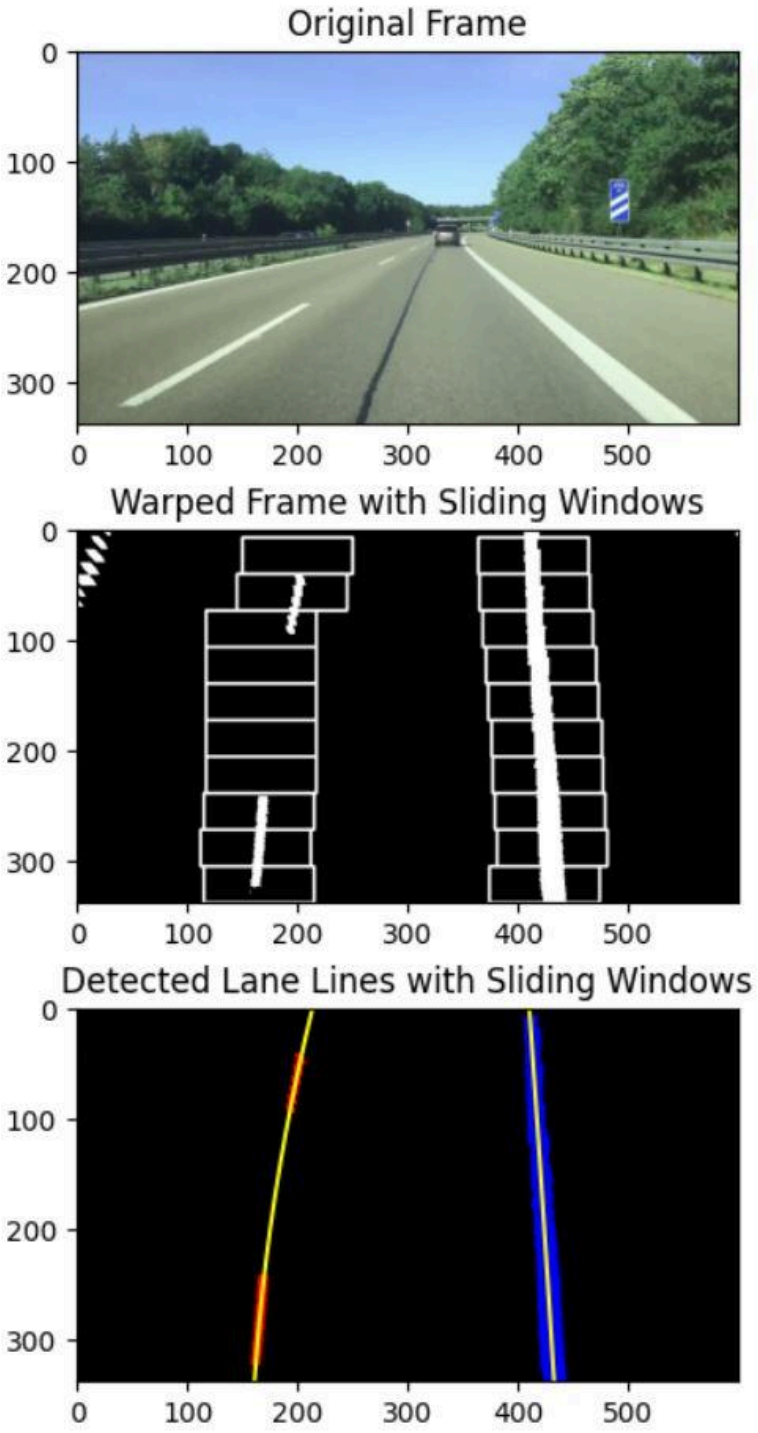


Fig. 2. De arriba abajo: Imagen original, imagen binaria procesada con filtro de color y cambio a perspectiva de vista de pájaro, estimación de las líneas detectadas. Imágenes sacadas de [7].

Curiosamente, en [4], aparte de tomar el algoritmo de Canny como una de las opciones sujetas a comparación, se usan también la Raspberry Pi y la Raspberry Pi Camera Module, y en [2] se usa una Raspberry Pi 2, de forma similar al hardware usado en este trabajo, una placa Raspberry Pi 5, un modelo de la misma gama, pero con mejores especificaciones (ver el apartado 3.2 Hardware a utilizar).

Otro punto importante consiste en que la mayoría de proyectos similares publicados (en repositorios de GitHub u otros artículos que presentan código) usan Python como lenguaje de programación [6][7]. Algunos de ellos, como [6], en sus conclusiones, justifican directamente el uso de C++ en este trabajo en comparación con sus soluciones, al mencionar un rendimiento no óptimo, una característica atribuible a Python: "Performance: The final performance is not as fast as I wanted to be, the fps of roughly 12 fps is not enough for real time scenarios. … An interesting aproach [sic.] would be porting this delopment [sic.] from C++ to Python [sic.] which suggests to speed up the undistortion 5x times." La gramática de esta cita no es del todo correcta, y además el orden de los lenguajes está cambiado. El trabajo está realizado en Python, no en C++, por lo que también deducimos de lo leído al final de esta cita que, en realidad quiere decir *from Python to C++*, y no al revés. Por otra parte, C++ suele mencionarse como uno de los lenguajes más adecuados para aplicaciones en las que la rapidez es necesaria, y OpenCV tiene también una interfaz en este lenguaje. Además, a bajo nivel, esta librería está programada también en C++. Por suerte, al menos el uso de la librería OpenCV es prácticamente idéntico en ambos lenguajes.

Al principio de esta investigación, no se encontraron demasiados ejemplos de IA (Inteligencia Artificial) aplicada a este tipo de procesos. Sin embargo, buscando soluciones específicamente en este campo para nuestro caso, se encontraron algunas aplicaciones novedosas de esta tecnología destinada al ámbito de la visión artificial y, más concretamente, a la detección de carriles de guiado [9][30]. [9] implementa una versión de la IA descrita en el artículo que cita, disponible en el archivo README.md de la página. Se puede observar que, entre otras tareas, sigue realizando la segmentación o separación de las líneas en una imagen binaria, sólo que en este caso se emplea una red neuronal. En cierto modo, esto valida también el método de filtrado por color que se ha mencionado anteriormente, ya que se busca el mismo resultado, pero con herramientas o filosofías distintas. De hecho, teóricamente, podrían combinarse con un algoritmo de visión artificial clásica a cargo de la segmentación, y que ese resultado se pasase a la otra red que, a partir de esa imagen binaria, fuese capaz de identificar y etiquetar

las líneas. Según algunas de estas referencias [20], la IA está tomando un protagonismo cada vez mayor y con mucha rapidez en este sector. Debido a la falta de tiempo y conocimiento de esta tecnología por parte de los investigadores, no se acabó haciendo uso de la misma, pero esto se discutirá más adelante en el apartado de Conclusiones.

2.1 Estado previo del prototipo

Debido a que este trabajo se origina sobre un proyecto de prototipo ya existente, basado en investigaciones anteriores, una buena parte de este epígrafe se dedicará a describir el estado previo del mismo, basándonos en el estado inmediatamente anterior a este TFG, descrito en anteriores trabajos. Por ejemplo, aquí se reproduce la tabla de especificaciones encontrada en https://blogs.upm.es/robogait/prototypes/:

TABLA I
Especificaciones técnicas con traducciones de la página web de ROBOGait

Especificaciones técnicas

Máxima velocidad (Top speed)	15 m/s
Radio de giro (Turn radius)	1.1 m
Rango de captura (Capture range)	0.5 – 10 m
Dimensiones (Altura x Ancho x Profundidad) [Dimensions (H x W x D)]	48 x 60 x 70 cm
Peso (Weight)	10 kg
Autonomía (Autonomy)	3 h
Consumición de energía (Power consumption)	15V 5A

En el TFM de Carlos Ferreira se describe el prototipo ROBOGait SPORT de la siguiente forma:

> La base robótica sobre la que se asienta la electrónica y el resto de componentes consiste en un chasis Radio-Control modelo Arrma Kraton EXB (Extreme Bash) de escala 1/5. ... Este chasis utiliza una cinemática Ackermann, donde las ruedas delanteras actúan como ruedas directrices

> y motoras al mismo tiempo, mientras que las traseras solamente operan como ruedas motoras. Además, el chasis incluye un sistema de recepción RC (Radio-Control) y un mando para controlar el robot de forma manual. ... El motor es alimentado y controlado a través de una controladora electrónica que recibe el nombre de ESC, por sus siglas en inglés Electronic Speed Controller. La función de este componente es compleja, a partir de la señal de entrada realiza la polarización adecuada en los polos del estator para generar un par continuo en el giro, que se traduce en una velocidad angular obtenida en el eje. La controladora recibe una señal PWM (Pulse Width Modulation) de 20 ms de periodo como input, con pulsos de un valor entre 1 y 2 ms. La relación entre la señal PWM y la velocidad obtenida en el eje es desconocida, ya que estos dispositivos suelen incluir un sistema de control propio y no accesible por el usuario. ... Por último, es necesario hablar de los dos servomotores de los que dispone el robot. El primero de ellos se encarga de controlar la dirección del robot y el segundo se encuentra instalado debajo de la cámara de seguimiento y permite orientar la cámara en la dirección del atleta de estudio, de forma que éste siempre quede centrado en la imagen [1, pp. 18-19].

De igual modo, en este mismo TFM se describen los sensores y el software dedicados a la adquisición de datos de la marcha y de la distancia existente entre el propio robot y el sujeto de estudio (el atleta). Lo relevante para nuestro proyecto es saber que este prototipo consta ya de un hardware y de diversos sistemas de control que aseguren una distancia concreta al sujeto estudiado, ajustando su velocidad a la marcha del atleta, gracias a las cámaras RGB-D [1, Sec. VII]. El sistema RGB de estas cámaras se refiere a los tres canales de color primarios (Red, Green and Blue), y la letra D se refiere concretamente al canal de distancia. Esta funcionalidad deberá permanecer siempre intacta, ya que no sólo mantiene constante la distancia de seguridad del robot con respecto al sujeto de análisis, el atleta, si no que también garantiza la calidad y rigor de los datos que el prototipo registra a partir de sus movimientos.

Como se ha mencionado en la introducción, la razón de ser de este TFG tiene su origen en una de las características no del todo satisfactorias de este estado previo de construcción, es decir, en el fallo en el sistema de navegación por GPS que se le había incorporado. El prototipo incorporaba en ese primer estado la sensórica necesaria, como la antena, y algunos dispositivos más para complementar la navegación, como un *lidar* (Light Detection and Ranging o Laser Imaging Detection and Ranging) y una cámara capaz de realizar un mapeado del terreno desde ROS 2. El objetivo sería hacer un sistema alternativo que pueda reemplazar al actual. Debido a la modularidad que nos proporciona el uso de ROS 2, que se discutirá más adelante, podría pensarse en algo parecido a una sustitución de piezas, para que estas últimas

envíen datos equivalentes o similares al sistema anterior para que el prototipo necesite los menores cambios posibles. Volviendo a la analogía de la pieza, se trataría de diseñar una distinta, pero que sea capaz de encajar en el hueco abandonado por la antigua.

Durante el desarrollo de este trabajo se descubrió que algunos sistemas que se presuponían funcionales no lo eran en realidad, como la odometría. Este hallazgo fue realizado por parte de otros compañeros que habían estado desarrollando sus propios trabajos de investigación en el mismo prototipo de forma paralela. También hay que apuntar el cambio de hardware que se produjo en la placa principal entre dos modelos distintos de Jetson Nano. El cambio de software se reflejó en la nueva versión de Ubuntu, el Sistema Operativo de ambos modelos de placa, que fue necesario para instalar ROS 2 *Humble*, ya que la versión usada hasta esa fecha (*Foxy*) iba a quedar desfasada en breve:

> En primer lugar, sería conveniente realizar una migración de la arquitectura de software a una versión de ROS 2 más reciente. La versión *Foxy Fitzroy* en la que se ha desarrollado el software ha terminado su vida de mantenimiento, por lo que sus paquetes no cuentan con los últimos parches de los desarrolladores [1, p. 97].

Estos cambios sucedieron durante el proceso de indagación realizado por otro compañero para su propio TFG, de forma simultánea y en paralelo al desarrollo e investigación del presente trabajo.

3| Marco teórico y herramientas utilizadas

3.1 Software a utilizar

ROS 2: 

El uso del framework ROS 2 (Robot Operating System) no se justifica únicamente por tratarse de un estándar en robótica [10], sino porque ya había sido implementado en el prototipo inicial. Si pretendiésemos cambiar ahora de paradigma, esto supondría no sólo cambiar el modo de funcionamiento del prototipo, sino que también nos obligaría a reprogramar muchas de las partes vitales del proyecto y a buscar nuevas alternativas de buena parte de los algoritmos que ya vienen implementados en este framework y en el prototipo. ROS 2 es un framework o "*middleware*" (que podría definirse como una plataforma de desarrollo) de carácter código abierto, destinada al desarrollo de robots, que permite comunicar dispositivos, sensores y actuadores de forma sencilla y transparente. Explicándolo de modo sencillo, este framework funciona comunicando nodos (que en sí se pueden definir como programas o rutinas) que pueden residir en dispositivos distintos, mediante *topics*, que, de forma metafórica, se podrían definir como canales o tuberías por los que se envía la información y a las que cada nodo puede elegir suscribirse para obtenerla. Los *topics* tienen un nombre propio para identificarse y una estructura de datos definida. Por ejemplo, podemos elegir enviar palabras o frases (`strings`) números enteros (`ints`), booleanos (`bool`, un valor que es binario, de álgebra de Boole, 1 ó 0, *`true`* o *`false`*), o combinaciones personalizadas de estos, como vectores. Un conjunto determinado de estos datos podría dar lugar a conceptos menos abstractos. Por ejemplo, juntando en un *topic* una `string` con un nombre y un `int` con un número de matrícula, se podría representar el concepto de un alumno de la ETSIDI. Esta es también una de las bases de la programación orientada a objetos (POO). Si el entorno de ROS 2 está correctamente configurado, cualquier nodo de la red o "dominio de ROS 2" (ya sea en un solo proceso o programa, en una sola máquina o en máquinas distintas en la misma red) puede elegir recibir la información o, expresado de forma más técnica, suscribirse a esta, teniendo en cuenta que necesita saber qué tipo de información va a recibir. Con esta estructura puede observarse que un *topic* no es un canal bidireccional de información, si no que es un único nodo quien envía información al *topic*, aunque, teóricamente, cualquier número de nodos puede recibirla (con límites debidos al hardware y software, porque, por ejemplo, sólo se pueden ejecutar un máximo de 120 nodos en una sola

máquina). Otra ventaja de ROS 2 consiste en que, en los lenguajes que soporta, como Python, que no nos incumben para este proyecto, si en un futuro se desease leer información de un *topic* de forma nativa, esto sería posible, sin mecanismos externos, lo que lo hace muy flexible y con garantía de futuro. Esta ventaja permite que cualquier información que mandemos mediante un *topic* se pueda leer por cualquier otro nodo o paquete de ROS 2 que lo incluya, de forma sencilla y rápida.

Esta es una de las formas que tienen los nodos en ROS 2 de comunicarse, pero existen otros modelos, como el del nodo servicio, que espera una entrada y envía un resultado de vuelta, o el nodo de acción, que es similar sólo que, al recibir el *input* y hasta llegar al resultado, puede ir recibiendo *feedback* (no es una computación tan básica como se espera de un servicio, sino un proceso más complejo).

Podríamos pensar en un nodo como en un trabajador que realiza una actividad muy concreta, y puede elegir enviar (publicar) o recibir (suscribirse) mensajes diversos, estableciendo canales de comunicación con otros nodos, que serían los *topics*. Estableciendo otra comparación, podríamos pensar en un *topic* como en una pizarra en la que un nodo puede escribir o publicar información de cierto tipo, mientras que otro nodo puede mirar o recibir la información que se escriba en esta, además de averiguar cuándo se escribe información nueva para poder leerla. ROS 2 permite que estos *topics* puedan viajar entre dispositivos a través de distintos protocolos de comunicación, como Ethernet o Wi-Fi, por nombrar algunos ejemplos, aunque también puede haber comunicación entre nodos en un mismo dispositivo, en cuyo caso se utilizan otros protocolos de forma automática y eficiente.

Como se comenta más adelante, la instalación de ROS 2 en Raspberry Pi OS conlleva el uso de Docker, que, a nivel muy básico, podríamos definir como un programa que permite tener "imágenes" o pseudo máquinas virtuales, llamadas contenedores. Podríamos decir que es como tener un ordenador con un Sistema Operativo que sí soporta ROS 2 dentro de nuestro propio Sistema Operativo, virtualmente.

OpenCV: 

Por otro lado, y tras analizar diversas comparativas de funcionalidad, rapidez, popularidad y mantenimiento de varias librerías de visión artificial, finalmente nos decidimos por el uso de la librería OpenCV. Según la página de esta librería [11], se trata de la más grande del mundo, usada por gigantes tecnológicos como Google, Microsoft, Intel, etc. La optimización y previo

desarrollo de muchos algoritmos extendidos en el campo, así como la documentación de sus funciones y explicaciones de conceptos, facilitaron enormemente muchas de las tareas necesarias para el desarrollo del presente trabajo. Los algoritmos implementados para realizar las técnicas necesarias se explicarán en detalle en los apartados correspondientes de metodología y en el propio núcleo del trabajo. Además, esta librería proporciona facilidades no estrictamente ligadas a la visión artificial, pero sí útiles, como poder mostrar las imágenes procesadas, poder guardarlas en disco, así como el procesamiento de inputs, la creación de interfaces de control (ver Fig. 25), etc.

libcamera: 

Al usar la cámara de Raspberry Pi, lo más lógico y recomendable, algo que sugiere también la propia empresa, sería usar una librería llamada libcamera, de código abierto, lo que nos debería facilitar a priori el proceso del control del sensor de dicha cámara. A pesar de esta ventaja de partida, compatibilizar su uso con las estructuras de uso y de datos de OpenCV supuso un reto añadido que también discutiremos más adelante.

CMake: 

Para compilar el código se utilizó CMake, una herramienta estándar a la hora de realizar proyectos de programación en sistemas UNIX o a bajo nivel en los lenguajes C/C++. Para comprobar la fiabilidad de esta elección, podemos observar que incluso OpenCV usa y recomienda CMake para su compilación. Simplificando mucho, podríamos decir que CMake se encarga de abstraer todo el proceso de traducción de código C/C++ a ensamblador, que es el lenguaje de instrucciones básicas que entiende un ordenador. Este proceso se llama compilación.

Editores de texto (Geany y Visual Studio Code):  

Para la escritura de código se empezó usando el editor de texto Geany, ya que este venía instalado por defecto en el Sistema Operativo de la placa utilizada (Raspberry Pi OS, basado en Debian, una distribución de Linux). Pero, posteriormente, se pasó a usar Visual Studio Code como IDE, aunque sólo se usaron las funciones de edición de texto/código. Este cambio facilitó el proceso de desarrollo, ya que la segunda herramienta cuenta con atajos de teclado,

sugerencias de texto (IntelliSense) y *syntax highlighting*, funciones que son de gran utilidad a la hora de escribir código, además de una CLI integrada que, debido al uso de CMake, era la interfaz necesaria para compilar y ejecutar el código.

GitHub:

Por último, para el control de versiones, se usó git, en concreto GitHub, en una rama propia salida del repositorio que contiene el software correspondiente al prototipo, siendo este privado [12]. Dicho control de versiones GitHub se usó principalmente como copia de seguridad.

3.2 Hardware a utilizar

Prototipo ROBOGait SPORT:

Todo el hardware utilizado y su justificación se explican con detalle en el TFM que los desarrolla [1]. A efectos del presente trabajo, se toma ese prototipo como una pieza de hardware cohesiva a la que se le añadirá funcionalidad. El estado previo de este se describe en el epígrafe 2.1 Estado previo del prototipo. Los cambios relevantes que sufrió durante el desarrollo del trabajo se describirán en el epígrafe 5. Desarrollo del sistema (ver Fig. 1)

Raspberry Pi 5:

Fig. 3. Microcontrolador/minicomputador Raspberry Pi 5.

El proceso de selección de esta placa se analiza en detalle más adelante, ya que es uno de los objetivos del trabajo. Se trata de una placa o microcontrolador con Sistema Operativo; de hecho, muchas veces el marketing la describe como un ordenador. El SO elegido para operar con el hardware fue Raspberry Pi OS. Para el almacenamiento del SO y del sistema de archivos se eligió una tarjeta SD. Las características específicas pueden encontrarse en la Tabla II.

Raspberry Cam Module 3:

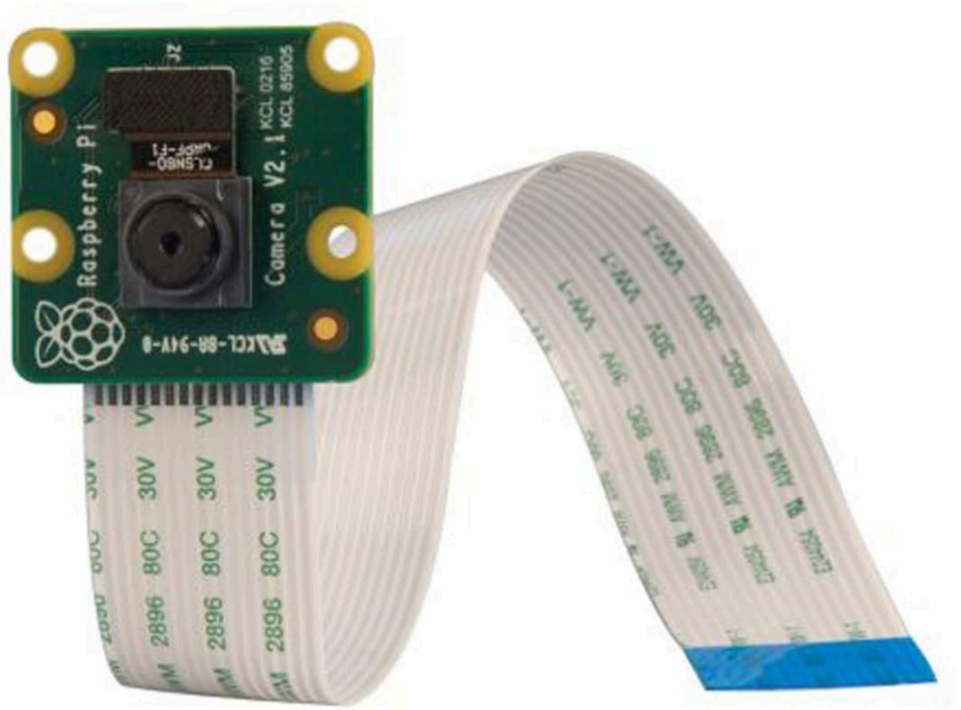


Fig. 4. Cámara Raspberry Cam Module 3 con cable de conexión.

Una vez se eligió trabajar con la gama Raspberry, se decidió también trabajar con su gama de cámaras debido principalmente a la compatibilidad entre ellas. La elección de esta cámara hizo necesario el uso de la librería libcamera para acceder a la información recogida por el sensor mediante código. Esta cámara cuenta con un sensor de 12 MP (MegaPíxeles) IMX708 Quad Bayer, con HDR (High Dynamic Range) y 4.74 mm de distancia focal. Usa el protocolo CSI-2, que permite obtener información de la cámara, y soporta los protocolos de comunicación serial I2C *fast mode* y *fast mode plus*.

Power Bank 24000mAh:

Fig. 5. Batería portátil de hganus.

Es una batería externa de seis celdas de polímero de litio para alimentar la placa. Posee conectores de carga y alimentación de tipo USB (Universal Serial Bus) C, 5V, 5A. Estas características corresponden a los requerimientos de alimentación de la Raspberry Pi 5, y se usará para su alimentación en pruebas de grabación en movimiento.

3.3 Marco teórico

Como preámbulo a este marco teórico, debemos explicar en primer lugar unos cuantos conceptos básicos que nos ayudarán a desarrollar constructos de mayor complejidad. Estos conceptos son los que a continuación se enumeran:

Píxel:

Es la unidad mínima de información visual, materializada en forma de cuadrado y organizada en forma de matrices o mosaicos compuestos por muchos de estos píxeles a la hora de representar una imagen digital [13]. Dependiendo de la forma en que se codifica la información, el píxel puede contener información sobre el color, por ejemplo, codificando la información en RGB, mediante esos tres valores primarios de la luz, y dividiéndolos en tres canales: un primer canal para el color rojo, otro para el verde y el último para el azul, con estructuras de mezclas cromáticas similares a las que se producen en nuestra visión humana

mediante las células fotosensibles al color, denominadas conos, alojados en la retina. Otros valores codificables en el píxel, además del tono de color, son el brillo y la saturación, o la luminancia (siempre y cuando estos valores se representen mediante el sistema HSV o en HSL), o simplemente representados mediante diversos valores tonales de gris, en una escala entre el blanco y el negro puros, cuando se trabaja en escala de grises, coloquialmente conocido como conversión a "blanco y negro". En la Fig. 6 se puede ver una imagen ilustrativa del funcionamiento de los píxeles:

Fig. 6. Imagen que muestra extractos a gran escala o tamaño de la composición de píxeles de la imagen digital de fondo. Extraída de https://www.uv.mx/personal/lenunez/files/2013/06/INICIACION-A-LA-FOTOGRAFIA-DIGITAL-DeCamaras.pdf

Kernel:

Es una matriz cuadrada, de dimensiones impares, centrada en el píxel central que queremos calcular o procesar. Dentro de esa matriz, cada elemento representa el peso que tendrá el valor del píxel sobre el que está produciéndose la operación de convolución que se genera con dicha matriz. Por esa razón, al kernel también se le llama matriz de convolución, máscara, o filtro [14][15][16]. Intentando aportar una explicación más básica, podríamos decir que el kernel marca los pesos en una media ponderada de los píxeles que tiene en cuenta. En la Fig. 7. se puede ver la representación matemática de un kernel muy común.

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Fig. 7. (S. R. Fernando, *3x3 Gaussian Kernel* [14]) Este es un kernel Gaussiano 3 x 3 usado en suavizado Gaussiano (desenfoque).

Difuminado/Desenfoque/Suavizado:

De forma resumida, podríamos definir la operación de desenfoque como esa que realizamos píxel a píxel. Usando un kernel adecuado, y realizando concretamente la convolución de suavizado de los píxeles (ya que con otros kernels podemos detectar bordes, realizar operaciones de *sharpen* o afilado de la imagen, por ejemplo [14]), se obtiene un difuminado de la imagen mediante una cierta pérdida de detalle [14][15][16]. Se podría decir que actúa como un filtro paso bajo, eliminando en gran medida el ruido de alta frecuencia que es inevitable que capture la cámara del robot. En la Fig. 8. Se puede ver un ejemplo del antes y el después de la aplicación de un “difuminado medio".

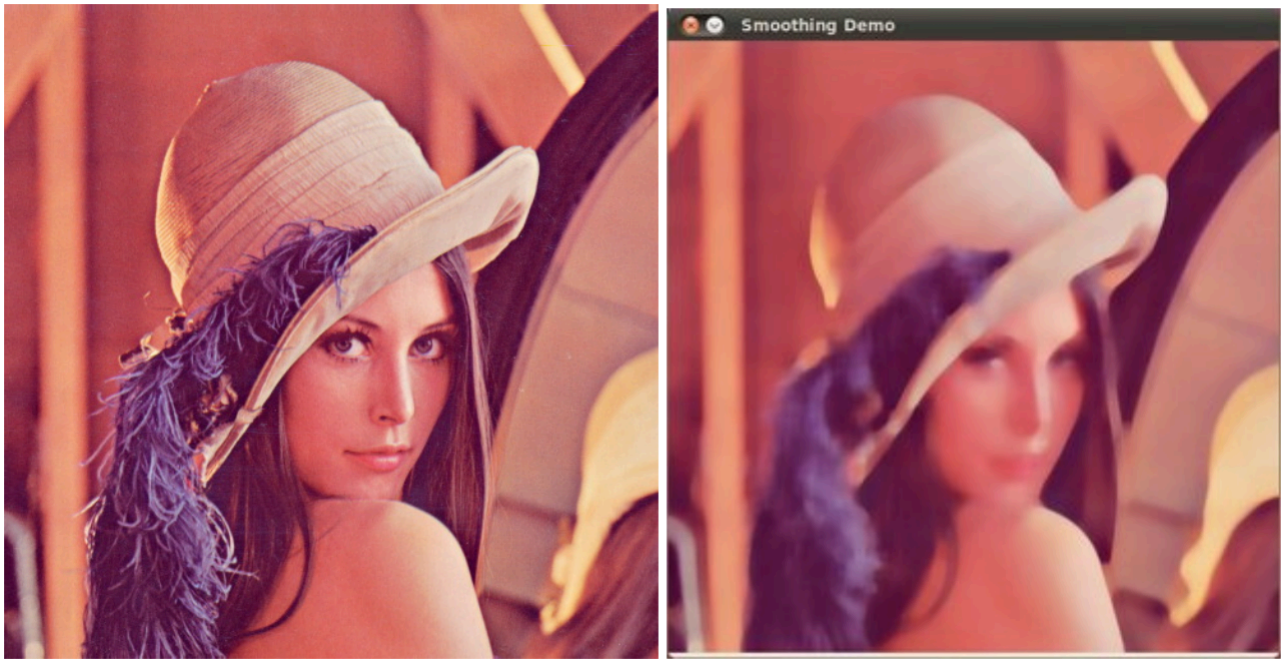


Fig. 8. (OpenCV documentation [16]) De izquierda a derecha: imagen original, e imagen aplicando la función *`medianBlurr()`*, que calcula un difuminado "medio".

Sobel:

La explicación de lo que es una derivada queda fuera del alcance de este trabajo. Pero, conociendo el concepto, y como se explica en [17], podemos ver que, si tenemos como función la intensidad de los píxeles de una imagen, los máximos al calcular su derivada nos darán como resultado los bordes de la imagen, es decir, los cambios bruscos de intensidad producidos en la imagen. Es aquí, precisamente, donde entran los operadores Sobel (1)(2), que no son más que kernels que nos permiten aproximar esta derivada de forma discreta:

Sobel horizontal 3x3:

$$\begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \quad (1)$$

Sobel vertical 3x3:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} \quad (2)$$

Cabe destacar también que podemos encontrar estos operadores con los símbolos cambiados. Sin embargo, esto no debería influir en los resultados finales por la forma en que se calculan [18].

En la documentación de la librería OpenCV encontramos también un operador llamado Scharr, que supuestamente da mejores resultados con un tamaño de kernel de 3x3. Este operador sería igual que el Sobel descrito, pero sustituyendo los números 1's por 3's y los números 2's por 10's. Sin embargo, la mayoría de la bibliografía consultada usa el kernel anteriormente descrito.

Una vez que se realiza la convolución de ambos operadores con la imagen filmada con la cámara, obteniendo G_x y G_y respectivamente, se calcula el gradiente total G con la ecuación (3):

$$G = \sqrt{G_x^2 + G_y^2} \qquad (3)$$

Como se puede ver en la ecuación anterior, al elevar al cuadrado ambos términos, eliminamos la posible discrepancia que nos daban los signos, como se mencionó anteriormente. En muchos casos, para simplificar el cálculo, en vez de esta operación, sencillamente se suman G_x y G_y , lo cual no es del todo exacto, pero, para muchas aplicaciones, es más que suficiente.

Con los resultados G_x y G_y y la ecuación (4) también se puede obtener la dirección estimada del borde, la cual requeriremos para realizar, por ejemplo, el algoritmo denominado Canny Edge Detection, que explicaremos a continuación.

$$\vartheta = atan(\frac{G_y}{G_x}) \qquad (4)$$

Para la tarea de detección de bordes en la imagen, hemos barajado las técnicas más comunes a lo largo de las publicaciones estudiadas en esta investigación, a saber:

Detector de bordes Canny (Canny Edge Detector):

Este algoritmo persigue una única detección por cada borde, logrando un error mínimo en la distancia entre el borde detectado y el real, y una tasa de error baja [19]. La explicación de la técnica proviene de [20], que a su vez se cita en [19], y de [21]:

1. Se recomienda realizar un desenfoque, difuminado o suavizado gaussiano para eliminar o suavizar el ruido y/o bordes menos pronunciados, que podríamos decir que son bordes que una persona no consideraría bordes reales. Al decir que el desenfoque es gaussiano, sólo nos estamos refiriendo a que el kernel utilizado se aproxima a una distribución normal, si tomamos el píxel central como la media/mediana. Esta solución también suele preferirse a un difuminado "medio" (*mean/normalized blurr*), que usaría un kernel en el que todos sus elementos son 1, ya que la solución descrita preserva mejor ciertos detalles [2].

2. A continuación, se halla el gradiente de intensidad, normalmente aplicando los operadores Sobel. Es importante decir que el kernel de Sobel tiene incorporado un filtro de Gauss ya incorporado, pero, aún así, hay autores que recomiendan hacer un desenfoque previo, como el descrito en el paso 1.

3. En tercer lugar, se realiza la tarea de eliminación de píxeles que no formen parte de líneas finas de los bordes máximos. En otras palabras, Sobel nos da unos bordes de grosor variable y, lo que se hace en este paso, es intentar detectar dentro de esta línea el "borde real" y dejar una línea de un solo píxel de grosor. Esta tarea se realiza utilizando la información de direccionalidad de los bordes que se obtiene con los operadores Sobel. Así, se examina si en un borde los píxeles a un lado o a otro del mismo son menos intensos (y, por tanto, si es más probable que el borde real se encuentre ahí), es decir, perpendicularmente a la línea. A lo largo del borde resulta indiferente si el borde se hace más pronunciado o menos. Se puede decir que se están buscando los máximos locales. Dejando sólo estos valores máximos, eliminamos el resto. En la Fig. 9. se pueden observar unas imágenes de [21], que tratan de dar una explicación gráfica a esta selección de máximos

Fig. 9. (M. Pound, Dibujos ilustrativos de [21]). A la izquierda, una aproximación de los valores atravesando un borde perpendicularmente. A la derecha, la línea recta representaría el máximo local de grosor mínimo con el que nos queremos quedar. Este se encuentra, al ser un máximo, comparando con los valores contiguos.

4. Por último, se realiza un proceso de filtrado por histéresis, aplicando dos *thresholds* (umbrales) o límites a los píxeles del gradiente obtenido: 1) un límite inferior, por debajo del cual se descartarán los píxeles que no lo superen, y 2) un límite superior por encima del cual estos píxeles sí que se consideran "bordes fuertes". Entre ambos límites, los píxeles resultantes se declaran como "bordes débiles". Y esos píxeles débiles se descartarán también si no aparecen contiguos a píxeles fuertes.

Esta técnica de detección de bordes es de las más populares por su efectividad y relativa sencillez. Tanto los subprocesos descritos como el algoritmo Canny completo están implementados en la mayoría de librerías de visión artificial. En la Fig. 10 se puede ver un ejemplo de los resultados:

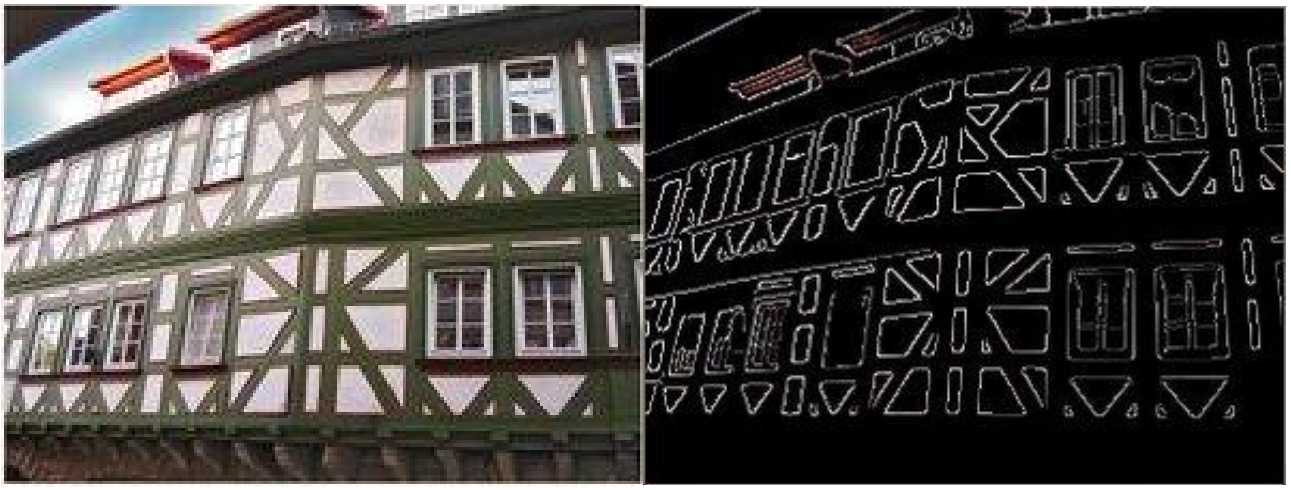


Fig. 10. Imagen de entrada al Detector de bordes Canny (izquierda), y resultado del algoritmo (derecha). Imágenes sacadas de [19].

HSV:

Es el espacio de color que, de forma similar al RGB, es el más comúnmente utilizado por las cámaras. Este espacio codifica la información visual de una imagen en tres canales que describimos a continuación:

El canal 1 corresponde al tono o hue (H), el canal 2 corresponde a la saturación o saturation (S) de color y el canal 3 corresponde al llamado valor o value (V).

Con el canal de Tono (H) obtendríamos la información que nos permite diferenciar entre los distintos tonos de color. Es decir, que este es el canal que permite discriminar y diferenciar claramente entre colores como el rojo, verde, amarillo, etc. El canal de saturación (S) nos da información sobre la "pureza" del color registrado, es decir, sería algo así como la intensidad del mismo. La saturación variará también en función de la cercanía o lejanía de un determinado color a la escala de grises. Por último, el canal de Valor (V) nos da una información que podríamos equiparar al brillo del color detectado en cada píxel [22].

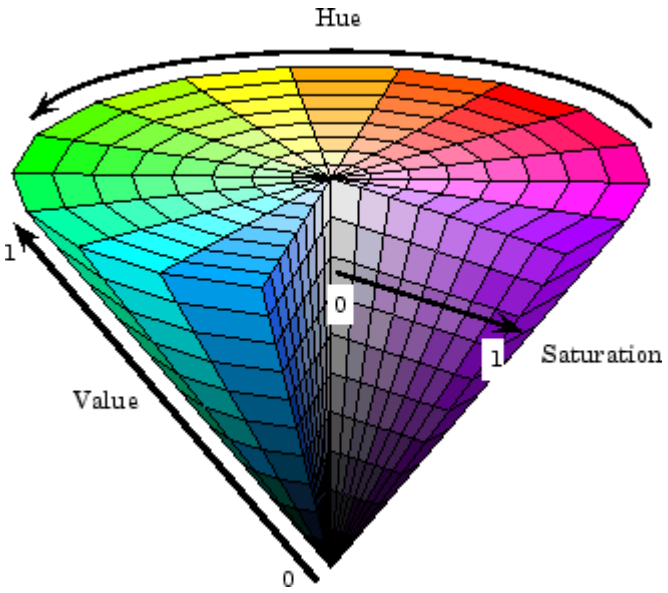


Fig. 11. Representación del espacio de color HSV en forma de cono, de [22].

Como se puede observar en la imagen superior, que representa el sistema HSV mediante un cono invertido (Fig. 11), la dimensión del tono (H) es circular. Esto supone un problema, ya que, al expresarse este valor de forma numérica, por ejemplo de 0 a 360 (como los grados trigonométricos), el valor 360 y el 0 son muy similares. Si los tonos en los que nos queremos centrar caen en este "corte", se podría generar un problema añadido por su discrepancia de valor en esa forma numérica. Es por esta razón por la que muchas veces se rota la dimensión H para que el corte entre esos valores extremos "caiga" sobre un tono que no sea relevante para nuestro objetivo en cuestión. En este trabajo este proceso de rotación de valor de tono ha resultado de especial relevancia.

4| Diseño del sistema

Basándose en los requisitos previos que presentaba el prototipo antes de acometer este trabajo, y en los que se pensaba que a priori también marcarían los objetivos planteados, el primer paso que se dio fue realizar un primer diseño del sistema, representado gráficamente en la Fig. 12:

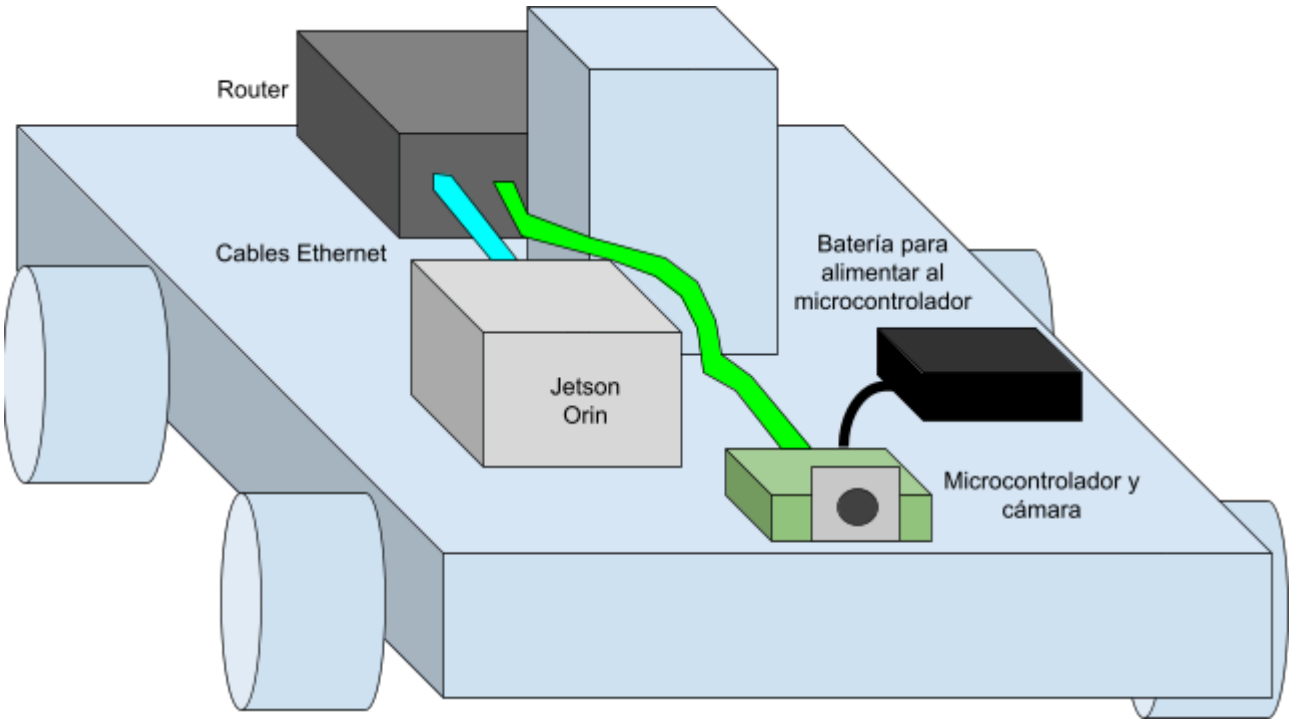


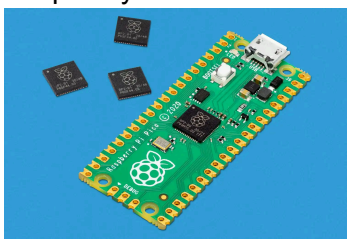
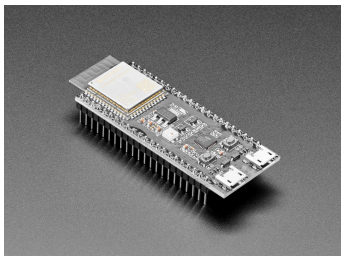
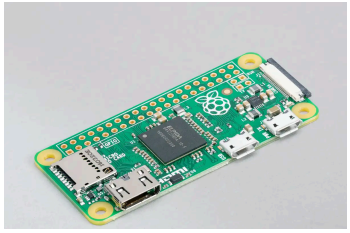
Fig. 12. Dibujo esquemático de la disposición y conexiones de los elementos del sistema existente en el prototipo y los que se incorporarían como parte del sistema de visión artificial. Fuente: Elaboración propia.

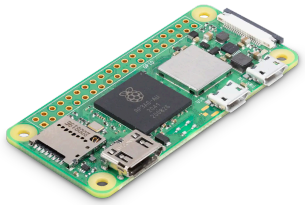
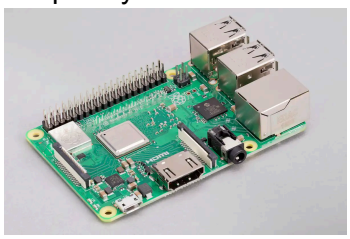
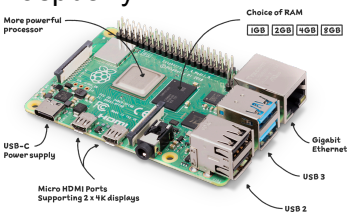
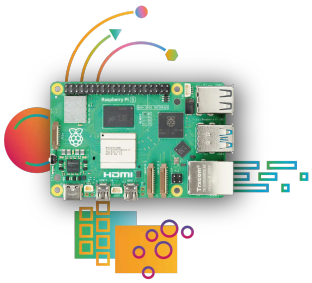
Como se puede observar, la cámara de guiado se ubicaría mirando en el sentido de avance del prototipo, ya que el router se encuentra en la parte posterior y la computadora de NVIDIA también se encuentra situada en la parte delantera. Las colocaciones deben distribuirse de esta manera ya que esta cámara de guiado necesitará dirigirse y enfocar a lo que el robot tiene delante. Esta disposición, además, mantiene cierta coherencia con la disposición que ya tenía el prototipo, dejando las máquinas de procesamiento (el nuevo microcontrolador, la placa de control a bajo nivel y la Jetson) delante, junto a sus fuentes de alimentación.

4.1 Elección de hardware

Con los requisitos y la estructura anterior en mente, se procedió a la elección del hardware adecuado para el trabajo de guiado del robot mediante un sistema de visión artificial determinado. Como ya hemos apuntado, comparando con otros microcontroladores, y teniendo en cuenta su tamaño, su capacidad de procesamiento y un valor añadido, como fue el hecho de tener incorporado un puerto Ethernet, se consideró que la placa Raspberry Pi 5, así como su cámara asociada, la Raspberry Pi Module 3, era la opción más adecuada para este propósito. La comparativa realizada para tomar esta decisión se presenta en la Tabla II:

TABLA II
Comparativa de microcontroladores

Modelo placa	Ethernet	Tamaño	RAM y procesador	Compatibilidad con cámaras
Raspberry Pi Gama Pico 	No	51 mm × 21 mm	Dual-core Arm Cortex-M0+ processor, flexible clock running up to 133 MHz 264kB on-chip SRAM	No se menciona nada en la página del producto, dando a entender que no dispone de puerto ni posiblemente potencia para hacer control de cámaras
Gama ESP 	No nativo	62mm x 25mm alrededor	ESP32-S3-WROOM -2 (1 GHz, 8 MB PSRAM)	Módulo ESP-CAM (no encontrado en adafruit)
Raspberry Pi Zero 	No nativo (posible shield?)	65mm x 30mm	1 GHz single-core CPU 512 MB RAM	Con un cable especial, es compatible y soportada por las cámaras de Raspberry y su biblioteca específica.

Modelo placa	Ethernet	Tamaño	RAM y procesador	Compatibilidad con cámaras
Raspberry Zero 2 W 	Igual que Raspberry Pi Zero pero: At the heart of Raspberry Pi Zero 2 W is RP3A0, a custom-built system-in-package designed by Raspberry Pi in the UK. With a quad-core 64-bit ARM Cortex-A53 processor clocked at 1GHz and 512MB of SDRAM, Zero 2 is up to five times as fast as the original Raspberry Pi Zero.			
Raspberry Pi 3 B+ 	2.4GHz and 5GHz IEEE 802.11.b/g/n/ac wireless LAN, Bluetooth 4.2, BLE Gigabit Ethernet over USB 2.0 (maximum throughput 300 Mbps)	88mm x 56mm	Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC @ 1.4GHz 1GB LPDDR2 SDRAM	Soporte nativo de cámaras Raspberry
Raspberry Pi 4 	2.4 GHz and 5.0 GHz IEEE 802.11ac wireless, Bluetooth 5.0, BLE Gigabit Ethernet	88mm x 56mm	Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.8GHz 1GB, 2GB, 4GB or 8GB LPDDR4-3200 SDRAM (depending on model)	OpenGL ES 3.1, Vulkan 1.0 Soporte nativo de cámaras Raspberry
Raspberry Pi 5 	Gigabit Ethernet, with PoE+ support (requires separate PoE+ HAT)	88mm x 56mm	Broadcom BCM2712 2.4GHz quad-core 64-bit Arm Cortex-A76 CPU. Up to 8 GB RAM (also 4GB version)	VideoCore VII GPU, supporting OpenGL ES 3.1, Vulkan 1.2 Soporte nativo de cámaras Raspberry

El puerto Ethernet incorporado en la propia placa parecía de gran utilidad, ya que el resto de puertos USB del prototipo se encontraban ocupados por otros componentes de funcionamiento, y era imprescindible encontrar una forma de comunicarse con el resto del prototipo de forma segura (ya que las comunicaciones inalámbricas pueden dar problemas). Además, nuestro prototipo ya incorporaba un router para otras funcionalidades, y [1] asegura que ya provee una red local, lo que seguramente haría dicha conexión todavía más sencilla.

El primer obstáculo con el que nos encontramos en las pruebas vino precisamente de la mano de este primer avance. El hardware, es decir, la placa y la propia cámara, tardaron demasiado tiempo en llegar a la sede de la ETSIDI (la facultad en la que se desarrolló el trabajo) desde que fueron encargadas. Por suerte, otro proyecto de TFG de esta misma universidad también utilizaba una Raspberry Pi 5 y su cámara compatible, Raspberry Cam Module 2 (una versión anterior a la elegida para nuestro proyecto). Esta versión de cámara y de placa nos fue prestada hasta que llegó el pedido respectivo a este trabajo. Sin embargo, en el paso de la placa Raspberry Pi de su versión 4 a la 5, también hubo un cambio significativo en el diseño del puerto de la cámara, haciendo necesario el uso de un adaptador que nos llegaría más tarde, con los componentes adquiridos. Mientras se esperaba su llegada, se comenzó a desarrollar el software con un modelo de cámara USB Logitech C920 HD Pro, una webcam que suele utilizarse para realizar videoconferencias, de la que ya se disponía.

En otro orden de cosas, para alimentar la placa durante las pruebas, cuando esta esté montada en el propio robot, se necesita una fuente de energía portátil, que también pueda ir montada en el robot. Cuando se estaba finalizando el trabajo, se estimó que, debido a su consumo, que es de 5V 5A, es decir, 25W (aunque la fuente de alimentación incluida soporta 27W), se podría alimentar portablemente con una fuente de alimentación portátil o batería externa de 24000mAh, ya que, entre sus características, asegura poder proveer este voltaje y amperaje. Se adquirió este componente recientemente, pero aún no se han hecho pruebas para comprobar que funciona correctamente.

De igual modo, y en paralelo a esta tarea, se fue realizando un proceso de búsqueda y rastreo de *papers* y artículos de investigación, así como de repositorios o proyectos relacionados con el que hemos abordado en este TFG. Dicho proceso de investigación y búsqueda de referentes bibliográficos se produjo de manera continuada durante todo el proceso de trabajo.

5| Desarrollo del sistema

5.1 Preparación de las imágenes

Al iniciar el proceso de experimentación práctica, se comenzó a desarrollar un algoritmo capaz de realizar dos tareas "a alto nivel" necesarias para el objetivo planteado. La primera consistía en la separación de las líneas de carril del resto del fondo. Debido a su pigmentación particular, lo más intuitivo, y de acuerdo con bastantes precedentes en la bibliografía consultada [6][7][8], era usar la información de color que alberga cada píxel para realizar un filtrado.

En primera instancia, la mayoría de la bibliografía revisada al efecto recomendaba el uso del espacio de color HSV para el filtrado por color. En [14], por ejemplo, hay una implementación básica en este espacio. Sin embargo, se realizaron algunas pruebas usando los espacios RGB y HSL. La hipótesis en ambos casos era opuesta: para el uso de canales RGB es más complejo describir un rango de colores sin que importe la iluminación sobre el objeto; en cambio, en HSL el modelo sugiere que, con filtrar únicamente en el canal L (luminancia), podríamos obtener cualquier tono blanco/claro sin importar demasiado su tono o saturación. Sin embargo, en ambos casos los resultados fueron inferiores a los obtenidos con el sistema HSV, que, a la postre, fue el método con el que se obtuvo mejor rendimiento. Aunque no se pueden cuantificar los resultados de precisión de forma numérica, sí se elaboraron varias comparativas a ojo entre los usos de estos tres sistemas para llegar a la conclusión final sobre lo pertinente del uso del HSV, que resultó finalmente el más apropiado para nuestros propósitos del sistema de filtrado más preciso del color de la línea separadora.

El segundo obstáculo con el que nos encontramos es el que ya hemos descrito en la propia definición del espacio de color HSV a la hora de discriminar valores numéricos entre diversos tonos en el canal H. Como hemos adelantado en el apartado anterior, este problema fue solucionado rotando el valor numérico de determinados colores en ese canal. Una optimización de este algoritmo de rotación (propio) se describe en el apartado 5.2 Mejoras en velocidad de procesamiento.

La segunda gran tarea del algoritmo consistió en clasificar los píxeles resultantes del filtrado en líneas que podemos clasificar, de alguna manera, identificando puntos de estas, por ejemplo, definiendo una sucesión de estos puntos, como un polinomio, una spline… etc. Esta clasificación se realizó de distintas formas a lo largo del trabajo, las propuestas se describirán más adelante en orden cronológico, tal y como se fueron produciendo a lo largo del desarrollo de este TFG.

En primera instancia, la primera versión de este algoritmo consistía en los siguientes pasos:

1. El paso de la imagen al espacio de color HSV, y la rotación del canal de tono (H) en función de las necesidades que se produjesen según el color con el que estuviese pintada la pista de atletismo (normalmente 50º, si se toma el rango de H como 360º. Este número se decidió a base de prueba y error, hasta comprobar que el color quedaba en un rango adecuado para filtrar, lo que se determina de forma visual, como se puede ver en la Fig. 13). Se usó la función `cvtColor()`, que toma como parámetros la imagen original, la imagen en la que se almacenará el resultado, y un tercer parámetro de tipo `enum`, que especifica los espacios de color de origen y destino.

2. El filtrado por valores en los tres canales, H, S y V (ajustado a mano al color de la línea en cuestión, ver Fig. 25). Este filtrado nos da una imagen en blanco o negro puros. Se usa la función `inRange()` que toma la imagen que deseamos filtrar y los límites inferiores y superiores en cada canal.

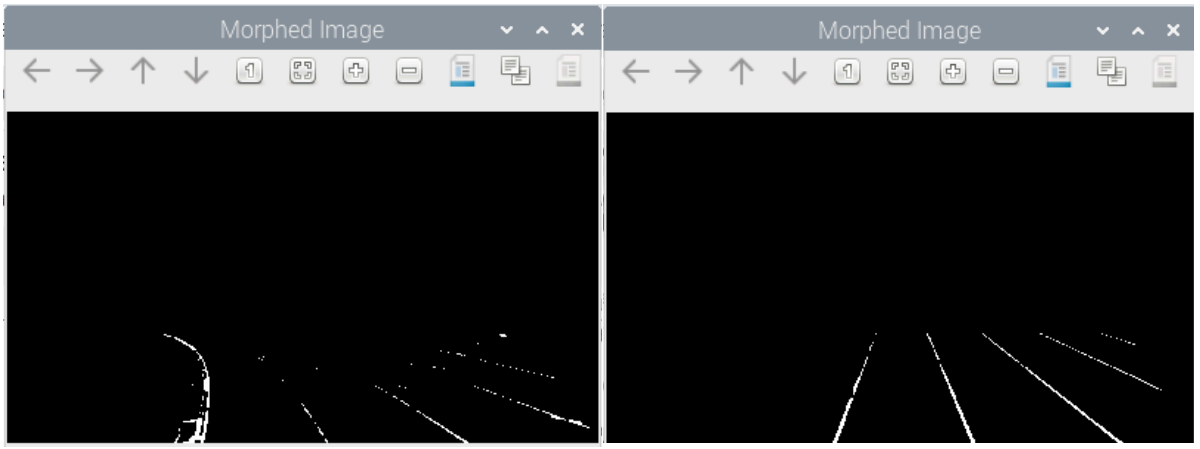


Fig. 13. Comparación de resultados obtenidos de filtrado en color con espacio de color HSL (izquierda) y HSV (derecha) con la rotación en el canal H y ajuste de filtros con mejores resultados y ROI (Region Of Interest) aplicada en ambos casos. Fuente: Elaboración propia.

3. Ejecución de la operación morfológica denominada *closing*, y que, como su propio nombre indica, consiste en "cerrar" los pequeños agujeros que quedan dentro de las líneas generadas con los procesos anteriores. Dicha operación utiliza dos convoluciones en sucesión: una primera de dilatación o *dilation* y, acto seguido, otra de erosión o *erotion* [23]. Podría haberse aplicado un proceso parecido de *opening* y, aunque se experimentó también con este, su rendimiento aumentaba la carga computacional sin producir resultados decentes e incluso, a veces, ni siquiera observables (en este caso, eliminar pequeños puntos fuera de las líneas). Este problema fue solucionado más tarde en una nueva versión del algoritmo que se describirá más adelante. La función concreta de OpenCV usada es `morphologyEx()`, que toma como argumentos: la imagen binaria por procesar, la imagen en la que guardar los resultados, un parámetro de tipo enum que elige la operación, y un objeto que representa el kernel. El tamaño de este último es de 3x3, ya que, con tamaños mayores, se observó que los agujeros más pequeños no se cerraban.

Fig. 14. Imagen original y resultado tras realizar la operación de cerrado, sacada de [23].

4. Después se trabajó en dos líneas de actuación distintas, que conllevaban dos soluciones que se experimentaron de forma paralela:
   (Opción 1) El uso de Canny Edge Detection, con la peculiaridad de utilizar la función `medianBlur()` (de la librería OpenCV) en lugar de `GaussianBlur()`, y que, según [2], es más efectivo para estos propósitos. Volviendo a la Fig. 10, se puede ver un ejemplo del resultado de estos pasos.
   (Opción 2) El uso de la función `findContours()`, también de la librería OpenCV, basada en [24].

5. Pintado digital de las líneas obtenidas con cualquiera de las dos opciones anteriores. En el caso de elegir la opción 1, utilizamos una función de elaboración propia. Y, en el caso de la opción 2, trabajamos con la función `drawContours()`. Estas dos operaciones realizadas sobre la imagen original de vídeo se probaron para comprobar que el resto de partes del algoritmo funcionaban tal y como se esperaba, de forma visual. Para hacerse a la idea, se puede ver la Fig. 18, aunque los puntos obtenidos y la forma de pintarlos no son exactamente iguales, como se explicará más adelante.

5.2 Mejoras en velocidad de procesamiento

Una vez que se obtuvo el primer algoritmo funcional y un par de vídeos con imágenes de pistas de atletismo, obtenidas a través del canal de vídeos YouTube y de Google Images, se probó la velocidad de procesamiento. Al principio, y con las primeras versiones del algoritmo, se logró obtener un resultado de velocidad de procesamiento de imágenes que variaba de 12 a 7 fotogramas por segundo, a partir de ahora “fps”. Buscando una solución para esta casuística, se pensó en utilizar la GPU (Graphics Processing Unit). Pero, al investigar un poco más, y a

criterio de los tutores, se acordó que esta solución era demasiado compleja y que, potencialmente, la GPU podría no ser tan potente como para que el tiempo empleado llevara a una mejora significativa. Por tanto, se realizó un proceso de paralelización mediante *threads*, reduciendo así el tiempo de procesado hasta 10 milisegundos. Como referencia, para alcanzar 24fps, una medida estándar, el tiempo entre frames debería ser de entre 41 y 42 milisegundos. En concreto, se emplearon *threads* para optimizar la rotación del canal H; se aplicó la función `parallel_for` de la librería OpenCV. Esta operación separa de forma inteligente todas las operaciones que se le proveen, y, en este caso, se aplicaría al cambio del canal H píxel a píxel, en *threads* o hilos de ejecución virtualmente paralelos, en vez de realizar cada operación una detrás de otra, de forma secuencial.

De igual modo, se ejecutaron algunas otras mejoras menores (como el uso de menos variables o pasos intermedios, la optimización de algunas llamadas a funciones, etc.). Los procesos de optimización de más peso, a nivel de código, fueron los siguientes:

- Paso de parámetros constantes como `const [type]&` (referencia constante): Evita el copiado de memoria cada vez que se pasa como argumento a la función. Esta mejora empieza a ser necesaria, sobre todo, cuando se empieza a trabajar con los contornos de las líneas, que son vectores de puntos. Esta mejora ha sido vital para evitar el copiado de datos de gran tamaño, como los descritos, cada vez que se llama a una función, utilizando los que ya existen, ya que sólo se van a leer, y no a modificar.

- Uso de `std::vector.emplace_back()`: Cuando creamos un objeto específicamente para meterlo en un `std::vector`, si usamos el método `push_back()`, creamos el objeto y luego el vector tiene que copiarlo. Sin embargo, con `emplace_back()`, en vez de pasar un objeto como argumento, se pasan únicamente los parámetros para construirlo, por lo que sólo necesitamos realizar una operación de memoria.

- Uso de `std::vector.reserve()`: una de las ventajas de la clase vector es que es de tamaño variable, y realiza las operaciones de aumento de tamaño en la memoria automáticamente, cuando es necesario. Sin embargo, cada vez que se añade un elemento y se necesita aumentar el tamaño del vector, la operación a nivel bajo que se encarga de todo esto es relativamente costosa. Si usamos `reserve()` en los vectores en

los que sabemos con anterioridad qué tamaño van a tener, esta operación sólo se hará una vez, en vez de repetirse cada vez que se añade un elemento.

La optimización mayor del proceso llegó como sugerencia de mis tutores. Esta solución consistía en reducir la resolución de cada imagen y, por tanto, el número total de píxeles por procesar. Bajando la resolución de 1080p a 720p, el algoritmo pasó a alcanzar en torno a los 24 fps, el *framerate* (o velocidad de fotogramas) original del vídeo de prueba, comprobando así que la efectividad de la solución resultaba óptima para nuestro propósito. A su vez, esta reducción de resolución no conllevaba una reducción apreciable de la exactitud del algoritmo. En el apartado 6. Pruebas y resultados se presenta un estudio de las mejoras descritas.

5.3 Algoritmo de detección de la posición de las líneas

Después de llegar a un consenso con los tutores del trabajo, se empezó a ampliar el algoritmo descrito en el epígrafe 5.1 Preparación de las imágenes:

Se usó la opción 2 del paso 4, descrito en ese epígrafe, ya que, con ella, se obtienen coordenadas de imagen (píxeles) de los contornos. Esta función nos da siempre contornos que logra definir como formas o polígonos cerrados. A tenor de esta información, se desarrollaron algoritmos capaces de obtener los centroides de estos. Un centroide, como su propio nombre sugiere, sería el "centro de gravedad" del área contenida en un contorno. Por cómo se definen estos centroides, pueden quedar fuera del área de la línea si se trata de una curva, por lo que esta vía no resultó muy fructífera. También se logró obtener uno solo de los dos bordes de la línea, dividiendo su contorno en dos mitades. Elegimos el lado izquierdo de la línea de forma arbitraria, en todos los casos, lo que nos daría una aproximación de la posición de la misma, ignorando su grosor. También se implementaron funciones con respecto al filtrado o ROI (Region Of Interest), que consisten en obtener sólo la información de una región dentro de la propia imagen. En este caso, se realizó este filtrado eliminando la información fuera de esta (en una imagen binaria, poniendo el negro, o el valor numérico fuera de nuestra zona de interés). La región escogida corresponde a un trapecio que intenta rodear únicamente la pista, dejando fuera parte del suelo de alrededor y también el cielo.

Con la información de las líneas obtenidas se planteó realizar un mapeo de coordenadas de imagen convertida a coordenadas en la pista, lo cual sería equiparable a obtener distancias en

el mundo real. Este mapeo se produciría dependiendo de la posición de la cámara en el robot, que debería ser siempre fija. En función de esa posición y del ángulo de la misma, y realizando mediciones a varias alturas en la pantalla, y mediante comparación con mediciones de distancia en el mundo real, se podría obtener una función matemática que relacionase ambas dimensiones:

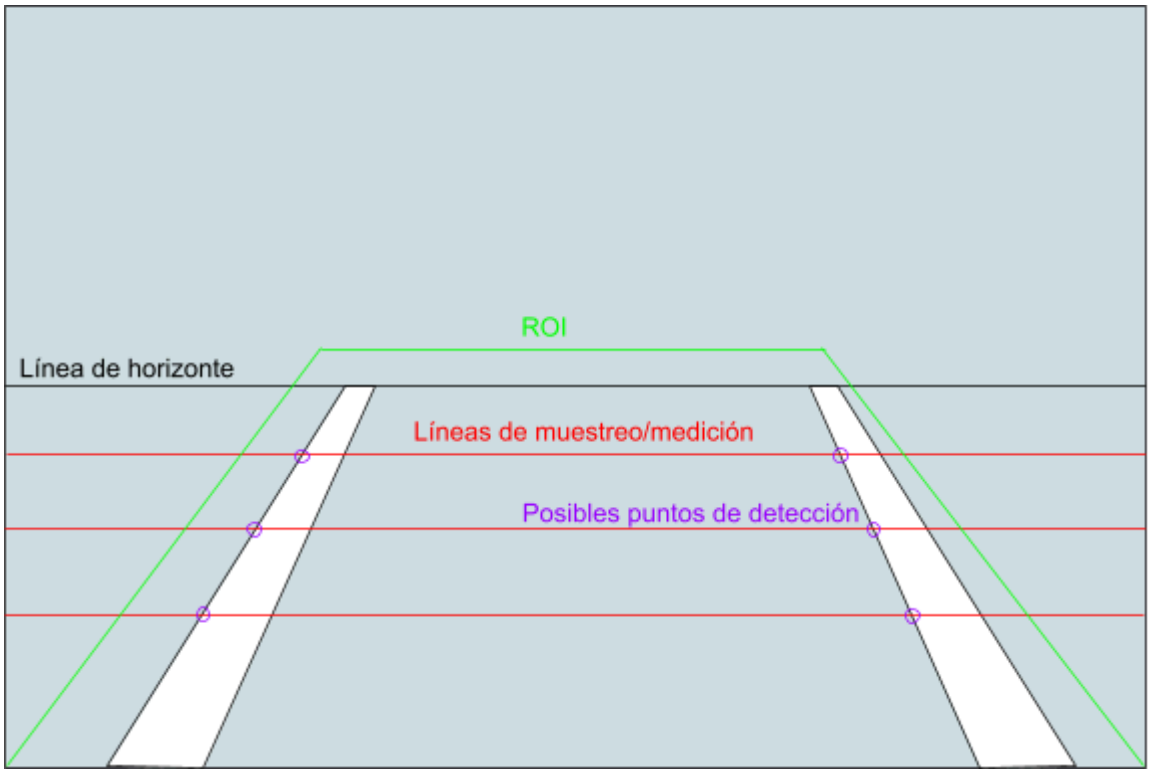


Fig. 15. Dibujo de frame con explicación de obtención de puntos. Se muestra un frame hipotético capturado por la cámara, con dos líneas a la vista. En este caso, se marcarían puntos del borde izquierdo de cada línea a tres niveles de altura. Fuente: elaboración propia.

La transformación de estos puntos en el espacio de imagen (coordenadas en píxeles) a puntos reales, permitiendo obtener datos de distancias, se planteó, en primera instancia, con dos metodologías. La primera consistía en tomar una imagen con distancias conocidas, y relacionar las coordenadas de los píxeles de los puntos conocidos con las distancias reales, utilizando, por ejemplo, el programa *MATLAB* para realizar el cálculo de la operación intermedia. La segunda opción consistiría en aprovechar primero una función ya implementada en la librería OpenCV, lo que permite transformar una imagen dando 4 puntos en la imagen original y sus coordenadas deseadas en la nueva imagen, calculando así una

matriz de transformación de perspectiva que permita transformar tanto en un sentido como en otro [25][7]:

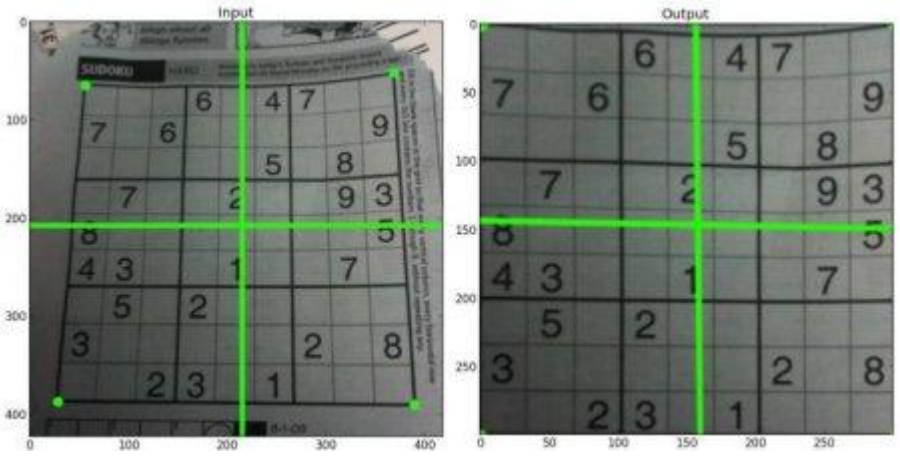


Fig. 16.1 Imagen de ejemplo de resultados del uso de las funciones mencionadas (*getPerspectiveTransform()* y *warpPerspective()*). A la izquierda, la imagen original, con líneas perpendiculares en la mitad de cada dimensión y las esquinas del papel (el plano a aplanar con la operación) marcadas en verde; a la derecha, la misma imagen corregida mediante las funciones. Imagen obtenida de [25].

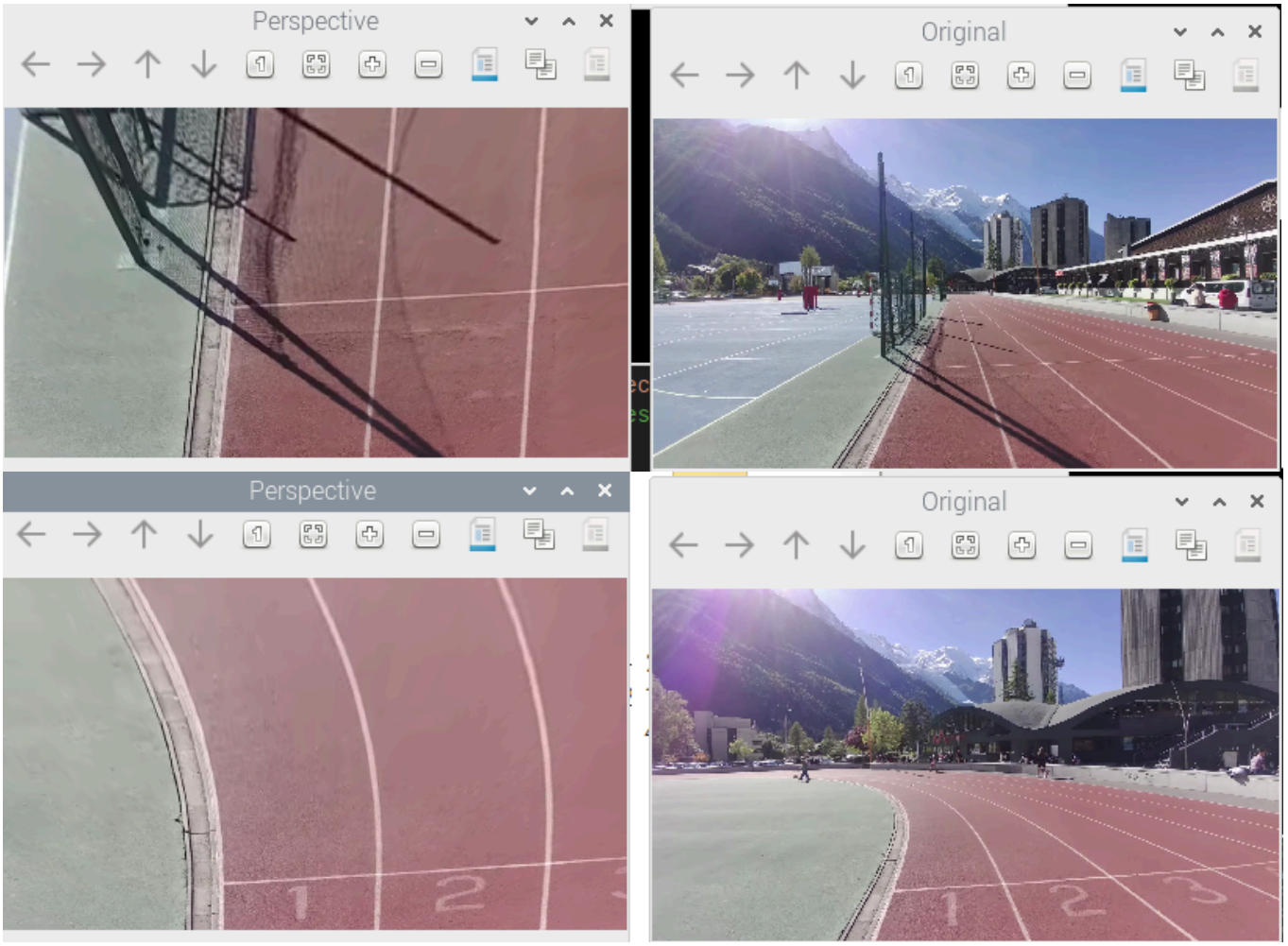


Fig 16.2 Ejemplos de implementación propia del cambio de perspectiva.
Fuente: Elaboración propia.

Con los puntos adecuados, se puede conseguir una *birds-eye view*, o vista de pájaro, que consistiría en una imagen que parecería haber sido tomada por encima del suelo, y en un ángulo perpendicular a éste, en un picado extremo, con el plano de la imagen paralelo al suelo, como se puede ver en la Fig. 16.2. Si se consiguiera esta solución, sólo se necesitaría hallar una relación, que ahora sería lineal, entre las distancias de píxeles en la imagen transformada y las distancias en el mundo real. Esta mejora o característica se comenzó a desarrollar de forma preliminar, pudiendo obtener la vista de pájaro, pero se deja y se discute también en el punto 5.2 Posibles futuras líneas de investigación.

Para tomar los puntos de las líneas en la imagen, que luego se parametrizarían, y siempre a criterio de los tutores, se cambió el paso 4 nuevamente, desarrollando una pseudo-API que

permite calcular unos histogramas particulares en imágenes binarias. Esta técnica, u otras implementaciones similares en concepto, aparecen en [2][6][7][8]. Estos histogramas consisten en un vector que contiene el número de píxeles blancos (y que, en nuestro caso, deberían corresponder a píxeles de las líneas de la pista, obtenidas en el paso 2 de filtrado de color en HSV del algoritmo, ver el epígrafe 5.1 Preparación de las imágenes) para cada columna de píxeles en una imagen. Los picos o máximos locales de este histograma, que se calculan con esta nueva API, basada en funciones de *OpenCV*, corresponderían a puntos de la línea, si se hace con la suficiente granularidad, es decir, dividiendo la imagen en franjas lo suficientemente pequeñas. Una representación gráfica de estos histogramas puede verse en la Fig. 17:

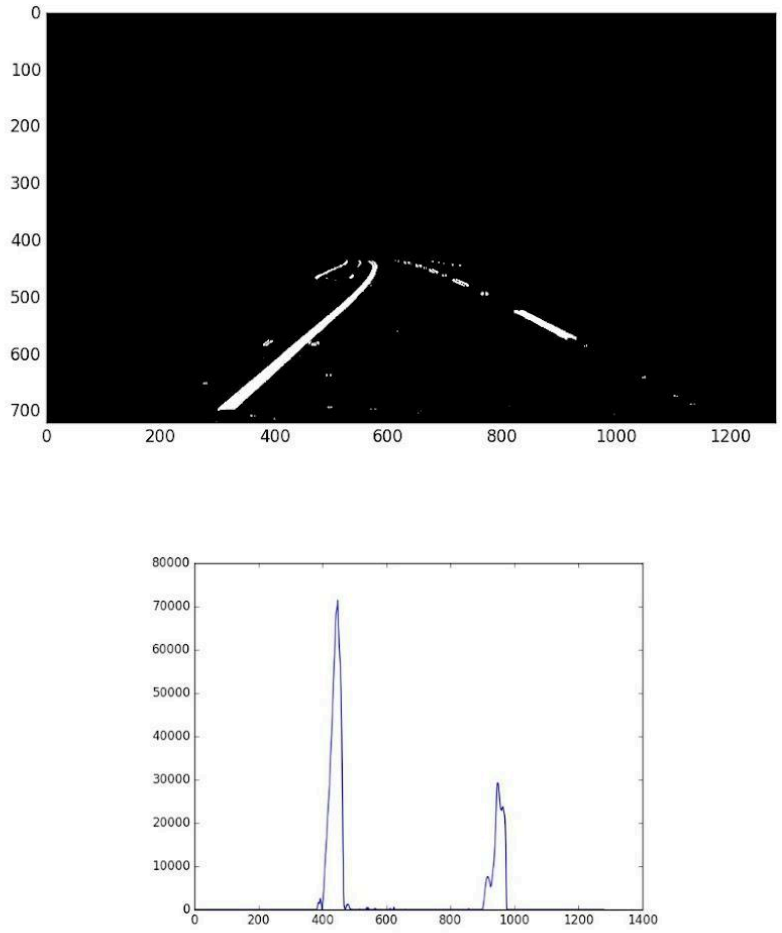


Fig. 17. Representación gráfica de una versión de histogramas obtenidos de píxeles blancos, sacada de [6].

Para evitar encontrarnos con posibles ruidos no deseados en la imagen, o líneas pequeñas, en este histograma se descartan columnas que contengan un número menor de píxeles que un determinado número específico, siendo este un parámetro ajustable a nuestras necesidades.

El concepto es similar a la propuesta anterior del paso 4, ya que ambas soluciones usan varias alturas en la imagen. Este histograma, y los puntos que luego se extraen de él, se calcularían también a distintas alturas de la imagen, en distintas franjas de la misma, para obtener distintos puntos de cada línea de carril. En nuestro caso, dividimos la imagen en franjas con el mismo ancho que la imagen original, pero en alturas que rondan los 20 píxeles (aunque es un parámetro ajustable). Sin embargo, este método es más ventajoso que el anterior, ya que no sólo podemos obtener el punto medio de cada línea (no haría falta aproximarla por su borde izquierdo o derecho), sino que también podemos escoger la granularidad y, por tanto, el número de puntos que tomamos. En la Fig. 18 se puede ver un fotograma de un vídeo de prueba, en el que se toman estos puntos a derecha e izquierda cada 20 píxeles verticales. En este caso, se calcula también la media de la posición horizontal de ambas líneas para obtener el carril medio (lo podremos llamar *midLane* de ahora en adelante), en azul:

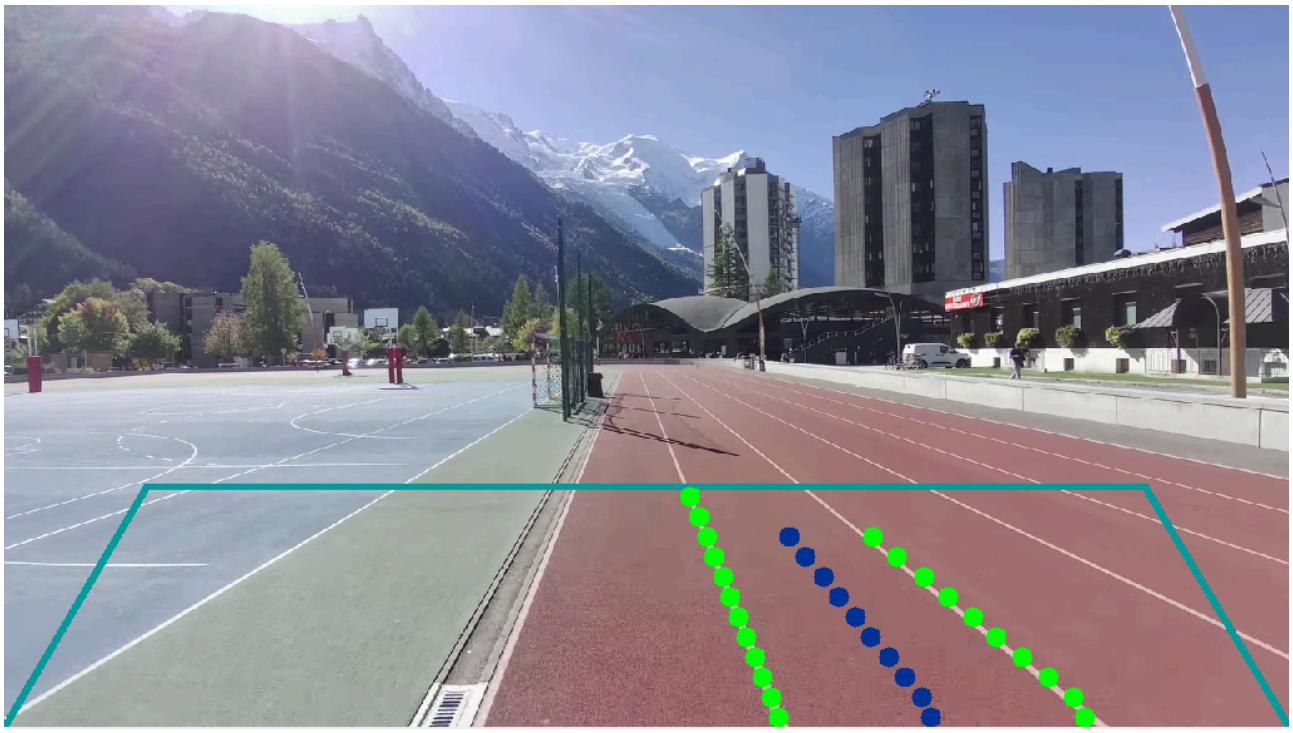


Fig. 18. Imagen con los puntos obtenidos superpuestos. Bordes de la ROI (azul verdoso o turquesa), puntos detectados de líneas izquierda y derecha del carril (verde claro/neón), puntos del carril medio o posición media del carril (azul oscuro). Fuente: Elaboración propia.

La precisión de estos puntos depende de la granularidad o número de subdivisiones que hagamos en la imagen. También es importante la granularidad para poder discernir si los puntos obtenidos son de la misma recta, ya que en la imagen es casi seguro que nos encontremos con varias líneas en cada fotograma. Para entenderlo mejor, esto se podría imaginar como un puzle de unión de los puntos marcados, en los cuales no hay una numeración ordenada. Pero, en ese caso, ¿cómo sabríamos qué puntos concretos hay que unir entre sí para formar la línea? ¿Cuáles constituyen una misma línea y cuáles no? Para resolver esta duda, los puntos del primer nivel (el de más abajo en la imagen) constituirán las "semillas" de las líneas, y cada nivel siguiente comparará sus posibles puntos con los puntos obtenidos en el nivel de altura anterior. Entre dos alturas contiguas con una granularidad adecuada, la distancia horizontal entre dos puntos de una misma línea no debería ser muy grande, incluso si la línea es curva. Por lo tanto, nuestro algoritmo tratará de buscar, de entre los máximos locales (posibles puntos de líneas), aquellos que estén a menos de una distancia especificada del punto anterior. Podríamos resumir el proceso diciendo que conseguimos resolver la incertidumbre mediante la proximidad entre puntos, es decir, que los puntos más cercanos y muy próximos entre sí pertenecerán a una única línea, ya que entre los puntos pertenecientes a distintas líneas hay una distancia mucho mayor.

Se empezó tratando de tomar dos líneas, delimitando un único carril, que se consideraron como carril izquierdo y derecho. Esto se consiguió tomando un "punto semilla" dentro del carril y extendiendo la búsqueda a derecha e izquierda, y tomando los primeros puntos que se encontrasen. Estos puntos serán en los que se base el algoritmo para encontrar los siguientes puntos de cada una de las líneas. Como primera aproximación, y también para ir adaptando los datos a una sola línea o trayectoria útil para el prototipo, para cada nivel de altura se hizo una media de la posición de ambas líneas, dando el punto medio del carril. Este mismo punto sirvió como semilla para encontrar los carriles en el siguiente fotograma. A menos que se produzca un movimiento muy brusco, el carril no debería desplazarse demasiado entre un fotograma y el siguiente y, por tanto, si lo buscamos desde la posición que conocemos ya, tenemos cierta seguridad de que las líneas que encontremos serán las del mismo carril. Esto supone una primera aproximación. Este mecanismo nos puede permitir enviar un mensaje de error o de advertencia si detectamos que el punto medio nuevo difiere mucho del que detectamos en el fotograma anterior, dando a entender que los nuevos datos seguramente no sean correctos.

Para parametrizar las líneas se experimentó con el uso de la librería *open source* o de código abierto "*Eigen"* y la librería *spline.h*, albergada en [26] y de uso libre bajo licencia *GPLv2*. La primera no llegó a producir resultados, pero se sospecha que fue debido a un fallo de implementación propio de este trabajo, y no debido a la librería. Sin embargo, con la segunda se pudieron crear *splines* de tipo "*cubic C1 Hermite spline"* para la parametrización de líneas en base a los puntos obtenidos. Se comprobó de forma visual (pintando el resultado de la parametrización sobre la imagen) que esta solución era correcta, coincidiendo con las líneas de la imagen, al menos a primera vista. También se comprobó que esta *spline* pasaba o incluía los puntos que se le daban como parámetros, lo cual es condición indispensable para que esta aproximación o parametrización sea correcta. Este pequeño avance queda implementado y a disposición de futuros objetivos e investigadores, tal como se describe en el epígrafe 7.2 Posibles futuras líneas de investigación.

El algoritmo final se representa en un flujograma elaborado en la siguiente figura:

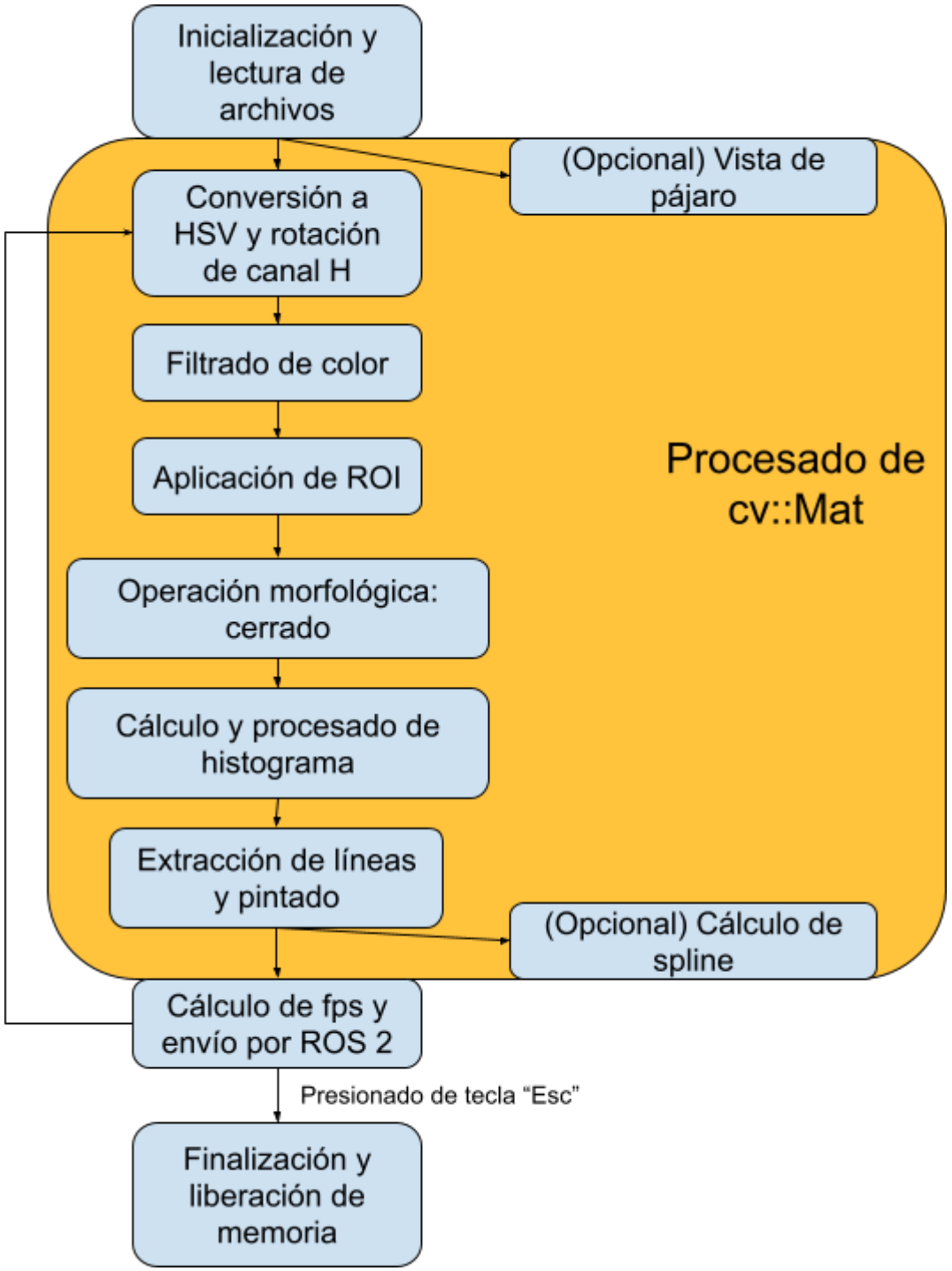


Fig. 19. Estructura de flujo del algoritmo final y funcionalidades opcionales. Fuente: elaboración propia.

5.4 Desarrollo de capa de compatibilidad con la cámara

Finalmente, y una vez que recibimos el hardware adquirido, se empezó a desarrollar la capa de compatibilidad de la nueva cámara de Raspberry con la librería *OpenCV* y se comprobó que

funcionaba correctamente. El proceso obtenido finalmente consiste en las siguientes etapas descritas a continuación en la Fig. 20:

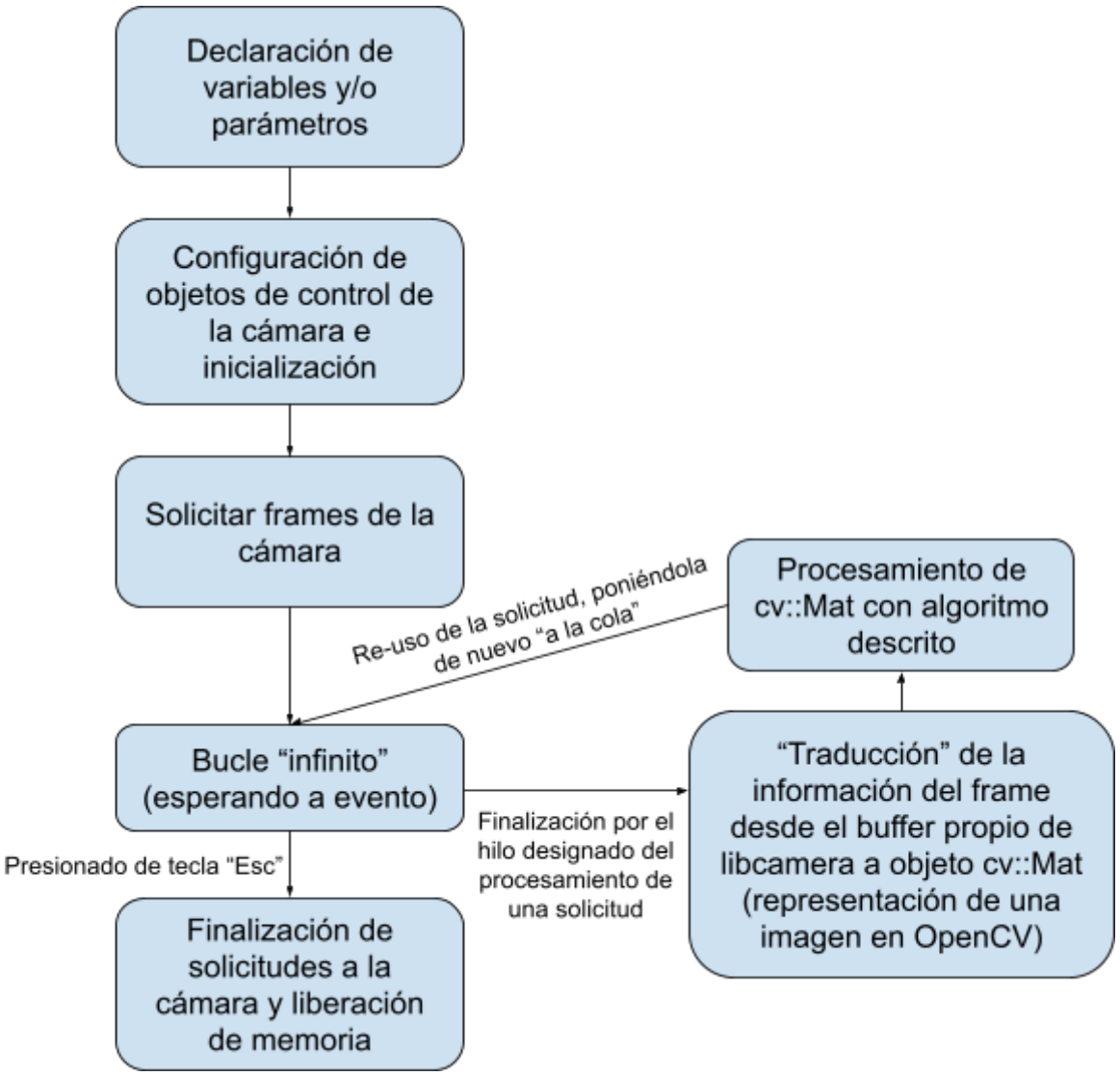


Fig. 20. Estructura del programa de compatibilidad entre libcamera y OpenCV. Fuente: elaboración propia.

Fue complicado encontrar información detallada del uso de la librería *libcamera*, y cierta información resultó ser confusa y/o errónea. Por ejemplo, las instrucciones para liberar la memoria asociada al objeto virtual que representa la cámara pueden funcionar correctamente en una ejecución, pero, en la mayoría de casos, suele quedarse bloqueada sin continuar con la ejecución, siendo necesario cerrar el programa forzosamente con el Sistema Operativo para que libere la memoria y termine la ejecución.

La funcionalidad que se realiza para obtener fotogramas con *libcamera* está basada en la aplicación *simple-cam* [27] que sirve como ejemplo de la propia librería. Se simplificaron o, en su defecto, se eliminaron los pasos que en la aplicación original servían para ejemplificar el uso de las clases, como la obtención del nombre de las cámaras. Además, se sustituyó el método de bucle por el propio de OpenCV: bucle `while`, interrumpido por la función `waitKey()`, que indica el punto en el que renderizar las imágenes y espera un tiempo o a la pulsación de una tecla específica; en este último caso, saldría del bucle, parando la ejecución.

A grandes rasgos, la metodología específica de *libcamera* utilizada sería la siguiente:

- Solicitar una lista de las cámaras conectadas al equipo (normalmente sólo hay una, la Raspberry Cam Module 3).

- Configurar el sensor de la cámara detectada para que envíe fotogramas con la resolución (dimensiones de píxeles) y espacio de color adecuados. Según *libcamera*, el hecho de solicitar los frames ya de la forma que se necesitan reduce el tiempo de procesamiento, ya que es el hardware de la cámara quien se encarga de la transformación, en vez de tener que redimensionar las imágenes en código y por cada fotograma.

- Configurar el *callback* del evento que nos dice cuándo hay una imagen disponible en el *buffer*. Esta rutina de atención (o *callback*) se encarga de tomar los datos de la imagen que nos proporciona el sensor, almacenados en un *buffer* de memoria, y se encarga de codificarlos en una estructura de datos que podamos usar en *OpenCV* (un objeto *Mat* en código, que representa una imagen o fotograma en este caso). Una vez hayamos convertido o "traducido" la información y la hayamos guardado en esta estructura de datos, la pasamos como entrada a nuestro algoritmo (ver el apartado 5.1 Preparación de las imágenes).

5.5 Desarrollo en ROS 2

Después de estos progresos, se empezó a experimentar con ROS 2 en el sistema, contenido en una imagen de Docker, tal como se recomienda en [28], para el uso del software en una Raspberry Pi 5. Se empezó a desarrollar un *publisher* básico para familiarizarse con el envío de

datos mediante *topics* en el entorno del framework y, más tarde, se empezó a adaptar la funcionalidad. El sistema de visión que se desarrolla en este trabajo sólo necesita la funcionalidad del *topic* para comunicarse. Se planteó tener dos en los que publicará: información de la posición del robot respecto al carril que está "viendo" o detectando, en forma de *offsets*, y en el otro, los parámetros que permiten representar la línea del carril de forma numérica (por ejemplo, los parámetros de la spline calculada). Para este propósito, se necesitó definir un tipo de datos que no está soportado por defecto en ROS 2, aunque sí nos da la posibilidad de desarrollarlo, consistente en un vector de números enteros (en C++ un `std::vector<int>`). Este tipo de datos se definió en su propio paquete de ROS 2, donde contiene su propio archivo .msg, que es el que se encarga de definir la estructura de datos del *topic* [29]. Una vez compilado mediante las herramientas del *middleware*, este mensaje se puede incluir posteriormente en un fichero de C++ como si fuera un archivo `.hpp` (*C++ header file*). Como se mencionó anteriormente, ROS 2 permite que este tipo de mensaje se traduzca automáticamente entre los lenguajes de programación que soporta, por lo que también se podría incluir o usar en un nodo escrito en Python, por ejemplo.

En primer lugar, se realizaron varias pruebas en dos terminales de la Raspberry Pi 5, de forma que la información generada por un nodo denominado *publisher*, lanzado en una terminal, fuera recibido por otro nodo llamado *subscriber*, lanzado de forma independiente en otra terminal. En realidad, cada una de estas terminales era un contenedor de docker independiente, como se ha mencionado antes. Mediante estas pruebas realizadas con éxito, se dio por sentado que ROS 2 nos garantizaría que este comportamiento también funcione cuando ambos nodos se ejecuten en máquinas distintas pero en la misma red local (LAN, local area network), siempre y cuando hayan sido configuradas con la misma variable ROS_DOMAIN_ID (un parámetro que permite tener funcionando varias redes de nodos en la misma red local sin que interfieran).

Cuando se probó, sin embargo, los intentos al principio no fueron exitosos. En [1] se menciona: "En cuanto al ordenador de a bordo, se utiliza el kit de desarrollo JNX30D de AUVIDEA … para la NVIDIA Jetson Xavier NX", sin embargo, durante el transcurso de este TFG, se cambió de modelo a un "Kit de desarrollo Jetson AGX Orin". La configuración de la red fue sencilla, se estableció una conexión mediante cable Ethernet entre la Jetson Orin y el router de a bordo del prototipo, y otro cable entre este y la Raspberry Pi 5 (una de las razones por la que se escogió el modelo, ya que disponía de su propio puerto Ethernet, para este propósito). La

máquina Jetson Orin es de forma efectiva "el cerebro" del prototipo, y se conecta con el resto de sensores o sistemas. De forma similar a la Raspberry Pi, dispone de un sistema operativo basado en Ubuntu Linux, con interfaz gráfica o GUI (Graphical User Interface) y la posibilidad de usar ROS 2 nativamente. El fallo en la comunicación, como se descubrió poco después, fue debido a que, por defecto, docker aísla los contenedores en una especie de sub-red virtual, independiente incluso de la propia máquina que los aloja (*host*, en nuestro caso, la Raspberry Pi). Esto, efectivamente, hace que los contenedores de docker por defecto no estén efectivamente en la misma red local que la Raspberry (aunque sí están en su propia red local, y por eso funcionaba entre dos contenedores), lo que hacía imposible que ROS estableciera la comunicación de forma correcta entre ellos y el prototipo. Por esa razón se adoptó el parámetro "`--network host`", lo que hace que el contenedor que se lanza adopte la IP y puertos de la máquina anfitriona, aunque, en nuestro caso, rompe otras funcionalidades, como los comandos apt-get, necesarios para actualizar y descargar software. Para evitar estos errores, se usó un contenedor persistente en el que se desarrollaba y compilaba el software, sin este parámetro, y se copiaba su carpeta ros2_ws (que alberga todos los archivos relevantes a la ejecución de código ROS creados por el usuario) a cada contenedor temporal a utilizar, que sí tendría esta configuración, además de otros parámetros para que se pudiera utilizar la pantalla de la Raspberry (tampoco accesible por defecto por los contenedores) para aplicaciones con GUI, por ejemplo. Se describen instrucciones detalladas de este proceso en el Anexo B.

Una vez que comprobamos que las pruebas de envío de datos fueron exitosas, se desarrolló una pequeña API para facilitar el envío de coordenadas del carril medio o *midLane* mediante este mecanismo. El envío de datos final consiste en restarle a estas coordenadas (en el eje horizontal, puesto que las coordenadas verticales no son relevantes) un número fijo; normalmente es la mitad de la dimensión de la imagen, aunque este es un parámetro ajustable, obteniendo así el número de píxeles a derecha o izquierda del centro de la vista. Un ejemplo de estos datos se puede ver en la Fig. 21, que también explica el diagrama de nodos. Si la cámara está centrada en el robot, este número sería proporcional a la desviación del robot con respecto al carril detectado. Como es posible que la cámara no esté centrada exactamente en el prototipo, el hecho de que el número a restar a las coordenadas sea ajustable resulta bastante útil. Como se discute en el epígrafe 7.2 Posibles futuras líneas de investigación, este número proporcional sería la información útil y necesaria para mandar comandos al robot y que este sepa centrarse y seguir el carril, constituyendo, por tanto, el cúlmen de este trabajo.

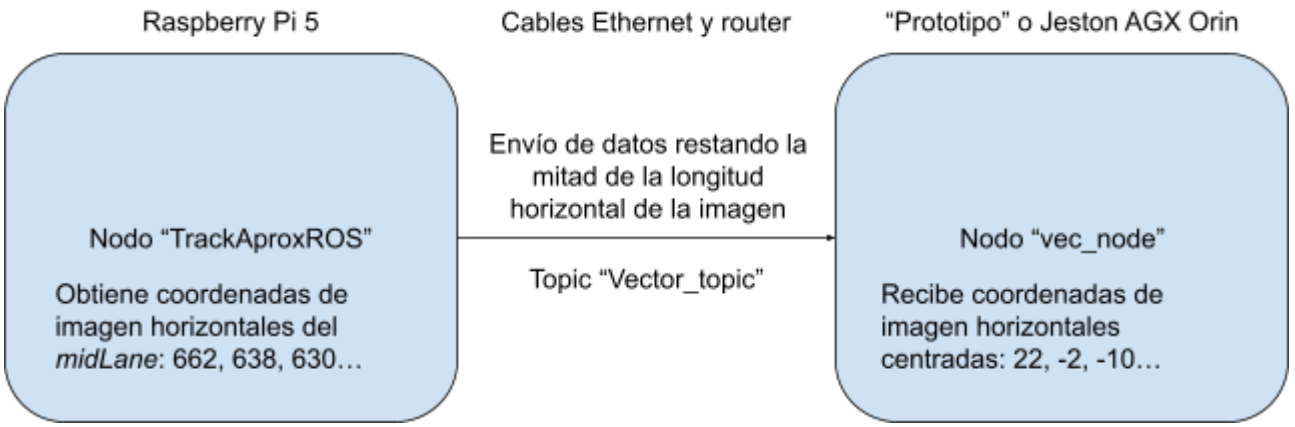


Fig. 21. Representación de la red de nodos y topics del sistema. Fuente: elaboración propia.

6| Pruebas y resultados

En primer lugar, se discutirán las mejoras de rendimiento obtenidas. En la tabla III se presenta un pequeño estudio estadístico evaluado en un extracto de uno de los vídeos de prueba antes de todas las optimizaciones, y después, ya que, sólo cuando se aplican todas a la vez, se aprecian resultados significativos. Cabe destacar que, al estar reduciendo los milisegundos entre frames, pero, como se están comparando fps (su inverso), las "ganancias" no son lineales, porque corresponden a la función 1/x, siendo x el número de segundos entre frames y el resultado, los fps. Es decir que, por ejemplo, la mejora necesaria en ms para pasar de 20 a 22 fps (aproximadamente 4,5 ms) no es igual a la que necesitamos para pasar de 60 a 62 (que es aproximadamente de 0,5 ms).

TABLA III
Comparativa estadística ante y después de aplicar optimizaciones

Muestra	Media de fps redondeados a 0 decimales (1/segundos entre frames)	Media de ms entre frames redondeados a dos decimales (1000/fps)	Desviación típica
Antes	9,35	106,95	1,5580
Después	22,6	44,25	1,7146

Para realizar muchas de las pruebas y decisiones tomadas durante el desarrollo del algoritmo, simplemente se hicieron comparaciones visuales entre los métodos, ya que una evaluación numérica hubiera sido difícil de definir y realizar. Un ejemplo de este tipo de comparación es visible en la Fig. 13.

Para probar la eficacia de los algoritmos, se realizaron grabaciones en el Centro Deportivo Municipal Estadio de Atletismo Vallehermoso, cuyos responsables, de forma generosa, permitieron la realización de estas grabaciones, con la condición de que fueran utilizadas solo para el propósito de este trabajo. Se realizaron vídeos diversos, por ejemplo, siguiendo un carril ensombrecido, un en el que cambian las condiciones lumínicas durante el transcurso del recorrido, cambiando de carril o incluso saliéndose de la pista.

En los casos en los que se cambiaba de carril, si se utilizaba la primera aproximación de la búsqueda de carril a partir del frame anterior, y si ese cambio no era demasiado brusco, se

observó que el seguimiento continuaba hasta que el carril salía del campo de visión (ver Fig. 18). En esos casos, se comenzaba a seguir el carril visible para el algoritmo que estuviera más cerca de donde se detectó el anterior.

Del mismo modo, se observó un fenómeno parecido en las pruebas en las que la cámara se sale de la pista. Se fue cambiando de carril según la cámara los iba cruzando, llegando al exterior de la pista, donde el algoritmo ya no devolvía información correcta, ya que la pista no estaba a la vista y, por tanto, sería imposible extraer información de su posición en la imagen.

En cuanto a las señalizaciones y líneas adicionales encontradas en pista, si estaban resaltadas con colores distintos y no cubrían las líneas propias de los carriles, no influían en absoluto en el guiado. Sólo en el caso de que "cortasen" a las líneas separadoras entre carriles se producían errores momentáneos, principalmente cuando llegaban a la parte inferior de la imagen. El caso de las líneas adicionales del mismo color es peculiar, como se puede observar en la Fig. 22:

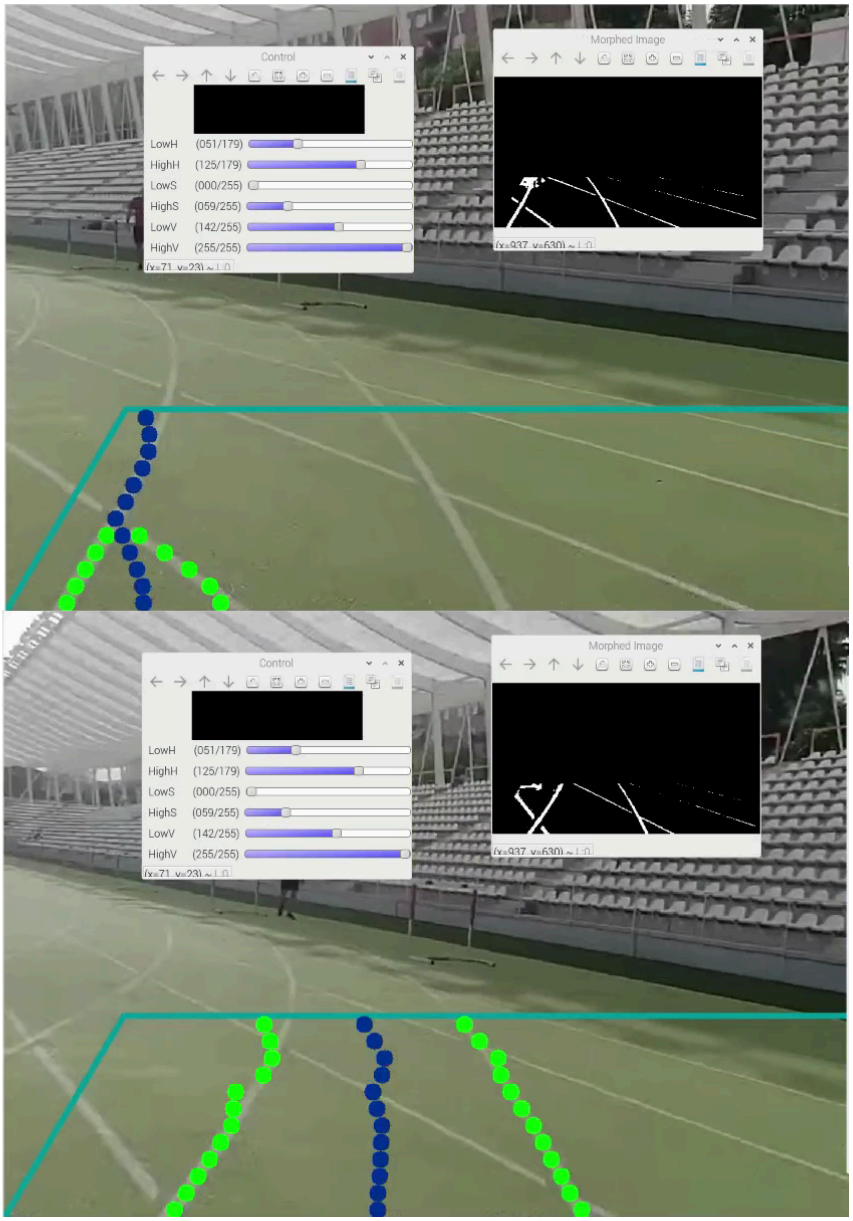


Fig. 22. De arriba abajo: Fotograma en el que las líneas auxiliares confunden al algoritmo, y fotograma en el que las coordenadas se ven alteradas, pero podrían tomarse como viables. Fuente: Elaboración propia.

En estas imágenes podemos ver cómo, en rápida sucesión, si las líneas auxiliares interfieren en el nivel inferior de la imagen, el del punto semilla, producen fallos en el algoritmo. Sin embargo, en los fotogramas en los que esto no sucede, la detección de las líneas es mayoritariamente correcta, aunque se produce cierto ruido debido a estas otras líneas auxiliares que se cruzan en las coordenadas detectadas.

En cuanto a las condiciones lumínicas, si la diferencia de color entre sol y sombra era lo suficientemente pequeña, se podían ajustar los filtros para soportar las condiciones de sombra y luz a la vez. Sin embargo, en bastantes casos, el cambio de iluminación hacía necesario un cambio del filtro in situ para que el algoritmo funcionara correctamente. Sin embargo, en los vídeos donde la sombra del propio corredor que realiza el vídeo se mete en la imagen y cubre la línea, se observa que esto no supone un problema, como se puede ver en la Fig. 23:

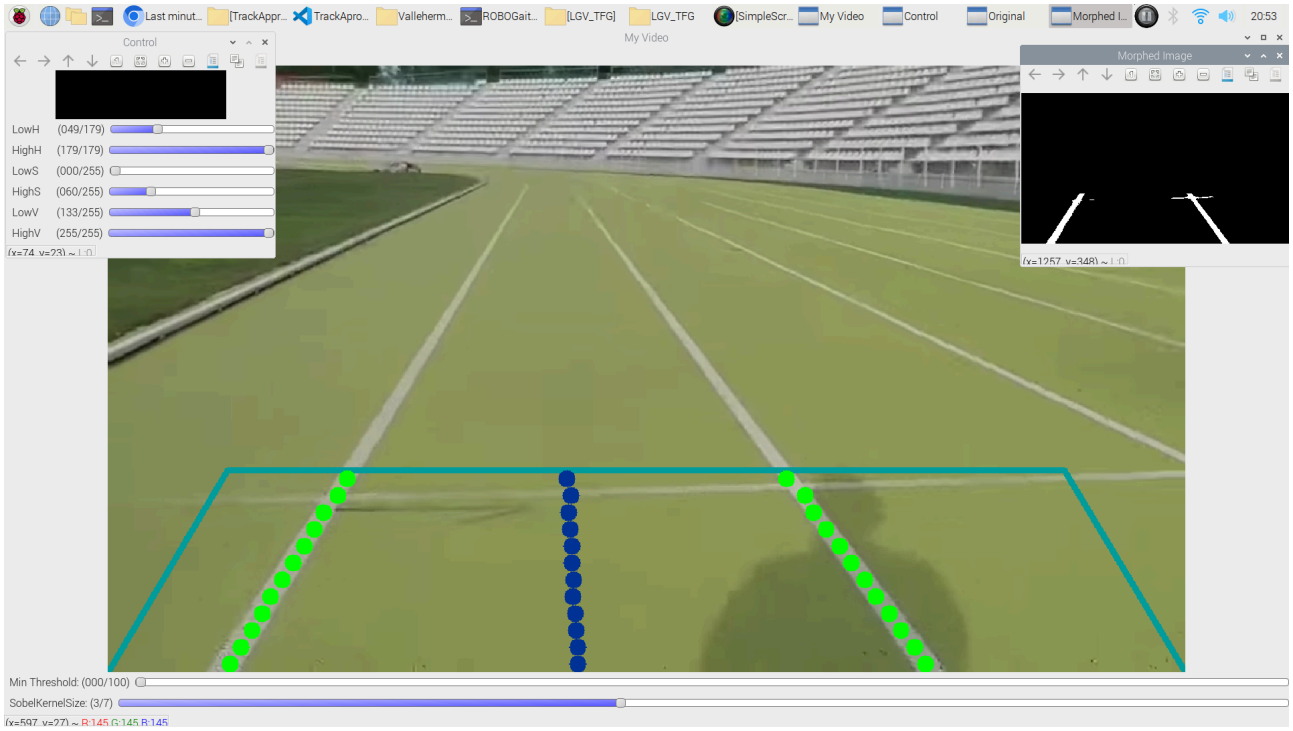


Fig. 23. Fotograma con detección correcta de carril a pesar de la sombra proyectada por el corredor sobre el carril derecho. Fuente: Elaboración propia.

La mejora realizada frente a estos errores, así como sus posibles soluciones, se discuten en mayor profundidad en el apartado 7. Conclusiones.

7| Conclusiones

Teniendo en cuenta la variedad de los objetivos planteados, así como las limitaciones y obstáculos encontrados durante el proceso de desarrollo y testeo del sistema de guiado en la presente investigación, el cumplimiento de buena parte de los objetivos marcados fue alcanzado satisfactoriamente.

El hardware, a pesar de las dificultades que ha generado, ha resultado a la postre muy útil en muchos otros aspectos, por ejemplo, al incorporar su propio Sistema Operativo con interfaz gráfica o GUI. Este SO permite por ejemplo la interacción con ROS 2, el guardado de varios de los programas desarrollados en disco, frente a la típica memoria flash de un microcontrolador, que sólo guarda el programa a ejecutar, y el desarrollo en el propio hardware. La cámara elegida también tiene sus ventajas, como su reducido tamaño, sin sacrificar la velocidad de captura de fotogramas o su resolución. Además, se ha comprobado que dicho hardware cumple con los requisitos que se le exigen, incluso aunque aún no le hayamos sacado todo el partido que requerimos a estos componentes.

El algoritmo desarrollado, a pesar de ser similar en naturaleza a algunos otros de los anteriormente citados como antecedentes, presenta optimizaciones frente a todos ellos para el caso que aquí se pretende solucionar, creando un enfoque único y que, con el ajuste adecuado de los parámetros del filtro de color, resulta de gran efectividad. Además, se adecúa a una situación que no muchos precedentes plantean, como son las curvas, y aplicada a un caso de nuevo uso y con sus propios obstáculos: el uso en pistas de atletismo, frente al uso en carretera o autopista. La mejora en su funcionamiento, con independencia de las condiciones lumínicas y ambientales que nos encontremos, así como el hecho de sustituir el ajuste manual del filtro de color por uno automático, no se han implementado aún, tanto por falta de tiempo como de referentes o de soluciones óptimas en la bibliografía encontrada.

La implementación en ROS 2 con el resto del sistema del prototipo se ha podido probar con éxito. Aunque la forma concreta de unir los resultados de este algoritmo con el programa de control del robot constituye un trabajo futuro, se ha probado que los datos que se creen necesarios para este propósito se envían con éxito. Esto implica una familiarización del alumno con el framework y, debido a las características requeridas, se ha conseguido alcanzar

un conocimiento aún más claro de la herramienta, teniendo que desarrollar la funcionalidad a nivel bajo (nivel de código, *topics* y mensajes o estructuras de datos propios, reemplazar el método de bucle propio de ROS 2, spin, con uno propio, etc.).

A continuación se plantean los obstáculos encontrados durante el proceso, así como posibles caminos a seguir para ampliar o mejorar el sistema desarrollado hasta este momento.

7.1 Limitaciones

Muchas de las limitaciones encontradas en este trabajo ya se han mencionado en el apartado anterior, y estas han sido generadas principalmente por cuestiones de carácter temporal, como el retraso en la llegada de varios de los componentes solicitados para este TFG.

Cabe destacar también la dificultad que hemos encontrado a la hora de realizar pruebas reales en pistas de atletismo, debido a la complejidad del desplazamiento y portabilidad del prototipo, como el uso de baterías, así como la disponibilidad de las instalaciones deportivas adecuadas.

Al ser este un prototipo sobre el que se realizan varios trabajos de investigación y experimentación al mismo tiempo, el escaso margen de tiempo con el que cuenta cada investigador responsable de cada trabajo para hacer sus propias pruebas con el robot también ha supuesto un reto de coordinación, que ha ido solventándose parcialmente al desarrollar buena parte del trabajo con una placa independiente del prototipo que se integrará posteriormente a él: la placa Raspberry Pi y su cámara videográfica compatible, la Raspberry Pi Module 3.

Con respecto a este hardware que elegimos, una de las principales razones que se tuvieron en cuenta fue la compatibilidad con la cámara y con el resto del prototipo. Sin embargo, esta característica era cierta sólo a nivel de hardware. En cambio, a nivel de software, el uso de la librería recomendada por el fabricante, *libcamera*, resultó complejo de implementar debido a la documentación escasa, confusa y, a veces, incluso desfasada o inexistente de sus funciones. Sin ir más lejos, muchas de las funciones o estructuras de código recomendadas o impuestas por la librería llegaron a dar fallos, pero, por suerte, no en las partes críticas del código. Los dos obstáculos principales con los que nos encontramos fueron:

- No mencionar la necesidad y posibilidad de elegir el espacio de color al configurar la cámara, información que se encontró en foros adyacentes. Una vez configurado este parámetro, sí se pudo leer bien la información de la imagen y, por tanto, hacer una capa de compatibilidad con OpenCV. El formato que la librería escogía por defecto causaba problemas.

- El bloqueo impredecible al usar la función `stop()` del objeto que representa a la cámara en la librería. Esta función es vital para parar la ejecución del bucle de obtención de frames, lo que después permite la correcta liberación de memoria (vital en cualquier programa para evitar *memory leaks*). Sin embargo, con mucha frecuencia, al llegar a esta función el programa se queda bloqueado en este paso. Por suerte, al disponer de un Sistema Operativo moderno, podemos cerrar forzosamente el proceso, lo que libera la memoria automáticamente, pero no es lo ideal.

Como se ha mencionado en el cuerpo del trabajo, el uso necesario de Docker para usar ROS 2 ha generado sus propios problemas. Este uso incluyó el aprendizaje en profundidad de la herramienta en sí, y aprender a guardar la información desarrollada en los contenedores, así como aprender a conectar la pantalla y los puertos de la Raspberry a estos para que los usen también, consiguiendo que los desarrollos funcionasen correctamente.

Con respecto al algoritmo, se observan aún estos errores:

- Se da una confusión momentánea con ciertas sombras, reflejos, líneas auxiliares y similares: En una cantidad limitada de fotogramas, pero no nula, los cambios de color producidos por sombras o reflejos causan errores en la detección del color de las líneas. Para el ojo humano es evidente que esas zonas son sombras y que, por tanto, los colores serán similares, si bien nos resultan un poco más oscuros o más claros. Sin embargo, para el algoritmo y los ordenadores esta suposición o cálculo no es directa ni obvia. En Fig. 24 se puede ver uno de estos casos:

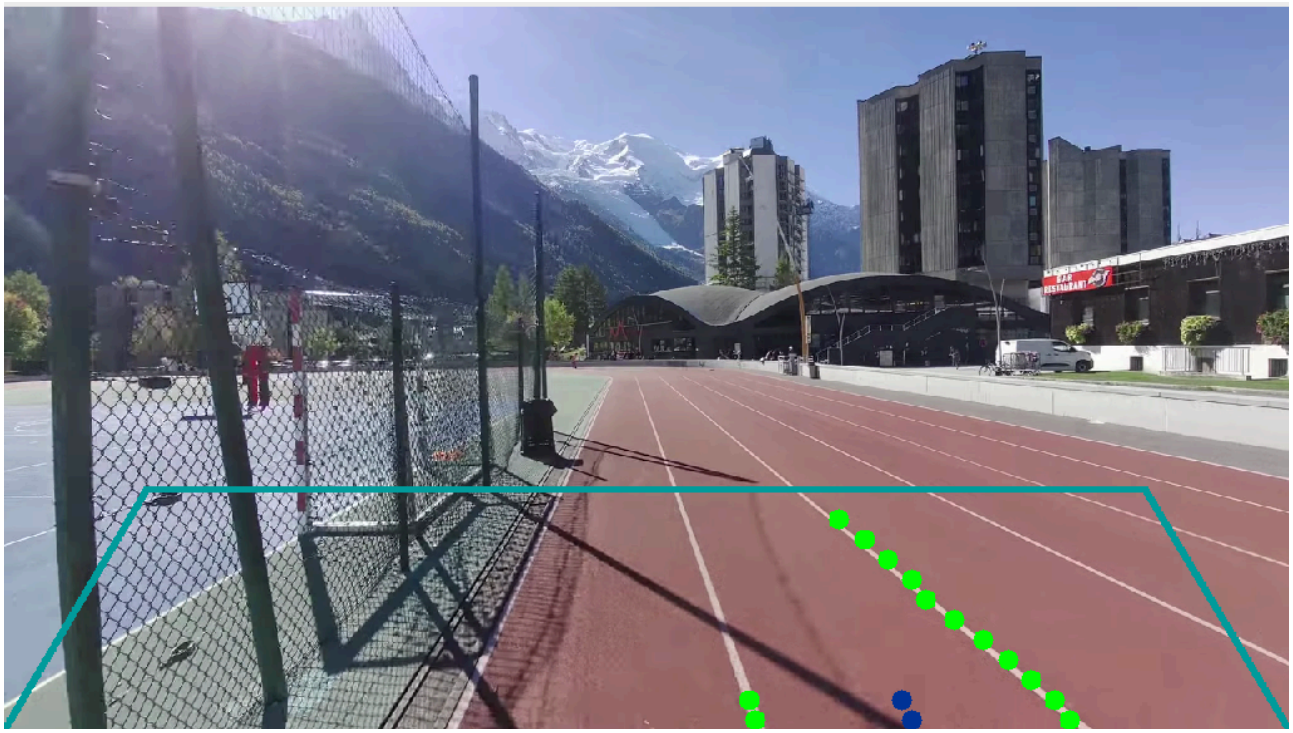


Fig. 24. Fotograma con cálculo erróneo de líneas debido a la sombra producida por el poste. Fuente: Elaboración propia.

- Puede producirse un error en la detección de puntos:

Cuando el algoritmo no detecta información que pueda discernir como líneas en el primer nivel de altura (el más bajo), no se intenta detectar ningún otro más. Esto es extremadamente infrecuente en las pruebas realizadas, pero podría plantearse una solución con la siguiente lógica: si no se detecta información de línea en el primer nivel, el algoritmo debería pasar al segundo; si no se detecta en el segundo, debería pasar al tercero, y así sucesivamente. Esta metodología podría solventar muchos de los fallos momentáneos antes apuntados. Como se ha observado, aparte de ser infrecuentes por cómo se ha programado el histograma, sólo aparecen durante un único frame.

- Falta de calibración automática de filtro de color:

Esto se ha discutido con mayor profundidad en el apartado anterior 5. Desarrollo del sistema. Pero, básicamente, en la versión actual del algoritmo, no sólo es necesario ajustar manualmente el filtro de color, si no que esta operación requiere de una pantalla y un ratón.

En otro orden de cosas, el ámbito de investigación sobre la visión artificial es un campo relativamente nuevo, lo que supone afrontar unos retos específicos, también nuevos y

desconocidos. Por poner un ejemplo, la mayoría de trabajos y estudios anteriores consultados a lo largo de esta investigación, que versan concretamente sobre la detección de líneas para el guiado, se centran esencialmente en líneas rectas, lo cual no es aplicable a nuestro trabajo, ya que, para poder trabajar en el circuito cíclico y cerrado de una pista de atletismo, es necesario contemplar obligatoriamente las líneas curvas.

7.2 Posibles futuras líneas de investigación

Por la complejidad del trabajo y la falta de tiempo para seguir experimentando con otras soluciones, no se aprovechó la GPU (Graphics Processing Unit) de la placa utilizada, lo que podría haber mejorado el rendimiento de los algoritmos de forma considerable, reduciendo así la carga sobre la CPU (Central Processing Unit). La arquitectura de una GPU está pensada para realizar operaciones iguales de forma paralela, como, por ejemplo, hacer el mismo cálculo sobre todos los píxeles de una imagen, como ocurre en nuestro caso. De hecho, el uso inicial de las GPU consistía en calcular los píxeles que debían mostrarse en las pantallas de los ordenadores con interfaz gráfica; de ahí su nombre y su acrónimo. A medida que este hardware ha ido alcanzando mayores potencias, se ha empezado a usar también en pesados cálculos científicos o en el minado en la *blockchain*, por ejemplo. Relegar estas operaciones de procesado, que pueden realizarse en paralelo en vez de forma secuencial a la GPU, podría acelerar considerablemente los tiempos de los algoritmos.

Tanto por falta de referencias bibliográficas al respecto, como por falta de tiempo en el desarrollo de las pruebas realizadas, no se ha planteado a corto plazo el uso de Inteligencia Artificial (IA) en los experimentos y funcionalidad del dispositivo. Según uno de mis tutores, y a la vista de algunas de las referencias consultadas [20], muchas de las tareas realizadas con las técnicas de visión artificial clásica ya están siendo sustituidas en la actualidad por otras técnicas, basadas en estas tecnologías propias de la IA. Por suerte, todas las tareas desarrolladas en este trabajo están bien separadas en el código. Es este planteamiento modular el que nos permitiría que, si se consiguiera una herramienta basada en IA capaz de identificar las líneas de la pista en una imagen, esta se pudiera incorporar sin problemas, sustituyendo fácilmente el algoritmo de detección implementado actualmente, basado en técnicas clásicas. Estas técnicas parecen ser también bastante robustas para evitar errores producidos por condiciones lumínicas, sombras, etc. Además, estas tecnologías no requerirían

de calibración para cada caso, si se entrenan correctamente, por lo que deberían poder extraer las líneas en las mismas situaciones en las que una persona podría hacerlo.

Otra de las mejoras que se han barajado consiste en la posibilidad de adaptar el algoritmo a las líneas pintadas en el firme de una carretera, como en la mayoría de algoritmos utilizados en desarrollos e investigaciones referenciadas, o aplicarlas también a otras líneas arbitrarias que se pinten en el suelo. En esta línea de investigación, también es posible plantear el seguimiento con visión artificial de cualquier otro color de línea, además del blanco, ya que este trabajo se centra esencialmente en su discriminación por tonos claros, que son los más utilizados y comunes para pintar las líneas de separación entre los distintos carriles de las pistas de atletismo. El requerimiento de un ajuste manual del filtro de color puede resultar una ventaja en este aspecto, ya que permitiría seleccionar cualquier color de línea (aunque cabe reseñar que se ha optimizado fundamentalmente para tonos blancos o claros).

Muchos de los futuros pasos para hacer que este sistema sea completamente funcional como guía del robot a alta velocidad se han tenido en cuenta a la hora de estructurar el código, y se plantean aquí como futuras líneas de investigación:

- Creación de sistema de calibrado:
Como ya se ha discutido, el filtro de color necesita ser calibrado para cada pista, para cada situación lumínica, etc. Además, este ajuste requiere el acceso al hardware mediante GUI para ajustar los parámetros, como se puede ver en la Fig. 25:

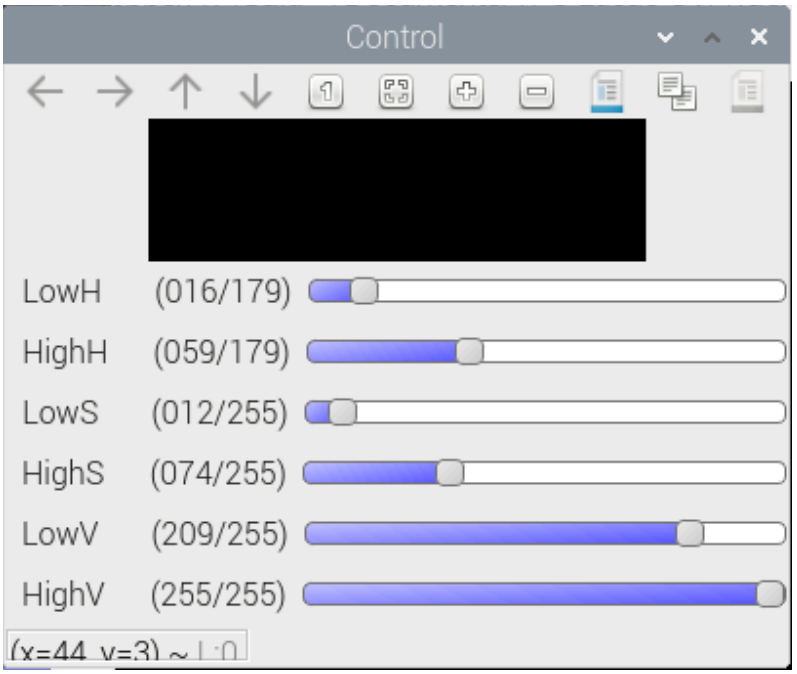


Fig. 25. Interfaz gráfica de los controles del filtro en HSV (LowH sería el mínimo valor admisible en el canal H, y HighH el máximo). Fuente: Elaboración propia.

Por tanto, sería muy útil desarrollar un sistema que, dada una primera imagen del robot situado en la pista, es decir, en parada, pudiera detectar el color de las líneas automáticamente, y que, en ese mismo momento, mandara una señal de OK, por ROS 2, por ejemplo. Este desarrollo sería útil, no sólo para que el comportamiento y el ajuste del filtro fuera más predecible, si no para no requerir interfaz gráfica en un entorno de pruebas al que sería complicado llevarse una pantalla. Existen técnicas ya implementadas en OpenCV, como, por ejemplo, la separación de Otsu (Otsu's thresholding), o la función `adaptiveThresholding()`. Pero estas técnicas funcionan correctamente en condiciones que nuestro caso no cumple, como que haya dos claros picos en el histograma de intensidad, o que no exista mucha información de color, respectivamente. Si se implementase un algoritmo basado en IA para esta segmentación, como ya se ha propuesto anteriormente, no haría falta una nueva calibración cada vez que se haga una prueba, ya que la red estaría entrenada ideal y previamente con todo tipo de situaciones.

- Implementación de búsqueda del carril inteligente:

Manteniendo un histórico de la localización del carril en fotogramas o momentos anteriores, lo cual puede ser útil también para alguno de los puntos posteriores. Conociendo datos de la odometría del robot (velocidad y dirección en la que se mueve, por ejemplo), podría implementarse un filtro de Kalman, como ejemplo de mis tutores, que haga que la predicción de la posición del carril en el siguiente fotograma sea más fiable y rápida. Además, también podría servir para corregir aquellos fotogramas en los que las sombras, los reflejos u otras líneas auxiliares producen errores momentáneos. Para este propósito sería también muy útil guardar las líneas parametrizadas en ese registro, en lugar de como una colección de puntos, ya que así se podría optimizar la memoria y hacer más sencillas las predicciones de posición en fotogramas posteriores.

- Etiquetado de líneas:

En vez de centrarse sólo en dos líneas que definan el carril que buscamos, como se hace con la versión actual del algoritmo, podría implementarse un sistema que etiquete todas las líneas visibles y que pueda discernir entre todas ellas las que delimitan el carril que se quiere seguir. Para ser de utilidad, este sistema de etiquetado tendría que ser persistente entre fotogramas, es decir, que, entre dos fotogramas, la "línea 1" tendría que ser la misma, a pesar de haber cambiado en el nuevo fotograma, para mantener una continuidad. Esta visión resulta intuitiva para el ojo y el cerebro humano que procesa las imágenes, ya que podemos identificar fácilmente las líneas de los carriles, avanzar un poco con los ojos cerrados y volver a identificar las líneas que estamos viendo en relación con las que vimos instantes antes, identificándolas como las mismas. Sin embargo, para un computador esta tarea no es para nada trivial. Si consigue implementarse este sistema, podría permitir incluso cambiar de carril en medio del desplazamiento, o evitar estos cambios de calle si se produjesen de forma errónea (es decir, confundir un carril con el de al lado en un movimiento brusco).

- Diseño de controlador, basado en los datos enviados a través de ROS 2:

En este trabajo ya se ha conseguido mandar una serie de números que son la cantidad de píxeles a derecha o izquierda de la posición del carril detectado con respecto a una línea vertical fija en la imagen (normalmente el centro). Esto, a grandes rasgos, nos da una indicación de si el robot está centrado en el carril, entre otras muchas cosas, y podría usarse como señal discreta en un sistema de control. Por ejemplo, podría lograrse que la dirección de las ruedas sea proporcional a este dato, como una primera aproximación. El diseño de un sistema de control adecuado es complejo (ya que es muy improbable que el ejemplo anterior

funcione), como se puede observar en [1]. Además, este diseño suele requerir un proceso de prueba y error, a la hora de estudiar el sistema que queremos controlar, así como probar la eficacia de varias técnicas, cambios de parámetros, etc.

- Definición completa de trayectorias:

Haciendo un primer mapeado de la pista a menor velocidad (ya sea manualmente, o bien con el sistema de control descrito en el punto anterior), los datos de cada fotograma podrían combinarse para crear una trayectoria completa del recorrido en la pista. Internamente, lo que hacen los nodos a los que queremos sustituir es algo parecido. Por ejemplo, cuando en ROS 2 se manda un punto de destino, lo que en realidad se calcula son una serie de puntos intermedios entre este y el punto en el que está el robot actualmente, evitando obstáculos que ya conoce por haber realizado un mapeado anterior, mediante ciertos algoritmos de *pathfinding* o "búsqueda de rutas".

Replicar esto resulta prácticamente imposible, si no se mantiene un registro histórico, ya que la cámara no puede comprender toda la pista desde una sola vista o fotograma, si no que analiza únicamente la parte de la misma que tiene inmediatamente delante de sí. Se podría componer una trayectoria, similar a las calculadas por los nodos de ROS 2 descritos anteriormente, a base de todos los datos obtenidos por la cámara al dar una vuelta completa a la pista. Para conseguir esto, seguramente también se necesitarían obtener datos de odometría del robot para localizar los datos relativos a cada fotograma en un mapa virtual. Una trayectoria ya tratada de esta manera por el sistema de visión artificial podría enviarse directamente a los nodos de navegación usados en el entorno de ROS 2 del prototipo, y en principio, podría navegar guiándose con ellos.

- Traducción de datos a distancias en metros:

Teniendo en cuenta la posición de la cámara y la perspectiva respecto al suelo, se podría hacer una transformación de las coordenadas obtenidas en el espacio de imagen o píxeles (una proyección 2D o plana de la realidad) a distancias relativas al robot. Hay ya algún avance desarrollado en este aspecto, como un primer intento de corrección de perspectiva a vista de pájaro, o *birds-eye view*, lo que supone un intento de transformación de la imagen obtenida en una vista perpendicular al suelo desde arriba, algo que ya hemos explicado anteriormente en el apartado 5. Desarrollo del sistema. Desde esta nueva pseudo-perspectiva, si se lograse hacer el cambio de perspectiva de forma precisa, mediante la transformación de distancias en píxeles a metros, sería una operación lineal en teoría, y podría ser una información muy útil

para el control y/o navegación, por ejemplo, para el cálculo de la velocidad o incluso para ayudar a la odometría del robot. Esto, además, sería un paso necesario para el objetivo previo.

Juntando varias de las mejoras anteriores a la parametrización de las líneas ya existente (principalmente el filtro de Kalman y técnicas de IA), se considera que la robustez que podría obtenerse frente a errores momentáneos, producidos por sombras, reflejos, obstáculos, líneas incompletas o auxiliares, aumentaría considerablemente. Con estas nuevas herramientas, no sólo se tomaría la información del frame que se tiene delante, si no que también se obtendría información de frames anteriores o la tendencia de la línea. Por ejemplo, y como ya se ha mencionado, en buena parte de la literatura estudiada se asume que se trabaja con líneas rectas. Esta suposición permite que, si hay una sombra en medio de una línea, o esta está incompleta por desgaste, se pueda completar igualmente con la información que se tiene del resto de la recta, aunque el algoritmo no pueda detectar estrictamente ese tramo que falta. Con la parametrización y algunas de estas técnicas mencionadas, se podría hacer un sistema parecido que también soportase curvas, aunque la complejidad siempre sería mayor. Este tipo de información es el que usan las personas también de forma análoga e inconsciente para identificar, por ejemplo, las líneas y poder asumir los tramos faltantes de pigmentación en esas líneas. Por desgracia, las ventajas que nos podrían ofrecer estas nuevas herramientas son proporcionales a su complejidad, por lo que no se han podido desarrollar en este trabajo con tiempo suficiente.

También se ha valorado analizar en un futuro el impacto sobre la efectividad del algoritmo de detección del uso de una cámara RGB-D, es decir, una cámara con un sistema de grabación que, además de los tres canales de color de la imagen, incorpora un cuarto canal con información de la distancia. Esto podría ayudar a no necesitar realizar los cálculos de perspectiva ya mencionados para obtener datos de distancia que se envíen al prototipo, por ejemplo.

Debido a la naturaleza iterativa y práctica de la investigación, el código ha sufrido bastantes revisiones de optimización e incluso de funcionamiento. A pesar de haber tratado de conseguir que este sea claro, limpio, generalista (para que se pueda ampliar y/o reutilizar en futuras investigaciones), y que no admita un uso erróneo, estos objetivos son complejos, incluso en desarrollos con múltiples personas, por lo que no podemos asegurar que esto resulte factible a largo plazo. La mejora del código principal y de las pseudo-librerías o APIs,

creadas para desarrollar dicho código centrado en estos aspectos, es un objetivo que puede parecer menos prioritario y que, de puertas afuera, no parecería tener mucho impacto. Sin embargo, esta mejora puede ser de vital importancia para la eficacia del algoritmo y para que futuros investigadores puedan entender el código existente de forma rápida y puedan reutilizar las funciones ya creadas.

8| Presupuesto y diagrama temporal

8.1 Presupuesto

Aquí se reproduce la tabla [1, Table 8.1], que describe los precios de los componentes anteriores a este TFG:

TABLA IV
Reproducción de [1, Table 8.1]

Producto	Distribuidor	Precio (€)
Chasis Arrma Kraton 1/5	RtrValladolid	749,99
Motor HobbyWing Xerun 42688SD 1600 KV	E1RC	136,95
ESC HobbyWing Xerun XR8 SCT 1/8 140 A	E1RC	108,90
Servo Digital RUIZHI 70 kg	Amazon	25,99
Servo SRT 25 kg 0,14 S	E1RC	49,90
Spektrum SLT3 2,4 GHz + Receptor SLR300	RtrValladolid	57,99
HobbyWing programador bluetooth OTA	RtrValladolid	57,90
U-TECH PRO 4s 14,8V 8000 mAh 20C	RCinnovations	151,80
Cargador ISDT D2 Mark 2	Amazon	125,83
AUVIDEA Jetson Xavier NX	SiliconHighway	599,99
STM32F411CEU6	Amazon	17,34
ZED 2 Depth Camera STEREOLABS	GenerationRobots	540,00
Luxonis OAK-D Lite	RobotShop	199,00
IMU WitMotion WT901C-TTL	Amazon	39,00
HUB USB 3.0 4 puertos	RSComponents	22,09
RPLIDAR S2	RobotShop	425,81
GPS Sparkfun ZED-F9P	Mouser	266,18
U-Blox ANN-MB1-00	Mouser	66,07
Router TP-LINK MR100	Amazon	52,99
Total		3693,72

No se considerará el precio de la nueva Jetson Orin, ya que no es una consecuencia directa de este trabajo, y ya se ha considerado el coste de una máquina aproximadamente equivalente en precio, la Jetson Xavier. Todo el software utilizado es gratuito, por lo que las únicas compras que se realizaron como consecuencia de este trabajo se detallan en esta tabla:

TABLA V
Precios directos del desarrollo de este trabajo

Producto	Precio (€)
Raspberry Pi 5 (kit completo)	150
Raspberry Camera Module 3	40
Power Bank 24000mAh de hganus	18.99
Total	208,99

TABLA VI
Sueldos aproximados basados en cálculos de [1, Table 8.2] y las fuentes que se utilizan en esta

Trabajador	Precio horario (€/h)	Horas	Coste de trabajador (€)
Alumno	13,50	440	5940,00
Tutor	27,09	40	1083,60
Co-tutor	18,50	40	740,00
Total		7763,60	

En la siguiente tabla se agregan todos los costes calculados:

TABLA VII
Costes totales

Concepto	Precio (€)
Piezas anteriores	3693,72
Piezas nuevas del sistema	208,99
Sueldos	7763,60
Total*	11.666,31

*No se han tenido en cuenta los costes indirectos, como el trabajo de otros alumnos en el prototipo, el coste de uso de las instalaciones, o las máquinas propias del alumno, ya que no

sólo son difíciles (o casi subjetivos) de calcular, si no que no tienen gran relevancia en el desarrollo del trabajo, en la mayoría de los casos.

8.2 Diagrama temporal

Teniendo en cuenta que el proceso de desarrollo e investigación ha sido mutable e impredecible, además de reiterativo (la investigación ha sido continua, se ha trabajado y vuelto a versiones preliminares de algoritmos, etc.), no se puede hacer un diagrama temporal exacto de las actividades. Sin embargo, se puede dividir la actividad en cuatro grandes bloques:

TABLA VIII
División de actividad en el tiempo (fechas del año 2024)

Inicios de febrero	Inicios de abril (día 4)	Mitad de julio	Principio de septiembre (hasta día 9)
Fase preliminar, prototipado, selección de piezas y comienzo de búsqueda de referencias	Se reciben los componentes necesarios para el desarrollo de algoritmos	Compatibilización con el prototipo y desarrollo y aprendizaje de ROS 2	Finalización del desarrollo, pruebas y toques finales a la memoria del trabajo

9| Referencias bibliográficas

[1] C. Ferreira, "*SISTEMA DE CONTROL Y NAVEGACIÓN DEL ROBOT PARA CORREDORES DE ATLETISMO: ROBOGAIT*", Madrid, España, Sep. 2023. Este TFM fue publicado como resultado del Máster Universitario de Automática y Robótica en la UPM. Available: https://oa.upm.es/76046/1/TFM_CARLOS_FERREIRA_GONZALEZ.pdf

[2] H. Yang, Y. Wang and L. Gao, "A general line tracking algorithm based on computer vision," 2016 *Chinese Control and Decision Conference (CCDC)*, Yinchuan, China, 2016, pp. 5365-5370, doi: 10.1109/CCDC.2016.7531957.

[3] D. J. Bora, "An optimal color image edge detection approach," *2017 International Conference on Trends in Electronics and Informatics (ICEI)*, Tirunelveli, India, 2017, pp. 342-347, doi: 10.1109/ICOEI.2017.8300945.

[4] M. H. Hasan and D. Kr Das, "Analysis of Edge Detection for Road Lanes through Hardware Implementation," *2021 Fourth International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, Erode, India, 2021, pp. 1-6, doi: 10.1109/ICECCT52121.2021.9616738.

[5] Z. Chunjiang and D. Yong, "Color Image Edge Detection using Dempster-Shafer Theory," *2009 International Conference on Artificial Intelligence and Computational Intelligence*, Shanghai, China, 2009, pp. 476-479, doi: 10.1109/AICI.2009.34.

[6] R. Sacchelli, "Lanes-Detection-with-OpenCV," *GitHub*, Accessed Aug. 30, 2024. [Online]. Available: https://github.com/rogersacchelli/Lanes-Detection-with-OpenCV?tab=readme-ov-file

[7] A. Sears-Collins, "The Ultimate Guide to Real-Time Lane Detection Using OpenCV," *Automatic Addison*. Accessed Aug. 30, 2024. [Online]. Available: https://automaticaddison.com/the-ultimate-guide-to-real-time-lane-detection-using-opencv/

[8] J. He, S. Sun, D. Zhang, G. Wang and C. Zhang, "Lane Detection for Track-following Based on Histogram Statistics," *2019 IEEE International Conference on Electron Devices and Solid-State Circuits (EDSSC)*, Xi'an, China, 2019, pp. 1-2, doi: 10.1109/EDSSC.2019.8754094.

[9] MaybeShewill-CV, "lanenet-lane-detection," *GitHub*, Accessed Sep. 9, 2024. [Online]. Available: https://github.com/MaybeShewill-CV/lanenet-lane-detection

[10] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, May 2022. doi: 10.1126/scirobotics.abm6074. Available: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074

[11] OpenCV. "About OpenCV." *OpenCV - About*. Accessed Mar. 17, 2024. [Online]. Available: https://opencv.org/about/

[12] L. Gómez, "LGV branch of ROBOGAIT/software_exteriores," *GitHub*. Accessed Aug. 28, 2024. [Online]. Available:

https://github.com/ROBOGAIT/software_exteriores/tree/LGV. Available publicly: https://github.com/Pigamer37/Trabajo-Fin-de-Grado

[13] RAE and ASLAE. "Píxel: Diccionario de la Lengua Española." *Diccionario de la lengua española - Edición del Tricentenario*. Accessed Mar. 18, 2024. [Online]. Available: https://dle.rae.es/p%C3%ADxel

[14] S. R. Fernando. "OpenCV Tutorial C++." *OpenCV-SRF*. Accessed Mar. 17, 2024. [Online]. Available: https://www.opencv-srf.com/p/introduction.html

[15] M. Pound, "*How blurs & filters work - Computerphile*," YouTube, (Oct 2, 2015). Accessed: Mar. 18, 2024. [Online Video]. Available: https://www.youtube.com/watch?v=C_zFhWdM4ic

[16] A. Huamán. "Smoothing images." *OpenCV documentation*. Accessed Mar. 18, 2024. [Online]. Available: https://docs.opencv.org/3.4/dc/dd3/tutorial_gausian_median_blur_bilateral_filter.html

[17] A. Huamán. "Sobel Derivatives." *OpenCV documentation*. Accessed Mar. 17, 2024. [Online]. Available: https://docs.opencv.org/3.4/d2/d2c/tutorial_sobel_derivatives.html

[18] M. Pound, "*Finding the edges (Sobel operator) - Computerphile*," YouTube, Accessed: Mar. 19, 2024. [Online Video]. Available: https://www.youtube.com/watch?v=uihBwtPIBxM

[19] A. Huamán. "Canny edge detector." *OpenCV documentation*. Accessed Mar. 18, 2024. [Online]. Available: https://docs.opencv.org/3.4/da/d5c/tutorial_canny_detector.html

[20] R. Szeliski, *Computer Vision: Algorithms and Applications*, 2nd ed. Springer, 2022.

[21] M. Pound, "*Canny edge detector - Computerphile*," YouTube. [Online Video]. Accessed: Mar. 19, 2024 [Online Video]. Available: https://www.youtube.com/watch?v=sRFM5IEqR2w

[22] Mathworks, "Understanding color spaces and color space conversion", *Mathworks,* Accessed May 12, 2024. [Online]. Available: https://es.mathworks.com/help/images/understanding-color-spaces-and-color-space-conversion.html

[23] A. Huamán. "More Morphology Transformations." *OpenCV documentation*. Accessed May. 12, 2024. [Online]. Available: https://docs.opencv.org/4.x/d3/dbe/tutorial_opening_closing_hats.html

[24] S. Suzuki and K. Abe, "Topological Structural Analysis of Digitized Binary Images by Border Following," *Computer Vision, Graphics, and Image Processing*, vol. 29, no. 3, p. 396, Mar. 1985. doi: 10.1016/0734-189x(85)90136-7.

[25] OpenCV documentation. "Geometric Transformations of Images." *OpenCV documentation*. Accessed Jul. 16, 2024. [Online]. Available: https://docs.opencv.org/3.4/da/d6e/tutorial_py_geometric_transformations.html

[26] T. Kluge, "c++ cubic spline interpolation library, spline.h", *kluge.in-chemnitz*, Accessed Jul. 19, 2024. [Online]. Available: https://kluge.in-chemnitz.de/opensource/spline/

[27] libcamera documentation. “libcamera simple-cam tutorial application.” *git.libcamera.org*. Accessed Jul. 11, 2024. [Online]. Available: https://git.libcamera.org/libcamera/simple-cam.git/tree/simple-cam.cpp

[28] ROS 2 documentation, “ROS 2 on Raspberry PI,” *ROS 2 documentation: Humble documentation*. Accessed Jul. 31, 2024. [Online]. Available: https://docs.ros.org/en/humble/How-To-Guides/Installing-on-Raspberry-Pi.html

[29] ROS 2 documentation, “Creating custom msg and srv files,” *ROS 2 documentation: Humble documentation*. Accessed Aug. 14, 2024. [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Custom-ROS2-Interfaces.html

[30] Z. Qin, P. Zhang and X. Li, “Ultra Fast Deep Lane Detection With Hybrid Anchor Driven Ordinal Classification”, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022, pp.1-14, doi: 10.1109/TPAMI.2022.3182097. Implementation in: https://github.com/cfzd/Ultra-Fast-Lane-Detection

ANEXO

Anexo A: Repositorio de GitHub y explicación del código

Como se ha mencionado con anterioridad, en [22] se puede encontrar el código del que se compone este trabajo, aunque al tratarse de un repositorio privado, puede que no sea accesible por todo el mundo. Por ello se incluyen dos fuentes, una es pública.

Notas:

Los nodos de ROS se desarrollaron en el entorno de docker provisto en [28], y se comprobó el funcionamiento de los paquetes necesarios para leer mensajes de tipo vector de enteros también en la Jetson Orin, cuyo SO es una versión de Ubuntu. No se puede garantizar que estos funcionen en otros SO, aunque, debido a la naturaleza de ROS 2, Linux y el propio C++, deberían ser portables. Todo el código externo usado permite su reproducción y su uso libre personal y comercial. El código externo o preliminar a ROS también debería ser portable al depender en su mayoría de OpenCV y de C++, que son multiplataforma. El código que depende de *libcamera* se sospecha que sólo funcionará en sistemas UNIX, ya que se ha desarrollado específicamente para ellos y sólo se ha testeado en los mismos.

Requisitos:

En este apartado se incluyen las librerías o dependencias necesarias para cada parte del código, y las formas de instalación en Ubuntu cuando sea posible. Además se marcará más tarde cada pieza de software con sus dependencias.

- OpenCV 👁️: `sudo apt install libopencv-dev python3-opencv` (para compilar y ejecutar respectivamente, necesarios en cualquier app que use imágenes).
- libcamera 📷: `sudo apt install libcamera` (para usar la cámara de la Raspberry).
- ROS 2 🤖: depende del sistema operativo (para cualquier nodo de ros2).
- spline.h: Incluída en el repositorio. (para funcionalidad de síntesis/aproximación de líneas).

Carpetas/piezas de software:

- OpenCV-test: test preliminar de uso de la librería en Visual Studio, en Windows. 👁️
- Paquetes de ROS 🤖:
    + vec_package: paquete que alberga el mensaje propio Vector. De este paquete dependen todos los que usan este tipo de mensaje.
    + my_package: publisher básico de mensajes de ROS propios, de tipo Vector (`std::vector<int>` en C++).

    + vec_sub: subscriber/listener básico de mensajes de ROS propios, de tipo Vector (`std::vector<int>` en C++).
    + Cam_package 👁️: algoritmo TrackApprox en entorno ROS, capaz de publicar datos internos en un *topic*. Contiene también **ROSPublish.hpp**, útil para tener una primera clase básica que permite publicar mensajes relevantes del algoritmo en ROS.

- Aux_OpenCV.cpp(*.hpp) 👁️: intento anticuado de librería custom para facilitar el uso de OpenCV. Fue sustituida después por OCV_Funcs.cpp(*.hpp).
- OCV_Funcs.cpp(*.hpp) 👁️: librería con variedad de funciones para implementar algoritmos relacionados con OpenCV, compatibilización con spline.h, extracción de datos, creación de los histograma descritos, y seguimiento inteligente de líneas con la clase `laneLogic`.
- DetectTrack.cpp 👁️: Versión anterior de TrackApprox.cpp.
- TrackApprox.cpp 👁️: Implementación de algoritmo en imágenes y vídeos.
- RaspiTrack.cpp 📷👁️: Implementación de TrackApprox con la cámara de la Raspberry, lo que supone un cambio de estructura.

Anexo B: Lanzar implementación del código/ tutorial de uso con Docker

Como se ha mencionado, en la placa ya existe un contenedor de Docker permanente en el que se puede compilar, almacenar y desarrollar código. Para entrar en una sesión de CLI en este, se debe lanzar este comando:

```
sudo docker start -a -i 2302d08e8bd2
```

Para copiar los contenidos de la carpeta ros_ws (los archivos relevantes del desarrollo en ROS 2) a nuestro sistema de archivos se debe ejecutar este comando:

```
sudo docker cp 2302d08e8bd2:/root/ros2_ws "/home/ROBOGait/Documents/Ros2 docker"
```

Para poder disponer de comunicación con ROS 2 y de la pantalla conectada a la placa Raspberry, se deberá lanzar un contenedor temporal con una instrucción similar a esta:

```
sudo docker run -it --rm --network host --name rostmp -e DISPLAY=$DISPLAY -v
/tmp/.X11-unix:/tmp/.X11-unix:rw ros:humble-ros-core
```

`--network host` : habilita que los puertos del contenedor se compartan con la Raspberry.

`-e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix:rw` : habilita el uso por parte

del contenedor de la pantalla de la Raspberry. Nota: es necesario *en la Raspberry* lanzar este comando también : `xhost +`

`-name rostmp` : simplemente se utiliza para hacer más fácil la identificación. Si se omite este comando, Docker asignará un nombre aleatorio a nuestro contenedor.

Para copiar los contenidos de la carpeta ros_ws del contenedor permanente que tenemos en `/home/ROBOGait/Documents/Ros2 docker` (no se puede copiar información directamente entre contenedores) al contenedor temporal, se ejecuta el siguiente comando:

```
sudo docker cp "/home/ROBOGait/Documents/Ros2 docker/ros2_ws" rostmp:/root
```

Si no se hace --network host, la herramienta apt-get no funcionará. Todos los paquetes necesarios se pueden instalar con los siguientes comandos:

```
apt-get update
apt-get install libopencv-dev
sudo apt install python3-colcon-common-extensions
apt-get -y install \
    firefox \
    libcanberra-gtk-module \
    libcanberra-gtk3-module
```

Cada una de estas líneas de comando realiza respectivamente las siguientes funciones: actualiza, instala OpenCV, instala las herramientas de colcon para compilar en ROS, instala las dependencias para que OpenCV pueda presentarse en pantallas.