Escuela Politécnica Superior de Jaén

Grado en Ingeniería Informática

Universidad de Jaén

Escuela Politécnica Superior (Jaén)

CONTROL DE UN ROBOT MÓVIL A TRAVÉS DE INSPECCIÓN OCULAR

Alumno: Francisco Joyanes Morales

Tutor: Prof. D. Diego Manuel Martínez Gila

Prof. D. Silvia María Satorres Martínez

Dpto: Departamento de Informática

Septiembre, 2022

Universidad de Jaén

Escuela Politécnica Superior de Jaén

Departamento de Informática

Don Diego Manuel Martínez Gila y Doña Silvia María Satorres Martínez tutores del Trabajo Fin de Grado Titulado: Control de un robot móvil a través de inspección ocular, que presenta Francisco Joyanes Morales, autoriza su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, noviembre de 2022

El alumno: Los tutores:

Firmado por JOYANES MORALES FRANCISCO - ***1581** el día 11/11/2022 con un certificado

SATORRES MARTINEZ SILVIA - 26741651B

Firmado digitalmente por SATORRES MARTINEZ SILVIA - 26741651B
Nombre de reconocimiento (DN): c=ES, serialNumber=IDCES-26741651B, givenName=SILVIA, sn=SATORRES MARTINEZ, cn=SATORRES MARTINEZ SILVIA - 26741651B
Fecha: 2022.11.15 13:07:40 +01'00'

Firmado digitalmente por MARTINEZ GILA DIEGO MANUEL - 77338812R
Fecha: 2022.11.16 11:54:59 +01'00'

Francisco Joyanes Morales Diego Manuel Martínez Gila

Silvia María Satorres Martínez

Indice

Figuras

Resumen

La aplicación permitirá al usuario controlar un robot móvil con la mirada gracias a inspección ocular. Para ello se utilizarán técnicas de tratamiento de imágenes y de reconocimiento facial implementadas en el lenguaje de programación Python. Además se usará Makecode para el control del robot móvil.

El proyecto se divide en dos aplicaciones. La primera aplicación reconocerá los movimientos y gestos de los ojos del usuario y los traducirá a órdenes que se enviarán a la segunda aplicación. La segunda aplicación interpretará las órdenes y activará el funcionamiento del robot móvil según las instrucciones recibidas.

La aplicación está dirigida a usuarios que padecen ELA. Dichos usuarios pueden presentar dificultades motoras en las manos que les impiden usar una silla eléctrica. En este proyecto se plantea la alternativa de usar inspección ocular en lugar de un mando eléctrico para facilitar el uso a estos usuarios.

Abstract

The application will let the user to control a mobile robot with him gaze thanks to ocular inspection. For this it will used image processing and facial recognition techniques implemented with the programming language Python. It will used MakeCode to control the mobile robot too.

The project is divided into two applications. The first application will recognize the movements and the gestures of the user’s eyes and translate them into orders that will be sent to the second application. The second application will read the orders and activate the functioning of the mobile robot according to this orders.

The application is intented for users suffering from ALS. These users could have motor difficulties in their hands that don’t let them to use an electric chair. In this project we propose the alternative of using ocular inspection instead of an electric control to facilitate the use for these users.

1. Introducción

El presente trabajo consistirá en la implementación del control de un robot móvil a través de inspección ocular. El guion estará dividido en los siguientes apartados:

1) **Introducción**: Explicación del trabajo a realizar, motivación y los objetivos del mismo, además del contexto sobre la situación actual de la enfermedad de la ELA y la investigación al respecto.
2) **Metodología**: Presentación de las diferentes partes de la metodología a seguir durante el desarrollo del trabajo.
3) **Tecnologías utlizadas**: Herramientas tanto software como hardware utilizadas.
4) **Reconocimiento facial: OpenCV vs Dlib**: Comparación de las herramientas de reconocimiento facial probadas para este trabajo y conclusión final sobre cuál de ellas utilizar.
5) **Detección ocular. Pestañeo y Eye Tracking**: Explicación en detalle de la implementación del programa capaz de realizar la detección del pestañeo y el eye tracking en tiempo real.
6) **Robot móvil**: Funcionamiento del robot móvil que se utilizará en este trabajo y explicación del programa implementado del mismo.
7) **Ejemplos de uso**: Uso del programa y calificación de su funcionamiento bajo diferentes circunstancias.
8) **Conclusiones**: Reflexiones finales sobre el desarrollo de este trabajo.
9) **Webgrafías**: Webs de las que se ha extraído información para este trabajo.

1.1 Motivación

La **esclerosis lateral amiotrófica (ELA)** es una enfermedad que afecta al tronco cerebral y la médula espinal, y entre otros síntomas uno de los más remarcables es el debilitamiento muscular. Se conoce además que una de las primeras partes del cuerpo que se ve afectadas son las manos.

Esta enfermedad suele llevar a los afectados a utilizar sillas de ruedas para facilitarles el movimiento. Sin embargo estas conllevan otro problema, ya que a día de hoy las sillas eléctricas más comunes funcionan con un mando que el usuario maneja con la mano. La sensibilidad de la mano puede ser limitada y esto dificulta o incluso imposibilita al usuario el uso de la silla eléctrica, pese a que en muchos casos es indispensable para que el usuario haga vida normal.

Este trabajo tratará de dar solución a este problema añadiendo controles mediante inspección ocular a la silla eléctrica. El objetivo es que el usuario pueda manejar hacia donde avanza su silla desviando ligeramente su mirada hacia izquierda o derecha, o avanzar recto si el usuario se mantiene mirando al frente.

1.2 Objetivos

Los principales objetivos de este trabajo podrían dividirse de la siguiente forma:

- Estudiar e implementar los métodos actuales de **detección facial** y estimar la precisión de cada uno para seleccionar el que más se adecúe a las exigencias del proyecto.
- Implementar métodos de "**Eye Tracking**" y realizar un seguimiento de la vista del usuario, almacenando información sobre hacia que dirección mira en tiempo real.
- Comunicar las diferentes partes de la implementación, traduciendo los movimientos oculares del usuario a instrucciones de movimiento para el robot móvil.
- Estudiar el funcionamiento del **robot móvil** proporcionado, haciéndolo funcionar mediante las instrucciones recibidas desde el seguimiento ocular del usuario.
- Ajustar los parámetros de velocidad del robot para proporcionar la mejor **experiencia de usuario** posible en su manejo.

1.3 Contexto y antecedentes

En el siguiente apartado se detallarán las características principales de la enfermedad de la Esclerosis Lateral Amiotrófica (ELA), como puede afectar a los que la padecen y como podríamos nosotros ayudar con este trabajo a estas personas. Además se verán ejemplos de los antecedentes tecnológicos que se han desarrollado para la ayuda a los enfermos de ELA, y como este trabajo se basa en parte en los fundamentos que ya establecieron anteriormente otros proyectos.

1.3.1 Esclerosis Lateral Amiotrófica (ELA)

La **ELA** es una enfermedad de las neuronas en el cerebro, el tronco cerebral y la médula espinal que controlan el movimiento de los músculos voluntarios. La consecuencia es una debilidad muscular que puede avanzar hasta la parálisis, extendiéndose de unas regiones corporales a otras. Amenaza la autonomía motora, la comunicación oral, la deglución y la respiración, aunque se mantienen intactos los sentidos, el intelecto y los músculos de los ojos.

Uno de cada 10 casos de ELA se debe a un defecto genético. La causa se desconoce en la mayoría del resto de los casos y actualmente no existe una cura para esta enfermedad mortal, pero si tratamientos para facilitar la vida de quienes la padecen, ya que estos necesitan cada vez más ayuda para realizar las actividades de la vida diaria.

Los síntomas no suelen presentarse hasta después de los 50 años, aunque no significa que no puedan aparecer en personas más jóvenes. Los primeros síntomas debilitan brazos y piernas, expandiéndose por otros grupos musculares a medida que la enfermedad empeora. Otros síntomas asociados son depresión, demencia y pérdidas de peso y de memoria.

Uno de los mayores problemas que esta enfermedad acarrea son los problemas respiratorios. Anteriormente se mencionaba que los grupos musculares se debilitan con el tiempo, y esto no se refiere exclusivamente a brazos y piernas. Los músculos necesarios para respirar también se ven afectados y esto suele llevar

a las personas con ELA a necesitar métodos de respiración artificial, como los respiradores que utilizan las personas que padecen apnea del sueño por las noches. Estos dispositivos ayudan a la respiración con presiones a dos niveles a través de una mascarilla. En algunos casos el paciente recurre a una traqueostomía que le ayuda a respirar a través de un orificio que se crea en la parte frontal del cuello y que conecta con la tráquea. La insuficiencia respiratoria es la causa más común de muerte para las personas con ELA.

Otro grupo muscular que sufre esta enfermedad es el que se encarga del habla. La persona empieza con dificultades leves en el habla, pero se agrava con el tiempo. Eventualmente se vuelve difícil de entender para otros. Para este problema actualmente ya existen numerosas soluciones tecnológicas similares a las que se proponen en este trabajo, haciendo uso de técnicas de visión por computación. Estas herramientas transcriben las palabras que la persona indica con los ojos, usando inspección ocular para determinar las letras y textos predictivos para agilizar el proceso. Una de las aplicaciones que se conocen para esta función es "**GazeSpeak**", de la cuál hablaremos más adelante.

Junto a los problemas para el habla, la debilidad que presentan en los músculos de la boca afecta también a la alimentación. La falta de control en la deglución ocasiona riesgos de que entren alimentos y líquidos en las vías respiratorias, lo que puede originar una neumonía. Actualmente la solución más común a este problema es el uso de sondas de alimentación.

Finalmente, la personas con ELA desarrollan problemas con la memoria y la toma de decisiones, que con el paso del tiempo se diagnostican como demencia. Concretamente se suele llamar a la demencia ocasionada por esta enfermedad como demencia frontotemporal.

Los factores de riesgo que se asocian al desarrollo de la ELA no están claros aún, pero si que se identifican unos patrones entre las personas que la padecen. Normalmente suelen estar asociados a factores hereditarios; entre el 5 y el 10% de las personas con ELA la heredaron. Los hijos de personas con ELA tienen un 50%

de probabilidades de desarrollar la enfermedad. Este factor viene de la mano de la genética, la cuál también se atribuye a las razones de la aparición de la enfermedad. Algunos estudios del genoma humano encontraron que efectivamente existen similitudes en las variaciones genéticas de las personas con ELA, tanto aquellas con familiares que la padecen como las que no.

Otras posibles causas son las asociadas a la edad y el sexo de la persona. La ELA se desarrolla por norma general entre los 40 y mediados de los 60 años. Respecto al sexo, antes de los 65 años se observa que la enfermedad es un poco más común en hombres que en mujeres. A partir de los 70 años esta diferencia disminuye.

Existen también factores de riesgo ambientales que pueden desencadenar la ELA. El primero de ellos es el tabaquismo; fumar es el factor de este tipo que más relación parece tener con la enfermedad. Otros factores de riesgo menos demostrados pueden ser la exposición a toxinas ambientales, como el plomo u otras sustancias en lugares de trabajo de alto riesgo, o el servicio militar, por razones similares como la exposición a sustancias químicas nocivas, aunque ninguna de estas últimas han sido demostradas con certeza.

1.3.2 Nuevas tecnologías como ayuda al enfermo

Como se ha mencionado anteriormente, la ELA no afecta a los sentidos (al menos en la mayoría de los casos). Es aprovechando esto que se han llevado a cabo diferentes experimentaciones tecnológicas que facilitan la vida de los pacientes de ELA. A continuación se detallarán algunas de las herramientas más usadas, las cuáles se han tomado como ejemplo para este trabajo.

- **Inspección ocular como ayuda al habla**

De la misma manera que en este trabajo se emplean técnicas de inspección ocular para controlar la dirección de una silla con los debidos conectores, existen varias aplicaciones que han explorado el uso de los ojos para escribir y traducir a

voz las palabras que una persona quiere decir. Algunas de estas aplicaciones son las siguientes:

GazeSpeak (Figura 1), desarrollada por Microsoft junto a investigadores de la Universidad de Washington, es una simple aplicación de móvil que proporciona a la gente con ELA una forma de “hablar con los ojos”.

La aplicación requiere que se apunte la cámara hacia los ojos del usuario. El usuario sabrá gracias a una pegatina en la parte trasera del móvil las letras de las que dispone mirando en cada dirección (arriba, abajo, izquierda, derecha). Primero debe mirar al cuadrante que desea y luego a la letra en concreto que quiere escribir. La escritura se facilita gracias al sistema de texto predictivo, que tratará de adivinar las palabras que se quieren escribir.

Figura 1: GazeSpeak siendo utilizada en un móvil con la pegatina y las letras

Otra aplicación con una funcionalidad similar es **Tallk** (Figura 2). A diferencia de la anterior, esta aplicación incorpora directamente un teclado, también ayudado por la funcionalidad del texto predictivo. Como añadido, la aplicación cuenta con una

sección de frases recurrentes que pueden ser muy útiles al usuario, facilitando aún más el habla.

Además, Tallk incorpora funcionalidades aparte del habla, como el control de la domótica en el hogar, que conectada con el dispositivo móvil permite al usuario apagar o encender dispositivos en casa, controlar las persianas, etc.

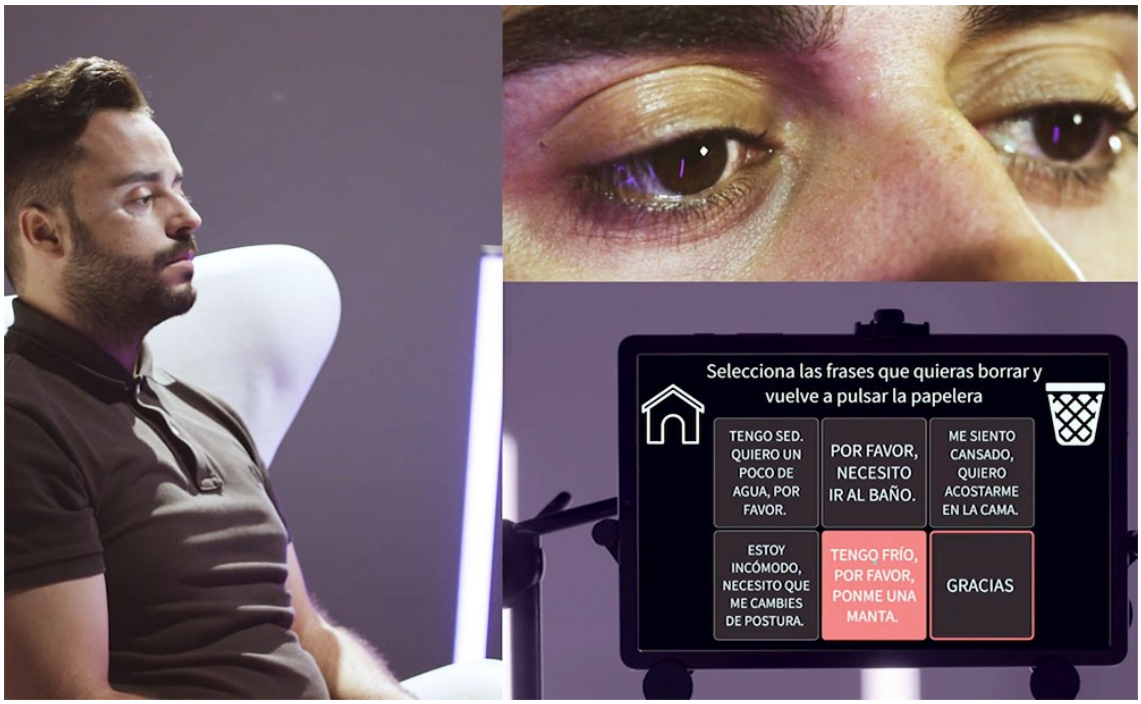

Figura 2: Tallk y su sección de frases recurrentes

- **Interfaz cerebro-computadora**

Existen también tecnologías más sofisticadas que conectan directamente el cerebro con los propios dispositivos electrónicos del usuario mediante señales cerebrales. Uno de estos dispositivos es Stentrode.

Stentrode es un dispositivo que ya se ha probado en personas que padecen ELA y ha probado ser muy útil para completar tareas rutinarias con dispositivos móviles u ordenadores tales como consultar la banca en línea, hacer compras o enviar mensajes de texto. Stentrode es lo que se conoce como una interfaz cerebro-

computadora, las cuales se basan en la conversión de los impulsos nerviosos desde la corteza motora en una señal digital.

Las pruebas que se realizaron con personas con ELA fueron muy positivas, pasando de un tiempo con supervisión a ver como los dispositivos facilitaban la vida diaria de sus usuarios una vez acostumbrados a los mismos. Sin embargo, esta tecnología aún se encuentra en desarrollo y salvo algunos casos no está disponible al público. Aún queda mucho por explorar en estos ámbitos.

- **Stephen Hawkins, “an exceptional man”**

Uno de los casos más conocidos de la ELA es sin duda el que afectó a **Stephen Hawkins** (Figura 3). El galardonado físico fue diagnosticado de ELA a muy temprana edad, 21 años. La enfermedad empezó debilitando sus músculos pero con el tiempo se extendió hasta dejarlo completamente paralizado y obligado a moverse con una silla de ruedas.

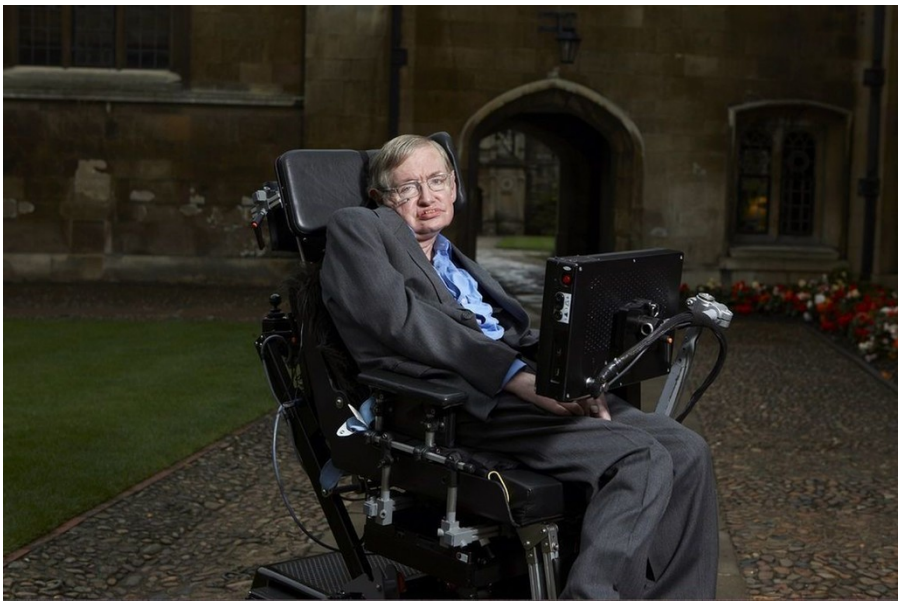

Figura 3: Stephen Hawking, físico especializado en el estudio de agujeros negros

Sin embargo, Stephen Hawking fue un caso excepcional dentro de las personas con ELA, ya que ha sido la persona más longeva con esta enfermedad, a

la que sobrevivió durante 55 años, cuando la esperanza media de vida es de aproximadamente 14 meses una vez diagnosticada la enfermedad.

Los neurólogos que estudian su caso se referían a él como “excepcional”, no solo por sobrevivir a la enfermedad durante tanto tiempo, sino porque la enfermedad parecía haberse desvanecido de su cuerpo. La ELA no cesa en el deterioro de las neuronas motoras, sin embargo, en el caso de Stephen y llegado a un punto de deterioro, este se frenó, haciendo posible que viviera por tantos años siendo el brillante científico que fue.

Hawking por supuesto tuvo ayuda de diversas tecnologías para su día a día y su trabajo. Fue sometido a una traqueostomía, por lo que tuvo que comunicarse mediante su sintetizador de voz, el cuál consistía en una interfaz que le dejaba escoger palabras mediante gestos de su cabeza, ojos y contracciones de sus mejillas. Del mismo modo, usaba movimientos de su cabeza y ojos para controlar su silla de ruedas, de forma similar a la que trataremos de replicar en nuestro proyecto.

Llegado a un punto crítico de deterioro, Hawking no era capaz de hablar a un ritmo superior a una palabra por minuto. Fue en 2011 cuando pidió ayuda a Intel, quienes diseñaron una nueva interfaz que implementaba predicción de texto, lo cuál ayudó no solo a Hawking, sino a todos los enfermos de ELA, ya que hoy en día es un estándar en la aplicaciones de ayuda al habla mediante inspección ocular.

2. Metodología del proyecto

El seguimiento de los movimientos oculares se basa en el uso de técnicas de **detección de objetos** y de **reconocimiento facial**. Se han evaluado dos posibles técnicas, tratando de obtener datos sobre cuál de ellas proporciona un mejor reconocimiento del ojo.

La primera técnica que será evaluada consiste en una serie de ajustes de **tratamiento de imágenes** en tiempo real sobre la imagen recogida por una cámara.

El primer objetivo es hallar la cara del usuario dentro de la imagen, y posteriormente el ojo de este. Una vez localizado, se trabajará la imagen para hallar la zona más cercana al color negro, puesto que esta por lo general se corresponderá con el la pupila o el iris. Con estas partes localizadas, simplemente se seguirá el centro en ellas y se tomarán los valores x e y de las mismas para determinar si el usuario estuviese mirando hacia una dirección u otra.

La segunda opción que se tiene en cuenta es utilizar **modelos de reconocimiento facial** ya entrenados.

Un modelo de reconocimiento es un conjunto de algoritmos que trata de detectar un determinado objeto o parte del cuerpo (en este caso, caras). Estos algoritmos mejoran sus predicciones gracias a técnicas de inteligencia artificial, las cuáles pueden consistir en usar el programa con una gran cantidad de imágenes e indicarle si el reconocimiento ha sido correcto o no. Gracias a esto, el programa de reconocimiento “aprende” y se vuelve mucho más preciso.

El modelo que se va a utilizar se basa en la localización de una cantidad de puntos clave que forman un rostro humano. De esta forma se podría en primer lugar encontrar la cara del usuario y una vez localizada trabajar solo con los puntos correspondientes al ojo del mismo.

Una vez decidamos la técnica a utilizar y se recojan los valores asociados al movimiento ocular del usuario se traducirían a señales que recibirá la silla eléctrica. También se incluirá un pequeño programa que nos ayude a calibrar la detección de la cara y los límites a partir de los cuáles la silla empiece a moverse, a gusto del usuario.

La implementación de la aplicación se realizará en **Python**, haciendo uso de las herramientas software que se detallarán a continuación. El programa será funcional en **Raspberry**.

Durante el uso normal de la aplicación se pretende hacer pasar al usuario por un proceso de calibración para ajustar el programa a las diferentes casuísticas en cuanto a posición del usuario, iluminación, etc. Tras este proceso de calibración, el programa comenzaría a funcionar y el usuario podría manejar el robot con la mirada.

El trabajo será probado con un **robot móvil** para verificar que la recogida de datos del movimiento de los ojos funcione correctamente. El robot estará programado en el lenguaje de bloques de **MakeCode** que pone a nuestra disposición la propia placa controladora **Micro:Bit** que se usará para manejarlo.

Se pretende en un futuro integrar el sistema en funcionamiento a una silla eléctrica, integrando la cámara necesaria a la silla eléctrica dirigida al usuario.

3. Tecnologías utilizadas

Para este trabajo se necesita un conjunto de herramientas tanto de carácter software como hardware.

La parte software del proyecto controlará diversos procesos como el seguimiento ocular, la traducción de los gestos de los ojos a movimientos del robot, el propio funcionamiento del robot o la interconexión entre los distintos programas.

Por otro lado, las herramientas hardware serán el propio robot que recibirá las instrucciones del programa y sus componentes. El robot está compuesto por una serie de placas de Arduino de diferentes tipos, los motores y las ruedas que le permiten moverse, además del cableado que conecta todas sus partes entre sí.

3.1 Herramientas software

Las **herramientas software** que se utilizarán se componen de lenguajes de programación, bibliotecas software para el tratamiento de imágenes, algoritmos y metodologías de reconocimiento facial y los entornos de programación que se usarán para gestionar el código de nuestros programas:

- **PyCharm**: Entorno de programación orientado al lenguaje Python. El reconocimiento facial y el seguimiento ocular se realizará con un programa en Python.
- **OpenCV**: Biblioteca libre para el tratamiento de imágenes basada en C++ con conectores para Python.
- **Dlib**: Biblioteca basada en deep learning para el reconocimiento facial implementada en C++ con conectores para Python.
- **MakeCode**: Entorno de programación en bloques. MakeCode es una herramienta de Microsoft para la programación de placas Micro:Bit de forma sencilla y didáctica.

3.2 Herramientas hardware

Las **herramientas hardware** son el propio robot móvil y las placas Arduino que controlan su funcionamiento:

- **Robot Maqueen para Micro:Bit**: Robot de programación gráfica para educación STEM diseñado para BBC Micro:Bit.
- **Placa Micro:Bit V2 - Controlador**: Modulo que controlará el funcionamiento del robot, se conecta de forma sencilla al mismo en su ranura asignada.
- **Módulo Bluetooth HC-06**: Módulo que permitirá mandar órdenes recibidas desde nuestro programa a la placa Micro:Bit vía Bluetooth.
- **Motores N20**: Motores del robot móvil.
- **Raspberry Pi 3 Model B+**: Placa Raspberry de computación capaz de ejecutar programas en Python que integra comunicación Bluetooth.
- **Cámaras de vídeo**: Para captar las imágenes que serán procesadas para el reconocimiento facial del usuario. En este caso la cámara frontal de un dispositivo móvil Xiaomi Redmi Note 8 de 13 MP y una cámara USB para la implementación en Raspberry.

3.3 Lenguajes de programación

Los lenguajes de programación utilizados en este trabajo son los siguientes:

Logo	Nombre
python	Python
MakeCode	Programación en bloques (Python/JavaScript)

3.3.1 Python

Python es un lenguaje de alto nivel de programación que destaca por la alta legibilidad de su código. Se dice que es un lenguaje de programación multiparadigma ya que soporta parcialmente la orientación a objetos, programación imperativa y también parcialmente la programación funcional. Es un lenguaje interpretado, dinámico y multiplataforma.

Está administrado por Python Software Foundation, posee una licencia de código abierto y es uno de los lenguajes de programación más populares. Se utiliza para desarrollar aplicaciones de todo tipo, por ejemplo Instagram, Netflix, Spotify, Panda 3D entre otros.

3.3.2 Programación en bloques MakeCode

Para la programación del robot con la placa Micro:Bit integrada se utilizará el entorno de programación que ofrece Microsoft: **MakeCode**. En MakeCode se utilizará programación en bloques, una programación de carácter didáctico y base en los lenguajes **Python** y **Javascript**.

Aunque se podría escribir código directamente en estos lenguajes, este tipo de programación permite un desarrollo rápido y sencillo arrastrando y ordenando bloques que desempeñan las distintas funciones.

3.4 Entornos de programación

Los entornos de programación utilizados en este trabajo han sido los siguientes:

Logo	Lenguaje de programación	Función
PyCharm	Python	Programación del programa de reconocimiento facial y seguimiento ocular.
Microsoft MakeCode	Programación en bloques (Python/JavaScript)	Programación de la placa controladora del robot.

3.4.1 Pycharm

Pycharm es un entorno de desarrollo integrado (IDE) utilizado en programación y especializado para el lenguaje de programación Python. Está desarrollado por la empresa JetBrains (IntelliJ anteriormente). Pycharm es multiplataforma y se publica la Licencia Apache, contando también con una Professional Edition accesible por suscripción.

Entre las facilidades que ofrece se incluyen análisis de código, un depurador gráfico, un probador de unidades integrado y un sistema de control de versiones. Además, incluye una consola Python desde la que podemos incluir las librerías que necesitemos de forma sencilla mediante comandos, además de configurar otras herramientas de Python.

3.4.2 MakeCode

La aplicación web de MakeCode consiste en un entorno de programación sencillo y de carácter didáctico para todo tipo de alumnos, desde alumnos de primaría dando sus primeros pasos en la programación a programadores en formaciones superiores. Esto se debe a que aunque la programación se realiza mediante **bloques** y por lo tanto de forma sencilla, las posibilidades que ofrece son muy amplias. En este caso se utilizará este entorno para programar la placa Micro:Bit, pero también ofrece soporte para programar circuitos electrónicos, programas en Minecraft e incluso pequeños juegos arcade.

La base de la programación en bloques es código en **Python** o **Javascript**. Al crear un proyecto se puede cambiar en cualquier momento de la programación en bloques a estos dos lenguajes para ver y manipular el código que hay debajo de nuestra estructura de bloques.

Se puede crear un nuevo proyecto de forma sencilla desde la web https://makecode.microbit.org/#

3.5 OpenCV

OpenCV es una biblioteca libre de visión artificial desarrollada por Intel. Sus herramientas resultan muy útiles para todo tipo de funcionalidades relacionadas con visión por computación. Es sin duda una de las herramientas más populares para estos propósitos debido a que es libre, multiplataforma y por la gran cantidad de documentación que ofrece en sus repositorios.

Las herramientas que ofrece OpenCV van desde las más sencillas modificaciones a una imagen, como la aplicación de filtros Gaussianos o simples tratamientos de imágenes, hasta técnicas de reconocimiento facial, de gestos, tracking visual, etc. (Figura 4). Por supuesto, las herramientas más sencillas tras varias aplicaciones son las que nos permiten llevar a cabo las técnicas de reconocimiento que vamos a utilizar en este proyecto.

La instalación de OpenCV en Python es sencilla, como la instalación de la mayoría de librería en Python basta con introducir la sentencia:

```
pip install opencv-python
```

Aunque se debe tener en cuenta que para el uso óptimo de la librería OpenCV es necesaria la librería **Numpy**, que se puede instalar de forma similar.

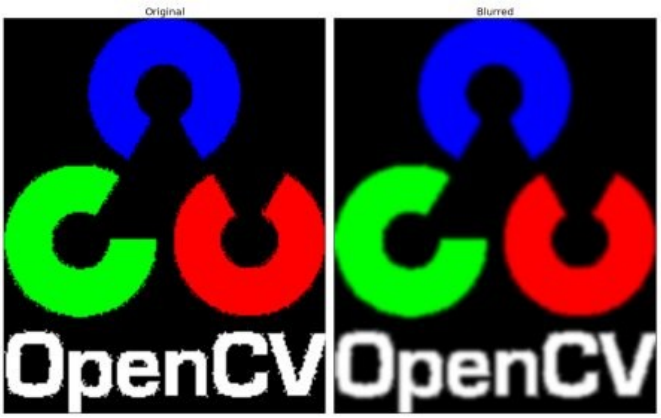

Figura 4: Logo de OpenCV antes y después de aplicar un filtro Gaussiano

3.6 Dlib

Dlib es otra biblioteca que contiene algoritmos y herramientas de machine learning para la resolución de problemas que comprenden los ámbitos de la robótica, la telefonía móvil o entornos de computación de alto coste. Al igual que con OpenCV, existe gran cantidad de documentación y ejemplos sobre esta librería y también ha sido probada en los principales sistemas operativos actuales.

Algunas de las funcionalidades que incluye son algoritmos de deep learning enfocados a la detección de objetos, detección de los mismos en diferentes capas y herramientas relacionadas con el tratamiento de imágenes como las que también dispone OpenCV.

La instalación de la librería vuelve a ser tan sencilla como con OpenCV, simplemente se ejecutará la línea de comando que instala Dlib en el entorno Python:

```
pip install dlib
```

Igual que OpenCV requería Numpy como biblioteca adicional, para Dlib se debe instalar antes **Cmake** en el entorno Python.

3.7 DroidCam

DroidCam es una pequeña aplicación tanto para dispositivos móviles (Figura 6) como de escritorio (Figura 5), que nos permite utilizar la cámara del dispositivo móvil como si fuera la propia de nuestro PC, pudiendo accederla desde los programas que se implementan.

El hecho de que tenga versiones para móviles y para PC es porque cada versión es diferente y la conexión se hace a través de las mismas. Desde la versión de móvil se proporciona una IP y un puerto de conexión. Estos datos se introducen en la versión de escritorio de la aplicación y se realiza la conexión.

Desde este momento podemos acceder a la cámara del dispositivo móvil desde el PC.

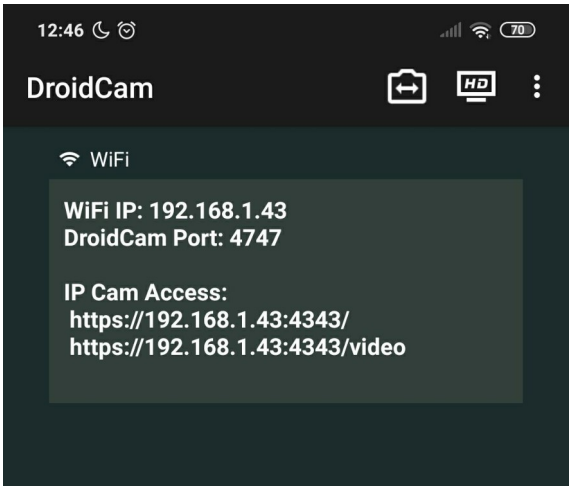

Figura 6: Versión móvil de DroidCam

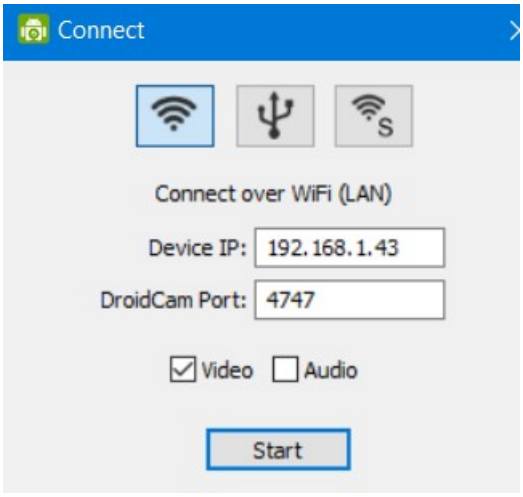

Figura 5: Versión de escritorio de DroidCam

Existe también la opción de conectar el móvil al PC vía USB para acceder a la cámara, pero la conexión vía WiFi otorga la facilidad de no necesitar cables y poder mover más libremente la cámara para realizar las pruebas necesarias. Una vez realizada la conexión se muestra la cámara en la aplicación de escritorio (Figura 7).

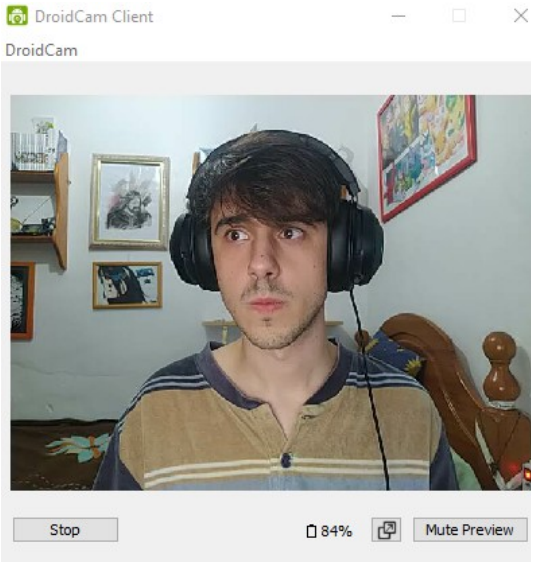

Figura 7: Cámara de móvil captada en PC mediante DroidCam.

4. Reconocimiento facial. OpenCV vs Dlib.

El **reconocimiento facial** o **detección de caras** es un tipo concreto de detección de objetos. La detección de objetos consiste en ubicar la posición de un determinado objeto dentro de una imagen o un vídeo. Existen actualmente diferentes métodos para la detección de caras por ordenador encuadradas dentro del marco de la visión por computación.

La primera implementación que se realizará será el programa que reconozca la cara del usuario y los gestos de la misma, que serán posteriormente traducidos a los movimientos del robot móvil. Como se indica en la anterior sección, para esta tarea se dispone de dos posibles herramientas: las librerías de OpenCV y de Dlib.

En está situación se optó por implementar el reconocimiento facial de dos formas distintas y observar cuál de las dos era más precisa, ya que es necesario que el programa no falle en el reconocimiento porque podría suponer perder el control del robot móvil o tener que suplir durante esos instantes las instrucciones por valores por defecto.

A continuación se detallarán el funcionamiento de ambas opciones y los resultados y las conclusiones obtenidas:

4.1 OpenCV y el clasificador Haar Cascade

OpenCV utiliza como herramienta de reconocimiento facial el clasificador **Haar Cascade**. Un clasificador consiste en un sistema de machine learning consistente en el entrenamiento de un algoritmo que clasifica elementos como positivos o negativos para un determinado problema, en este caso, la detección o no de un cara en una imagen.

Haar Cascade presenta una característica diferenciadora de otros algoritmos similares: su tratamiento de una misma imagen "**en cascada**" (Figura 8). Para determinar una imagen como positiva o negativa en la detección de caras, el algoritmo se ejecuta varias veces para diferentes áreas de la imagen, identificando si existe o no un rostro o parte del mismo en esa parte de la imagen. Una vez determinado si en alguna de esas zonas existe una cara o parte de la misma, y tras poner en común todas esas zonas para reconstruir un posible rostro, es cuando el algoritmo determina si la imagen tiene un rostro y cuál es el área en la que se encuentra.

OpenCV proporciona esta herramienta ya entrenada, es decir, el algoritmo Haar Cascade después del proceso de recibir gran cantidad de imágenes en las que podía haber rostros o no, y habiéndolas clasificado como tal. Esto es lo que se denomina como "aprendizaje" del algoritmo, lo cuál nos garantiza una mayor precisión de la herramienta.

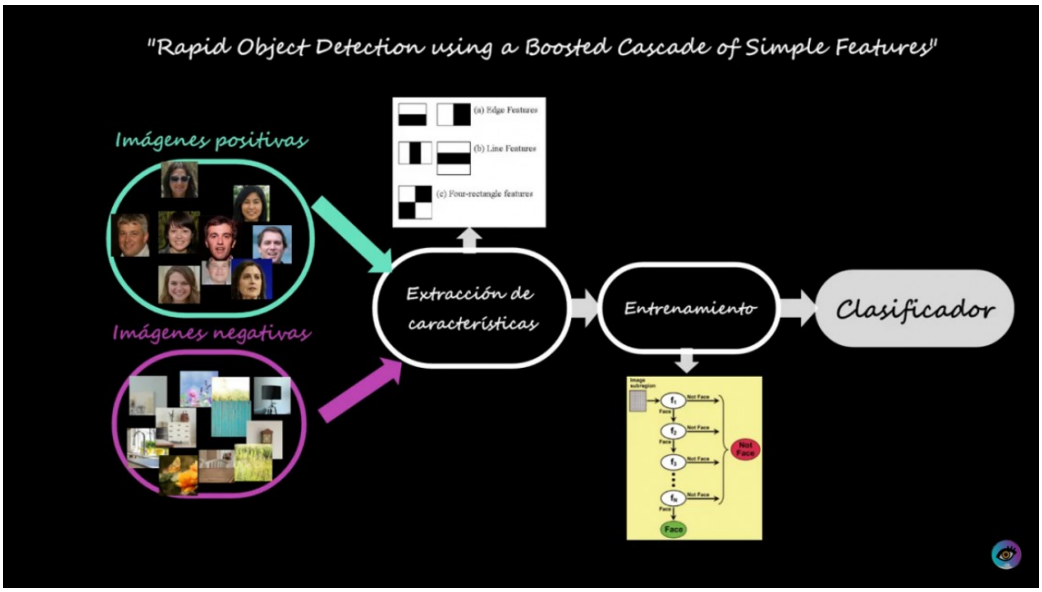

Figura 8: Método de entrenamiento del modelo Haar Cascade

4.1.1 Aplicación de Haar Cascade con OpenCV

Teniendo entonces el algoritmo de Haar Cascade entrenado, se procede a aplicarlo mediante las herramientas de OpenCV. OpenCV ofrece no solo detectores faciales al uso, también ofrece detectores de ojos, de sonrisa e incluso detectores faciales de animales. Todos estos detectores están en formatos .xml, y se pueden descargar desde su repositorio en Github:

https://github.com/opencv/opencv/tree/master/data/haarcascades

En este trabajo se usarán los detectores faciales y de ojos, los correspondientes a los ficheros: **haarcascade_frontalface_default.xml** y **haarcascade_eye.xml**. Con los ficheros añadidos al proyecto Python, se utilizan las siguientes líneas de código para cargarlos en el programa (Figura 9):

```
face_cascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
eye_cascade = cv2.CascadeClassifier('haarcascade_eye.xml')
```

Figura 9: Carga en Python de los modelos Haar Cascade para rostros y ojos.

Tras realizar algunos tratamientos sobre la imagen que hace la detección facial más precisa y con los modelos cargados, OpenCV permite obtener las zonas dentro de la imagen que el algoritmo considera una cara con las siguientes instrucciones (Figura 10):

```
faces = face_cascade.detectMultiScale(gray, 1.3, 2)
eyes = eye_cascade.detectMultiScale(roi_gray)
```

Figura 10: Zonas detectadas para ojos y caras con Haar Cascade

Lo que se almacena después de esta instrucción en los objetos “faces” y “eyes” son las dimensiones de las zonas que el algoritmo ha detectado como una cara o un ojo. Con estos objetos se puede dibujar con instrucciones sencillas de OpenCV rectángulos que identifiquen en una imagen o video los elementos detectados (Figura 11).

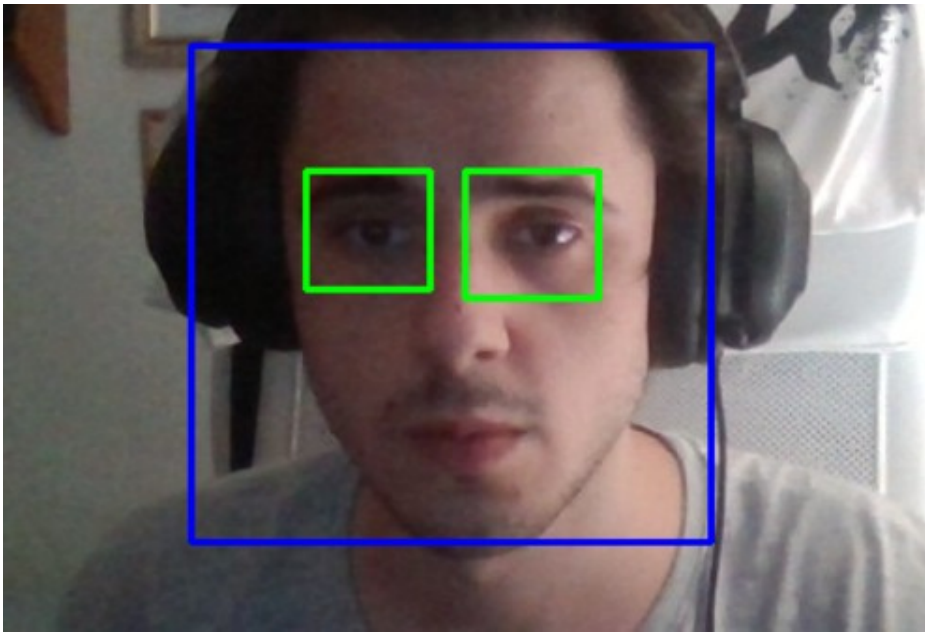

Figura 11: Cara y ojos detectados en una imagen mediante Haar Cascade

Tras comprobar el funcionamiento en cámara a tiempo real de este método se ha observado que es algo inconsistente. En un entorno despejado, claro e iluminado, únicamente con un rostro en pantalla, al realizar movimientos leves esperables en una ejecución real del programa, se producen fallos como las pérdidas momentáneas de la detección o la aparición de falsos positivos (dibuja rectángulos donde no hay caras).

Se llega a la conclusión temporal de que Haar Cascade funciona especialmente bien para imágenes estáticas, pero quizás no resulta tan útil para videos, y ya que este trabajo requiere la detección facial en tiempo real, es posible que no sea la solución ideal.

4.2 Dlib y Face Landmarks Detector

Dlib ofrece un método distinto a Haar Cascade para la detección facial. En lugar de un **modelo entrenado** en la detección de caras, Dlib detectará rostros a partir de un modelo de puntos que componen el contorno básico de una cara humana. De esta forma, dada una imagen en la que detectar un rostro, el programa compara las zonas de la imagen tratando de encontrar similitudes con el modelo cargado. Así, se buscarán en la imagen formas similares a la siguiente (Figura 12):

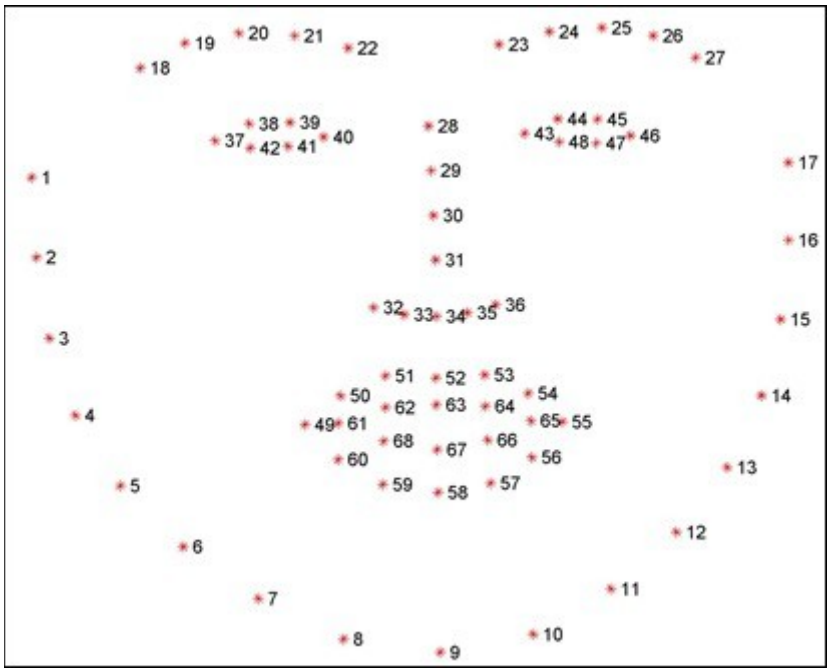

Figura 12: Modelo detector facial mediante landmarks numeradas

Como se puede observar, los puntos que componen el modelo están numerados. Esto será muy útil ya que con este método no se dispone de una forma directa de detectar los ojos en la imagen, pero si que se almacena el lugar exacto donde se identifica la similitud con cada uno de los puntos del modelo, por lo que se puede trabajar obteniendo las posiciones de los puntos que corresponden únicamente a los ojos.

4.2.1 Aplicación de Face Landmarks Detector con Dlib

La forma en la que se utilizará el **Face Landmarks Detector** en Dlib es similar a la forma en la que se utiliza el algoritmo de Haar Cascade con OpenCV. Los datos respectivos a los puntos que conforman el rostro por defecto pueden descargarse a partir de la propia web de Dlib:

dlib.net/files/shape_predictor_68_face_landmarks.dat.bz2

Este fichero consiste en un set entrenado de imágenes etiquetadas manualmente con los landmarks que conforman un rostro humano y especificando los valores x e y de las propias marcas. A partir de este fichero, la detección de rostros se realiza buscando en la imagen relaciones entre pares de posible píxeles identificados como landmarks del rostro humano. Estas relaciones funcionarán como un árbol de regresión estimando la posición del rostro cuando se estima que la detección de los puntos y la separación entre los mismos se adecúa a lo que el modelo entrenado reconoce como un rostro.

Al igual que utilizando Haar Cascade, se cargan los datos en el programa con las siguientes instrucciones (Figura 13):

```
detector = dlib.get_frontal_face_detector()
predictor = dlib.shape_predictor("shape_predictor_68_face_landmarks.dat")
```

Figura 13: Carga en Python del modelo Face Landmarks Detector

En primer lugar, se carga un detector de la cara y luego la predicción de los puntos exactos que conforman el rostro de forma similar a nuestro modelo. Tras unos ajustes de tratamiento de imágenes similares a los utilizados con la anterior técnica, se procede a obtener las caras detectadas y los puntos obtenidos (Figura 14):

```
faces = detector(gray)

for face in faces:
    landmarks = predictor(gray, face)
```

Figura 14: Carga de los rostros y los puntos asociados a las Landmarks

El detector almacena en “faces” los rostros detectados en la imagen, para seguidamente recorrerlos y almacenar en “landmarks” los puntos exactos que conforman el rostro detectado.

Con los landmarks detectados, se pueden extraer aquellos que sean de interés para la detección de zonas concretas de la cara, en el caso de este trabajo, los ojos. Se comprueba en la imagen del modelo que las marcas asociadas al ojo derecho van desde la 37 a la 42 y las del ojo izquierdo desde la 43 a la 48. Para identificar en tiempo real donde se está detectando el ojo, se dibuja un polígono que recorre todos los puntos de cada ojo de la siguiente manera (Figura 15):

```
left_eye_region = np.array([(facial_landmarks.part(36).x, facial_landmarks.part(36).y),
                            (facial_landmarks.part(37).x, facial_landmarks.part(37).y),
                            (facial_landmarks.part(38).x, facial_landmarks.part(38).y),
                            (facial_landmarks.part(39).x, facial_landmarks.part(40).y),
                            (facial_landmarks.part(41).x, facial_landmarks.part(41).y),
                            (facial_landmarks.part(42).x, facial_landmarks.part(42).y)], np.int32)

cv2.polylines(frame, [left_eye_region], True, (0, 0, 255), 2)
```

Figura 15: Dibujo de un polígono alrededor de los landmarks del ojo derecho

Nótese que los puntos almacenados en un vector son los correspondientes menos 1, ya que el vector cuenta con la posición 0. El resultado de este código sería el siguiente (Figura 16):

Figura 16: Dibujo de polígonos detectando la posición de ambos ojos.

Tal y como se muestra en la imagen, la detección es bastante precisa. No solo eso, además es más consistente que la obtenida con Haar Cascade, produciendo menos falsos positivos y manteniendo más estable la detección tanto de la cara como de los propios ojos, de los cuales se hace un seguimiento más que decente en tiempo real.

4.3 Conclusión y solución final

Sin duda la forma más precisa de las dos estudiadas es la predicción mediante comparación con el **modelo de Facial Landmarks**. Entre las diferencias en el funcionamiento de ambas opciones destacan la consistencia en la detección del rostro y la precisión para ubicar el mismo e incluso las partes más concretas como los ojos.

Además de ser más preciso en la detección de ojos, utilizando la comparación con las Facial Landmarks y precisando de esta forma la detección clara de cada una de sus partes, se observa que aparecen muchos menos falsos positivos, mientras que el algoritmo de Haar Cascade, que se puede intuir que detecta ojos como zonas pequeñas oscuras rodeadas de zonas mas claras, da lugar a muchos fallos ya que

encuentra con facilidad partes de la imagen con esas características fuera de la cara, en cualquier lugar del fondo que enfoque la cámara.

Por todo esto finalmente se ha determinado que para este trabajo la mejor opción para la detección de rostros y seguimiento ocular es la que ofrece la librería **Dlib** con sus Facial Landmarks Detector. No obstante, la librería **OpenCV** sigue disponiendo de muchas herramientas de tratamiento de imagen e incluso de dibujado que serán muy útiles en el programa, por lo que aunque no se utilizarán para la detección facial, se mantendrán para utilizar otras muchas herramientas de las que ofrece.

Todas las pruebas han sido comprobadas utilizando la cámara frontal de 13 MP de un dispositivo móvil o una cámara USB, y tratando que las condiciones de luz y claridad en la imagen fueran lo mejores posible.

5. Detección ocular. Pestañeo y Eye Tracking

Tras la selección de las herramientas que se utilizarán para detectar los ojos en cámara, se buscará traducir los diferentes gestos del usuario a los movimientos del robot. En primer lugar, se debe determinar que tipos de movimientos realizará nuestro robot. Los resultados esperados deberían permitir controlar si el robot avanza o no y modificar la dirección del avance tanto a izquierda como a derecha.

El avance puede detenerse o reanudarse mediante un mismo comando que actúe como switch o interruptor. Un gesto que se puede detectar y que actúe de esta manera es el pestañeo o los guiños del usuario. Se debe tener en cuenta que el usuario va a pestañear de forma natural y no siempre significará que quiere activar o desactivar el dispositivo. Por ello, se tratará de detectar un guiño durante un tiempo determinado.

Por otra parte está la dirección a la que el usuario quiere que se dirija el robot. Para ello se tendrá en cuenta la intuitiva dinámica de hacer que el usuario mire hacia donde quiere ir. Mediante técnicas de “eye tracking” se determinará hacia donde esta mirando el usuario, en este caso diferenciando entre tres posibles posiciones: izquierda, centro y derecha. Si la persona está mirando hacia el centro el avance seguirá recto, mientras que si mira a las otras direcciones el robot girará hacia estas mismas direcciones reposicionándose para continuar su avance.

Los parámetros que hagan el robot manejable en cuanto a velocidad y control de dirección serán calculados más adelante tras realizar las pruebas necesarias.

A continuación se detallará como se van a detectar cada uno de los dos tipos de movimientos mencionados.

5.1 Detección del pestañeo

Para la detección del pestañeo se han investigado dos metodologías diferentes: una teniendo en cuenta la relación de aspecto de las landmarks cuando el ojo esta abierto o cerrado y otra observando la luminosidad de la imagen en la zona del ojo del usuario. A continuación se detallarán ambos procesos y cuál de los dos se ha utilizado finalmente.

5.1.1 Relación de aspecto de landmarks del ojo

Tras la detección de los ojos anteriormente mencionada mediante la comparación con la plantilla de landmarks, disponemos de una serie de puntos localizados en la imagen a tiempo real y que representan los ojos del usuario. Estos puntos pueden utilizarse para detectar la forma en la que se mueven los ojos y, entre otras cosas, el propio **pestañeo**.

Se comenzará por realizar una detección de los puntos de intersección que servirán para determinar si el ojo esta abierto o cerrado. Como se observaba en la figura anterior, cada ojo está enmarcado por **6 puntos**. A partir de dichos puntos se dibujarán las líneas de intersección.

Para ello bastará con coger tanto las posiciones de los puntos más a la izquierda y más a la derecha del ojo, como los puntos intermedios entre los puntos superiores e inferiores del ojo. Mediante la herramienta de dibujado de líneas de OpenCV, se comprobará que efectivamente se están tomando las líneas intermedias para cada ojo (Figuras 17 y 18).

```
left_point = (facial_landmarks.part(eye_points[0]).x, facial_landmarks.part(eye_points[0]).y)
right_point = (facial_landmarks.part(eye_points[3]).x, facial_landmarks.part(eye_points[3]).y)
center_top = midpoint(facial_landmarks.part(eye_points[1]), facial_landmarks.part(eye_points[2]))
center_bottom = midpoint(facial_landmarks.part(eye_points[5]), facial_landmarks.part(eye_points[4]))

hor_line = cv2.line(frame, left_point, right_point, (0, 255, 0), 2)
ver_line = cv2.line(frame, center_top, center_bottom, (0, 255, 0), 2)
```

Figura 17: Toma de los puntos medios de cada ojo y dibujado de líneas

Figura 18: Líneas intermedias horizontales y verticales de cada ojo

Estas líneas no son solo de carácter informativo. Podrán utilizarse para obtener lo que se denominará como "**blinking_ratio**". Este ratio vendrá determinado por la longitud de cada una de las líneas y la relación entre ambas longitudes.

Lo primero que se debe obtener es la longitud de cada línea, las cuales se obtienen de forma sencilla ya que están almacenados los valores x e y de cada punto. La fórmula que determina la distancia entre dos puntos en un plano es la siguiente:

$$d(A,B) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Esta fórmula es de hecho la misma que se utiliza para calcular la hipotenusa, por lo que se podrá utilizar directamente la fórmula que proporciona la librería "math" de Python. Una vez obtenidas las distancias, se calcula el ratio dividiendo la distancia de la línea horizontal con la de la línea vertical (Figura 19):

```
ver_line_lenght = hypot((center_top[0] - center_bottom[0]), (center_top[1] - center_bottom[1]))
hor_line_lenght = hypot((left_point[0] - right_point[0]), (left_point[1] - right_point[1]))

ratio = hor_line_lenght / ver_line_lenght
```

Figura 19: Cálculo de distancias medias en el ojo y "blinking_ratio"

Este valor será arbitrario y dependiente de la posición de la cámara y de la distancia entre esta y el rostro del usuario, pero se ve claramente afectado cuando el usuario pestañea. Tras una calibración y calcular los umbrales, se puede determinar que valores están asociados al ojo cerrado y al abierto.

Por último, hasta ahora se ha trabajado con el "blinking_ratio" de uno de los dos ojos. Podemos suponer que el pestañeo del usuario se realizará de ambos ojos a la vez y usar la media de ambos ratios para aumentar la precisión del programa. Para ello simplemente se calcula los ratios de cada ojo por separado y se realiza la media aritmética entre ellos (Figura 20).

```
left_eye_ratio = get_blinking_ratio([36, 37, 38, 39, 40, 41], landmarks)
right_eye_ratio = get_blinking_ratio([42, 43, 44, 45, 46, 47], landmarks)

blinking_ratio = (left_eye_ratio + right_eye_ratio) / 2
if blinking_ratio > blinking_umbral:
    cv2.putText(frame, "BLINKING", (25, 100), font, 5, (255, 0, 0))
```

Figura 20: Media de los ratios para cada ojo y estimación de pestañeo

Se puede mostrar cuando el programa está detectando un pestañeo con la herramienta "putText" de OpenCV, escribiendo en la propia cámara la palabra "BLINKING". Los resultados son los siguientes (Figura 21):

Figura 21: Detección del pestañeo al cerrar los ojos

Se observa como se contraen las líneas que marcan los puntos medios de los ojos al cerrarlos, especialmente la línea vertical. De esta forma se usará la relación entre ambos ojos en lugar de únicamente la longitud de esta línea para que el programa funcione correctamente sin importar la distancia entre la cámara y el usuario, aunque se espera que la cámara se mantenga fija una vez configurada.

La **detección del pestañeo** obtenida con este método es lo suficientemente precisa para el propósito de este trabajo. Apenas existen falsos positivos y en la mayor parte del tiempo que el ojo se mantiene cerrado se calcula correctamente que se esta realizando un pestañeo.

5.1.2 Luminosidad de la zona del ojo

Al observar la zona del ojo exclusivamente teniendo en cuenta la **luminosidad** de la misma, hay una diferencia entre la que se obtiene cuando el ojo esta abierto y cuando está cerrado. Esto se debe lógicamente al color blanco del **globo ocular** en contraposición al más oscuro de la **pupila**, o directamente la homogeneidad del color más claro de la **piel** al cerrar el ojo y no dejar ver ninguna parte del ojo. Basándose en esto se puede extraer información sobre si el ojo del usuario esta abierto o cerrado.

Para obtener esta información en primer lugar se recortará el frame de la cámara exclusivamente a la zona que engloba el ojo del usuario. Esto puede realizarse de forma sencilla tomando los landmarks respectivos al ojo y creando un nuevo frame limitado a esos puntos.

Una vez recortado el frame se aplicarán una serie de tratamientos de la imagen del mismo, consistentes en un paso a **escala de grises** previo a la **umbralización** de la imagen, de forma que queden diferenciados las zonas más claras y más oscuras.

Teniendo la imagen del ojo del usuario de forma que los píxeles más oscuros se dibujen como negros y los más claros como blancos, podemos hacer una relación

entre los píxeles blancos respecto al número total de píxeles en el frame para obtener el nuevo ratio que usaríamos para determinar si el ojo está abierto o cerrado.

Figura 22: Ojo abierto umbralizado en blanco y negro.

Figura 23: Ojo cerrado umbralizado en blanco y negro.

Como se puede observar en las figuras anteriores (Figuras 22 y 23), la **relación entre la cantidad de píxeles blancos y negros** dentro del frame del ojo varía con el ojo abierto y cerrado. Obteniendo el valor porcentual de píxeles blancos respecto al total de la imagen obtenemos nuestro nuevo “blink_ratio” (Figura 24).

La **detección del pestañeo** mediante este método es también lo suficientemente eficiente para el uso de nuestro programa, obteniendo pocos falsos positivos y se realizan bien los cálculos cuando el ojo esta cerrado o abierto. Sin embargo, se ha dado preferencia a esta metodología sobre la anterior por adaptarse mejor a los diferentes tipos de **luminosidad ambiente**, ya que usando este método se umbraliza la imagen automáticamente en función de la propia luminosidad.

```
blink_ratio_num = np.sum(threshold_eye_blink == 255)
heightT, widthT = threshold_eye_blink.shape
tam_frame = heightT * widthT
blink_ratio = blink_ratio_num / tam_frame
```

Figura 24: Obtención de blink_ratio en relación a los píxeles blancos del frame.

5.2 Eye Tracking

Eye Tracking es como se conoce a las técnicas tecnológicas, ya sean a nivel hardware o software, que pretenden extraer la información de los movimientos oculares de un usuario. Actualmente es una tecnología que se utiliza para mejorar la experiencia de usuario en aplicaciones, mejorar imágenes de marca, etc. Sin embargo, también se puede hacer un uso más directo que eso para este trabajo.

Siendo técnicas ya utilizadas para otros fines, existen diferentes métodos para implementar “Eye Tracking”. Por norma general se recurre a herramientas de rayos infrarrojos apuntados desde el propio monitor al usuario. En su lugar, este trabajo se mantiene dentro del ámbito del tratamiento de imágenes y se emplearán otras técnicas para determinar en tiempo real hacia donde está mirando nuestro usuario.

El principal recurso será tener en cuenta la forma del ojo. Se puede describir el ojo visualmente como una zona particularmente oscura (la pupila) rodeada de una zona mucho más clara (el resto del globo ocular). Teniendo en cuenta esto, es posible implementar en el programa una forma de detectar en que zona dentro del área que se reconoce como el ojo se encuentra la parte más oscura. En función de la posición de la zona más oscura y un valor asociado a dicha posición se determina hacia donde mira el usuario.

Empezando pues con la implementación, el primer paso será **separar la región del ojo** del resto, ya que al buscar la parte más oscura en la imagen, las zonas oscuras del fondo o incluso del mismo usuario como el pelo podrían considerarse como falsos positivos.

Para separar la zona del ojo del resto de la imagen se pueden almacenar todos los puntos de la región del ojo en un array. Alrededor de esta región se dibujará un polígono de forma similar a la vista hasta ahora para identificar el ojo.

Se creará entonces una máscara compuesta de ceros del tamaño del frame original, es decir, del tamaño de la resolución de la cámara. Teniendo esta máscara se realiza una intersección entre la imagen y ella, seleccionando las partes que anteriormente quedaron encerradas en el polígono que dibujado. El resultado de esta intersección será similar al siguiente (Figuras 25 y 26):

```
eye_region = np.array([(facial_landmarks.part(eye_points[0]).x, facial_landmarks.part(eye_points[0]).y),
                       (facial_landmarks.part(eye_points[1]).x, facial_landmarks.part(eye_points[1]).y),
                       (facial_landmarks.part(eye_points[2]).x, facial_landmarks.part(eye_points[2]).y),
                       (facial_landmarks.part(eye_points[3]).x, facial_landmarks.part(eye_points[3]).y),
                       (facial_landmarks.part(eye_points[4]).x, facial_landmarks.part(eye_points[4]).y),
                       (facial_landmarks.part(eye_points[5]).x, facial_landmarks.part(eye_points[5]).y)], np.int32)

height, width, _ = frame.shape
mask = np.zeros((height, width), np.uint8)

cv2.polylines(frame, [eye_region], True, 255, 2)
cv2.fillPoly(mask, [eye_region], 255)
eye = cv2.bitwise_and(gray, gray, mask=mask)

cv2.imshow("Eye", eye)
```

Figura 25: Intersección con la zona del ojo para trabajar solo con ella

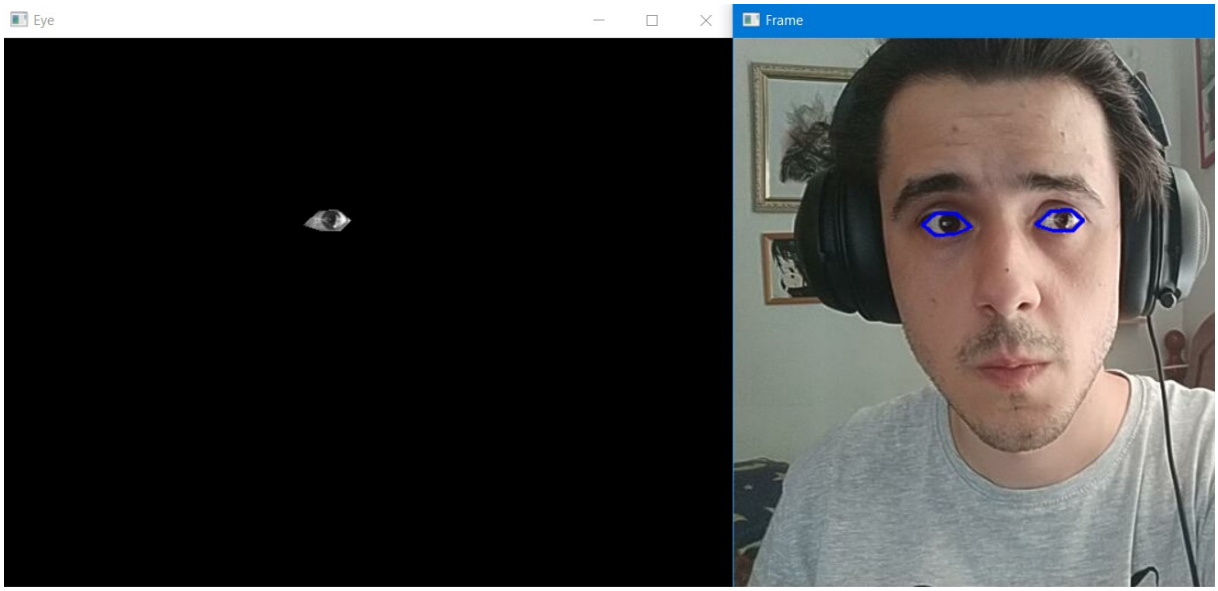

Figura 26: Separada la zona del ojo del resto del frame

Una vez separada la zona del ojo, se puede trabajar en una nueva ventana generada en cada frame que englobe únicamente a la posición del ojo en cada momento. Para ello basta con tomar los valores máximos para x e y dentro del array de dimensiones de la zona del ojo y crear un nuevo frame a partir del anterior, tomando esas dimensiones como las nuevas.

A la imagen obtenida se le aplicará un filtro de **thresholding**. El filtro thresholding recorre la imagen y calcula para cada uno de sus píxeles si supera o no un umbral previamente calculado de forma automática, ya que estamos usando el algoritmo de thresholding **OTSU**, que calcula dicho umbral para cada imagen en función de la luminosidad de la misma. Si el píxel supera el umbral se considera que es una zona oscura y se le asigna el color (0, 0, 0), es decir, el negro. De igual forma, si no se supera el umbral, ese pixel queda dibujado en blanco (Figura 27).

El algoritmo de thresholding OTSU es utilizado cuando el objeto que se quiere extraer de la imagen tiene un número de píxeles elevado en comparación con el tamaño de la imagen completa. En este caso se intenta extraer la pupila del plano que forma el rectángulo alrededor del ojo, por lo que el objeto a detectar constituye una gran parte del plano. Al detectar únicamente la pupila como objeto a detectar y el resto del plano como fondo, el histograma de la imagen será bimodal.

Tal y como se mencionaba anteriormente, OTSU haya el umbral para la umbralización de forma automática. Este umbral se calcula buscando la máxima varianza entre las clases diferenciadas, conocida como varianza intra-clases o varianza residual y determinada por la siguiente fórmula:

$$U_{opt} = arg\,max \frac{P(T)\left[1-P(T)\right]\left[m_d(T)-m_f(T)\right]^2}{P(T)\sigma_d^2(T)+\left[1-P(T)\right]\sigma_f^2(T)}$$

El resultado de aplicar este filtro será entonces una imagen del ojo en blanco y negro, quedando en negro las partes más oscuras, que serán la pupila y las zonas alrededor del ojo, y en blanco el resto del globo ocular (Figura 28). Se obtiene después de esto algo similar a la siguiente demostración:

```
min_x = np.min(eye_region[:, 0])
max_x = np.max(eye_region[:, 0])
min_y = np.min(eye_region[:, 1])
max_y = np.max(eye_region[:, 1])

gray_eye = eye[min_y: max_y, min_x: max_x]
u, threshold_eye_blink = cv2.threshold(gray_eye, 0, 255, cv2.THRESH_OTSU)
```

Figura 27: Aplicación del filtro de threshold al frame recortado del ojo

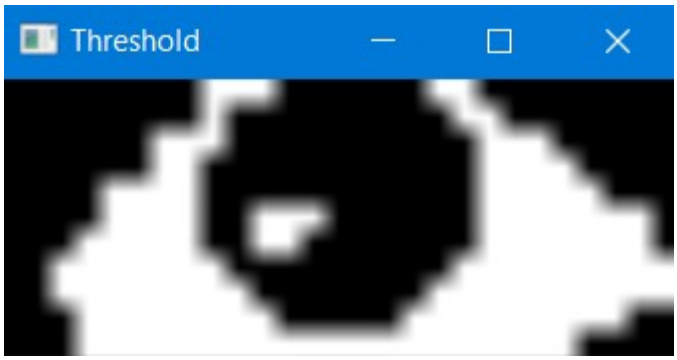

Figura 28: Ojo en blanco y negro tras aplicar thresholding

Es importante que el frame al que se aplica el filtro de thresholding esté en escala de grises, ya que de otra forma no sería igual de preciso. Se puede hacer que cualquier frame o imagen aparezca en escala de grises con la función cvtColor de OpenCV.

En la imagen anterior se muestra un ojo claramente posicionado en el centro. Esto ocasiona una concentración del negro en el centro de la imagen. Se podría decir que por la forma simétrica del ojo, mientras el usuario mira al centro existe una cantidad similar de píxeles negros a izquierda y derecha del frame, y que por la misma razón, si el usuario mira a izquierda o derecha, la relación entre la cantidad de píxeles negros a un lado y otro de la imagen variará. Será entonces en base a esto como se determinará el "**gaze_ratio**", el valor que determinará si el usuario mirando a una dirección u otra en nuestro programa.

Para calcular este ratio en primer lugar se separará el frame anterior en dos mitades simétricas. Una vez separadas, se utilizarán funciones de OpenCV que permiten contar el ratio de pixeles distintos de 0, es decir, distintos de negro. Con este valor calculado para los frames de izquierda y derecha independientemente, se llamará “gaze_ratio” a la división entre los mismos, y será el valor que variará en función de donde miré el usuario y determinará entonces hacia donde está mirando (Figura 29).

```
height, width = threshold_eye.shape
left_side_threshold = threshold_eye[0: height, 0: int(width / 2)]
left_side_white = cv2.countNonZero(left_side_threshold)

right_side_threshold = threshold_eye[0: height, int(width / 2): width]
right_side_white = cv2.countNonZero(right_side_threshold)

gaze_ratio = 1 #Por defecto, centro
if right_side_white != 0:
    gaze_ratio = left_side_white / right_side_white

return gaze_ratio
```

Figura 29: Separación del ojo en dos mitades y cálculo del "gaze_ratio"

Como apunte adicional al cálculo del “gaze_ratio”, se establecerá que será 1 por defecto, el cuál para las pruebas tomadas es un valor dentro de los umbrales de una mirada al centro. También se evitará el cálculo si casualmente la cantidad de píxeles en el lado derecho fuera igual a 0, evitando la división por cero y devolviendo el valor por defecto en tal caso.

Se sabe que normalmente al mirar hacia una dirección ambos ojo se orientan hacia esa misma dirección. Con esto en mente se puede mejorar la precisión de la detección aplicando los mismos métodos a ambos ojos, obteniendo dos “gaze_ratio” diferentes y realizando la media aritmética entre ambos.

Con el “gaze_ratio” final calculado solo queda designar los umbrales que tras una calibración se considerarán como “Izquierda”, “Derecha” y “Centro”. Para cada frame se comprobarán en cual de los umbrales se encuentra el “gaze_ratio” para saber la dirección a la que mira el usuario (Figura 30).

```
gaze_ratio_left_eye = get_gaze_ratio([36, 37, 38, 39, 40, 41], landmarks)
gaze_ratio_right_eye = get_gaze_ratio([42, 43, 44, 45, 46, 47], landmarks)
gaze_ratio = (gaze_ratio_left_eye + gaze_ratio_right_eye) / 2

if gaze_ratio < right_limit:
    cv2.putText(frame, "DERECHA", (50, 100), font, 2, (0, 0, 255), 3)
elif right_limit < gaze_ratio < left_limit:
    cv2.putText(frame, "CENTRO", (50, 100), font, 2, (0, 0, 255), 3)
else:
    cv2.putText(frame, "IZQUIERDA", (50, 100), font, 2, (0, 0, 255), 3)
```

Figura 30: Cálculo de "gaze_ratio" final y dirección de vista del usuario

Tras todo este proceso se verá en tiempo real como el programa determina a qué dirección mira el usuario. En las siguientes imágenes (Figuras 31 y 32) se muestra un ejemplo de funcionamiento, nótese que al ser imágenes tomadas desde una cámara el eje está invertido:

Figura 31: Detección mirada hacia la derecha

Figura 32: Detección mirada hacia la izquierda

La detección de la dirección de la mirada es lo suficientemente precisa como para manejar la dirección del robot. El valor que se mandará a la aplicación encargada del control del robot consistirá en un parámetro de tipo “int”, el cuál valdrá 4 si la mirada está dirigida a la izquierda, 1 si está dirigida al centro o 3 si está dirigida a la derecha.

Al estar ligada a la propia detección del rostro del usuario, su funcionamiento se ve condicionado por la calidad del mismo. Después de todas las pruebas realizadas se ha comprobado que la detección del rostro funciona mejor cuando la cámara se encuentra fija, así que aunque DroidCam nos proporciona la facilidad de operar sin cables y libremente con la cámara de nuestro dispositivo móvil, se usará una plataforma que mantenga la cámara siempre en la misma posición. En el caso de la cámara USB se usará un trípode que mantendrá la cámara fija también.

A continuación se muestra un **diagrama de secuencias** a modo de resumen del tratamiento sobre la imagen captada por la cámara hasta obtener los ratios que determinan la detección del pestañeo y el proceso de eye tracking (Figura 33).

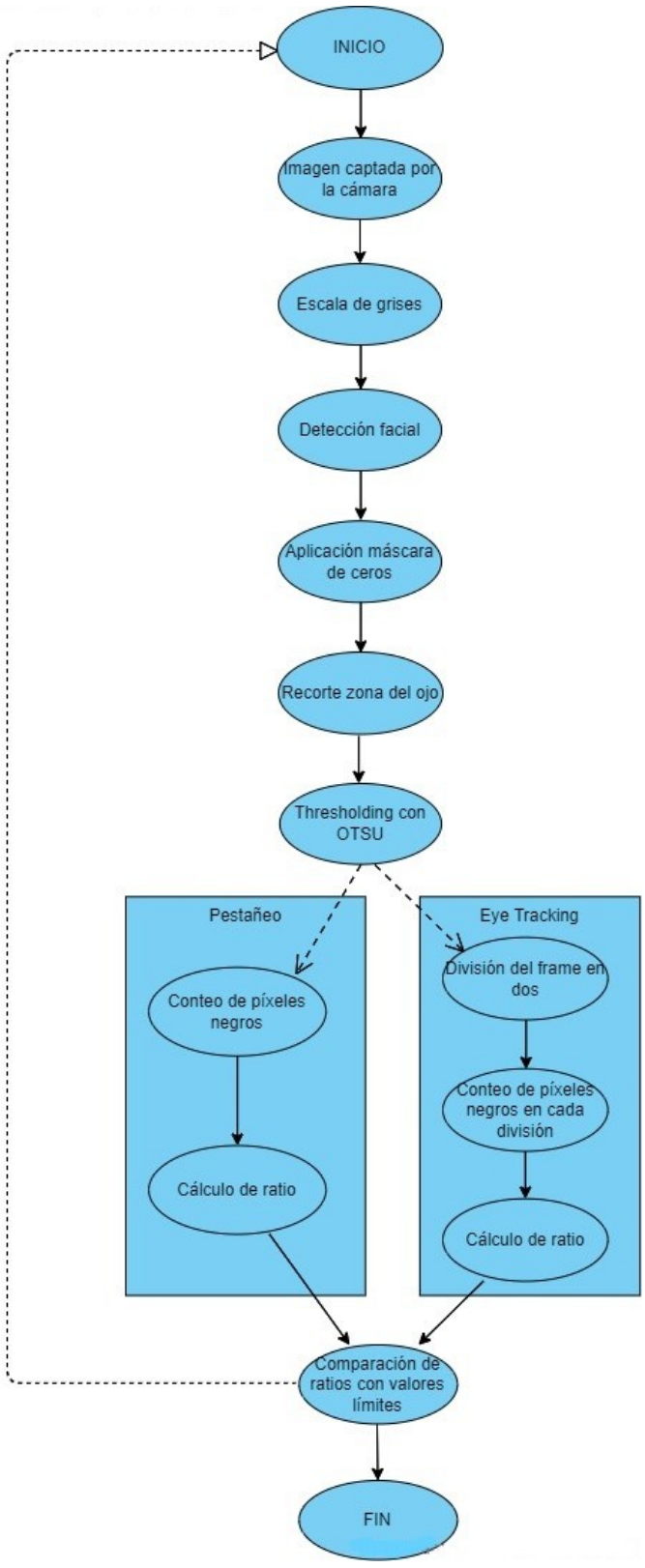

Figura 33: Diagrama de secuencias detección pestañeo y eye tracking

5.3 Calibración

Aunque durante el funcionamiento del programa se va a utilizar un trípode para mantener la cámara fija, no se conoce en una primera instancia en que posición exacta dentro del plano estará situado el usuario. Tampoco se conocen otros parámetros incluso más importantes, los ratios que obtiene el programa para cada situación según donde se tomen las imágenes.

Todos los ratios que se han utilizado para determinar la posición de los ojos o incluso si están cerrados o no son dependientes de parámetros como la distancia del usuario con la cámara, el tamaño de sus propios ojos, etc. Estos parámetros cambian para cada usuario y cada situación, por lo que es necesario un método para hallarlos cada vez que se ejecute el programa.

Para solventar este problema se ha implementado un método de **calibración** que hay que llevar a cabo antes de cada ejecución del programa. Este método obtendrá los **valores máximos y mínimos** para lo que se identifica que un usuario está mirando a la izquierda o a la derecha y para lo que se identifica como unos ojos cerrados y unos abiertos.

En primer lugar se pasa por una etapa de **inicialización de variables**, declarando los valores que se considerarán como nulos para los todos los límites que se tratan de calcular. Al iniciar el programa, se muestra al usuario en pantalla a través de la cámara y aparecerán sucesivamente las instrucciones para el calibrado de la aplicación (Figura 34).

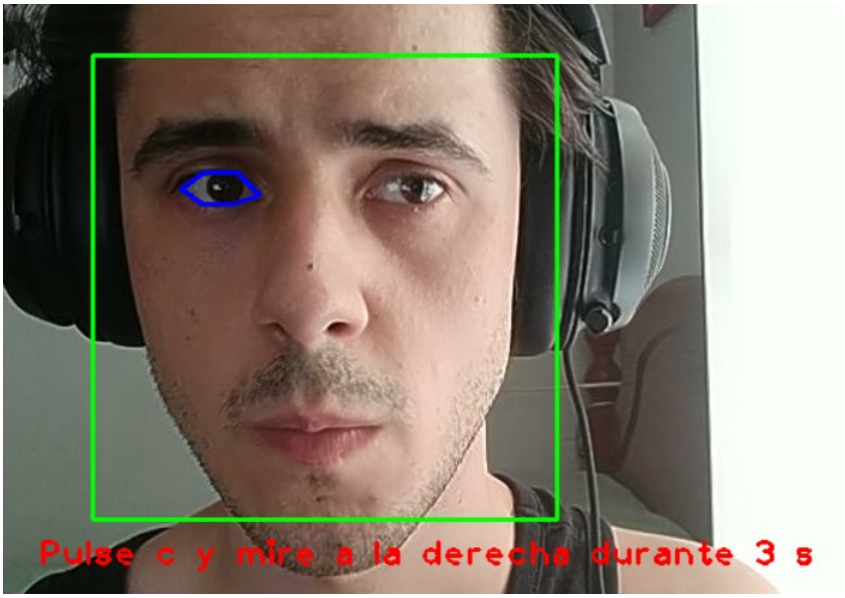

Figura 34: Instrucciones de calibración al inicio del programa.

Se observa la instrucción que pide pulsar la tecla ‘c’ y mantener la vista hacia la derecha durante 3 segundos. Cuando el usuario pulsa la tecla ‘c’, la aplicación toma los valores que se utilizan para determinar la posición del ojo como se ha explicado anteriormente y los almacena como “right_limit” en este caso. Seguidamente, la aplicación pedirá lo mismo para el límite hacia la izquierda o “left_limit” y, por último, pedirá al usuario que tras pulsar la tecla ‘c’ cierre los ojos durante 3 segundos y luego que los vuelva a abrir también, tomando así los límites para determinar si el ojo esta abierto o cerrado, los “blink_limit”, útiles para la detección de los pestañeos.

Tras observar el comportamiento del programa y los ratios obtenidos en pruebas anteriores, se sabe que por norma general el ratio cuando el usuario mira hacia la derecha es menor al obtenido cuando mira hacia la izquierda. Teniendo esto en cuenta, el programa se anticipará a malos usos o errores en la calibración. Si al final del proceso se detecta que el límite de la derecha es mayor que el de la izquierda, se considerará la calibración como nula, retornando todos los valores a sus valores por defecto y volviendo a iniciar el proceso.

Por último, se debe tener en cuenta que los valores obtenidos en un caso habitual de calibrado serán valores cercanos a los máximos posibles para el usuario

en todos los casos. Si se mantienen esos valores como están, la detección de todas las casuísticas podría complicarse, por lo que se realizan unos ajustes a los valores obtenidos para establecer los límites a partir de los cuales se considera que se cumplen los requisitos. Dichos ajustes consisten simplemente en permitir margen de entre un 10% y un 20% respecto de los valores tomados, aplicando los siguientes cálculos (Figura 35).

```
diferencia = left_limit - right_limit
margen_der = diferencia * 0.2
margen_izq = diferencia * 0.1
left_limit = left_limit - margen_izq
right_limit = right_limit + margen_der
margen_blink = (blink_limit_close - blink_limit_open) * 0.15
blink_limit = blink_limit_close - margen_blink
calibration = True
```

Figura 35: Ajustes de márgenes sobre los límites obtenidos.

Tras estos ajustes, la calibración se da por concluida y se podrán utilizar los límites obtenidos durante el resto de la ejecución del programa. Estos valores podrían almacenarse de forma permanente si la aplicación funcionase siempre para un mismo usuario y una cámara fija siempre a la misma distancia del rostro del usuario. De esta forma no tendría que pasar por el proceso de calibración cada vez que iniciara la aplicación.

5.4 Envío de datos al robot móvil

El programa logra una detección en tiempo real sobre hacia donde mira el usuario. Estos datos pueden utilizarse para dar instrucciones al robot móvil de forma sencilla. Para controlar nuestro robot se necesitará otro programa implementado en MakeCode, por lo que también se requiere un método para pasar parámetros desde la aplicación Python a la aplicación en funcionamiento del robot. Esto puede realizarse de forma sencilla vía **Bluetooth**.

Para mandar datos vía Bluetooth desde Python se utilizará la misma metodología que para mandarlos por el puerto serie. Gracias a la librería "**serial**", se creará un objeto "Serial" que dispone de las instrucciones necesarias para estas tareas. En primer lugar se emparejará el dispositivo Bluetooth, el módulo HC-06, con el PC.

Tras emparejarlo, se puede acceder a la configuración y ver el puerto por el que se comunica, en este caso es '**COM10**'. Este dato es necesario para la configuración de la comunicación desde Python, que se realiza de la siguiente manera (Figura 36).

```
print("Start")
port = "COM10"
try:
    bluetooth = serial.Serial(port, 9600)  # Start communications with the bluetooth unit
    time.sleep(2)
    print("Connected")
    bluetooth.flushInput()  # This gives the bluetooth a little kick
except serial.SerialException as e:
    print(e.errno)
```

Figura 36: Inicialización de la comunicación Bluetooth desde Python

Desde el objeto Serial "bluetooth" se tiene acceso a la función **bluetooth.write()** que mandará la cadena de texto que tenga como parámetro al puerto serie que se le indique, en este caso 'COM10', a la **velocidad en baudios** también indicada, en este caso 9600. Estos valores deberán ser los mismos con los que se trabajará en la aplicación para controlar el robot móvil.

Se establecerá la comunicación de forma sencilla, por ejemplo asignando la detección de la vista hacia la derecha, izquierda y centro a unos valores numéricos en específico. La decisión sobre la dirección a la que mira el usuario se realiza mediante la media aritmética de los ratios obtenidos y almacenados en un **buffer** en los últimos frames, aumentando así la precisión de las instrucciones.

En el caso de la detección del parpadeo, debido a que el usuario parpadeará de forma natural sin querer activar o desactivar, se ha implementado el mismo **buffer** de ratios obtenidos que usamos para determinar hacia donde mira el usuario, pero en este caso de mayor tamaño. El resultado de esta implementación será requerir que el usuario **guiñe el ojo durante un par de segundos** para detectar su instrucción. Además se ha implementado un **delay de 2 segundos** entre activaciones, para evitar detectar una única instrucciones como varias.

Por último, si desaparece el rostro del usuario de la grabación se mandará al robot la orden de parar sus motores, por lo que en caso de que la cámara perdiera de vista al usuario o dejará de funcionar la detección facial, el robot se parará inmediatamente.

Podemos resumir la secuencia de la comunicación entre los programas con el siguiente diagrama de secuencias (Figura 37).

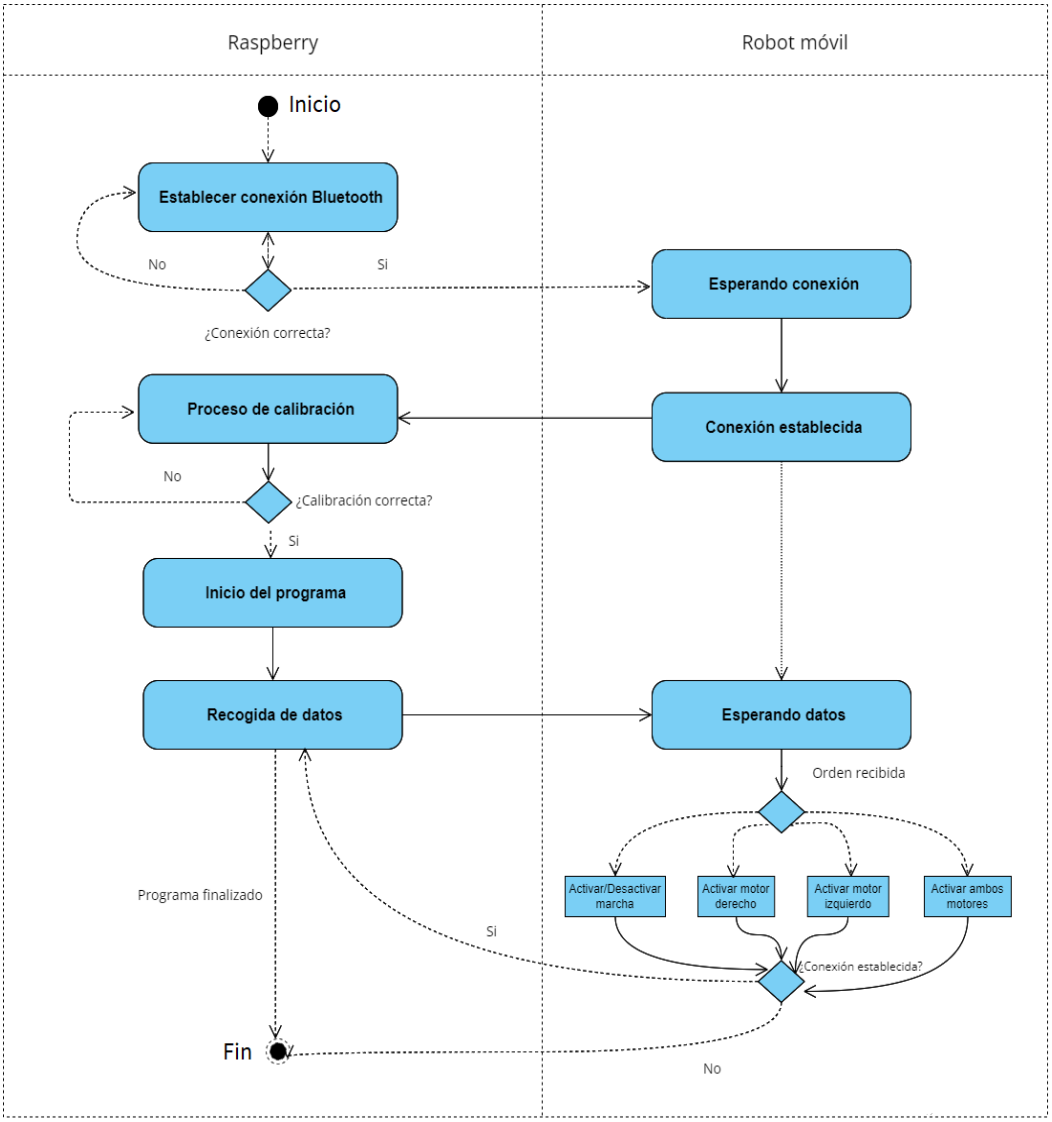

Figura 37: Diagrama de actividades de la conexión entre programas

5.5 Aplicación móvil

Se quiere que nuestra aplicación sea lo más portable posible. Hasta ahora se ha usado un ordenador para ejecutar el programa y un dispositivo móvil que actúa únicamente como cámara, por el simple hecho de tener una mejor calidad de imagen que ayude a hacer el reconocimiento facial más preciso. El hecho de utilizar dos dispositivos, y entre ellos un PC que no es fácilmente manejable ni transportable suponía un problema, por lo que se pensó en trasladar el programa a una aplicación para dispositivos móviles.

El objetivo era conectar el propio dispositivo móvil al robot mediante Bluetooth, y con la aplicación realizando el seguimiento ocular a través de la cámara del dispositivo, enviar las instrucciones correspondientes. Para ello se plantearon diferentes opciones:

5.5.1 “Wrap” de Python a aplicación móvil

En primer lugar se trató de “**wrappear**” el script en Python directamente a una aplicación móvil. “Wrappear” el script en Python consiste en añadir al programa una interfaz con unos requisitos determinados que permitan la navegación y el uso del mismo desde un dispositivo móvil.

Existen varias librerías en Python para traducir el script a un archivo .apk ejecutable en Android. De entre ellas se han utilizado tres: **BeeWare**, **Kivy** y **Chaquopy**. Tanto utilizando BeeWare como Kivy se llegó a la misma problemática, y es que la herramientas de OpenCV y Dlib que permitían el manejo de la cámara y el tratamiento de imágenes presentaban problemas que se han podido solucionar. Sobre Chaquopy se hablará más adelante ya que aunque presentará otros problemas, si que fue la herramienta más estable de entre las que se probaron.

5.5.2 Aplicación móvil con código nativo en C++

Tratando de buscar otros métodos se intentó crear una aplicación desde cero e integrar en la misma el script de detección facial. Para ello se utilizó **Android Studio**, un entorno de desarrollo oficial para la plataforma Android. Desde Android Studio se pueden crear aplicaciones móviles tanto en **Kotlin** como en **Java**. El lenguaje elegido para la aplicación fue **Java**.

Android Studio además ofrece la opción de crear programas con código en el lenguaje **C++ integrado** de forma nativa, ofreciendo funciones implementadas desde un archivo en C++ dentro del proyecto para utilizarlas dentro de nuestro código en Java. Como las librerías OpenCV y Dlib cuentan con versiones para C++, se trató de rehacer el programa en Python también desde cero en estos lenguajes de programación, integrando así las funciones con la aplicación móvil directamente.

Sin embargo la configuración para el correcto funcionamiento de esta integración conllevó muchos problemas de **incompatibilidades entre versiones** de las librerías necesarias. En primer instancia, no era posible integrar las librerías de OpenCV en el código C++ nativo. Los “includes” necesarios de las librerías no podían añadirse al proyecto de una forma tan sencilla al tratarse de código nativo. Se intentó numerosas veces modificar las versiones tanto de Android Studio como de OpenCV a instalar, pero no se llegó una solución real.

Además, la comunicación entre Java y C++ también era problemática dentro del proyecto. El objetivo, al igual que con el programa funcionando en Python, era tratar cada frame detectado por la cámara, detectar si había un rostro en el frame, hacer un seguimiento de los ojos en el rostro y mandar instrucciones a nuestro robot. Al separar la ejecución en dos lenguajes diferentes, se tuvo que separar que parte del programa se encargada de unas tareas y cuáles de otras.

Dado que al final se encontró una forma de implementar OpenCV en Java, la única parte del proyecto que no se podía ejecutar desde Java era la detección facial mediante Dlib. El objetivo de la comunicación entonces era **enviar el frame a cada momento** desde el código en Java a C++, utilizar el detector facial en C++ y devolver a Java un array con los 68 landmarks detectados en forma de puntos (x, y).

Tras varias pruebas y viendo los continuos problemas de incompatibilidades que impedían el uso de librerías externas en C++ como código nativo, se continuó buscando otras opciones para hacer funcionar la aplicación.

5.5.3 Dlib en Java

En este punto se sabía que lo único que no podíamos hacer desde el código Java era la detección facial en **Dlib**, por lo que se trató de buscar una forma de hacerlo funcionar. Se encontraron dos bibliotecas en Github creadas por usuarios que trataban de hacer funcionar Dlib en Java:

https://github.com/radium226/dlib-java

https://github.com/tahaemara/jdlib

Sin embargo, estas bibliotecas tampoco ofrecían una solución real al problema debido a más incompatibilidades, en este caso de **Jdlib** con Windows y de **Dlib-Java** con Android Studio, ya que el script que utilizaba esta librería para la detección de imágenes requería un objeto **BufferedImage** en Java, objeto del paquete **AWT** de Java que **no está soportado en Android Studio**. Android Studio cuenta con objetos de características similares a este, pero las funciones no funcionaban correctamente con estos.

También se intentó buscar método de "parseo" de un tipo de objeto a otro pero tampoco resultó. Con este problema se descartó también la opción de utilizar Dlib en Java.

5.5.4 Aplicación móvil con código en Python

Finalmente se retornó a la primera de las opciones, utilizar un "wrapper" de Python para aplicaciones en Android. En este caso se utilizó **Chaquopy**, que funcionaba de forma diferente a los anteriormente mencionados. Mientras que Kivy y BeeWare eran herramientas que daban interfaz a un script en Python, **Chaquopy se integra directamente en Android Studio** y utiliza la interfaz que se le asigna a la aplicación desde el mismo entorno.

La configuración consiste en incluir las herramientas de Chaquopy en el proyecto de Android e inicializar una instancia de Python que haga referencia al archivo implementado en Python dentro del proyecto. Con este objeto se podrían realizar llamadas a funciones definidas en Python de forma similar a la que se pretendían con la aplicación con código nativo en C++.

La inclusión de librerías con este método fue mucho más sencilla. Al igual que con Python, bastaba con instalar las librerías necesarias mediante el comando "pip install". La diferencia es que estos comandos se integraban dentro de la aplicación en el archivo "build.gradle" a nivel de app de la siguiente manera:

El siguiente problema a solventar fue la **traducción de la imagen** del formato que ofrecía Java dentro de Android Studio a un formato que reconociera las instrucciones de Dlib en Python. Para ello se ha optó por codificar la imagen en Base 64, obteniendo un string que puede ser recompuesto en la implementación en Python (Figura 38):

```
private String getStringImage(Bitmap bitmap) {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    bitmap.compress(Bitmap.CompressFormat.JPEG,  quality: 10, baos);
    byte[] imageBytes = baos.toByteArray();
    String encodedImage = android.util.Base64.encodeToString(imageBytes, Base64.DEFAULT
    return encodedImage;
}
```

Figura 38: Codificación de Bitmap a Base64 (string) desde Java

Teniendo por fin la imagen en Python se podían obtener los landmarks aplicando la función de detección facial de la misma forma que en el script original y devolver los puntos en un array que puede ser reconstruido y utilizado desde Java.

Ahora que por fin se tenían los landmarks calculados a cada frame en nuestro código en Java, solo faltaba operar con ellos, recortar el frame para encuadrar el ojo del usuario y calcular los ratios asociados a la posición de la pupila.

Pero antes de realizar esto, mientras se realizaban pruebas se descubrió el problema que no se ha podido solucionar de ningún modo y es el motivo por el que finalmente no ha sido posible trasladar el programa a una aplicación móvil. Y es que tanto en emuladores de Android en PC como corriendo la aplicación en un dispositivo móvil (Xiaomi Redmi Note 8) se observó que el uso de la cámara a la vez que la aplicación **ralentiza el video** hasta un punto que haría muy complicado el funcionamiento del robot controlado por los ojos.

Se ha tratado de dar solución a este problema de diferentes maneras. En primer lugar se ha reducido lo máximo posible la **resolución** de la imagen captada

por la cámara antes de aplicar el detector facial, pero pese a que mejoró mucho la velocidad de vídeo, sigue sin superar los apenás 3 FPS, insuficientes para el uso de programa. Se ha pensado también en cambiar el método de detección facial, pero un cambio a uno menos preciso como el que ofrece OpenCV sería demasiado irregular para la ejecución del programa, detectando continuamente **falsos positivos** como se comentaba en el estudio de los algoritmos.

Se investigó también respecto a como otras aplicaciones realizaban esta detección facial, ya que nuestra aplicación no es la primera que integra esta funcionalidad en el mercado de la Play Store. Sin embargo, lo que se encontró fueron aplicaciones capaces de detectar rostros en imágenes estáticas, o como mucho aplicaciones que grababan un vídeo de unos segundos para devolver unas estadísticas que determinaban hacía donde estaba mirando el usuario durante el vídeo, proceso insuficiente para la aplicación de este trabajo.

Debido a esto, se abandonó la idea de trasladar la aplicación a dispositivos móviles, ya que se consideró la ralentización de la cámara totalmente incompatible con el funcionamiento de nuestro programa (Figura 39).

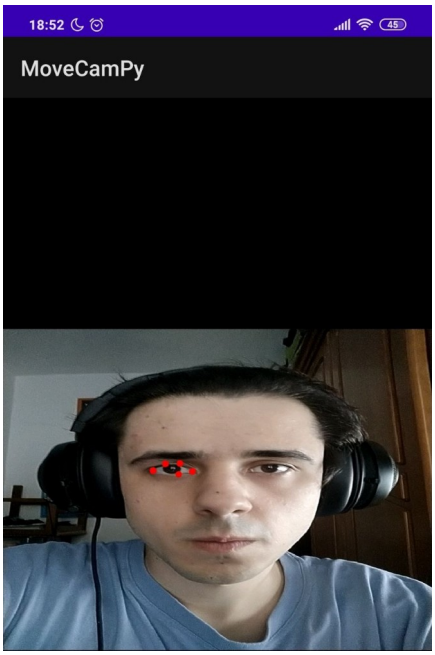

Figura 39: Detección facial funcionando en aplicación móvil

5.6 Aplicación en Raspberry

Tras intentar integrar el programa en una aplicación, se ha optado por implementarla en una **Raspberry**. Las Raspberry incluyen **intérpretes Python** por lo que no se tendrían los mismos problemas de librerías que tuvimos haciendo la aplicación móvil.

En este caso se va a utilizar una **Raspberry Pi 3 Model B+** (Figura 40). Esta Raspberry es una placa computadora que podemos conectar a una pantalla y que incorpora cuatro puertos USB para usar teclado y ratón, además de ser compatible con otros accesorios para incrementar sus funcionalidades. El procesador es un ARM Cortex A53 de cuatro núcleos, tiene una memoria RAM DDR3L de 1 GB y una caché de 2 MB. Además incorpora Bluetooth 4.2 que será útil para la conexión con el robot.

Figura 40: Raspberry Pi 3 Model B+

El uso del programa en esta Raspberry se puede realizar de forma sencilla. Simplemente se guardará el fichero .py en la Raspberry y se instalarán las librerías necesarias de forma similar a la que se instalan en Windows. La ejecución es diferente, ya que para acceder a ciertas instrucciones se requieren permisos de administrador, por lo que el script debe utilizarse mediante el comando “sudo python” y el nombre de nuestro archivo .py.

En cuanto al código del script, requiere unas leves modificaciones. En primer lugar, la instrucción “switch-case” que se utiliza en Windows no está soportada en el sistema operativo que incluye nuestra Raspberry, por lo que se debe cambiar por una secuencia de “if” anidados.

Otra modificación es la conexión al **puerto Bluetooth**. Para conectar la Raspberry por Bluetooth al robot se deberá configurar en primer lugar para ello. Esta configuración consiste simplemente en emparejar la Raspberry con el dispositivo Bluetooth del robot con el comando “pair” y abrir la conexión con el mismo mediante “sudo rfcomm connect hci0” más la dirección física Bluetooth del robot. Esta instrucción se mantiene en ejecución durante el funcionamiento del programa. Por último, el nombre del puerto dentro del programa dejará de ser “COM10” para ser la dirección física del robot.

El último cambio afecta al funcionamiento del programa de forma negativa. Al ejecutar el programa en Raspberry se nota una ralentización similar a la que se sufría con la aplicación móvil. Sin embargo esta es de menor importancia y es solventable aplicando algunas de las modificaciones que se probaron para la aplicación móvil.

El principal cambio ha consistido en hacer **más pequeña la resolución** de las imágenes captadas por nuestra cámara. Reajustando el tamaño de la ventana a 320 x 240 px, los FPS obtenidos con nuestra cámara en ejecución siguen sin ser todo lo fluido que podrían, pero si que se obtiene una **velocidad estable** que permite usar el programa sin muchas complicaciones. El cambio de resolución no afecta a la detección facial siempre que no sean valores extremadamente bajos, que no es el caso.

Finalmente, como la Raspberry no incorpora una cámara, utilizamos una **cámara USB** apoyada ya sea en un trípode o en un brazo móvil que permite más movilidad incluso (Figura 41).

Figura 41: Cámara USB utilizada desde Raspberry.

Una vez esté todo conectado, se puede observar que el funcionamiento del programa en Raspberry es similar al resultado que se obtenía en PC. Aún sufriendo la ralentización que hemos comentado anteriormente, la detección facial sigue siendo constante y permite mandar las instrucciones al robot de forma estable. La velocidad del robot no se ve afectada, aunque si que es posible notar un ligero retraso en la detección del cambio de órdenes.

La **calibración** por otra parte también sufre de esta ralentización, y es posible que no se tomen valores correctos desde el principio. Esto puede desembocar en manejos mas complicados del robot, ya que los valores límite de izquierda y derecha pueden estar más cercanos de lo que deberían y hacer muy difícil detectar cuando el usuario quiere ir recto (ojos mirando al centro).

6. Robot móvil

Para este trabajo, desde el departamento de electrónica se ha prestado un **robot móvil Maqueen Plus** (Figura 42) para micro:bit que dispone de todo lo necesario para recibir las instrucciones que se mandarán mediante inspección ocular. Entre los objetivos del trabajo encontramos el estudio de este robot, sus componentes, su comportamiento y la forma de manejarlo con las instrucciones que se obtengan mediante la inspección ocular del usuario.

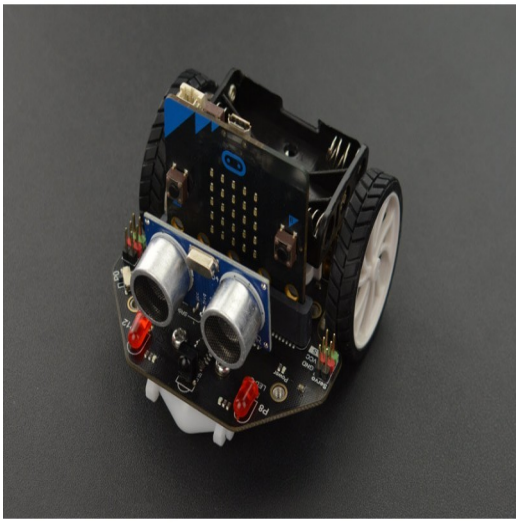

Figura 42: Robot móvil Maqueen Plus utilizado para el proyecto

Como se comentaba al inicio del proyecto, el contexto del mismo trata de dar solución a la pérdida de movilidad en las manos de la gente que padece ELA y ayudarles a mover su propia silla eléctrica mediante inspección ocular. Extrapolando esa utilidad a este robot, se tratará de hacer que el control sea lo más cómodo posible, ajustando parámetros como la velocidad o la velocidad de giro. Dichos valores podrían configurarse de forma distinta si el proyecto estuviera realizándose con una silla eléctrica y usuarios reales.

6.1 Componentes del robot

El robot está compuesto por diferentes componentes, tanto componentes incluidos en el propio chasis del robot como módulos externos que se detallan a continuación.

6.1.1 Chasis Robot Maqueen PLUS

El **robot Maqueen PLUS** incorpora todo lo necesario para las pruebas a realizar en este proyecto, además de otras muchas funcionalidades.

En este trabajo solo se utilizarán los motores que mueven sus dos ruedas para las pruebas de traducción de inspección ocular a movimiento. Sin embargo, este robot contiene muchas otras funcionalidades, tales como sensores para detectar y seguir líneas dibujadas en el suelo, LEDs RGB, sensores de infrarrojos, diferentes puertos de expansiones y hasta un zumbador para reproducir sonidos.

Aunque no es el objetivo de este trabajo, el robot Maqueen puede usarse también como método de enseñanza para niños. Combinado con la placa Micro:Bit V2 que puede programarse de forma sencilla a través de MakeCode, una web de programación por bloques, puede ser un kit de enseñanza muy interesante para los primeros pasos en programación para niños.

6.1.2 Placa Micro:Bit V2

El **placa Micro:Bit V2** (Figura 43) es un pequeño microprocesador ARM programable con varios sensores para medir luz, temperatura o aceleración, 25 leds programables, radio y Bluetooth, pins para conexiones externas, un par de botones programables e interfaz USB.

Esta placa será la encargada de almacenar y ejecutar el programa controlador del robot Maqueen. Leerá las instrucciones recibidas en el puerto serie y las traducirá a los movimientos o acciones correspondientes.

Del mismo modo que el robot Maqueen y trabajando de forma conjunta, el control del robot es solo uno de los posibles usos de esta placa, la cuál también se usa para introducir a niños en la programación de forma sencilla, además de otros muchos usos posibles.

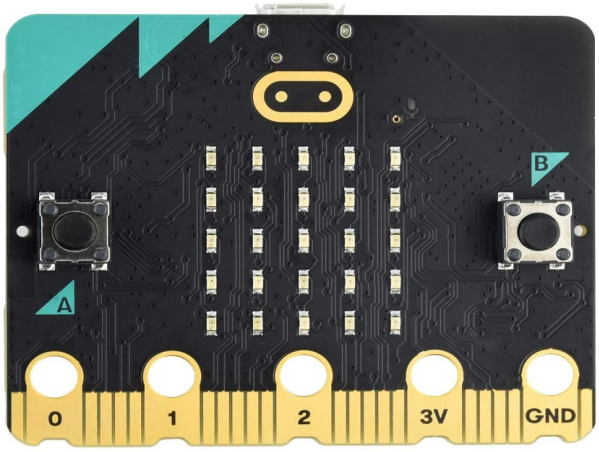

Figura 43: Placa Micro:Bit V2

6.1.2 Módulo Bluetooth HC-06

El **módulo Bluetooth HC-06** (Figura 44) consiste en un módulo que puede conectarse directamente al robot Maqueen y le proporciona la posibilidad de recibir comando vía Bluetooth. Es la versión mejorada del módulo HC-05, diferenciándose principalmente en su número de pines.

Aunque la placa Micro:bit dispone de conexión Bluetooth, la conexión directamente a este módulo es más sencilla de configurar, ya que la placa Micro:bit no puede conectarse de forma sencilla a un PC o a una Raspberry de la misma forma que a un dispositivo móvil, cuya conexión se realiza a través de la misma aplicación de Micro:Bit.

La conexión del módulo al robot Maqueen se puede hacer de forma sencilla a los pines que incorpora el robot, conectando los pines 0 y 1, el voltaje y la toma a tierra.

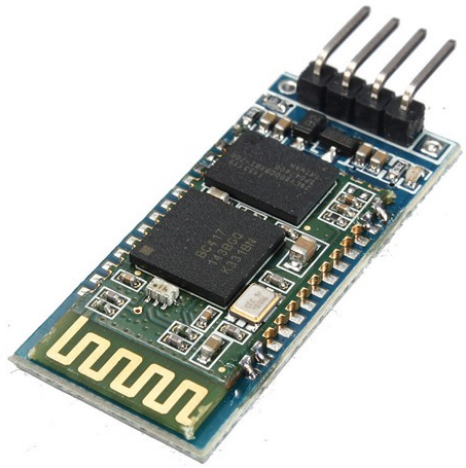

Figura 44: Módulo Bluetooth HC-06

6.2 Funcionamiento del robot

Para programar las placas Micro:Bit se puede utilizar el entorno de desarrollo web **MakeCode**. MakeCode ofrece una programación sencilla mediante bloques. En el caso del funcionamiento de nuestro robot será suficiente ya que su trabajo consistirá principalmente en recibir órdenes y activar o desactivar los motores en consecuencia. A continuación se explica el funcionamiento para nuestro robot y los ajustes que han sido necesarios para implementarlo.

En primer lugar dentro del entorno MakeCode se debe instalar la extensión Maqueen para el robot, ya que no viene instalada por defecto en el entorno. Esto se puede realizar de forma sencilla en el menú de extensiones del entorno. Una vez instalada se tendrá acceso las funciones para controlar el robot desde la placa Micro:Bit.

Por defecto MakeCode cuenta con dos bloques donde añadir el resto de funciones: “**on start**” y “**forever**”, siendo “on start” donde se las variables necesarias y “forever” el programa que se ejecutará continuamente mientras la placa esté encendida (Figura 45).

Figura 45: Programa robot Maqueen en placa Micro:Bit

La inicialización consistirá en abrir el puerto serie redirigido a los pines a los que está conectado nuestro módulo Bluetooth, P0 y P1 en este caso. También se establecerá el "baud rate" en 9600, igual que en el programa en Python. Otra parte de la inicialización es almacenar continuamente el dato recibido por Bluetooth en una variable "**order**".

En el bloque "forever" simplemente se comprobará cuál es el valor recibido almacenado en "order" y se activarán unos motores u otros en función del mismo. En este caso simplemente se han asociado los parámetros del 1 al 4 a diferentes direcciones (1 para seguir recto, 2 para ir marcha atrás, 3 para girar a la izquierda y 4 para girar a la derecha) y el valor 0 para hacer que el robot se pare.

Desde MakeCode descargaremos el programa de forma sencilla en **fichero .hex**. Para instalar el programa en la placa Micro:Bit basta con enviar el archivo a la placa conectada al USB. El programa se instalará automáticamente.

7. Ejemplos de uso

Ahora que todo está configurado, se plantearán diferentes casuísticas de uso para el programa, y se observará como afectan algunas condiciones externas a su funcionamiento.

En primer lugar se debe tener en cuenta que el correcto funcionamiento del programa viene determinado en gran medida por una **buena calibración**. El programa considerará como errónea una calibración en la que el valor límite de mirada hacia la derecha sea mayor al de mirada hacia la izquierda. Sin embargo, existen otros resultados de la calibración que también pueden conllevar errores, como valores **demasiado próximos** que hagan el margen de mirar hacia el frente muy pequeño, o valores límite de un lado u otro **demasiado altos** que dificulte la detección de la mirada hacia esos lados.

Con esto presente, es posible que sea necesario repetir el proceso de calibración hasta dar con una **calibración adecuada**. Una vez obtenida una calibración correcta, el uso del programa puede verse afectado por factores externos. El factor que más se debe tener en cuenta es la **luminosidad**, que puede afectar a la calidad de la imagen captada por las cámaras. Una luminosidad mayor o menor puede estar determinada por la iluminación dentro de una estancia o simplemente por el momento del día en el que se encuentra el usuario, habiendo menos luz a la noche de forma generalizada.

A continuación se analizará el comportamiento del programa en distintas situaciones, teniendo en cuenta si el usuario está en interior o exterior y si es de día o de noche.

7.1 Programa en interior y de día

En primer lugar se estudiarán las condiciones óptimas en las que no se debería encontrar ningún factor externo afectando a nuestro programa, utilizándolo desde una **habitación bien iluminada durante el día**.

Como ya se ha comentado, sin tener mayor problemática que la de encontrar unos valores de calibración que realmente se adecúen bien a movimiento de los ojos real del usuario, el programa funciona correctamente.

Se debe tener en cuenta a la hora de la calibración ciertos factores que pueden afectar a la calibración:

- El usuario debe mantener la cabeza lo **mas estática posible**. Si se altera la posición de los ojos dentro de la grabación podrían verse alterados los ratios recogidos por el detector facial y esto alterar la ejecución del programa.
- Cuando se pide mirar a izquierda y derecha se espera que el usuario mueva **únicamente los ojos** y no la cabeza, es decir, no queremos que el usuario gire todo el cuello. Nuestro detector facial funciona mejor con la mirada al frente, girar la cabeza normalmente conllevará dejar de detectar la cara del usuario y detendrá el robot o, en el caso de la calibración, obtendrá valores nulos.
- Cuando se pide que el usuario cierre los ojos, **no debe cerrarlos con fuerza**, sino con fuerza natural. Cerrarlos de forma forzosa suele hacer que el usuario frunza el ceño y puede resultar en valores erróneos de calibración.

De cualquier modo, estas advertencias sirven tanto para esta casuística como para el resto que veremos, ya que forman parte del uso correcto de la aplicación. Cumpliendo estos patrones la aplicación funcionará correctamente con la cámara recogiendo las imágenes de forma nítida.

7.2 Programa en interior y de noche

Para esta prueba se ha probado el programa de noche en una **habitación iluminada muy levemente** por la pantalla donde se estaba ejecutando el programa.

Los resultados han sido los esperables, y es que mientras más oscuras sean las imágenes captadas más difícil será para el programa reconocer un rostro dentro de la imagen.

Aún así, cabe destacar que dentro de la poca luminosidad de la habitación, si nos acercábamos un poco a la pantalla encendida, la cuál emitía muy poca luz, el programa empezaba a funcionar de forma bastante correcta, por lo que se puede deducir que **mientras haya un mínimo de luz** la detección se efectúa de forma muy notable.

7.3 Programa en exterior y de día

El resultado en condiciones normales en el **exterior durante el día** es el esperable existiendo la luz adecuada. De la misma forma que en una habitación iluminada las imágenes captadas por la cámara son lo suficientemente nítidas como para realizar la detección facial, durante el día y en condiciones normales habrá suficiente luz para captar el rostro del usuario y ejecutar el programa con normalidad.

Un factor que podría afectar en estas condiciones sería los días especialmente soleados, o incluso dependiendo de la posición de la cámara, los **reflejos** que pueda causar en ella el Sol. Si la cámara pierde la nitidez a la hora de grabar la cara del usuario podría causar fallos en el programa. En cualquier caso, la forma en la que se detecta la imagen y la calidad de la misma, así como la forma de gestionar problemas como el brillo excesivo en este caso, dependerán de la cámara utilizada y podría mejorarse con una mejora de hardware.

7.4 Programa en exterior y de noche

Finalmente se planteó la situación más adversa, el funcionamiento del programa **de noche y en zonas poco iluminadas en el exterior**. Los resultados fueron muy negativos debido a la poca luz.

Para plantear el caso más extremo posible se han realizado pruebas en el patio interior de un domicilio, evitando así zonas exteriores alumbradas por las farolas de las calles. Siendo también conscientes de que en el uso normal del programa el usuario transitará por zonas alumbradas en una ciudad, se han simulado también diferentes grados de iluminación ambiente.

Los resultados han sido similares a los obtenidos en una habitación oscura por la noche. Como se comentaba en ese caso, la detección del rostro será más difícil conforme baja la iluminación. A continuación se muestran diferentes grados de iluminación y su desempeño (Figuras 46 y 47):

Figura 46: Mínima iluminación para la detección facial

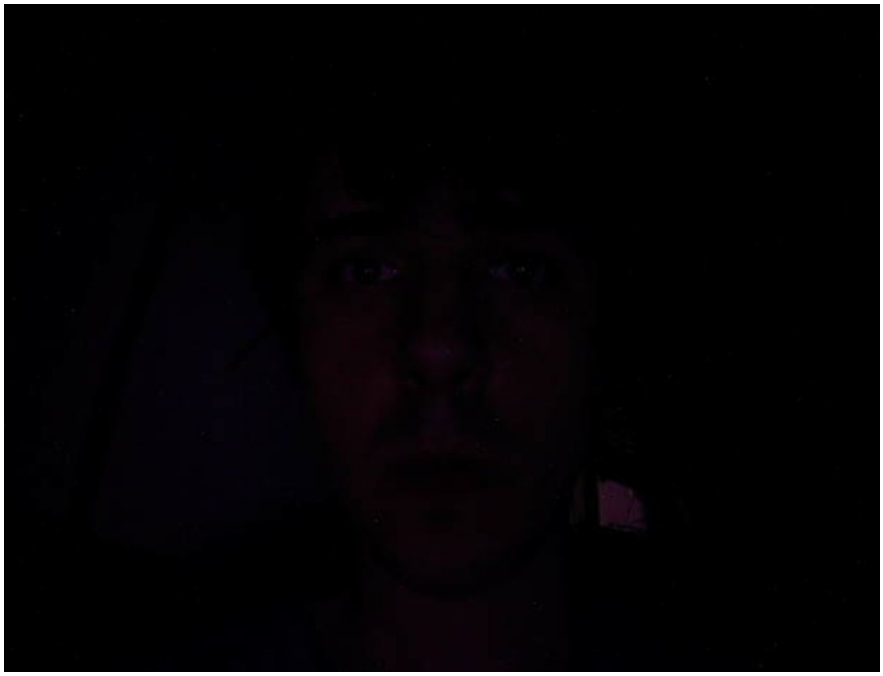

Figura 47: Iluminación insuficiente para la detección facial

La primera imagen fue tomada con una **luz encendida de fondo** en una habitación adyacente al patio, mientras que la segunda únicamente contaba con la **iluminación de la pantalla de mi ordenador** portátil de fondo.

Aunque en la primera imagen vemos como el programa es capaz de detectar el rostro del usuario, la detección deja de ser tan fluida, perdiendo el rostro durante varios segundos y causando que el robot **interrumpa la marcha continuamente**. Extrapolando el programa a un caso real de uso de silla eléctrica, estas interrupciones podrían ser muy incómodas para el usuario. En la segunda imagen vemos como llega un punto en el que aunque en la imagen se distinga levemente al usuario, no es capaz de detectar su cara en el programa, por lo que no funcionará.

En condiciones normales, las calles de un pueblo o ciudad **estarán iluminadas** con mayor intensidad de la simulada en este caso gracias a las farolas del municipio. Aunque se debe tener en cuenta que aún existen zonas que pueden estar mejor iluminadas que otras, por lo que el programa podría llegar a fallar en estos casos.

Una posible solución a este problema sería incorporar un **luz flash** desde el objetivo de la cámara, aunque habría que estudiar la intensidad de la misma necesaria para el correcto funcionamiento y una experiencia de uso agradable para nuestro usuario.

7.5 Video demostración

A continuación se muestra un pequeño **vídeo demostración** de la aplicación funcionando. En el vídeo vemos el robot móvil Maqueen mantenido estático para mejor visualización de su funcionamiento que se marca gracias a los LEDs, a la vez que se muestra como se realiza la calibración antes de ejecutar el programa y sucesivamente el programa en funcionamiento:

Vídeo: https://www.youtube.com/watch?v=3hf2BbTmm7Y

8. Conclusiones

Este trabajo se ha planteado como un supuesto de un uso más avanzado y complejo del mismo, como es el control de una silla eléctrica manejada por su propio usuario. La extrapolación del mismo a la aplicación real conllevaría muchos ajustes, tanto a nivel de software como de hardware. En primer lugar, deberían incluirse una configuración personalizada más allá de la calibración inicial, ya que cada usuario dentro de sus capacidades haría un uso distinto de la herramienta.

Sin embargo, y aún teniendo en cuenta que el proyecto queda lejos del objetivo real por motivos tanto de tiempo como económicos, trabajar en este supuesto me ha hecho ver de primera mano el amplísimo rango de aplicaciones que puede tener la informática.

Durante la investigación de métodos a utilizar para el funcionamiento del proyecto he descubierto que la visión por computación es un campo en el que se ha ido evolucionando mucho en estos últimos años, que si bien no es algo reciente, los últimos avances no distan mucho de la fecha actual, y que probablemente aún haya mucho espacio para la mejora.

Una mejora en el campo puede abrir la puerta a nuevas aplicaciones o a optimizaciones de las ya conocidas, o incluso a la normalización y el acceso de estas herramientas al público general, que actualmente están en fases de experimentación o directamente alcanzan precios más elevados de los que podrían permitirse de forma normal una familia media.

Además de la visión por computación, trabajar en este proyecto y anteriormente en la asignatura de Inteligencia Ambiental me ha enriquecido mucho, haciéndome ver de una forma más tangible lo que podemos conseguir mediante la programación y la gran cantidad de herramientas de las disponemos actualmente: sensores de todo tipo, robots autónomos, comunicaciones de todo tipo, etc.

Comprenden un área de la informática en la que me gustaría trabajar más en profundidad, por su gran capacidad de dar soluciones a problemas reales del día a día.

Creo que la informática estará integrada en el futuro, mucho más de lo que ya está actualmente, en gran parte por funcionalidades de este tipo que harán la vida más sencilla a todo el mundo, ya sea a nivel de usuario como a nivel de empresa, optimizando procesos industriales gracias a los avances tecnológicos.

9. Webgrafía

Paul Viola & Michael Jones (2001). “Rapid object detection using a boosted cascade of simple features”

https://ieeexplore.ieee.org/document/990517

Tallk, tecnología para que enfermos de ELA puedan hablar con la mirada

https://www.tecnobility.com/es/noticia/tallk-tecnologia-para-que-enfermos-de-ela-puedan-hablar-con-la-mirada

GazeSpeak, nueva aplicación móvil para personas con ELA

https://adelaweb.org/gazespeak-nueva-aplicacion-movil-para-personas-con-ela/

El dispositivo Stentrode permite el control de dispositivos por parte de pacientes con ELA

https://www.fundela.es/documentacion/publicaciones/general/el-dispositivo-stentrode-permite-el-control-de-dispositivos-por-parte-de-pacientes-con-ela/

Stephen Hawking, “An exceptional man”

https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1123440/

Stephen Hawking dice haber resuelto la "paradoja de la pérdida de información"

https://www.sopitas.com/geek/stephen-hawking-dice-haber-resuelto-la-paradoja-de-la-perdida-de-informacion/

OpenCV-Python: Procesamiento de imágenes IV en OpenCV. 16 Suavizado de imagen.

https://www.codetd.com/es/article/12713150

Detección de rostros con Haar Cascade Python – OpenCV

https://omes-va.com/deteccion-de-rostros-con-haar-cascades-python-opencv/

Detección de rostros, caras y ojos con Haar Cascad.

https://unipython.com/deteccion-rostros-caras-ojos-haar-cascad/

Detección facial con OpenCV y Python (Haar Cascad).

https://github.com/opencv/opencv/tree/master/data/haarcascades

Eye motion tracking – OpenCV with Python

https://pysource.com/2019/01/04/eye-motion-tracking-opencv-with-python/

OpenCV face recognition

https://pyimagesearch.com/2018/09/24/opencv-face-recognition/

Eye detection. Gaze controlled keyboard with Python and OpenCV

https://pysource.com/2019/01/07/eye-detection-gaze-controlled-keyboard-with-python-and-opencv-p-1/

Esclerosis lateral amiotrófica (ELA)

https://medlineplus.gov/spanish/ency/article/000688.htm

Asociación Española de Esclerosis Lateral Amiotrófica.

https://adelaweb.org/la-ela/la-enfermedad/

Librería Maqueen Arduino

https://github.com/kd8bxp/micro-Maqueen-Arduino-Library