Trabajo Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Desarrollo de sistema biométrico a partir de la anatomía vascular del dedo índice.

Autor: Francisco José Garrido Flores
Tutores: Manuel Ángel Perales Esteve y María del Mar Elena Pérez

Dpto. de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Proyecto Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Desarrollo de sistema biométrico a partir de la anatomía vascular del dedo índice.

Autor:
Francisco José Garrido Flores

Tutores:
Manuel Ángel Perales Esteve
Profesor Titular de Universidad
María del Mar Elena Pérez
Profesor Contratado Doctor

Dpto. de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2021

Proyecto Fin de Grado: Desarrollo de sistema biométrico a partir de la anatomía vascular del dedo índice.

Autor: Francisco José Garrido Flores

Tutores: Manuel Ángel Perales Esteve
María del Mar Elena Pérez

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2021

El Secretario del Tribunal

A la gente que creyó en mí

Agradecimientos

La ingeniería no ha sido precisamente un camino de rosas. El trayecto ha estado plagado de baches y obstáculos que han hecho que en múltiples ocasiones me cuestione el fruto de mi trabajo. No obstante, poder superar estos reveses es parte de ser un buen ingeniero. El grado me ha enseñado a no rendirme nunca y a trabajar sin descanso disfrutando de ello. Es por eso por lo que quiero, desde aquí, agradecer profundamente la dedicación que los profesores han tenido por ofrecernos una docencia que nos ha ayudado a todos los alumnos a asimilar los conceptos necesarios para convertirnos en futuros ingenieros.

Por supuesto, no habría sido capaz de seguir adelante si no fuera por mi familia, la que siempre ha creído en mí más que nadie, motivándome siempre a seguir adelante y siendo siempre positivos. Desde aquí doy las gracias a mi familia, especialmente a mi madre, que ha estado siempre a mi lado, apoyándome de todas las maneras posibles. A todos, os quiero.

Tampoco hubiera sido lo mismo si no fuera por mis maravillosos amigos, los cuales siempre me han apoyado, han creído en mí y me ayudan continuamente a desconectar pacíficamente del trabajo. Sin vosotros no hubiera sido lo mismo. Gracias Isaac. Gracias Valentín. Gracias Sergio. Gracias Tomás. Gracias David(es).

Por último, quiero mandar un enorme agradecimiento a mi querida novia. Gracias por estar siempre ahí apoyándome e intentando ayudarme en mi trabajo (aunque no te deje). Por todo esto y mucho más; muchas gracias, Ángela. Te quiero.

Francisco José Garrido Flores

Sevilla, 2021

Resumen

Hoy en día, es algo muy habitual y estandarizado verificar la identidad de alguien mediante medidas biométricas. El ejemplo más evidente son los teléfonos móviles inteligentes, los cuales (en su mayoría) incluyen un lector de huellas dactilares, para facilitar el desbloqueo del dispositivo al usuario.

No obstante, existen otros tipos de medidas biométricas para identificar unívocamente a una persona, como puede ser el patrón de la retina o el patrón de venas y arterias que se encuentra debajo de un dedo de la mano. A este último caso lo denominaremos anatomía vascular del dedo y es el objeto de estudio de este trabajo académico.

Esta tecnología se encuentra en auge, y en diversas partes del mundo ya se utilizan lectores biométricos de este tipo para la identificación segura de personas en cajeros automáticos. El mayor exponente de este caso es Japón. Se utiliza este tipo de medida biométrica porque es mucho más segura que las medidas descritas anteriormente, ya que además, es muy difícil engañar al lector, algo que no sucede en los casos anteriores, siendo la biometría vascular la seguridad que devuelve menas falsas aceptaciones.

Abstract

Nowadays, verifying someone's identity by biometric measures is something quite common and standardized. The most obvious example is smartphones. Most of them already include a fingerprint sensor to ease the unlocking of the device to the user.

Nevertheless, more types of biometric measures that can identify a person unmistakably do exist. Examples are retina scan, or the pattern of the veins and arteries placed under the finger of a hand. This last case is called vascular anatomy of the finger and is an object of study in this academic work.

This technology is currently rising, and biometric readers of this type can be found in some countries. These are used to identify a person securely in ATMs. The greatest exponent of this case is Japan. This kind of measure is used because it is safer than the other types of measures described in this abstract. In addition, it is rather difficult to deceive this sensor, something that does not happen in the previous cases; being vascular biometry the security that return less false acceptances.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

FAR	False Acceptance Ratio
FRR	False Rejection Ratio
API	Application Programming Interface
SIFT	Scale Invariant Feature Transform
SURF	Speeded-Up Robust Features
ORB	Oriented FAST and Rotated BRIEF
NIR	Near InfraRed
CLAHE	Contrast Limited Adaptive Histogram Equalization
LED	Light-Emitting Diode
PCB	Printed Circuit Board
IR	Infrarrojo/Infrared
RAM	Random Access Memory
SBC	Single Board Computer
RP4	*Raspberry Pi 4*
ARM	Advanced RISC Machine
GPU	Graphics Processing Unit
SoC	System on Chip
ABS	Acrylonitrile Butadiene Styrene
IC	Integrated Circuit
IDE	Integrated Development Environment
CGF	Circular Gabor Filter
OS	Operative System
FLANN	Fast Library for Approximate Nearest Neighbors
DB	Database
SQL	Structured Query Language
UI	User Interface
ACK	Acknowledge

1 INTRODUCCIÓN

Sólo hay un bien: El conocimiento. Sólo hay un mal: La ignorancia.

Sócrates, 470 AC - 399 AC

En este primer capítulo se realizará una introducción al proyecto con el fin de colocar al lector en contexto y se expondrán los motivos por los cuales el trabajo fue elegido. Además, se describirán ligeramente los problemas a los que nos enfrentaremos y las soluciones y caminos tomados.

1.1. Contexto

Hoy en día, todo el mundo tiene en su teléfono móvil inteligente un sensor de huellas dactilares. Esto es muy conveniente ya que nos ahorra una cantidad de tiempo significativa en el acceso al dispositivo. El pensamiento generalizado es, además, que es un sistema muy seguro, ya que sólo debería funcionar con la huella dactilar del propietario. Realmente, esto no es exactamente así.

Como cualquier sistema de seguridad, las medidas biométricas no son completamente impenetrables, y en concreto, la huella dactilar es de las menos seguras. Esto es así porque los sensores de huella dactilar tienen una ratio conocida como ratio de falsas aceptaciones (FAR de ahora en adelante) más alto que en otros sistemas de identificación biométrico. Esto quiere decir que existe una probabilidad de que cualquier persona distinta a la autorizada consiga entrar en el dispositivo sin permiso. Es por ello por lo que la huella dactilar no es utilizada en aplicaciones de muy alta seguridad.

En caso de necesitar una muy alta seguridad, y por tanto, un bajo FAR; recurrimos a otras medidas biométricas como pueden ser el escáner de retina, de anatomía vascular ocular, de la palma de la mano o dedos. De estas medidas de seguridad, las que utilizan la anatomía vascular de una persona son considerablemente más seguras que la huella dactilar o incluso el escáner de retina. Este último ofrece una buena seguridad, pero como contrapartida, su costo es extremadamente elevado con respecto a los escáneres.

Por lo tanto, llegamos así a la conclusión de que se puede realizar un sistema muy seguro con un costo bajo y es por ello; que en este trabajo académico hemos optado por realizar un escáner que sea capaz de identificar a una persona mediante la anatomía vascular de su dedo índice.

1.2. Descripción del problema

Se plantea un problema a la hora de identificar personas de manera fiable. Actualmente, la forma más extendida de esto es la identificación mediante huella dactilar. Sin embargo, como se ha descrito anteriormente, esta medida no es completamente fiable e incluso a través de una simple foto es posible obtener las huellas dactilares de una persona sin su consentimiento. Es por esto por lo que se plantea recurrir a un sistema más fiable y seguro, como puede ser la identificación a través de la anatomía vascular del dedo índice de una persona.

Este tipo de sensores, ya existen y se pueden encontrar en el mercado. Sin embargo, el coste de estos dispositivos los hace muy poco competentes en el mercado y por lo tanto, son usados en muy pocas aplicaciones. Es por ello, que en la propuesta de proyecto se plantea conseguir un sensor de este tipo a un precio competitivo aunque con características reducidas.

Por lo tanto, se propone como solución un sistema simplificado que consta de una cámara sin filtro infrarrojo, iluminación LED en el espectro infrarrojo cercano (de ahora en adelante, NIR), un ordenador para procesar las imágenes y una carcasa impresa en 3D para mantener todo en su lugar.

1.3. Organización de la memoria

La memoria consta de diversos apartados que engloban individualmente una parte concreta del trabajo, a excepción de este apartado de introducción, cuyo único objetivo es explicar ligeramente el proyecto y poner en contexto al lector.

En el apartado dos, se explicará todo lo relacionado al diseño del *hardware* del sistema, tales como la elección de iluminación, elección de ordenador a bordo, material para la carcasa y diseño de piezas y electrónica del sistema.

En el apartado tres, en contraposición al anterior, se explicará todo el diseño *software* del programa encargado de ejecutar los algoritmos de procesamiento de imagen, manejo de la base de datos y, por tanto de identificación y registro de usuarios en última instancia. Por lo tanto, en este apartado, se excluye deliberadamente el desarrollo de la interfaz de usuario.

La interfaz de usuario se explicará en su propio apartado, el apartado cuatro. En este punto, se detallará cómo se ha desarrollado y cómo se utiliza de cara al usuario.

En el punto cinco, se detallarán los resultados obtenidos tras la unión de todo lo desarrollado en los apartados anteriores, haciendo un análisis de las características más importantes del sistema, el ratio de falsas aceptaciones y el ratio de falsas aceptaciones.

En el apartado seis, se hará un análisis del presupuesto necesitado para el desarrollo del proyecto así como la planificación seguida para la realización de éste.

En el apartado siete se realizarán unas conclusiones obtenidas tras el desarrollo de la totalidad del proyecto y en el apartado ocho se nombrarán posibles líneas futuras de trabajo relaciona al proyecto.

Por último, se agrupará todo el contenido no necesario para la comprensión del documento en anexos al final del trabajo. Estos anexos incluirán instalación de librerías y lo más importante, el código fuente desarrollado.

1.4. Estado del arte

La identificación de personas mediante medidas biométricas está presente desde hace ya bastantes años. Sin embargo, la identificación mediante el patrón vascular del dedo de la mano es una tecnología relativamente reciente, y el primer dispositivo comercial llegó de las manos de *Hitachi* en 1997 [1]. Sin embargo, la patente más antigua [2] corresponde a un inventor francés en el año 1984, que ideó esta tecnología debido a que le fue robada su identidad bancaria. Hoy en día, esta tecnología se utiliza en cajeros alrededor de todo el mundo, incluyendo Japón, China y Polonia.

Ilustración 1-1: Cajero con biometría vascular en Polonia [24]

Dentro de los sensores de este tipo, hay diversas maneras de tomar las medidas. En primer lugar, como en la figura anterior, se puede utilizar tecnología de reflexión; es decir, la luz se refleja en el dedo y la cámara capta el producto de esta reflexión de luz. Sin embargo, existen más tecnologías, como la transmisión de luz, en la que la luz atraviesa el dedo y la cámara capta la imagen desde el otro lado. Por último, también existe la posibilidad de iluminar el dedo desde ambos laterales y captar la imagen desde abajo. Estas tecnologías se pueden ver de forma gráfica en las figuras de abajo:

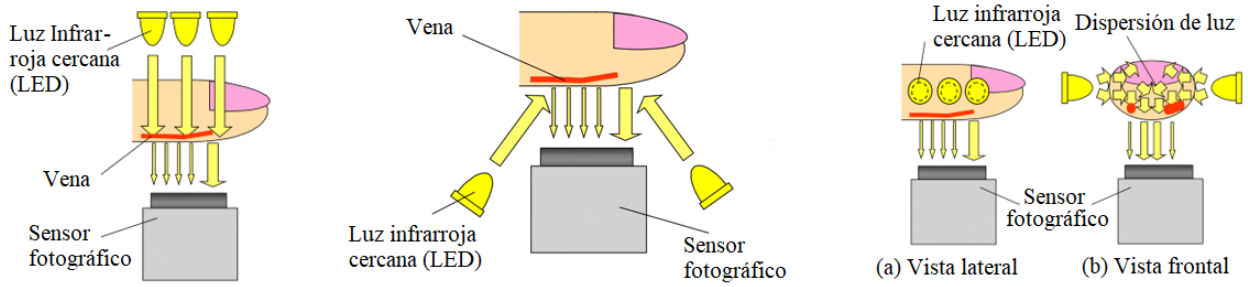


Ilustración 1-2: Tipos de tecnologías de iluminación para obtener la anatomía vascular del dedo [25]

Observando los tipos de tecnologías anteriores, consideramos que la solución más sencilla es utilizar la tecnología de iluminación directa, ya que tal y como se puede observar en la siguiente figura, muestra de forma más diferenciada la anatomía vascular que en el caso de luz reflejada. Gracias a esto, el procesamiento de imagen necesario para la obtención de las características será notablemente más sencillo, aunque en contrapartida, el tamaño del dispositivo será mayor.

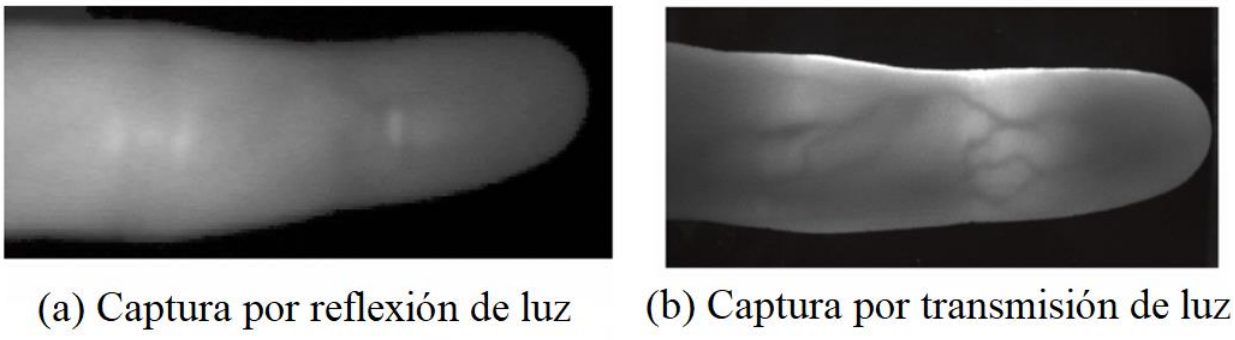


Ilustración 1-3: Comparación entre métodos de iluminación por reflexión y directa [25]

Por último, seguiremos un esquema de diseño similar al que se muestra en la siguiente figura. Donde la unidad de autentificación será el ordenador a bordo del sistema.

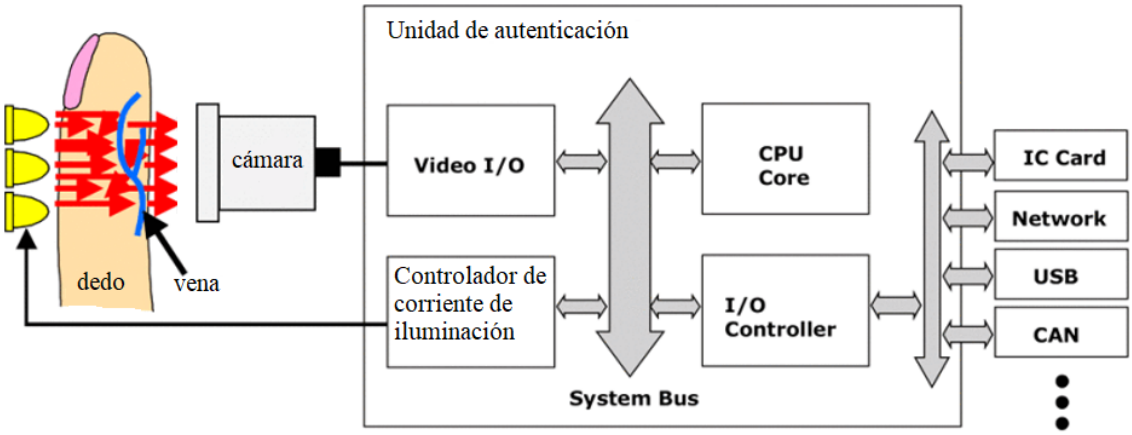


Ilustración 1-4: Diagrama de bloques para la identificación por patrón vascular [25]

1.5. Solución propuesta

En este apartado se describirán los objetivos y requisitos del proyecto a realizar así como las decisiones tomadas que influyen en el resultado del trabajo.

1.4.1 Alcance

1.4.1.1 Objetivos

- OBJ.1: El sistema debe ser capaz de tomar imágenes en el espectro *NIR*.
- OBJ.2: El sistema debe ser capaz de obtener la anatomía vascular del dedo índice de un sujeto.
- OBJ.3: El sistema debe ser capaz de detectar patrones característicos de imágenes binarias.

1.4.1.2 Requisitos

- F.1: El sistema será capaz de obtener imágenes.
- P.1.1: La resolución de las imágenes no debe ser inferior a $640x480$.
- P.1.2: El espectro capturado por la cámara debe incluir el infrarrojo cercano.
- F.2: El sistema será capaz de realizar procesamiento de imágenes.
- P.2.1: El sistema debe procesar las imágenes mediante *OpenCV4* [3].
- F.3: El sistema debe ser capaz de obtener la anatomía vascular del dedo índice mediante el traspaso de luz *NIR* a través de éste.
- P.3.1: La anatomía vascular obtenida debe formatearse en una imagen binaria.
- P.3.2: El sistema debe utilizar luz *NIR* en las longitudes de onda de $890nm$ y $940nm$ pico conjuntamente.
- F.4: El sistema debe ser capaz de identificar puntos característicos en imágenes binarias.
- P.4.1: El sistema detectará puntos clave mediante los métodos *SIFT*, *SURF*, *ORB* o *KAZE* [4].
- F.5: El sistema debería ser capaz de reconocer a una persona mediante la anatomía vascular de su dedo índice.
- F.6: El sistema debería incluir una base de datos para el reconocimiento de personas.
- P.6.1: La base de datos contendrá los puntos característicos de la anatomía vascular del dedo índice de varias personas.
- O.1: La captura de imágenes se realizará mediante la pulsación de una tecla.
- SEC.1: El sistema no debe causar daño alguno al usuario.
- SAF.1: La base de datos del sistema debería estar protegida mediante encriptación.
- C.1: El presupuesto máximo del proyecto será de 150 euros.
- C.2: El procesamiento se debe realizar sobre una *Raspberry Pi 4* (4GB).
- C.3: La captura de imágenes se debe realizar mediante la cámara *Raspberry Pi NoIR Camera*.
- C.4: El sistema debe contar con una fuente de alimentación externa tipo *USB-C* de $5V$ con una potencia mínima de $15W$.
- C.5: La salida de vídeo del sistema debe conectarse a un monitor externo.
- C.6: La cámara debe estar a $9cm$ del dedo índice del usuario.

1.4.2 Calendario de entregables

1.4.2.1 Criterios de aceptación

El proyecto se aceptará cuando se hayan implementado y verificado todos los requisitos según la matriz de verificación del siguiente apartado.

1.4.2.2 Matriz de verificación

Requisito	Nombre requisito	I	A	D	T	Nombre prueba	Estado
F.1	Adquisición de imágenes			X		Test1	Terminado
P.1.1	Resolución			X		Test1	Terminado
P.1.2	Espectro			X		Test1	Terminado
P.1.3	Captura por pulsación				X	Test1, Test2	Cancelado
F.2	Procesamiento de imágenes		X			Instalación *OpenCV4* [3]	Terminado
F.3	Anatomía vascular			X	X	Test3	Terminado
P.3.1	Binarización				X	Test3, Test4	Terminado
P.3.2	Luz *NIR*		X			Estudio *datasheet*	Terminado
F.4	Puntos característicos				X	Test4, Test5	Terminado
F.5	Reconocimiento				X	Test5, Test6	Terminado
P.5.1	Base de datos		X			Creación base de datos	Terminado

Tabla 1: Matriz de verificación

1.4.2.3 Plan de pruebas

- Análisis 1: Adquisición de imágenes.
    - Nombre de la prueba: Test1.
    - Descripción: El sistema debe tomar una o varias imágenes en las que se pueda ver que cumple con la resolución mínima requerida y se observe que se incluye el espectro infrarrojo cercano.
    - Realización: Se crea un código fuente con *OpenCV4* [3] y *C++* que capture imágenes y las muestre por pantalla en bucle, creando así una especia de *webcam NIR*. Posteriormente se ejecuta el programa resultado de la compilación y se observa el resultado.
- Análisis 2: Captura por pulsación.
    - Nombre de la prueba: Test2.
    - Descripción: El sistema debe tomar una única imagen cuando se presione una tecla en el mismo. Posteriormente, se muestra la imagen capturada por pantalla.
    - Realización: Se crea un código fuente con *OpenCV4* [3] que ejecute el programa cuando se pulse una tecla cualquiera a excepción de la tecla ESC, que es utilizada para terminar el programa.

- Análisis 3: Obtención de la Anatomía vascular del dedo índice.
    - Nombre de la prueba: Test3.
    - Descripción: El sistema debe ser capaz de umbralizar el dedo del sujeto y posteriormente resaltar las venas y arterias dentro de éste.
    - Realización: Se crea un código con *OpenCV4* [3] y *C++* que umbralice el dedo índice de la persona y realice un procesamiento de imagen que resalte las venas y arterias del sujeto en la imagen. Esta prueba debe realizarse en el sistema construido, ya que la obtención de imágenes en un entorno similar es muy importante para el funcionamiento del sistema. Por último, se ejecuta el programa creado y se comprueba el resultado.
- Análisis 4: Binarización de la Anatomía vascular del dedo índice.
    - Nombre de la prueba: Test4.
    - Descripción: El sistema, a partir de los datos obtenidos en la prueba anterior, debe realizar un procesamiento de imagen para obtener las venas y arterias del dedo y posteriormente convertirlas en una imagen binaria, donde los píxeles en los que se encuentran éstas serán 255 y el resto, 0.
    - Realización: Se escribe un código en *C++* con *OpenCV4* [3] que realice varias acciones; entre ellos, una igualación de histograma adaptativa (*CLAHE*), filtrado de imagen en frecuencia y umbralización binaria. Cuando se ejecute el código, deberían mostrarse las arterias y venas en blanco sobre un fondo negro.
- Análisis 5: Obtención de puntos característicos.
    - Nombre de la prueba: Test5.
    - Descripción: El sistema, a partir de la imagen binarizada de la Anatomía vascular del dedo, debe ser capaz de obtener puntos característicos de la imagen que la identifiquen unívocamente.
    - Realización: Se genera un programa con *OpenCV4* [3] y *C++* que utilice los métodos de obtención de puntos característicos y descriptores SIFT, SURF, ORB y KAZE, con el fin de comprobar cuál otorga mejores resultados. Las características que obtenemos con estos métodos nos ayudarán a diferenciar la Anatomía vascular del dedo de una persona de la de otra.
- Análisis 6: Reconocimiento de personas
    - Nombre de la prueba: Test6.
    - Descripción: El sistema, a partir de los puntos característicos obtenidos de la Anatomía vascular del dedo, coteja los datos obtenidos con una base de datos que almacena las características de la Anatomía vascular del dedo índice de varias personas y es capaz de determinar cuál de esas personas (si es una de ellas) es.
    - Realización: Se crea un código con *OpenCV4* [3] en *C++* que a partir de los puntos característicos obtenidos con el anterior análisis, realice una comparación de estas características con las existentes en una base de datos para buscar similitudes y determinar si coinciden con alguna entrada ya existente. En ese caso, la persona está identificada, en caso contrario, es una persona desconocida para el sistema.

- Análisis 7: Procesamiento de imágenes
    - Nombre de la prueba: Instalación *OpenCV4* [3].
    - Descripción: Se realizará la descarga y compilación de las librerías de código abierto para procesamiento de imagen *OpenCV* [3] en su versión 4.5.2 sobre el sistema operativo *Raspbian* de la propia *Raspberry Pi 4*.
    - Realización: Se clonarán los repositorios de los paquetes *opencv_4.5.2* y *opencv_contrib_4.5.2* de sus respectivas páginas de *Github*, github.com/opencv/opencv y github.com/opencv/opencv_contrib. Una vez descargadas, con la herramienta de *CMake* y las dependencias necesarias, se realiza la compilación y por tanto, instalación de las librerías en el sistema operativo. Después de esto, el usuario puede utilizar *OpenCV* [3], siempre y cuando realice la compilación con los argumentos *'pkg-config –cflags –libs opencv'*.

1.4.3 Diseño

La solución propuesta para alcanzar todos los requisitos y limitaciones mencionados anteriormente consiste en crear una carcasa mediante impresión 3D. El material elegido para el proyecto es *PLA+*, debido a su reducido coste, buena flexibilidad y resistencia, facilidad de impresión y por ser biodegradable e industrialmente compostable. La impresora utilizada para la impresión de todas las piezas es una *Prusa i3 MK3s*.

El diseño consta de cinco piezas distintas, de manera que el diseño del prototipo es modular. La unión de las piezas se realiza mediante cuatro tornillos M4. Los archivos de las piezas se encuentran en el enlace https://www.dropbox.com/s/xu84nvobmh45e9s/piezas.zip?dl=0. A continuación se describen individualmente las distintas piezas que componen el sistema:

1.4.3.1 Pieza principal

Esta pieza se denomina la pieza principal por ser la de mayor tamaño. La única función que tiene es alojar la carcasa en la que se encuentra la *Raspberry Pi* y la cámara, además de dejar un hueco de aproximadamente $9cm$ hasta la siguiente pieza, siguiendo la restricción C.6. La pieza se compone únicamente de cuatro paredes y cuatro bloques pequeños con hendiduras para introducir tuercas M4. Los tornillos irán enroscados en estas tuercas cuando se cierre el sistema completo.

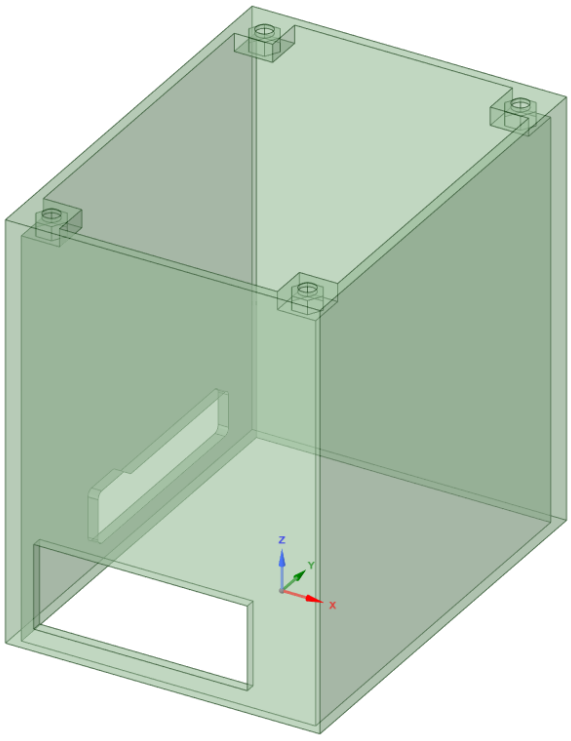


Ilustración 1-5: Pieza principal

1.4.3.2 Módulo dedo

Esta pieza, tal y como su propio nombre indica, es el módulo en el cual el usuario debe introducir el dedo. Es una parte relativamente simple, ya que sólo se compone de un agujero para introducir el dedo, unas paredes que limitan la movilidad de éste y unos pasadores para la correcta alineación de los tornillos.

1.4.3.3 Módulo LED

En este módulo, se ubicarán siete LEDs infrarrojos con distintas longitudes de onda y la PCB encargada de suministrarles corriente, de la cual hablaremos más adelante. En cuanto a la iluminación, se eligen 4 LEDs con una longitud de onda de 890nm, y tres con una longitud de onda de 940nm. Para contener todos estos elementos, la pieza se compone de cuatro paredes con pasadores para los tornillos en las que dentro se encuentra a su vez un recinto modelado de manera que encaje de manera justa con la PCB diseñada. Por supuesto, se encuentran siete agujeros por los que saldrán la iluminación hacia la pieza de abajo, el módulo dedo, encajando los LEDs en estos agujeros. Además, se añaden tres huecos para tuercas M2 para poder atornillar la placa a la pieza de manera que quede fija y un rebaje donde se coloca el circuito integrado para tener mayor margen. A continuación se muestran en las figuras el módulo LED y módulo dedo individualmente.

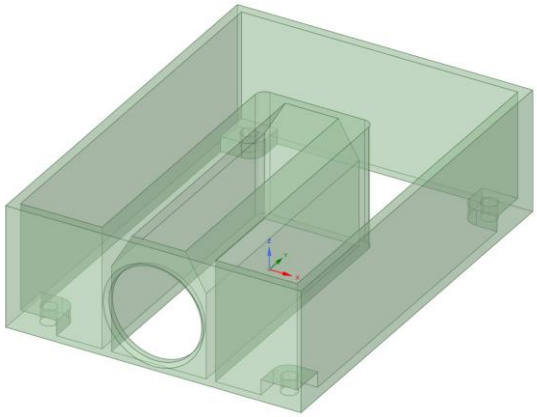


Ilustración 1-7: Módulo dedo

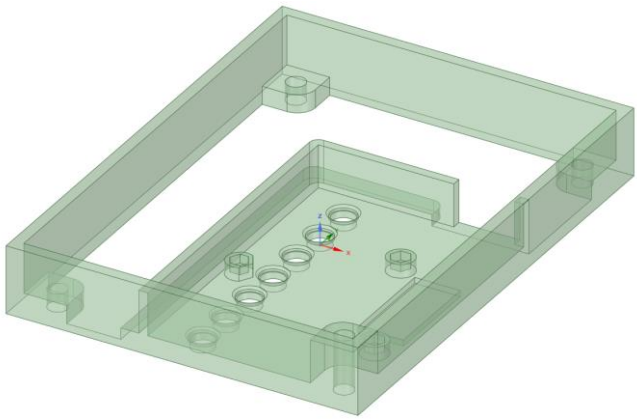


Ilustración 1-6: Módulo LED

1.4.3.4 Tapas inferior y superior

Estas dos piezas, simplemente se tratan de tapas para cerrar el sistema por arriba y por abajo. La tapa inferior está diseñada para encajar en la pieza principal, e incorpora unos huecos para ayudar a la ventilación del computador. Cabe destacar que para que esta ventilación funcione, se le debe incorporar además cuatro patas de goma adhesivas a esta tapa, para elevarla con respecto al suelo. En cuanto a la tapa superior, esta se trata de un panel sólido con cuatro agujeros avellanados para tornillos M4.

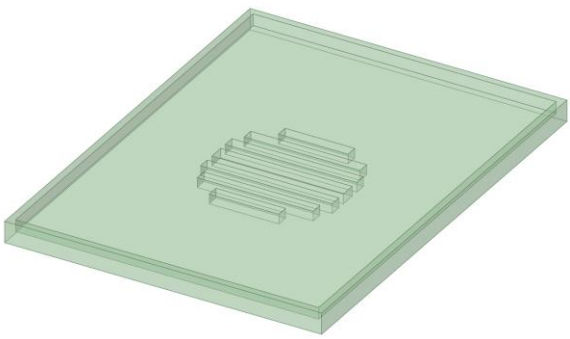


Ilustración 1-9: Tapa inferior

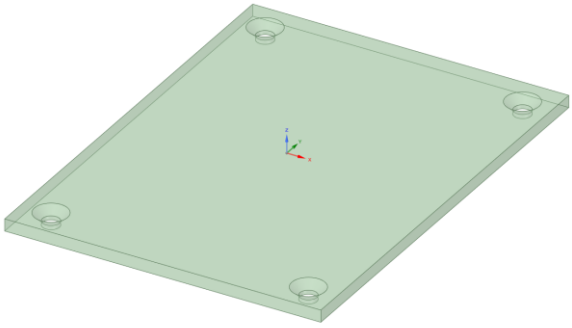


Ilustración 1-8: Tapa superior

1.4.3.5 Montaje completo

Juntando todas las piezas, el sistema queda totalmente montado. En la parte inferior se alojará la carcasa con la *Raspberry Pi* y la cámara sin filtro IR. El objetivo del diseño es que la cámara capte una imagen del dedo desde abajo mientras, con el módulo LED, éste está siendo iluminado con luz infrarroja desde arriba. Además, para ayudar a disminuir la dispersión de luz en el hueco para el dedo, se reviste de una esponja negra flexible que permite que el usuario introduzca el dedo sin ofrecer resistencia, pero que reduzca o elimine la reflexión de luz en las paredes del dispositivo, haciendo difícil la tarea de separar regiones posteriormente. A continuación, se puede observar una foto del montaje completo.

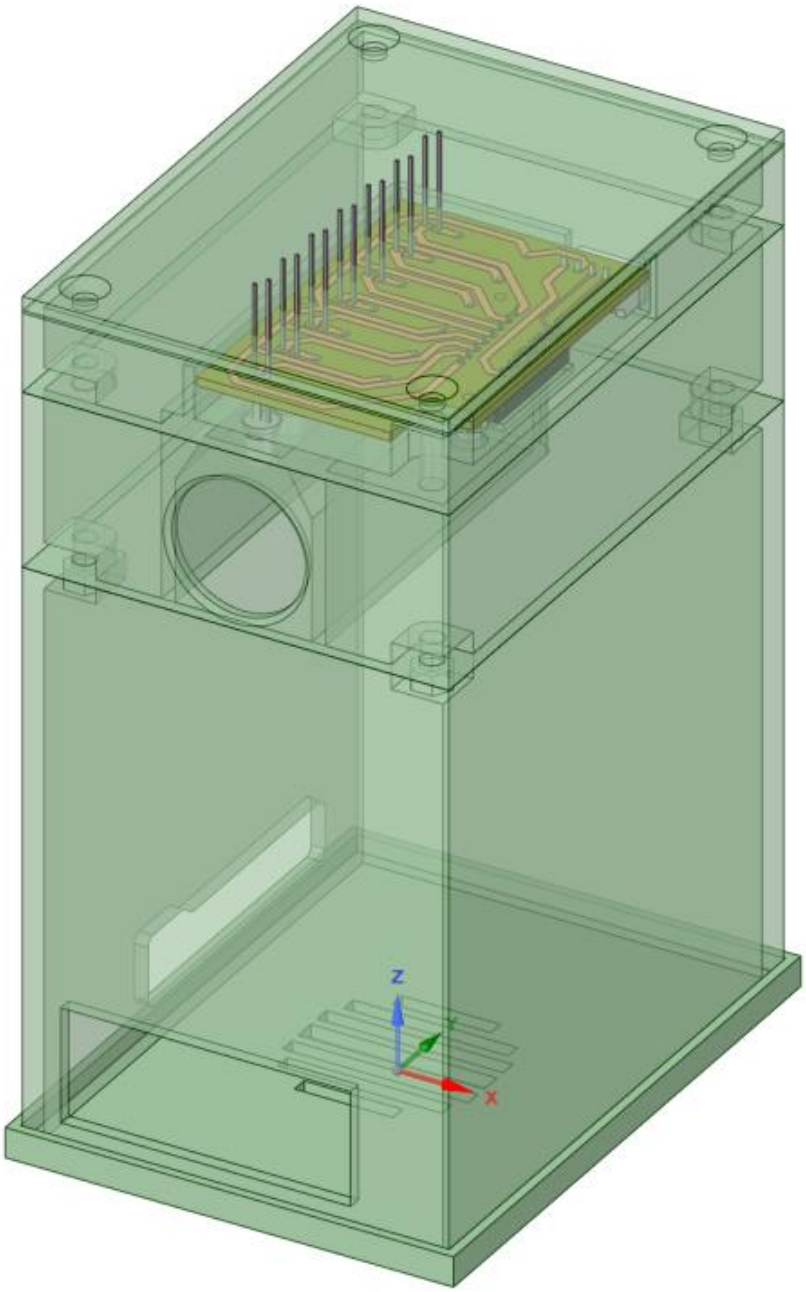


Ilustración 1-10: Montaje completo

2 DISEÑO Y ELECCIÓN DEL HARDWARE

Hay una fuerza motriz más poderosa que el vapor, la electricidad y la energía atómica: La voluntad.

Albert Einstein, 1879-1955

En este capítulo vamos a explicar con mayor nivel de detalle el proceso de creación de los diseños del proyecto, especialmente el diseño de la electrónica. Además, se detallará completamente todo el *Hardware* utilizado para la creación del prototipo y se hará un repaso sobre los errores encontrados y las iteraciones que han sido necesarias en el diseño.

2.1. Electrónica para la iluminación

Para comenzar el diseño de la electrónica del sistema, partimos de ciertas limitaciones. Para conocer mejor estas restricciones, nos remontaremos antes a los componentes que sabemos con total seguridad que necesitaremos, como son una fuente de alimentación, y los siete LEDs NIR. Por supuesto, también necesitaremos resistencias para estos LEDs, para limitar el flujo de corriente y evitar la rotura de éstos.

Para calcular las resistencias que necesitamos, primero necesitamos conocer la alimentación nominal de ambos tipos de LED, por lo que estudiamos la *datasheet* de ambos. Primero, buscamos lo necesario para el LED de $940nm$ pico, en concreto, el modelo TSAL6100 [5]. En la *datasheet*, podemos ver que la intensidad típica es de $100mA$, y que la caída de tensión es de $1.35V$ típicamente. Por lo tanto, podemos calcular el valor de la resistencia de la siguiente manera:

$$R = \frac{V_{cc} - V_{LED}}{I_f} = \frac{5V - 1.35V}{0.1A} \rightarrow R = 36.5\Omega$$

$$W = I^2 \cdot R = (0.1A)^2 \cdot 36.5\Omega \rightarrow W = 0.365W$$

Como se puede ver en los valores obtenidos en las ecuaciones anteriores, necesitaremos una resistencia de al menos $½\,W$ de un valor cercano a los 36.5 ohmios. En el laboratorio de la escuela se nos fueron proporcionadas resistencias de 34Ω, las cuales consideramos totalmente válidas para esta aplicación aunque la corriente sea ligeramente mayor que la estimada inicialmente.

A continuación, debemos realizar el mismo cálculo con el otro tipo de LED NIR, en concreto, el modelo QED223 [6]. Si observamos la *datasheet*, podemos observar que tanto la corriente típica como caída de tensión es similar al caso del LED anterior, por lo que las resistencias que utilizaremos serán las mismas.

Ahora ya conocemos el valor de intensidad que queremos que pase por cada diodo, por lo tanto sólo nos queda pendiente conocer cómo vamos a alimentar los LEDs. No podemos ponerlos en paralelo porque desconoceríamos la intensidad que circula por cada uno de ellos e incluso puede llegar a ser peligroso. La mejor solución es utilizar un IC controlador de LEDs. Sin embargo, tuvimos que descartar esta opción ya que la corriente necesaria para alimentar los diodos es demasiado alta para usar este tipo de integrado. Por lo tanto, decidimos utilizar un controlador más genérico, como puede ser un IC de pares Darlington; en concreto, el integrado ULN2003AN [7] de Texas Instruments. Este integrado es capaz de manejar siete salidas con sus respectivas entradas individuales, por lo que es perfecto para esta aplicación. Conectaremos las salidas a los LEDs a través de las resistencias y puentearemos las entradas de manera que con una señal podamos encender todos los LEDs de $940nm$ y con otra, los de 890; tal y como se aprecia en el circuito de la siguiente figura, donde los LED de número impar son QED223 y los de número par, TSAL6100:

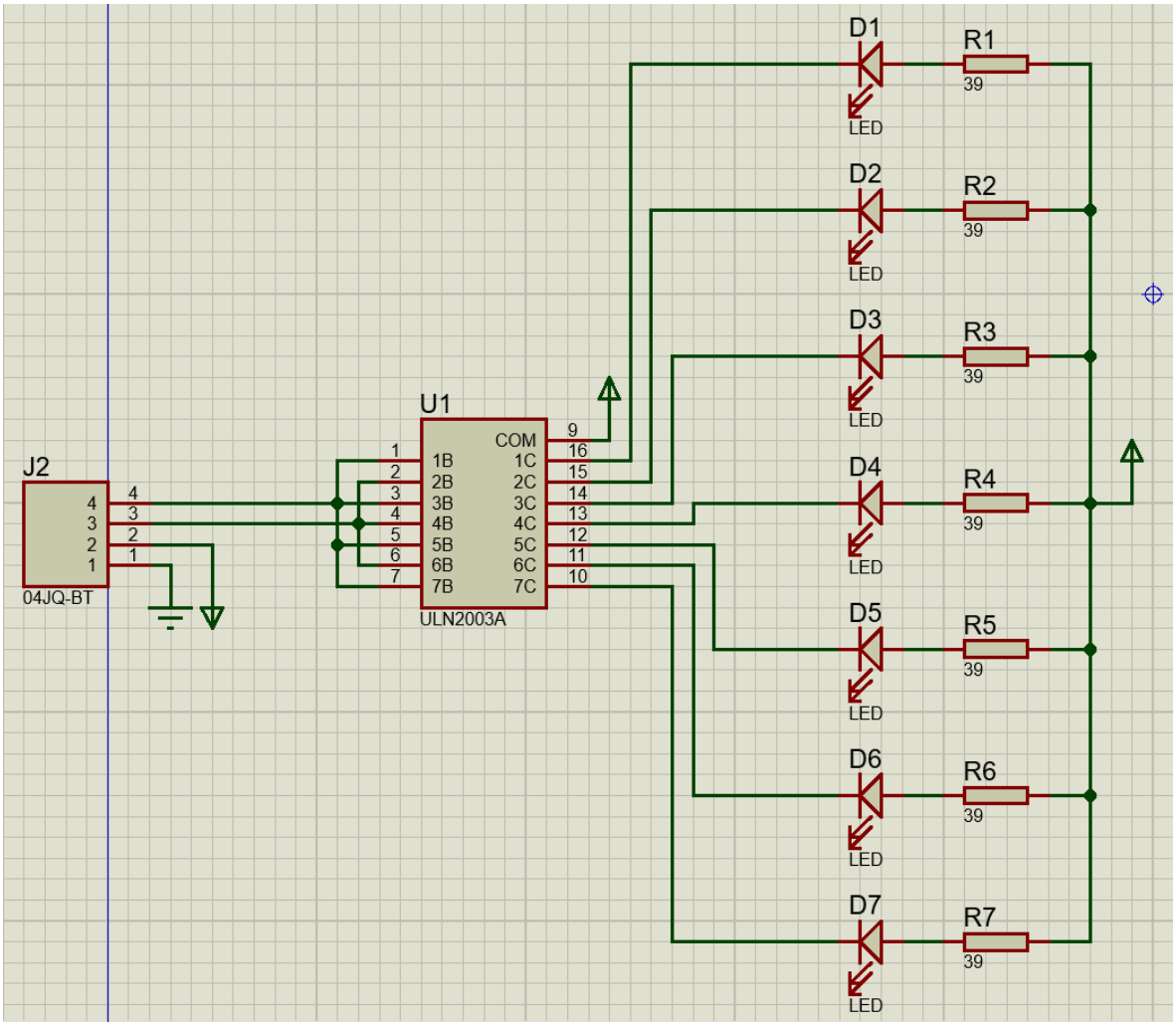
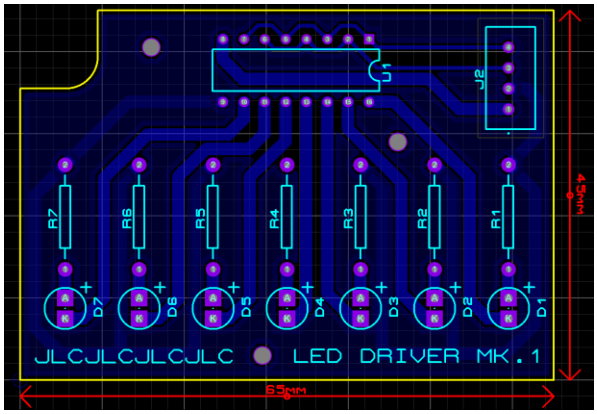


Ilustración 2-1: Esquemático de conexión de la placa de alimentación LED

Para otorgar la alimentación y las señales de control al circuito integrado, optamos por utilizar un conector de JST (Japan Solderless Terminals) tipo XH de cuatro pines. De esta manera, la conexión entre la placa y el ordenador no es permanente, ya que se puede desconectar simplemente el cable, permitiendo así la modularidad. Por último, tras verificar el circuito, se procede a diseñar la PCB de manera que sólo sea necesaria una cara de cobre. Teniendo en cuenta los requisitos de corriente de las pistas, la placa queda rutada de la siguiente manera:



Además, como se puede apreciar, se añaden tres agujeros M2 por los que pasarán tornillos que mantendrán sujeta firmemente la placa a la pieza de plástico junto con tuercas. También se añade un plano de V_{cc} para ahorrar ácido y/o tiempo a la hora de fabricar la placa. Las marcas que se ven en cian corresponden a las inscripciones que contiene la placa por el lado sin cobre, para identificar rápidamente el componente que se tiene que soldar en una determinada posición. Por último, a continuación, se muestra el resultado de fabricación de la placa, y el resultado tras realizar las soldaduras de forma manual.

Ilustración 2-2: Rutado de la placa de alimentación LED

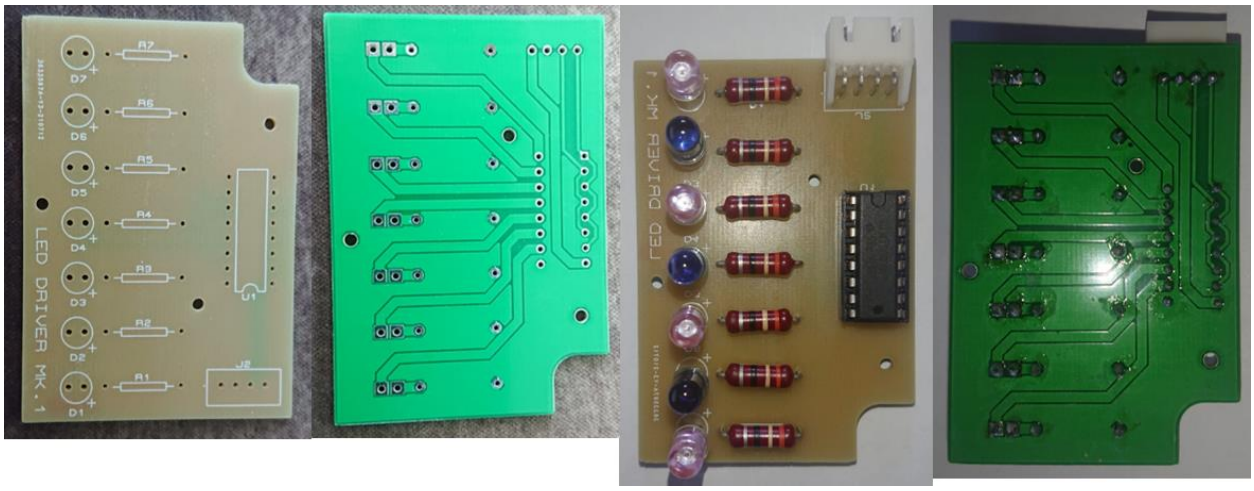


Ilustración 2-4: PCB

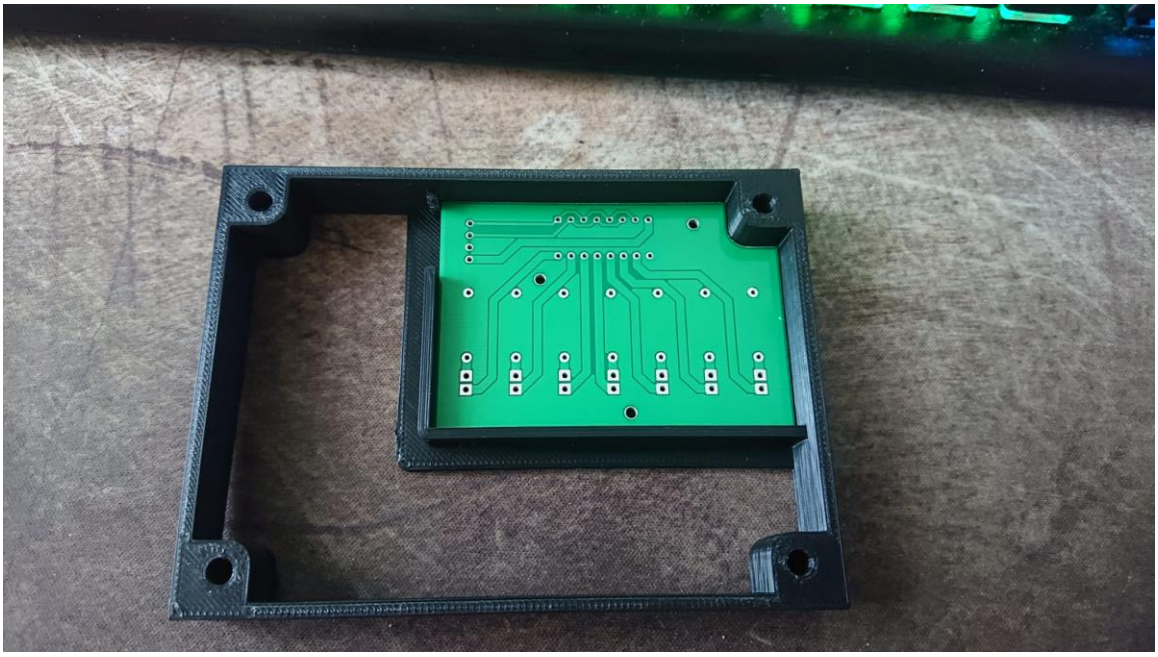


Ilustración 2-3: Muestra de encaje de PCB

Para la conexión de la placa con la *Raspberry Pi*, utilizaremos cuatro cables 22AWG con terminales crimpados de JST XH hembra por un lado, y Dupont hembra por el otro extremo, el cual será conectado al ordenador del sistema. El cable queda de la siguiente manera:

Ilustración 2-5: Conector PCB-Ordenador

No obstante, antes de fijar el montaje de la placa, se verifica si el circuito funciona correctamente. Para ello, decidimos soldar a una PCB igual unos LEDs con espectro visible, en este caso, rojos. En lugar de utilizar resistencias de mayor potencia como en el caso anterior, utilizamos resistencias de $^1/_4\,W$ de 330Ω. La placa obtenida queda de la siguiente manera:

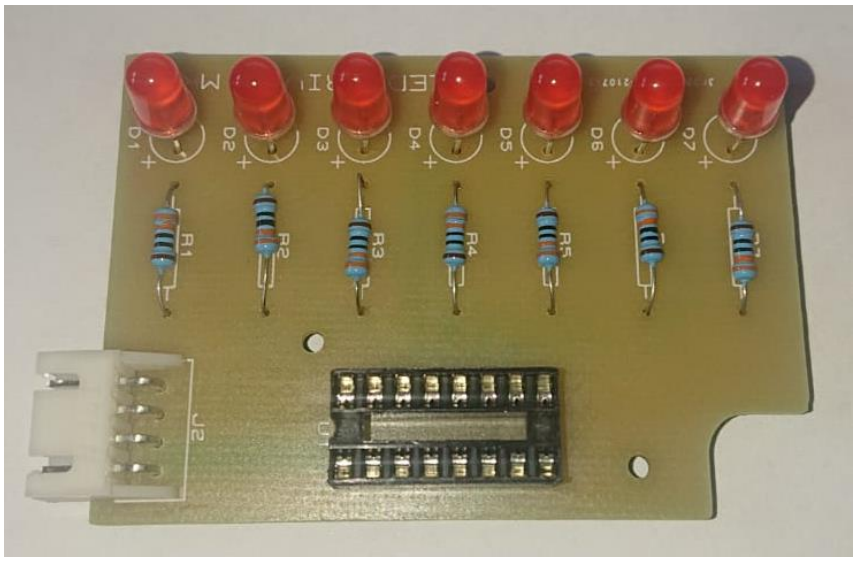


Ilustración 2-6: PCB alternativa

De esta forma, podemos verificar rápidamente si el circuito funciona o no, ya que esta luz sí es visible para el ojo humano, en contraposición al caso de las luces NIR. Para mandar las señales de encendido y apagado de las luces, utilizaremos una placa Arduino Uno R3. Colocaremos dos pulsadores que leerá el microcontrolador mediante *polling* para activar o desactivar las señales de control de las luces. El código se encuentra en el anexo E y el esquemático seguido es el siguiente:

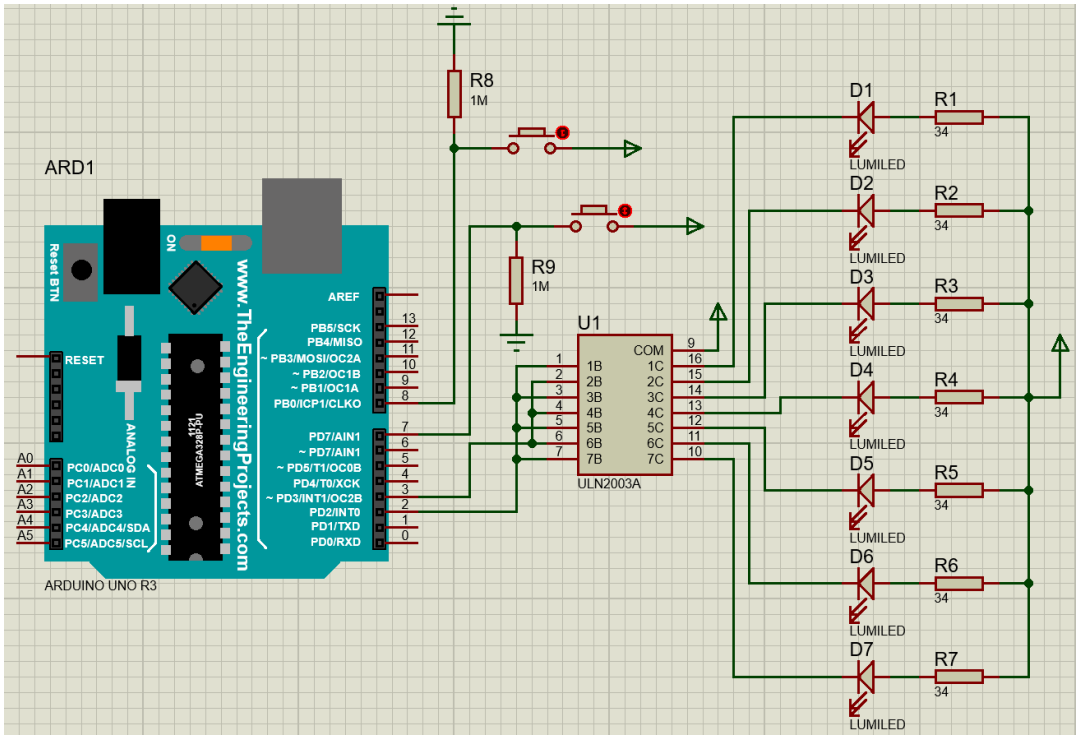


Ilustración 2-7: Montaje de prueba con Arduino

2.2. Ordenador a bordo

Como se ha indicado en apartados anteriores, la elección del ordenador del sistema es la placa *Raspberry Pi* 4B en su versión de 4GB de memoria RAM. El principal motivo para elegir este ordenador es la relación prestaciones/precio. Además, soporta aceleración de *Software* por paralelización y autovectorización, que es a su vez compatible con la API de *OpenCV*4 [3]. Esto, sumado a su bajo coste, la hace la placa adecuada para la aplicación de este proyecto. No obstante, a continuación, repasaremos las otras posibles opciones.

2.2.1 Raspberry Pi 4B

La Raspberry Pi 4 se trata de un ordenador en una única placa (SBC) del tamaño de una tarjeta de crédito. Es un ordenador totalmente funcional y completo. Se puede conectar hasta a dos pantallas 4K, contiene cuatro puertos USB para conectar cualquier periférico de este tipo como, teclado y ratón o memorias USB. Se compone de un SoC con un procesador Cortex A-72 de cuatro núcleos de arquitectura ARM v8. Además de esto, tiene un puerto específico para conectar cámaras y soporta APIs gráficas como OpenGL y Vulkan. También tiene un *header* de pines de entrada y salida programables GPIO además de puerto *Ethernet*, *Wi-Fi* y *bluetooth*. Se alimenta mediante una fuente de alimentación de tipo USB-C.

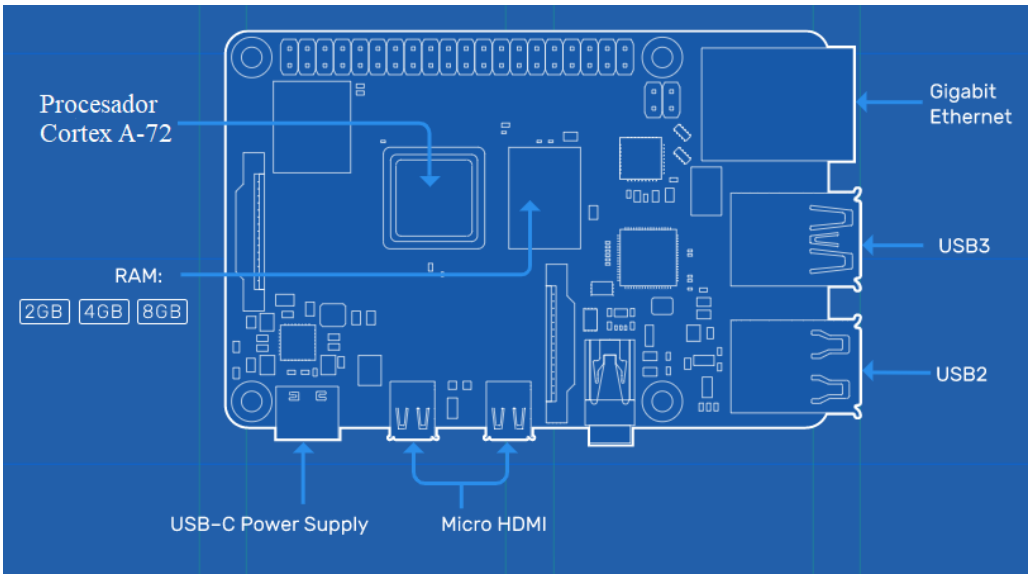


Ilustración 2-8: Raspberry Pi 4B

Además, dispone de una GPU VideoCore 6, con la que podremos acelerar tareas gráficas, haciendo este SBC ideal para nuestra aplicación con *OpenCV* [3]. Además, gracias a sus entradas y salidas GPIO, podremos alimentar y dar señal a la placa que encargada de la iluminación LED NIR.

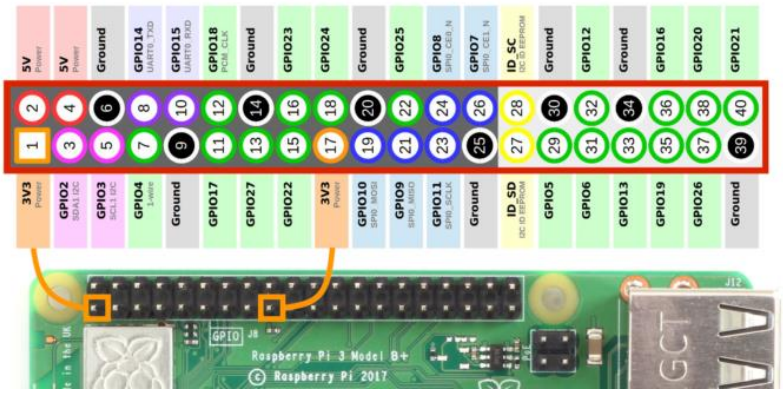


Ilustración 2-9: Esquema GPIO de la Raspberry Pi 4B [29]

2.2.2 NVIDIA Jetson Nano [8]

Este ordenador de única placa o SBC es una propuesta muy interesante, ya que tiene una gráfica completamente dedicada de 128 núcleos. Sin embargo, el procesador se queda un poco atrás con respecto a la *Raspberry Pi* 4. No obstante, el tamaño es menor, aunque al necesitar un disipador de aluminio, es notablemente más alta. Lo más destacable y lo que provoca que se descarte esta opción sin ninguna duda es el elevado precio del producto. El coste de esta placa es aproximadamente el triple que la *RP4*, por lo que esta opción queda descartada.

Ilustración 2-10: NVIDIA Jetson Nano Developer Kit [8]

2.2.3 Otras opciones

También se barajaron otras opciones, aunque se consideran menos importantes que el punto anterior, por lo que simplemente se enumerarán y se comentarán de forma breve.

- *Orange Pi 4B* [9]. Esta opción es una propuesta interesante porque incluye un procesador ARM con arquitectura big.LITTLE[1], por lo que el consumo podría verse reducido notablemente, además de incorporar una gráfica dedicada compatible con las APIs gráficas más recientes. Por otra parte, existe una escasez de este SBC y el precio es muy elevado.
- *BeagleBoard-X15* [10]. Esta opción fue rápidamente descartada ya que además de su elevado precio, las prestaciones se quedan atrás con respecto a las placas anteriores, incluso llegando a poder ser insuficientes para cumplir los requisitos del sistema.

2.3. Diseño e impresión de las piezas

Para el diseño del proyecto, se realizaron varias iteraciones dependiendo de cada pieza. Lo más destacable fue que en la primera iteración completada, no fue posible encontrar unos tornillos adecuados para cerrar el prototipo, por lo que fue necesario cambiar el calibre de los agujeros de todas las piezas a excepción de la tapa inferior. A continuación se comentarán las iteraciones necesarias hasta obtener el prototipo final pieza por pieza.

[1] Arquitectura de computación heterogénea que consiste en acoplar los procesadores más lentos y que consumen menos energía (LITTLE) con los procesadores más potentes y que consumen más (big). En general, sólo un lado se activa a la vez, por lo que se manifiesta que se puede ahorrar hasta un 75% de energía en algunas tareas. [25]

2.3.3 Tapa inferior

La primera iteración de esta pieza, al igual que todas las siguientes, fue impresa en plástico ABS gris con la impresora BCN3D SIGMA R17. Sin embargo, este es el único caso en el que el modelo no fue cambiado para la siguiente iteración. Simplemente, en la segunda y última iteración de esta pieza, ésta fue impresa en PLA+ negro en una impresora Prusa i3 Mk3s. Cabe destacar que esta pieza está pensada para incluir patas de goma adhesivas en las esquinas y a diferencia de las demás piezas, que llevan tornillos; esta pieza está diseñada para encajar con la siguiente. A continuación, en la figura, se puede observar las dos iteraciones impresas.

Ilustración 2-11: Iteraciones de tapa inferior

2.3.4 Pieza principal

Al igual que la pieza anterior y las siguientes, la primera iteración de esta pieza fue impresa en ABS. No obstante, en la siguiente y última iteración de esta pieza, hay un cambio notable. En la primera iteración de esta pieza, los agujeros para los tornillos son de tamaño M3, al igual que el hueco para las tuercas. Sin embargo, en la segunda iteración se amplió el diámetro de estos agujeros hasta M4, ampliando, por tanto, el tamaño para las tuercas correctas. El resto de las partes de la pieza se mantuvieron iguales.

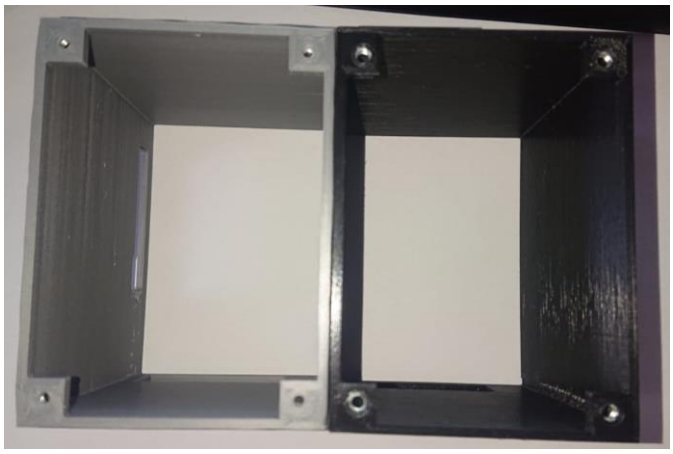


Ilustración 2-12: Iteraciones de pieza principal

2.3.5 Módulo dedo

Para esta pieza, también fueron necesarias únicamente dos iteraciones. En la segunda iteración se realizaron varios cambios. Para comenzar, el diámetro de los agujeros paso de M3 a M4, la cavidad para introducir el dedo se alargó ligeramente y, por último, se añadió una subida en pendiente con el fin de disminuir la dispersión de luz, motivo por el que también se le añadieron a la pieza esponjas forradas con tela opaca. A continuación, se observan la primera y segunda pieza y un detalle de la pendiente mencionada.

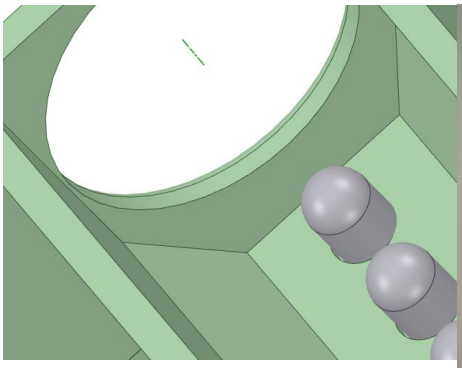


Ilustración 2-13: Detalle módulo dedo

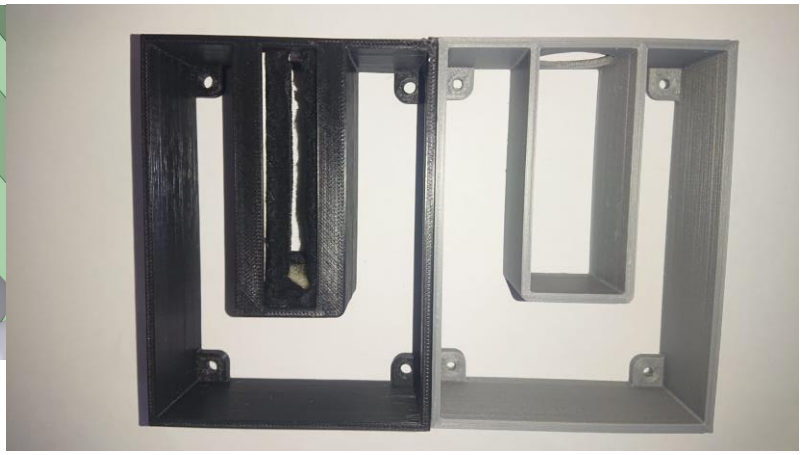


Ilustración 2-14: Iteraciones de módulo dedo

2.3.6 Módulo LED

Esta pieza fue la que más iteraciones necesitó, siendo tres en total. La primera iteración se realizó antes de tener el diseño de la placa de control de los LEDs, por lo que la segunda iteración cambia de manera notable. Para comenzar, elimina plástico de donde no es necesario, para reducir el tiempo de impresión, posteriormente se añaden paredes para la PCB, de manera que encaje de manera justa, además de añadir los huecos correspondientes para las tuercas. Por último, se aumenta ligeramente la altura de la pieza y se aumenta el calibre de los agujeros de los tornillos de M3 a M4. Sin embargo, estos cambios no fueron suficientes y se necesitó una tercera iteración en la que se aumentó aún un poco más la altura de la pieza, se aumentó la altura de los huecos para las tuercas, se eliminó más plástico innecesario para agilizar la impresión y, por último, se realizó un rebaje en la zona en la que se encontrará el IC, ya que se había detectado que chocaba con el suelo de la pieza. Tras estos cambios, la pieza se da por válida. En la siguiente figura se pueden apreciar todas las iteraciones una tras otra.

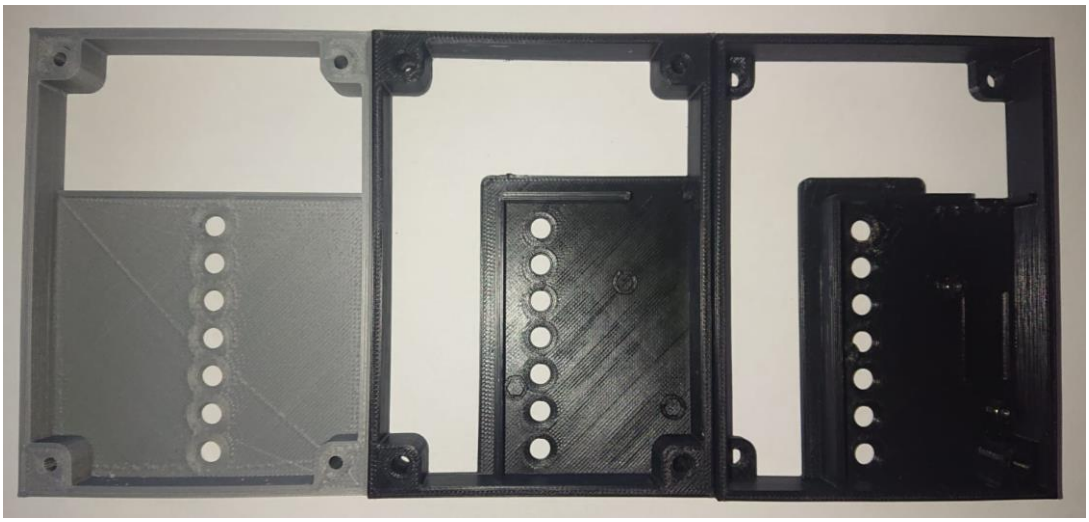


Ilustración 2-15: Iteraciones de módulo LED

2.3.7 Tapa superior

Para la tapa superior, fueron necesarias únicamente dos iteraciones. El cambio más notable es el paso del diámetro de los agujeros de M3 a M4. Además de esto, se disminuye ligeramente la altura de la pieza y se realiza un avellanado en los agujeros para los tornillos, ya que para el montaje se utilizarán tornillos M4x60mm de cabeza avellanada. En la siguiente figura se aprecia la ligera diferencia entre las dos iteraciones.

Ilustración 2-16: Iteraciones de la tapa superior

2.4. Montaje del prototipo

Por último, únicamente resta ensamblar el prototipo. Para ellos, debemos seguir una serie de pasos en orden, que se indicarán como puntos para facilitar la lectura:

1. Conectar la cámara, el ventilador y los cables de la placa de alimentación de los LEDs a la *RP4* en la carcasa de ésta, de manera que el cable rojo se conecta a $5V$, el negro a masa, el amarillo a GPIO.4, y el verde a GPIO.5, tal y como se observa en la figura.

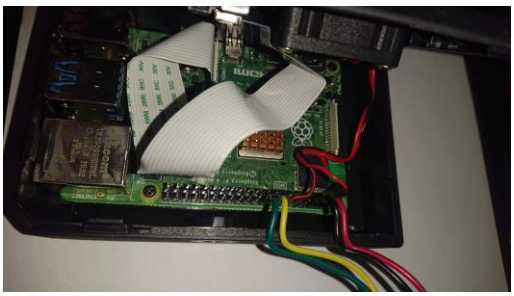


Ilustración 2-17: Conexión de cableado

2. Colocar las tuercas M4 en la pieza principal.
3. Colocar la caja contenedora de la placa sobre la tapa inferior y encajar la pieza principal encima.
4. Colocar encima el módulo dedo.
5. Introducir las tuercas M2 en el módulo LED y encajar y atornillar la PCB a la pieza.
6. Colocar encima del módulo dedo el módulo LED y conectar el cable de la placa a ésta.
7. Colocar la tapa superior encima del conjunto.
8. Introducir los tornillos desde arriba y atornillar.
9. Montaje finalizado.

3 DISEÑO DEL SOFTWARE

Un ordenador te permite cometer más errores y más rápido que cualquier otra invención en la historia de la humanidad, con las posibles excepciones de las pistolas y el tequila.

Mitch Radcliffe

Este apartado va a tratar sobre todo el desarrollo de código realizado en el proyecto, así como las decisiones de diseño *Software* que se han tomado y problemas hallados en el camino. Se explicará con detalle todos los procedimientos seguidos y se añadirán imágenes mostrando los resultados obtenidos a cada paso. También se comentará el entorno de programación elegido y el control de versiones utilizado.

3.1. Entorno de programación

En este proyecto, se ha trabajado desde diversos entornos y sistemas operativos. El trabajo principal se ha desarrollado desde la conexión de un sistema *Windows* 10 con un sistema *Raspbian* funcionando sobre la *Raspberry Pi 4* a través de un escritorio remoto. Dentro de este sistema, el IDE utilizado ha sido *Visual Studio Code*, ya que permite realizar *debugging*[2]. Además, las pruebas que se han planteado han sido probadas en su mayoría no sobre la *RP4*, sino sobre una máquina virtual ejecutando *Ubuntu* 21. En esta máquina también se ha utilizado el IDE de *Visual Studio Code* y el motivo de ejecutar las pruebas sobre este sistema es por el simple hecho de la comodidad y rapidez, ya que la máquina tiene mucha mayor potencia que el ordenador del prototipo.

Para el control de versiones, se utiliza el *Software* libre de *Git*, funcionando sobre un repositorio público [11] en *Github*. El esquema seguido es muy simple y existe únicamente una rama en la que se publican todos los avances realizados en el código.

3.2. Captación de imágenes e iluminación

El primer paso, antes de poder comenzar a realizar código, consiste en instalar *OpenCV* 4.5.2 [3]. Esto es un proceso relativamente largo de explicar, por lo que el proceso de instalación y configuración en el IDE puede encontrarse en el anexo B. Una vez tenemos preparada la API, podemos comenzar a programar.

Para comenzar, empezaremos con lo más simple, es decir, la captación de imágenes a través de la cámara. Para ello necesitamos agregar las cabeceras *highgui.hpp*, *imgproc.hpp* y *imgcodecs.hpp*. Ahora, podemos definir un tipo *Mat* para almacenar las imágenes que tome la cámara y para ello, además creamos un objeto de tipo *VideoCapture* para ejecutar la función *read*, que provocará que la cámara tome una imagen y sea almacenada en el contenedor definido anteriormente. Posteriormente, solo resta mostrar la imagen tomada por pantalla mediante el uso de la orden *imshow*. Si compilamos el programa y lo ejecutamos, debería aparecer una imagen muy oscura, ya que estamos tomando una foto de un lugar sin apenas luz. Para conseguir vislumbrar las venas y arterias del dedo, necesitamos hacer uso de la iluminación NIR, de manera que ésta atraviese el dedo y la cámara recoja una imagen en la que la anatomía vascular del dedo se vea oscura con relación al resto del dedo.

Para poder activar la iluminación, necesitamos acceder a las salidas y entradas de la *Raspberry Pi 4*. Para ello, necesitamos incluir la cabecera *wiringPi.h*. Ahora, inicializamos la librería con la función *wiringPiSetup*,

[2] Proceso de identificación y corrección de errores de programación.

seleccionamos los pines 4 y 5 como salidas y los colocamos en estado lógico *HIGH*. Después de tomar la imagen, cambiamos el estado lógico a *LOW* para desactivar la iluminación, de manera que ahorramos energía y evitamos calentar los componentes del sistema. Ahora, tras compilar y ejecutar el programa se puede vislumbrar claramente el dedo introducido y de forma sutil, la anatomía vascular del dedo, tal y como se puede apreciar en la siguiente figura.

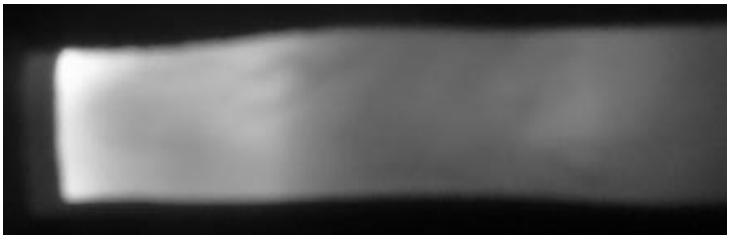


Ilustración 3-1: Imagen de dedo sin preprocesamiento

3.3. Preprocesamiento de imagen

En este punto tenemos un objetivo muy claro: Obtener el patrón de venas y arterias del dedo a partir de la imagen tomada por la cámara. Este proceso no es trivial y es necesario un gran número de operaciones para conseguir un resultado satisfactorio. Principalmente, nos basaremos en el preprocesamiento seguido en el desarrollo de *OpenVein Toolkit* [12] de la universidad de Salzburgo.

3.3.1 Umbralización del dedo

En la imagen anterior, las zonas de la imagen que no pertenecen al dedo no son totalmente oscuras, por lo que será necesario umbralizar para aislar el dedo del fondo de la imagen. Para ello, creamos una imagen de máscara que será el resultado de aplicar un umbral de valor fijo binario, de manera que el fondo se convierta en negro y el dedo en blanco. Posteriormente podremos usar esta imagen umbral como máscara.

Ilustración 3-2: Máscara de dedo

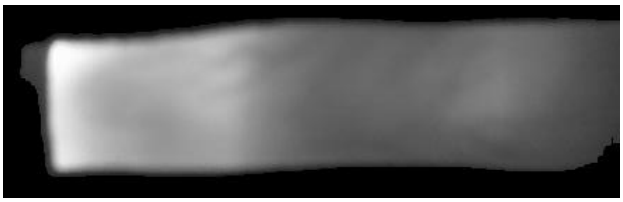


Ilustración 3-3: Dedo umbralizado

3.3.2 Ajuste de contraste

Como se puede observar, la imagen original tiene un contraste con margen de maniobra. Para mejorarlo, usaremos una técnica de igualación de histograma adaptativa. *OpenCV4* [3] incluye el algoritmo del método CLAHE, por lo que simplemente tenemos que crear un puntero de tipo CLAHE, inicializarlo y aplicarlo a la imagen. El resultado obtenido es una imagen en la que es mucho más fácil apreciar la vascularidad del dedo.

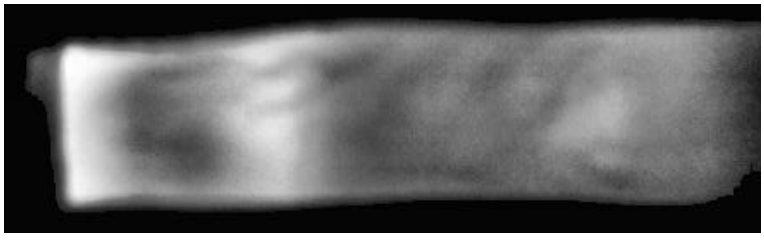


Ilustración 3-4: Resultado de igualación de histograma adaptativa

3.3.3 Filtrado en frecuencia

Con los pasos anteriores, la anatomía vascular del dedo se va viendo de forma más clara, sin embargo, aún debemos seguir procesando la imagen para resaltarlas de una manera más obvia antes de umbralizar. Para ello, podemos realizar un filtrado de alta frecuencia en la imagen. En el proceso de *OpenVein Toolkit* [12] se menciona el uso del filtro de énfasis en alta frecuencia. En este proyecto, además de implementar el mencionado filtro, también planteamos el uso del filtro gaussiano de paso alto, en sustitución del anterior filtro o para el uso conjunto de ambos. Más adelante compararemos los resultados obtenidos.

Para implementar ambos filtros, el proceso es totalmente similar, y, de hecho, únicamente cambia el valor de los parámetros, por lo que crearemos una función que sea capaz de ejecutar ambos tipos de filtrado. El primer paso por realizar es ejecutar la DFT de la imagen que se quiere filtrar, entonces, dependiendo si vamos a implementar un filtro HPF o HFE [13] [14], elegimos parámetros acordes. Primero, realizamos una función de transferencia que aplique un filtro de paso alto de la siguiente manera:

$$H_{HP}(u,v) = 1 - e^{-\frac{\left(u-\frac{N}{2}\right)^2+\left(v-\frac{M}{2}\right)^2}{2\cdot D_0^2}}$$

Donde (u, v) es un píxel cualquiera de la imagen, M y N son el número de filas y columnas de la imagen, respectivamente y D_0 es una cantidad específica no negativa [14].

A continuación, se realiza la operación $G = k_1 + K_2 \cdot H_{HP}$, operación por la cual se obtiene el espectro en frecuencia del filtro que se va a aplicar. A continuación, se aplica el filtro a la transformada de Fourier de la imagen que se desea filtrar, mediante multiplicación píxel a píxel. Al resultado se le realiza una FFTShift[3], y por último, se le realiza la IDFT, se normaliza la salida, y se realiza una igualación de histograma.

Los cambios entre el filtro HPF y HFE son únicamente cambios en las constantes D_0, k_1 y k_2. Para el primer caso, $k_1 < 0$ y $k_2 > 0$. Sin embargo, para otro filtro, ambas constantes deben ser positivas. También es de mención que la constante D_0 debe ser positiva en ambos casos, pero no necesariamente la misma. En las figuras de a continuación se pueden observar los resultados obtenidos con ciertos parámetros tanto como para los filtros individuales como la combinación de ambos. El resultado más valioso parece ser la aplicación de un filtro de énfasis en alta frecuencia seguido de un filtro de paso algo gaussiano.

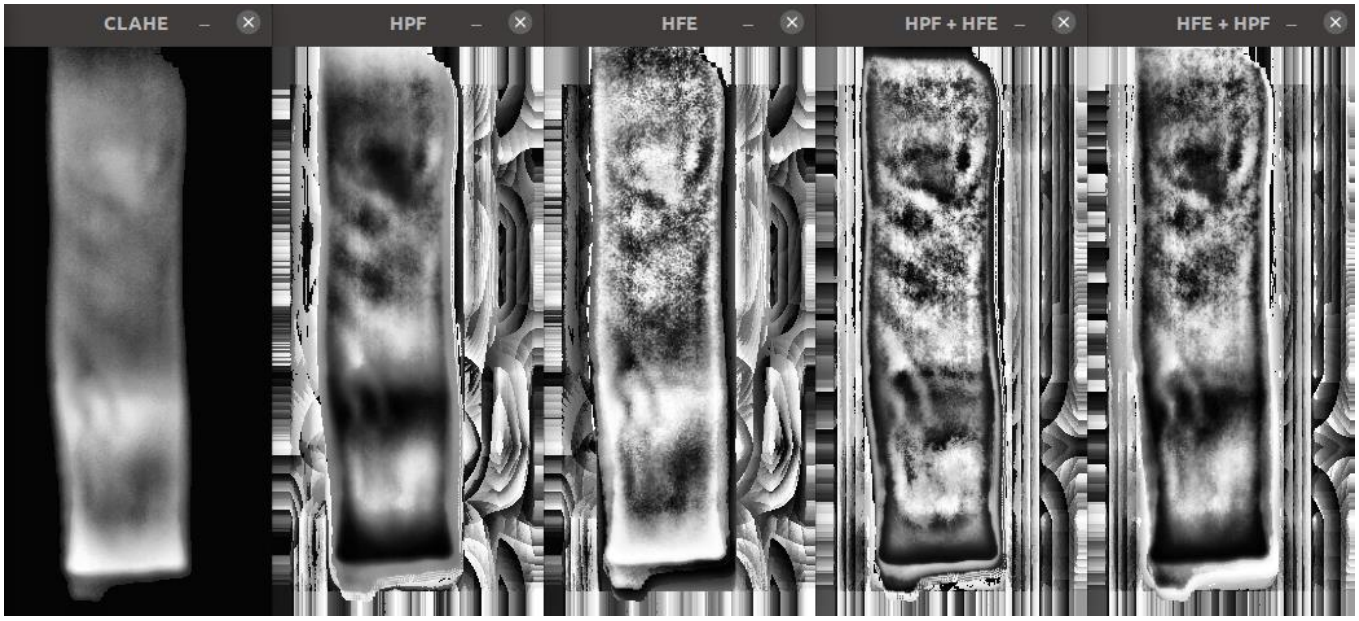


Ilustración 3-5: Aplicación de filtros en frecuencia

[3] La operación de FFTShift recoloca una transformada de Fourier dada desplazando la componente de frecuencia cero al centro de la imagen.

3.3.4 Binarización de la imagen

Para convertir la imagen anterior en una imagen binaria, se debe realizar algún tipo de filtro o umbralizar mediante valores de gris. No obstante, como se puede apreciar, la imagen anterior tiene una cantidad notable de ruido, por lo que esta opción es rápidamente descartada. Finalmente, utilizaremos el filtro circular de Gabor [15] (CGF de ahora en adelante), una variante del filtro de Gabor, utilizado para separación de texturas en imágenes. Este filtro es idóneo, ya que nos separará las venas y arterias del resto mediante valor de gris y además, ruido (textura). Para la implementación, seguimos los pasos detallados en el siguiente artículo [16].

Para utilizar la máscara hallada con el proceso anterior, únicamente es necesaria una correlación, sin necesidad de utilizar el dominio de la frecuencia. Para esto, utilizaremos la parte real de la máscara, ya que se trata de una imagen en el dominio complejo. La parte real es la útil para nuestro propósito, mientras que la parte compleja sirve como detector de bordes. En la siguiente figura se puede observar la parte real y compleja de la máscara del CGF con unos parámetros específicos y de tamaño $40x40$.

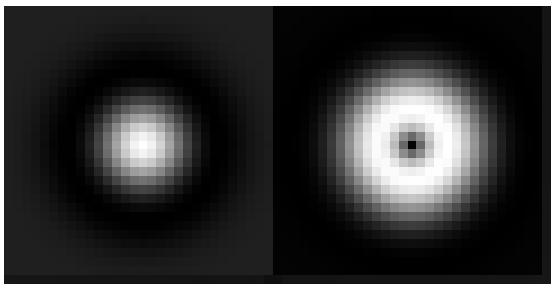


Ilustración 3-6: Máscara CGF: Partes real y compleja

El resultado obtenido, tras realizar una serie de ajustes, es muy prometedor, y mantiene en blanco tanto las venas y arterias, como los bordes del dedo, dejando el resto de la imagen a negro. Al resultado obtenido, se le realiza una operación de cerrado seguido de una de apertura, para corregir desperfectos del filtrado. Más adelante, se podrán obtener puntos característicos para identificar a las personas. En la siguiente imagen se puede observar el resultado obtenido para una configuración de preprocesamiento concreta:

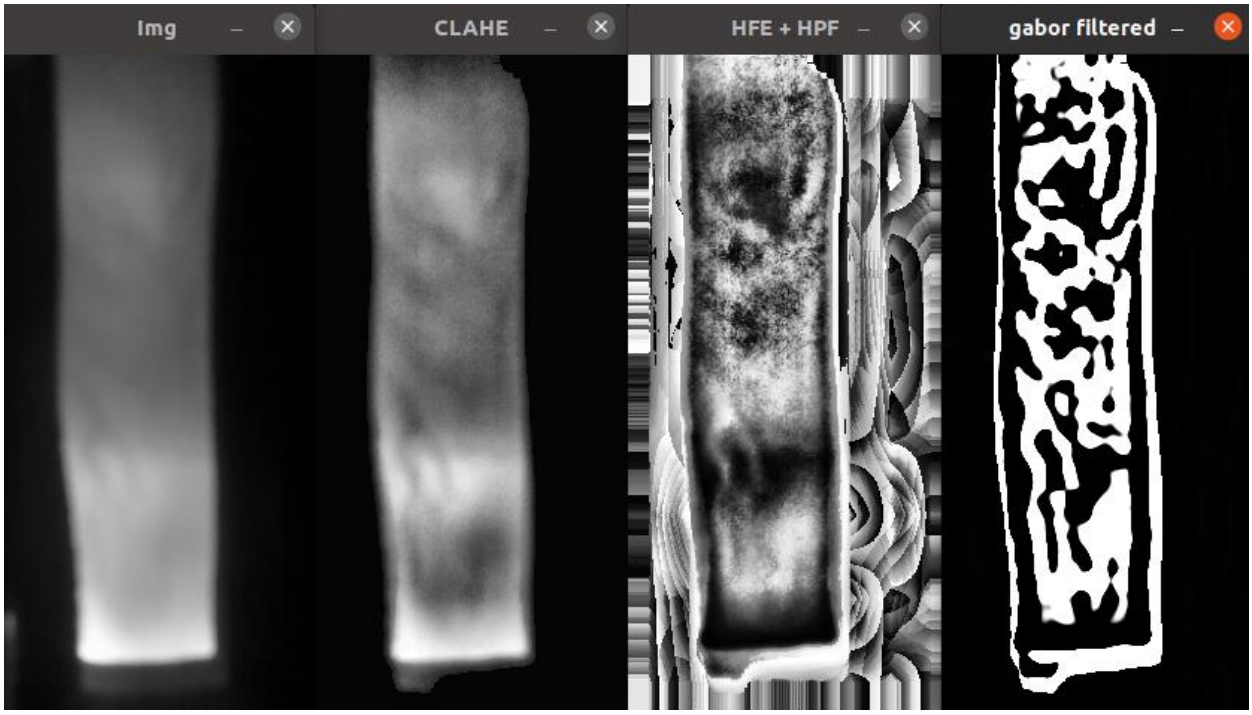


Ilustración 3-7: Aplicación de CGF

3.4. Obtención de puntos característicos

Para la obtención de puntos notables en la imagen binaria anterior, primero vamos a realizar un estudio de cuál es el método que resulta más útil para la aplicación. Es por ello, que a continuación, compararemos entre la utilización de cuatro métodos distintos: SIFT, SURF, ORB y KAZE.

El inconveniente de utilizar el método SURF es que es un algoritmo patentado [17] y por lo tanto, su uso no es gratuito en aplicaciones comerciales. Por otro lado, la patente de SIFT [18] ha expirado recientemente, y por tanto, su uso es gratuito. Además, exististen otras alternativas, como el detector y descriptor ORB de *OpenCV* [3] y el detector y descriptor KAZE.

Para la implementación de estos métodos en el código, necesitaremos hacer uso tanto de la librería *features2d.hpp* como de la librería *xfeatures2d.hpp* de *OpenCV* [3], que contienen los métodos de SIFT, SURF, ORB y KAZE. A continuación, creamos funciones para cada detector, muy similares entre ellas. Cada función devolverá una tupla de vector de puntos clave y matriz descriptora, con la única entrada de la imagen de la que se quieren detectar los puntos notables. Internamente, se crea un puntero del método a utilizar y se configura para encontrar alrededor de 1000 puntos. Entonces, se detectan y computan estos puntos y se devuelven en una tupla. En el caso de SURF, utilizaremos un hessiano mínimo de 5000, y en el caso del descriptor KAZE, utilizaremos la configuración por defecto, a la que únicamente le cambiaremos el parámetro *upright* de *false* a *true*.

Pero antes de aplicar estos métodos a la imagen, debemos tener en cuenta que los extremos de ésta contienen o pueden contener información errónea, debido a una mala umbralización o incorrecta iluminación. Es por esto por lo que eliminaremos estas zonas con una máscara binaria, de manera que las partes no útiles se convertirán a un valor 0, es decir, negro. Es necesario recalcar que la imagen no se recorta; el tamaño de la imagen es el mismo que antes de aplicar la máscara. Este proceso se realiza de esta manera para que el detector pueda incluir puntos en los bordes exteriores del dedo, que sí es información valiosa.

Para poder verificar si los resultados obtenidos con los detectores y descriptores anteriores son válidos y mejores que los demás, necesitamos además implementar un comparador de características para comprobar la correspondencia de los puntos hallados entre dos imágenes. Debido a la variedad de descriptores utilizados, necesitaremos dos comparadores distintos, un comparador de fuerza bruta para el método ORB que utilice la distancia de *Hamming* como parámetro, ya que ORB es un descriptor binario y por tanto esta distancia es la adecuada. Para el resto de los métodos, también podría utilizarse un descriptor de fuerza bruta que funcionara con la distancia euclídea; sin embargo, utilizaremos un comparador FLANN, al ser más rápido que el caso anterior. Además, en todos los casos filtraremos las correspondencias halladas con la prueba de relación de distancia propuesta por David G. Lowe [19] con un ratio umbral de 0.72. Tras esto, sólo resta crear una imagen con la correspondencia de puntos y mostrarla en pantalla.

Además de utilizar el ratio de Lowe, implementamos un método que descarte los emparejamientos que superen una inclinación umbral, ya que si es el mismo dedo, los emparejamientos deberían realizarse en línea recta. Para ello, a cada emparejamiento considerado válido por las normas descritas anteriormente, le calculamos el ángulo de inclinación con respecto a $y = 0$ mediante la arcotangente.

Por lo tanto, el proceso seguido es el siguiente: Primero se leen las imágenes que se van a comparar y se preprocesan para conseguir la imagen binaria. Entonces, se obtienen los puntos notables y descriptores en el par de imágenes con todos los métodos descritos anteriormente. Tras esto, únicamente resta aplicar los comparadores; de fuerza bruta en el caso del descriptor ORB y comparadores basados en FLANN para el resto de los métodos. A continuación, en las siguientes figuras, se muestran los resultados obtenidos para los cuatro métodos probados.

Como se puede observar en las ilustraciones de más adelante, el método que otorga el mejor resultado se trata del detector y descriptor KAZE [4], seguido de ORB. En la figura es capaz de encontrar 76 emparejamientos de aproximadamente 850 puntos en cada imagen, ORB empareja 33 puntos y tanto SIFT como SURF, únicamente 13 puntos característicos. Los métodos SIFT y SURF se quedan muy atrás respecto a sus rivales, aunque si no hubiéramos decidido validar emparejamientos por inclinación, el método ORB quedaría sin duda en el último puesto, debido a la enorme cantidad de falsos emparejamientos que encontraría. En conclusión, de ahora en adelante el detector de puntos característicos y descriptor utilizado será el método KAZE.

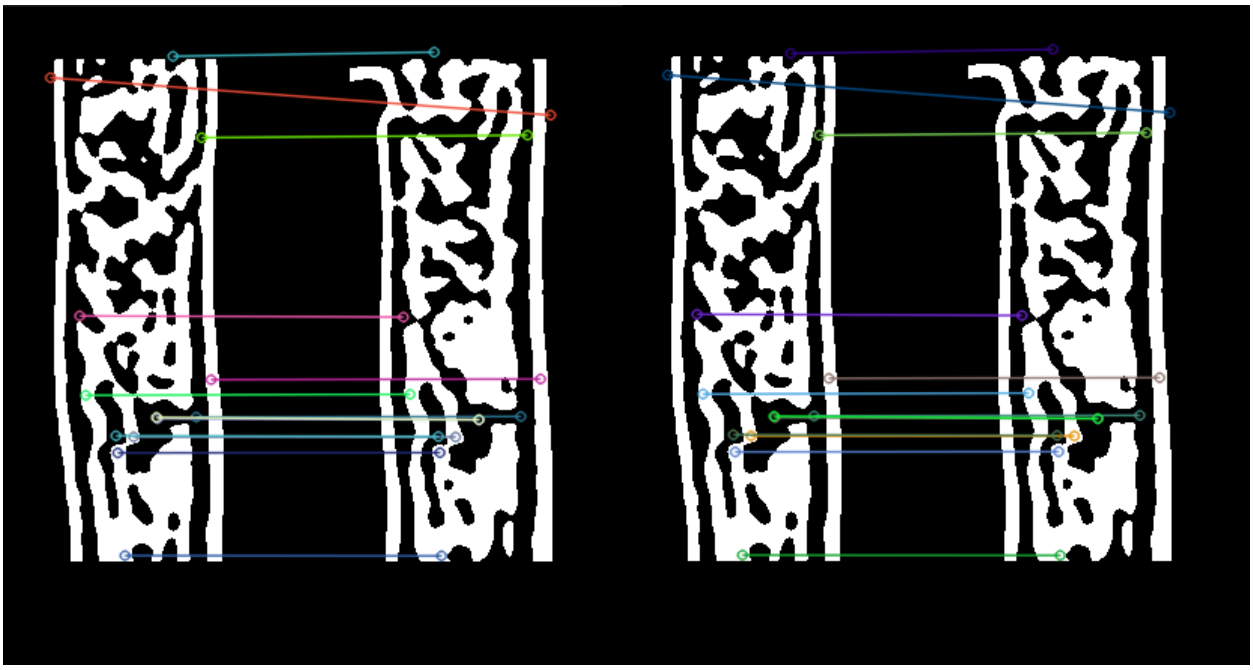


Ilustración 3-11: Comparación características SIFT

Ilustración 3-10: Comparación características SURF

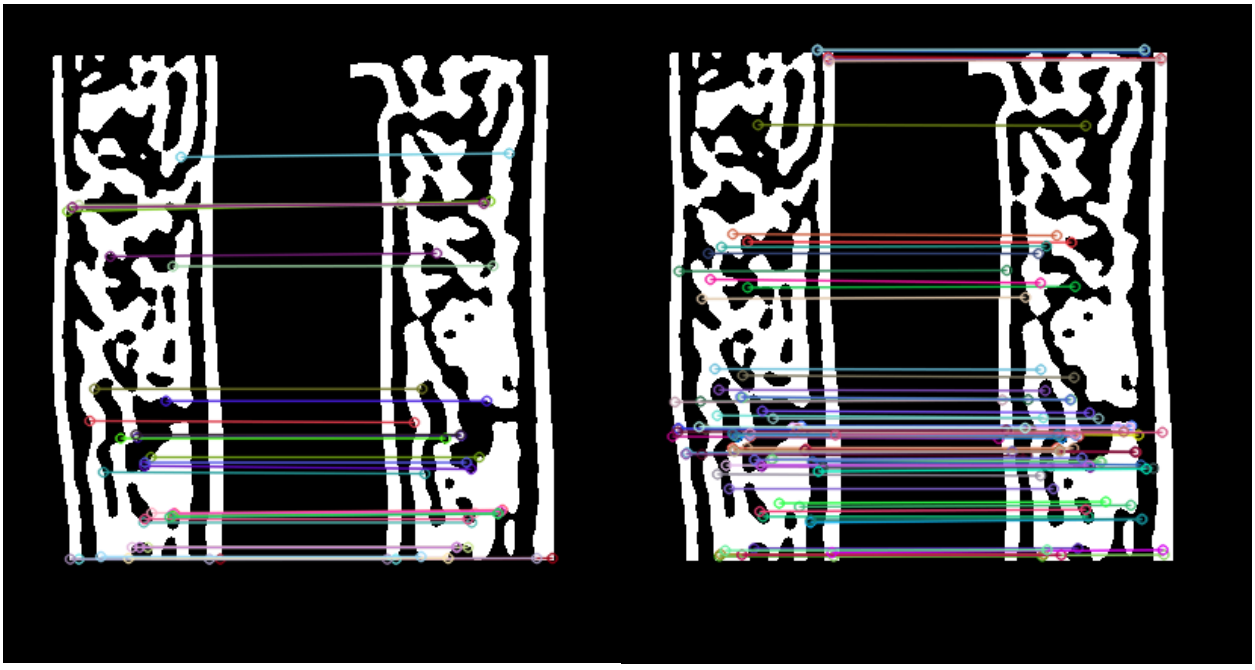


Ilustración 3-8: Comparación características ORB

Ilustración 3-9: Comparación características KAZE

3.5. Base de datos

Una vez obtenidas las características del dedo de una persona, necesitamos compararlas con otras para medir el parecido entre ellas. Para poder conseguir esto, es necesario almacenar de alguna manera las características del dedo de los usuarios habilitados. La solución más lógica es crear una base de datos, que de alguna manera, contenga la información del usuario tal como su nombre y características de anatomía vascular.

La solución más sencilla consistiría en crear una tabla que contuviera únicamente tres entradas: un número de identificación único, el nombre de dicha persona y la imagen binarizada de su anatomía vascular. Posteriormente, al recuperar los datos en el programa, únicamente habría que volver a obtener las características a partir de la imagen. No obstante, esto puede representar un problema de seguridad, ya que en caso de robo de la base de datos, el criminal tendría en su poder las imágenes de la anatomía vascular de las personas registradas. Es por ello por lo que optaremos por otra solución más compleja.

Una posible solución podría consistir en encriptar las imágenes de la base de datos anterior, sin embargo, esta puede ser una solución enormemente compleja, por lo que optaremos por almacenar directamente las características de la persona en lugar de una imagen. De esta manera, eliminamos el problema de seguridad, ya que el vector de puntos clave y descriptor de las características no suponen ningún dato sensible para el usuario, debido a que no es posible obtener la anatomía vascular a partir de estos datos. Por tanto, crearemos una tabla que contenga los siguientes campos: Número de identificación único, nombre del usuario, número de puntos clave, los puntos notables codificados en binario y, por último, el descriptor codificado en binario.

Para el manejo de la base de datos, utilizaremos un motor de bases de datos SQL[4]. Para su implementación, utilizaremos la conocida librería de SQLite [20], en su versión 3.36.0. Para ello, se instala desde la consola de Linux con el siguiente comando:

```
$ sudo apt-get install sqlite3 libsqlite3-dev sqlitebrowser
```

Posteriormente, tendremos que incluir la librería en el código mediante $\#include\ "sqlite3.h"$ y añadir el parámetro $-lsqlite3$ a las *flags* del compilador de C++. Tras estos pasos, la librería queda instalada y está totalmente lista para su uso.

3.5.1 Creación de DB

El primer paso consiste en crear la base de datos desde la aplicación *DB Browser for SQLite* instalada en el comando anterior. Una vez dentro de este programa, crearemos una nueva base de datos dándole al botón *New Database*. Entonces, le asignaremos un nombre (*database.db* en este caso) e inmediatamente tras crearla, procederemos a crear una tabla. En este caso, le daremos el nombre de data a la tabla y crearemos cinco campos mediante el botón *Add*. El primer campo lo llamaremos "id", será de tipo INTEGER y además, marcaremos las flags de NOT NULL(NN), PRIMARY KEY(PK) y AUTO INCREMENT(AI). El segundo campo lo llamaremos "name" y será de tipo TEXT. El tercer campo lo denominaremos "nKeypoints", será de tipo INTEGER y lo utilizaremos para almacenar el número de puntos clave que hay almacenados en el siguiente campo, que llamaremos "keypoints" y asignaremos un tipo BLOB. El último campo, lo denominaremos "descriptor" y también será de tipo BLOB. Entonces, pulsaremos el botón *Ok* y tendremos la tabla creada y por tanto, la base de datos lista.

Repasando lo que acabamos de realizar, el campo "id" será un número entero no nulo que se irá incrementando automáticamente con cada entrada que realicemos a la base de datos y será el principal identificador de cada entrada en la tabla. El campo "name" almacenará únicamente el nombre del usuario. El campo "nKeypoints" almacenará el número entero de puntos clave hallados mediante el programa, número muy importante para recuperar el resto de los datos. Por último, los dos últimos campos serán de tipo BLOB, ya que es un tipo que se utiliza para almacenar datos binarios y es exactamente lo que necesitamos para guardar los puntos clave y descriptor serializados.

[4] El Structured Query Language (SQL) es un lenguaje de dominio específico diseñado para administrar y recuperar información de sistemas de gestión de bases de datos relacionales. [26]

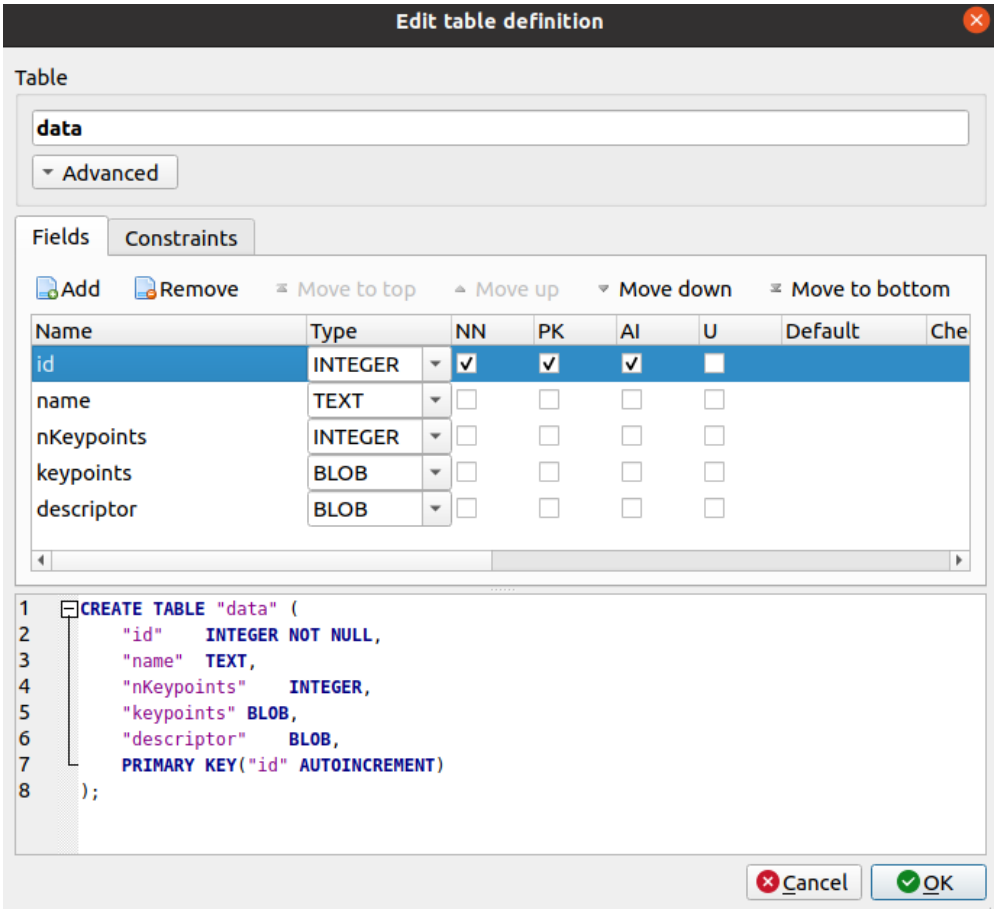


Ilustración 3-12: Creación de una tabla en DB Browser for SQLite

3.5.2 Escritura de datos

Para la escritura de datos, primero debemos solucionar el problema más complejo, serializar el descriptor. OpenCV [3] ya incluye una función para codificar imágenes en formatos conocidos. Sin embargo, sólo permite imágenes tipo *unsigned char*, es decir, imágenes cuyos píxeles tengan valores enteros comprendidos entre 0 y 255. Sin embargo, el descriptor de las características se trata de una imagen de tipo *float*. Es decir, que sus píxeles son valores decimales no necesariamente positivos y en este caso, cada píxel se corresponde con 4 bytes, tamaño del tipo *float*, a diferencia del caso anterior, que únicamente utiliza un byte por píxel. Por tanto, tendremos que hallar una manera propia de serializar la imagen para poder escribirla como una cadena de bytes en la base de datos.

La función de codificación consta únicamente de un *buffer* para el cual reservamos la memoria necesaria mediante *malloc*, para así liberar la memoria con *free* o *delete[]* más adelante. Entonces, recorremos cada píxel del descriptor individualmente y convertimos el valor *float* obtenido en una cadena de bytes de longitud cuatro. Entonces, recorremos esta cadena valor a valor y almacenamos los valores leídos en el *buffer* en orden. Tras estos bucles, únicamente resta devolver el *buffer* creado, quedando la imagen codificada en él.

Una vez solucionado el problema anterior, también necesitamos serializar el vector de puntos clave, la cual realizaremos de manera similar. Primero crearemos un *buffer* al cual le asignaremos tamaño para contener todos los puntos, entonces convertimos individualmente cada *Keypoint* a una cadena de bytes y, al igual que en el caso anterior, la recorremos añadiendo los valores leídos en orden al *buffer*. Entonces, ya en este punto, hemos resuelto ambos problemas de serialización[5] y por tanto, podemos escribir los datos en una nueva entrada de la base de datos.

[5] La serialización es el proceso de convertir un objeto en una secuencia de bytes para almacenarlo o transmitirlo a la memoria, a una base de datos o a un archivo. El proceso inverso se denomina deserialización. [27]

Para escribir los datos en la base de datos, debemos realizar ahora una serie de procesos, tales como abrir la misma en el programa, mediante el uso de la función $sqlite3\_open\_v2$. Una vez abierta la base de datos, antes de escribir una nueva entrada debemos comprobar si el usuario que se quiere introducir ya se encuentra en la base de datos. Para ello, crearemos una consulta que extraiga todas las entradas de la *database*. La *query* que necesitamos para esto es la siguiente:

```
SELECT * FROM data
```

Entonces, tras ejecutar la consulta, iteramos a través de todas las filas devueltas y comprobamos si el nombre de la entrada que se quiere introducir coincide con alguna de las existentes, devolviendo el número 1 en caso afirmativo. Si, por el contrario, el usuario no existía en la base de datos, termina la ejecución de la consulta y entonces se prepara para escribir los datos.

El primer paso para escribir los datos es serializar el vector de puntos clave y descriptor tal y como se ha descrito anteriormente. Una vez tenemos ambos *buffers*, continuamos y creamos una nueva consulta SQL para introducir los datos en una nueva columna. Esta nueva consulta contendrá parámetros representados por signos de interrogación para más adelante, introducir los datos antes de ejecutarla. La *query* es la siguiente:

```
INSERT INTO data(id, name, nKeypoints, keypoints, descriptor) VALUES(NULL, ?, ?, ?, ?)
```

Tras enlazar correspondientemente los datos a los parámetros anteriores, ejecutamos la consulta y si no devuelve error, terminamos la ejecución y cerramos la base de datos. En este punto, la nueva entrada ha sido introducida en la base de datos y sólo resta liberar la memoria utilizada para los *buffers*. En todo momento, si ocurre un error en cualquier paso, la función devolverá error, deteniendo la ejecución del programa. Si la función sale correctamente, devuelve el valor 0 indicando que no ha ocurrido error alguno.

3.5.3 Lectura de datos

Previamente hemos observado que hemos necesitado serializar los datos para poder escribirlos en la base de datos, por lo tanto, ahora que tenemos que leer esos datos escritos previamente, vamos a necesitar deserializarlos para poder almacenarlos en los tipos correspondientes. El proceso seguido es estrictamente el inverso y una vez comprendido el anterior, este procedimiento es simple. Comenzamos explicando la deserialización del descriptor.

La función encargada de convertir la cadena de bytes del descriptor a una imagen necesitará dos parámetros. El *buffer* de datos obtenido de la base de datos y el número de puntos clave, también obtenido de la base de datos. Entonces, crearemos una imagen en negro de tipo *float* que tenga como filas el número de puntos notables y 64 columnas, longitud de la cadena descriptora. Entonces, recorremos individualmente cada píxel de la nueva imagen y para cada uno, vamos sacando cuatro bytes del *buffer*, los cuales reinterpretaremos como *float* e introduciremos el valor obtenido en el píxel de la imagen. Entonces, destruimos el vector temporal utilizado para almacenar los cuatro bytes que componen el número flotante. El proceso se repite hasta que la imagen ha sido completada y entonces, se devuelve, obteniendo correctamente el descriptor.

Para deserializar el vector de puntos clave, primero obtenemos cuántos hay y el *buffer* de datos escrito en la base de datos. Entonces, creamos un vector de *KeyPoint* vacío y creamos un bucle que se repetirá tantas veces como puntos notables haya. Dentro de este bucle, crearemos un vector temporal en el que almacenaremos los 28 bytes que componen un punto clave y los reinterpretaremos a su tipo correcto, entonces, lo introducimos en el vector de puntos notables y destruimos el vector temporal. Seguimos repitiendo el bucle hasta que no haya más puntos que leer en el *buffer*. De esta manera, al terminar el bucle, tendremos un vector con todos los puntos clave originales.

Una vez conocemos como reinterpretar estos *buffers* para obtener los tipos de datos adecuados, podemos leer la entrada deseada de la base de datos. Para ello, crearemos una función que devuelva en una tupla el nombre del usuario, su vector de puntos clave y su descriptor, con el único parámetro del número de identificación de la entrada (id). Además, añadiremos un parámetro adicional optativo con el que se da la posibilidad de pasar un puntero a una base de datos abierta, de manera que esta función se podría llamar desde otra función que, a su vez, está haciendo ya uso de la base de datos. Más adelante se explicará en que caso utilizamos esta posibilidad de la función.

El procedimiento que seguir en la función es similar a la de escritura de datos, siendo el primer paso abrir la base de datos. No obstante, en este caso puede no ser necesario abrir la base de datos. Si se detecta que se ha pasado un puntero a una base de datos, ésta no se volverá a abrir de nuevo, ya que en ese caso, devolvería un error. Entonces, creamos una consulta para obtener los datos de una entrada específica con el único parámetro de id. La consulta queda de la siguiente manera:

```
SELECT * FROM data WHERE id=?
```

Tras preparar la consulta anterior, enlazamos el parámetro de la consulta con el valor de *id* introducido en los parámetros de la función. Una vez realizado este proceso, ejecutamos la consulta y leemos la fila devuelta si la hay. En caso de que no exista la fila, la función devolverá un error. Si por el contrario, sí existe una fila con ese número de identificación, se leen todas las columnas (excepto la de *id*) y se deserializan el vector de puntos clave y el descriptor. Entonces, terminamos la consulta, cerramos la base de datos, limpiamos la memoria de los *buffers* y devolvemos una nueva tupla con el nombre, vector de puntos notables e imagen descriptora de la anatomía vascular del dedo de la persona con el *id* introducido.

3.5.4 Búsqueda de la mejor coincidencia

Realmente, lo que nos interesa para el funcionamiento del programa, es que éste sea capaz de encontrar la mejor coincidencia entre todas las entradas de la base de datos, devolviendo la entrada con la que tenga mayor similitud. Esto lo haremos en base al número de puntos que el dedo introducido es capaz de emparejar con las entradas de la base de datos. Para ello, crearemos una función que obtenga todas las filas de la base de datos y realice esta comprobación para cada una de ellas, devolviendo finalmente la entrada con la que el usuario tenga mayor correspondencia de puntos.

La función creada devolverá una tupla que contenga el nombre del usuario de la entrada, su vector de puntos clave y su descriptor, mientras que el único parámetro de entrada será una tupla que contenga las características del usuario que introduce el dedo, es decir, su vector de puntos notables y su descriptor. A continuación realizaremos una descripción del funcionamiento de esta rutina.

El primer paso que debemos realizar es la apertura de la base de datos y preparar una consulta para obtener todas las filas y columnas de la base de datos, tal como la primera *query* del punto **3.5.2**. Entonces, creamos una tupla del tipo de salida de función para contener la mejor coincidencia y una variable de tipo entero para almacenar el número de correspondencias de la mejor coincidencia, inicialmente cero. En este momento, comenzamos a leer todas las entradas de la base de datos individualmente. Esto lo conseguimos leyendo el *id* de la fila y utilizando la función del punto **3.5.3**, a la que le pasaremos el parámetro opcional de puntero a la base de datos, ya que la tenemos abierta en esta función. Esto nos devolverá una entrada de la base de datos y la compararemos con nuestro comparador de tipo FLANN con los *features* introducidos en la función. Ahora, si la cantidad de puntos emparejados es mejor que la actual, actualiza tanto el valor de esta variable como la variable de entrada de mejor coincidencia. Este proceso se repite hasta que no haya más entradas que leer y cuando termine el bucle, se devuelve la entrada de mejor coincidencia o un error en caso de que no se haya conseguido emparejar ningún punto.

Para comprender de manera más sencilla el funcionamiento de la rutina anterior, en la figura de más adelante se muestra un diagrama de flujo del proceso seguido por la función.

En este punto, ha quedado explicado todo lo relacionado con la gestión de la base de datos, tanto la parte de escritura y lectura como la parte de serialización y deserialización. En el siguiente apartado se explicará todo lo relacionado con la interfaz del programa principal, más de cara al usuario que de diseño *Software*. Por supuesto, todo el código desarrollado tanto para este como para el resto de los apartados puede consultarse en los anexos que se encuentran al final de este documento.

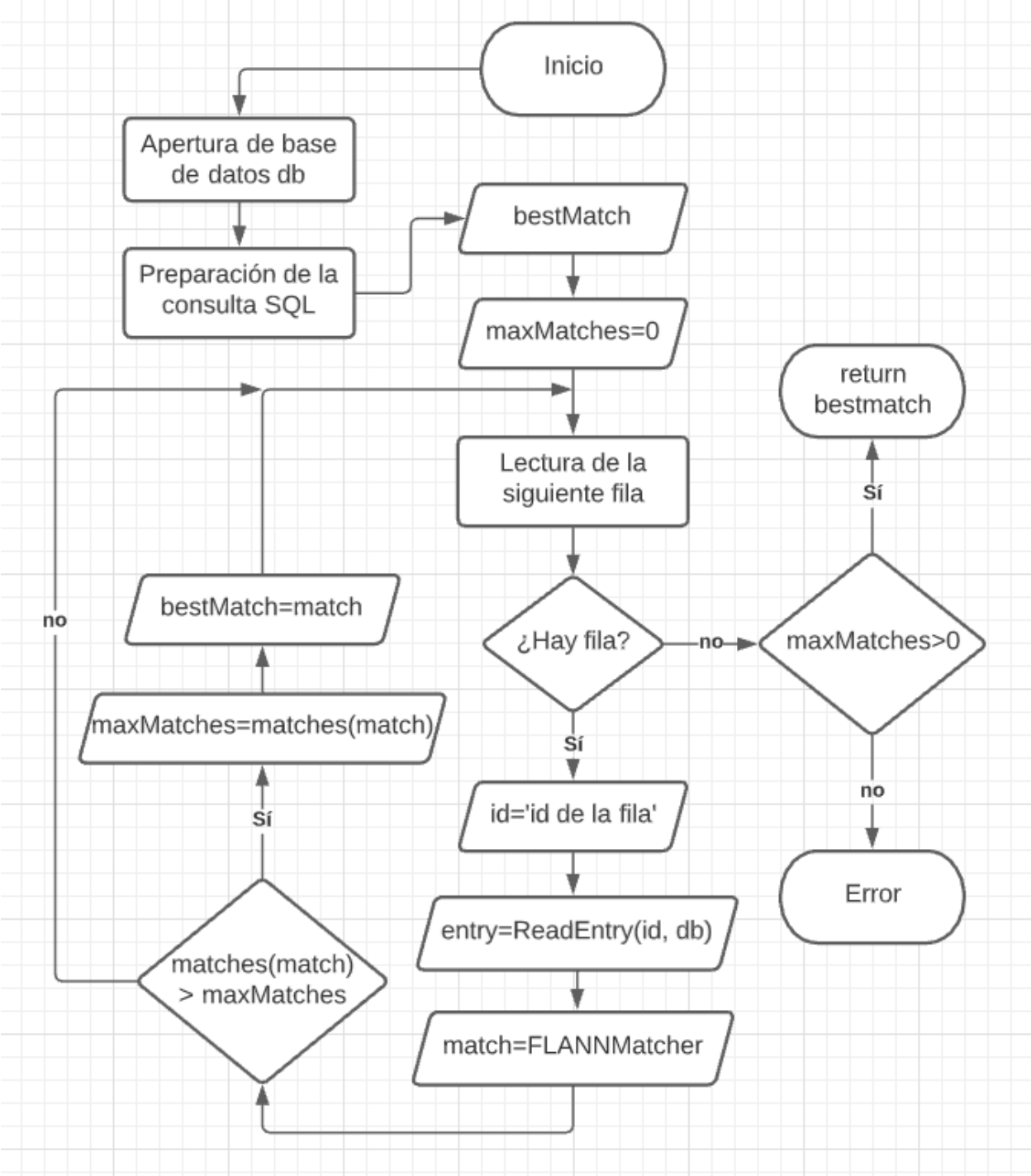


Ilustración 3-13: Diagrama de flujo de función de búsqueda de mejor coincidencia

4 INTERFAZ DE USUARIO

En la vida hay algo peor que el fracaso: El no haber intentado nada.

Franklin D. Roosevelt

El objetivo de este apartado es conseguir una interfaz simple e intuitiva para el usuario, en el que éste pueda elegir registrarse en la base de datos o identificarse. Por supuesto, por motivos de seguridad, lo lógico es que esta interfaz no la controle el usuario que se dispone a utilizar el dispositivo sino una tercera persona encargada de la seguridad del sistema que se pretende proteger con la biometría vascular. Esta decisión se deja a cargo del propietario del dispositivo.

Para el desarrollo de esta interfaz, decidimos utilizar QT Creator, un IDE para crear aplicaciones gráficas de ventanas o *forms* parecido a Visual Studio. Programaremos la aplicación en C++ junto con la API QT dentro del mencionado IDE. Esta interfaz será muy simple y únicamente se compondrá de dos botones y una caja para introducir texto. Con el IDE QT Creator, podemos diseñar la interfaz de forma gráfica, por lo que crearemos un nuevo proyecto que denominaremos TFG_GUI y nos pondremos a ello.

4.1. Creación de la interfaz gráfica

Una vez hemos creado el proyecto, para poder editar la ventana principal del programa, debemos abrir el desplegable del proyecto y hacer doble clic sobre el archivo de extensión .ui, que se encuentra en la dirección $\boldsymbol{TFG\_GUI} \rightarrow Forms \rightarrow mainwindow.ui$. Entonces, se abrirá el modo de diseño de ventanas.

El primer paso que vamos a seguir es eliminar los objetos que no vamos a usar en la pestaña *Object* de la derecha. En este caso, únicamente mantendremos *centralwidget* y *statusbar*. A continuación, fijaremos el tamaño de la ventana en la pestaña en la propiedad *Geometry*. Ajustaremos los parámetros *witdth* y *height* a 400 y 120, respectivamente. Además, ajustaremos las propiedades de *minimumSize* y *maximunSize* a los mismos valores, con el fin de evitar que la ventana sea de tamaño ajustable. Por último, cambiaremos el nombre de la ventana en la propiedad *windowTitle* a $Security\ System$.

Una vez tenemos la ventana bien ajustada, podemos comenzar a añadir los demás elementos. Comenzaremos por los botones, los cuales colocaremos aproximadamente centrados. Haciendo doble clic en ellos, podemos cambiar la etiqueta de éstos, ajustándolas a $Register$ y $Login$. Además, debemos añadir la acción de clic para más tarde programar la acción que ocurrirá cuando los botones se pulsen. Para ello, hacemos clic derecho en el botón y seleccionaremos *Go to slot*. En este menú, seleccionamos *clicked()* y pulsamos el botón OK.

A continuación, añadiremos un *lineEdit*, que funcionará como entrada y salida de texto para el usuario. En este objeto se indicará al usuario si ha sido identificado entre otras alertas. También se escribirá el nombre de usuario con el que se creará el registro en la base de datos si se elige la opción de registro. Colocaremos este componente encima de los botones, ocupando la mayor anchura posible. Además, a la izquierda de este objeto, colocaremos un *label* en el que colocaremos el texto $User$: para facilitar la comprensión de la interfaz al usuario. En la figura de a continuación, se muestra la ventana creada.

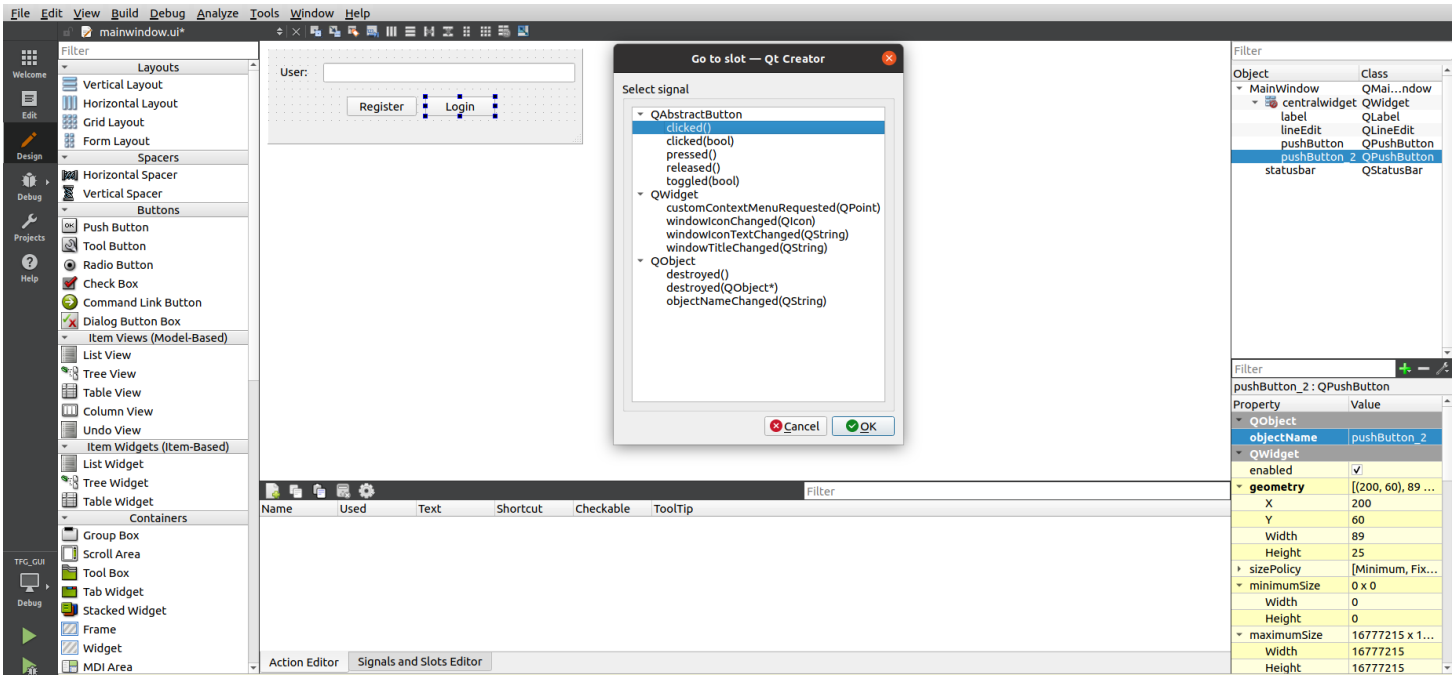


Ilustración 4-1: UI creada en QT Creator

4.2. Programación de la interfaz

En principio, este apartado puede parecer totalmente trivial. Sin embargo, es necesario resolver un problema de comunicación entre la interfaz y la aplicación que ejecuta todo el proceso de identificación. Recordemos que la UI y el programa que ejecuta OpenCV [3] son dos procesos distintos, y se necesita una comunicación bidireccional. Debido a esto, tendremos que hacer modificaciones en el programa principal.

La solución tomada para resolver este problema es la siguiente: La aplicación UI ejecutará en un nuevo proceso el programa principal al cual se le pasarán argumentos que modelarán el funcionamiento de éste. Sumado a esto, se utilizarán señales de tiempo real que se esperarán síncronamente en ambos procesos para la sincronización de los mismos. Se enviarán señales SIGRTMIN con datos desde la aplicación principal a la UI y ésta responderá con otra señal SIGRTMIN al proceso como ACK si espera recibir más datos. Los parámetros con los que se determina la funcionalidad del programa principal son los siguientes:

- Parámetro 1: Número entero igual a cero o uno. En caso de que sea cero, se realizará el proceso de identificación (login). Si es igual a uno, se realizará un nuevo registro en la base de datos. En caso de que no sea ninguno de estos dos números, la aplicación saldrá con error.
- Parámetro 2: Este parámetro es el *Process ID* del proceso que ejecuta la UI.
- Parámetro 3: Este parámetro sólo se usa en la operación de registro y contiene el nombre de usuario que se quiere registrar.

Si se ejecuta el programa sin pasarle ningún parámetro, realizará una prueba consistente en mostrar una imagen de la anatomía vascular del usuario. No obstante, la aplicación de la UI no contempla esta posibilidad y si se desea este funcionamiento, debe ejecutarse en un terminal.

Una vez realizado el proceso de identificación o registro, el programa devuelve datos mediante señales de tiempo real a la aplicación de la UI. Si se deseaba un registro, se enviará una señal con dato 1 si se ha llevado a cabo exitosamente o 0 si ya existía un usuario registrado con ese nombre en la base de datos. Si en cambio se deseaba realizar un *login*, el programa devolverá 0 mediante una señal si no se ha podido identificar al usuario o la longitud del nombre en caso contrario. Si se pudo identificar al usuario, además, esperará una señal de ACK del programa UI y comenzará a enviar carácter a carácter el nombre del usuario identificado mediante el uso de señales SIGRTMIN con datos. Antes de enviar el siguiente carácter también esperará un ACK.

De esta manera, se consigue la comunicación entre ambos procesos y se logra el funcionamiento del sistema. El código desarrollado para este apartado se encuentra en los anexos. El resultado de ejecutar la aplicación principal sin argumentos es el siguiente:

Ilustración 4-2: Ejecución de programa sin argumentos

4.3. Guía de usuario

La interfaz de usuario consta de muy pocos elementos, haciéndola no sólo simple visualmente, sino también a la hora de ser utilizada por el usuario. Consta únicamente de dos botones y una caja de introducción de texto, por lo que el funcionamiento es incluso predecible. No obstante, explicaremos detalladamente el uso en este apartado.

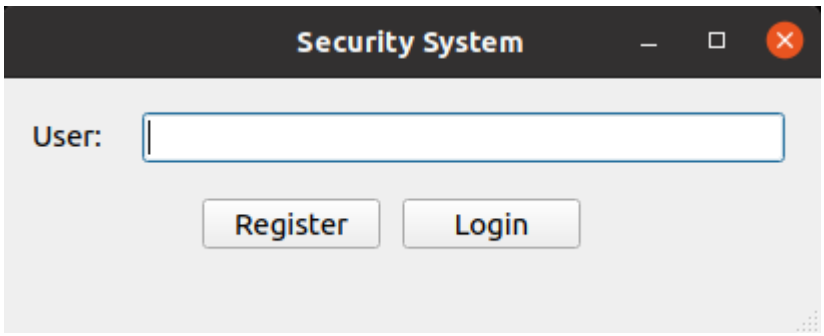


Ilustración 4-3: Aplicación UI en Ubuntu

En esta interfaz, el usuario puede realizar únicamente dos opciones: Registrar un nuevo usuario o tratar de identificar a una persona. Para ello, existen los dos botones ubicados en la parte inferior de la ventana. El botón *Register* es utilizado para registrar un usuario, y el botón *Login* es utilizado para identificación de usuarios. La caja de texto sirve tanto como entrada como salida de texto. La salida de texto es utilizada en ambos procesos e indica, en caso de registro, si el usuario se ha registrado correctamente o, si por el contrario, ya existía un

usuario registrado con ese nombre. En caso de identificación, la salida de texto sirve para indicar si no se ha podido identificar al usuario o, en caso contrario, mostrar el nombre del usuario identificado. Como entrada de texto, sirve únicamente para indicar el nombre con el que se va a registrar al usuario si se pulsa el botón de registro. Si esta caja está vacía, también colocará un mensaje de error indicando que el nombre no es válido.

Por lo tanto, si se desea registrar un nuevo usuario, se debe seguir el siguiente proceso:

- El usuario introduce su dedo índice en el sistema físico hasta que toque la amortiguación situada al fondo, aproximando el dedo hacia la parte superior de la ranura en la medida de lo posible.
- El administrador (persona a cargo de la UI), introduce el nombre de la persona en la caja de texto.
- El administrador presiona el botón *Register*.
- A continuación, se pueden dar dos situaciones:
    - Aparece el mensaje *User registered successfully*, lo que quiere decir que el usuario ha sido registrado exitosamente y no son necesarios más pasos.
    - Aparece el mensaje *User is already registered*, lo que quiere decir que ya existe un usuario registrado con ese nombre y por tanto se debe utilizar otro nombre o negar el registro.

Si se desea identificar al usuario, se sigue el siguiente proceso:

- El usuario introduce su dedo índice en el sistema físico hasta que toque la amortiguación situada al fondo, aproximando el dedo hacia la parte superior de la ranura en la medida de lo posible.
- El administrador presiona el botón *Login*.
- A continuación, se pueden dar dos situaciones:
    - Aparece el mensaje *User XXXX successfully logged in*, indicando que el usuario ha sido identificado correctamente y no son necesarias más acciones.
    - Aparece el mensaje *User is not registered*, indicando que no se ha encontrado una coincidencia fiable en la base de datos, por lo que se puede volver a intentar la identificación o registrar al usuario si no estaba registrado.

Para ejecutar esta interfaz, creamos un archivo .sh en el escritorio del sistema *Raspberry Pi* que ejecute la aplicación. El código es muy simple:

```
#!/bin/bash
cd ~~/Documents/TFG/
./TFG_GUI
```

Basta con hacer doble clic a este archivo que denominaremos *SecuritySystem.sh* en el escritorio para ejecutar la interfaz gráfica. Podemos elegir ejecutarlo en una consola si deseamos la salida de depuración o sin el terminal si no se desea la salida de *debug*. A continuación se muestran ejemplos de los mensajes mencionados anteriormente:

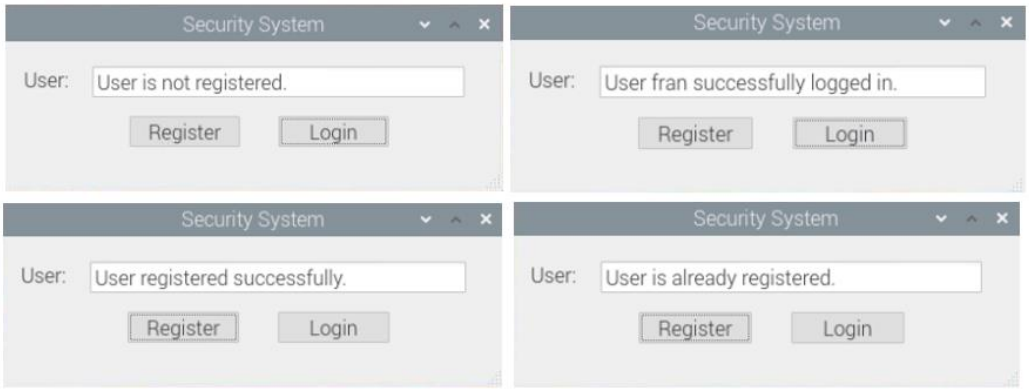


Ilustración 4-4: Mensajes de aplicación UI en Raspbian

5 RESULTADOS OBTENIDOS

Perder la paciencia es perder la batalla.

Mahatma Gandhi

Tras todo lo explicado en apartados anteriores, podemos llegar, al fin, a un análisis de los resultados obtenidos. Resumiendo, en este punto, tenemos un sistema basado en *Raspberry Pi 4* con *OpenCV4* [3] cuyo objetivo es obtener una imagen binarizada de la anatomía vascular del dedo índice de una persona cualquiera. Además, es capaz de obtener puntos característicos y cotejarlos con otros en una base de datos de *SQLite3* [20], devolviendo la entrada con la que tiene mayor coincidencia, identificando así a una persona.

Por supuesto, todo lo anterior funciona de forma teórica y únicamente ha sido probado con un conjunto extremadamente pequeño de usuarios e imágenes sobre el sistema Ubuntu, por lo que es necesario realizar un conjunto de pruebas más amplio analizando los ratios FAR y FRR, con especial hincapié en el primero de ellos. Por tanto, a continuación, se realizarán pruebas para una base de datos con 5 usuarios con distintas configuraciones, con el fin de comprobar cuál es la más adecuada. No obstante, aunque existan cinco usuarios, comprobaremos la coincidencia únicamente con cuatro, por lo que el quinto únicamente influirá en el detalle de FAR. Lo ideal es conseguir un FAR y FRR lo más bajos posibles. Realizaremos 50 pruebas por usuario.

Umbral	**Usuario 1**		**Usuario 2**		**Usuario 3**		**Usuario 4**		**Usua. 5**
	FAR	**FRR**	**FAR**	**FRR**	**FAR**	**FRR**	**FAR**	**FRR**	**FAR**
40	0%	84%	0%	96%	0%	100%	0%	S/ datos	0%
30	0%	46%	0%	40%	0%	88%	0%	S/ datos	0%
20	0%	45.8%	0%	47%	0%	50%	0%	S/ datos	2%
15	0%	4.26%	1.3%	57.8%	1.3%	26.9%	4%	S/ datos	4.7%

Tabla 2: Análisis de fiabilidad del sistema

Si realizamos la media para cada valor umbral, obtenemos las siguientes características de fiabilidad:

- Umbral 40: 0% FAR, 93.3% FRR
- Umbral 30: 0% FAR, 58.0% FRR
- Umbral 20: 2% FAR, 47.6% FRR
- Umbral 15: 11.3% FAR, 29.3% FRR

Observando los datos, es fácil llegar a la conclusión de que el funcionamiento del sistema es mediocre incluso en los casos que colocamos el umbral de coincidencias muy bajo. Como era previsible, la tendencia es que el FRR disminuye con el umbral y el FAR aumenta al disminuir el umbral. También llegamos a la conclusión de que un umbral de 40 es demasiado elevado y 15 es demasiado bajo. Una posible solución sería colocarlo entre 20 y 30. Sin embargo, nos damos cuenta, gracias a la salida de *debug* habilitada en la consola del programa, que el control de descarte por inclinación es demasiado estricto, descartando demasiados puntos que probablemente son válidos. Por lo tanto, desactivaremos esta función y repetiremos el mismo experimento, comparando los resultados obtenidos con los anteriores.

Umbral	Usuario 1		Usuario 2		Usuario 3		Usuario 4		Usua. 5
	FAR	FRR	FAR	FRR	FAR	FRR	FAR	FRR	FAR
65	0%	0%	0%	2%	0%	0%	0%	0%	0%
50	0%	0%	0%	0%	0%	0%	0%	0%	0.5%
40	1.5%	0%	0%	0%	0.5%	2%	0%	0%	0.5%

Tabla 3: Análisis de fiabilidad del sistema 2

Si realizamos la media para cada valor umbral, obtenemos las siguientes características de fiabilidad:

- Umbral 65: 0% FAR, 0.5% FRR
- Umbral 50: 0.5% FAR, 0% FRR
- Umbral 40: 2.5% FAR, 0.51% FRR

Tras observar estos datos, rápidamente podemos darnos cuenta de que el sistema funciona de manera mucho más fiable cuando no se descartan emparejamientos por inclinación. Al aumentar la cantidad de emparejamientos, es lógico aumentar también el umbral. Como se puede observar, los mejores resultados se obtienen para el umbral 65. No obstante, podría incluso aumentarse hasta 75 y seguir funcionando de manera completamente similar. Entonces, llegamos a la conclusión de que el sistema desarrollado en este trabajo es prometedor y aparentemente fiable. Por supuesto, habría que realizar una mayor cantidad de pruebas en una base de datos más extensa para confirmar la fiabilidad del sistema.

Si hablamos con relación al tiempo que tarda el proceso de identificación, debido a las limitaciones del ordenador a bordo del sistema, es considerablemente alto, sobre todo teniendo en cuenta que el tiempo de identificación crece linealmente con la cantidad de entradas de la base de datos. Por lo tanto, para un mayor número de usuarios, mayor tiempo de identificación.

6 PLANIFICACIÓN Y PRESUPUESTO

En este apartado vamos a mostrar la planificación seguida para el desarrollo del proyecto y haremos un análisis del costo total de éste, incluyendo horas de trabajo de ingeniero, componentes y elementos fungibles, entre otros.

6.1. Planificación

A continuación, se muestra un diagrama de Gantt que representa la planificación seguida para el desarrollo del proyecto. Por supuesto, sólo se puede tomar de manera orientativa y los plazos seguidos pueden haberse retrasado o adelantado indistintamente.

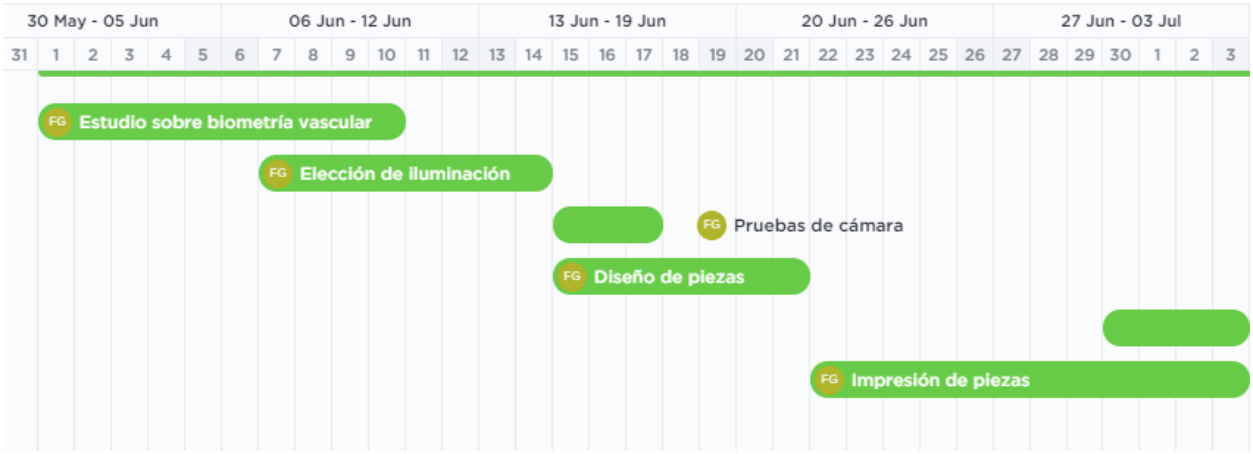


Ilustración 6-1: Planificación 30 mayo – 3 julio

Ilustración 6-2: Planificación 4 julio – 7 agosto

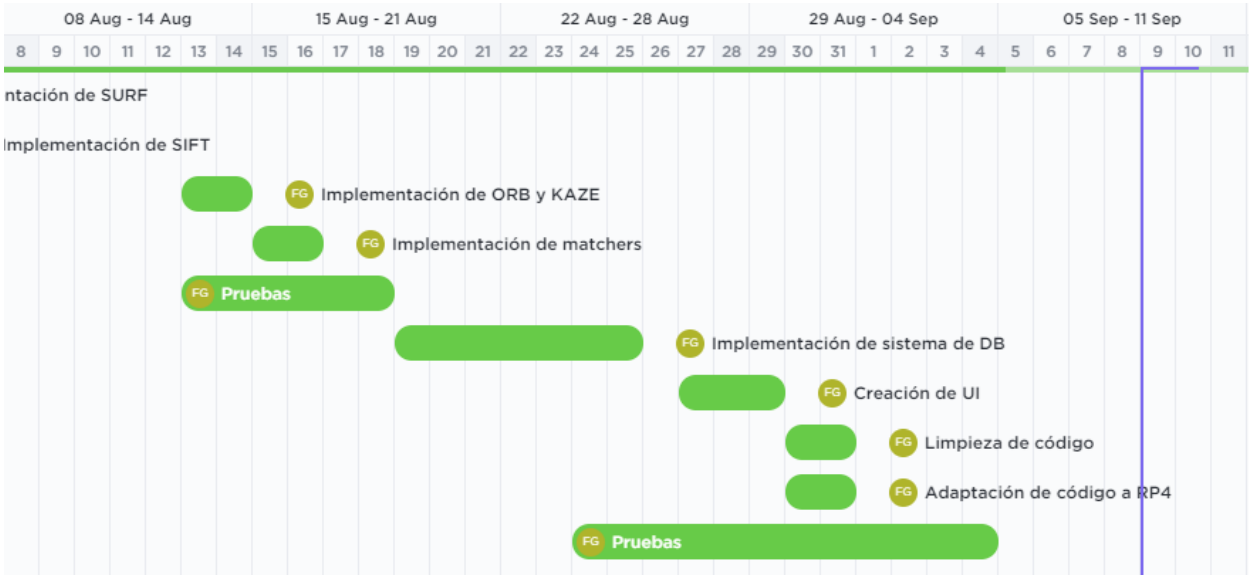


Ilustración 6-3: Planificación 8 agosto – 11 septiembre

Aunque no se incluye en este diagrama, la redacción de este documento se ha realizado durante todo el desarrollo del proyecto, siendo la tarea más extensa de todas.

6.2. Presupuesto

En este subapartado desglosaremos el coste total del proyecto, componente a componente.

Componente	Unidades	Precio unitario	Coste total	Proveedor
Kit Raspberry Pi 4B 4GB	1	79 €	79 €	Amazon
LED QED223 (890nm)	4	1.07 €	4.28 €	Farnell
LED TSAL100 (940nm)	3	3.81 €	11.43 €	Farnell
Tornillería	-	-	0.50 €	Proveedor local
Piezas impresas	213.66 g	19.99 €/kg	4.27 €	-
Cámara Raspberry Pi NoIR	1	34.99 €	34.99 €	Amazon
Fabricación y envío PCBs	5	2.77 €	13.86 €	JLCPCB

Cables y conectores	-	-	4 €	Amazon, RS
Trabajo ingeniero	300 horas	12.80 €/hora	3843 €	-
Resistencias de 34 ohmios 1/2W	7	0.199 € (min 10)	1.99 €	Mouser electronics
Integrado ULN2003AN	1	0.762 €	0.762 €	Mouser electronics
TOTAL			**3995.33 €**	

Tabla 4: Presupuesto

Desglosando el trabajo de ingeniero, podemos obtener el número de horas y sueldo de la siguiente manera. Como el trabajo de fin de grado se compone de 12 créditos ECTS y cada uno de éstos corresponde con 25h de trabajo, podemos obtener así el número de horas: $12cr * 25h/cr = 300h$. En cuanto al salario, el sueldo medio de un ingeniero en España [21] es de 2050 € netos mensuales. Si suponemos que se trabajan 40h semanales y suponemos que un mes se compone de cuatro semanas, obtenemos el sueldo por hora de la siguiente manera: $\frac{2050\frac{€}{mes}}{40\frac{h}{semanas} * 4\frac{semanas}{mes}} = 12.81\frac{€}{h}$.

7 CONCLUSIONES

Antes de concluir este documento, es imperativo volver la vista hacia atrás y resumir el objetivo del proyecto, la solución propuesta y los resultados obtenidos. La propuesta trataba de conseguir un sistema de identificación biométrico a través de la anatomía vascular del dedo índice de una persona cualquiera. La solución propuesta consistía en tomar una imagen del dedo con una cámara sin filtro IR mientras éste estaba siendo atravesado por luz NIR en dos longitudes de onda distintas. Posteriormente, la imagen obtenida se procesaría con *OpenCV 4* [3] en un SBC como es la *Raspberry Pi 4* y se cotejaría con entradas en una base de datos para determinar o no la identidad de la persona.

Como hemos visto ya en el apartado cinco, los resultados obtenidos, son realmente prometedores y definitivamente es un objeto de estudio más profundo muy interesante. Para conseguir todo este funcionamiento han sido necesarias muchas horas de programación en C++, en conjunto con las APIs de *OpenCV* [3], *SQLite3* y QT. Además, ha sido necesario diseñar *hardware* propio, tal como la carcasa de PLA en la que se encuentra el sistema y el circuito de alimentación de los LEDs.

Mencionado todo lo anterior, podemos comenzar a obtener conclusiones sobre la solución tomada para resolver el problema propuesto, las dificultades encontradas y sobre el grado de satisfacción con el resultado final obtenido.

La conclusión que se puede obtener en torno a la solución aportada es que se ha desarrollado un método fiable de identificación de personas y además se ha implementado en conjunto a una interfaz gráfica simple e intuitiva para facilitar el uso al usuario. Sin embargo, como no se ha probado de forma extremadamente extensiva, también obtenemos la conclusión de que sería necesario realizar un estudio de las características más profundo para poder comercializar el producto y sobre todo, usarlo en aplicaciones de alta seguridad como puede ser la seguridad bancaria o médica.

Un punto muy destacable que añadir a estas conclusiones es la enorme cantidad de trabajo que ha supuesto conseguir realizar el proyecto a un nivel aceptable. Ha sido muy importante la constancia y el apoyo de mis tutores que siempre me han motivado a dar lo mejor de mí. La mayor conclusión a la que llego en este proyecto y en concreto la más importante para mí, es que estoy completamente orgulloso de lo que he logrado en este trabajo académico y ha sido una enorme experiencia en la que he asimilado una gran cantidad de conceptos que sin duda me serán útiles en mi carrera profesional.

8 LÍNEAS FUTURAS

En este último capítulo se proponen posibles mejoras o adaptaciones al sistema desarrollado para aumentar su utilidad o precisión en el futuro, o incluso añadirle más funcionalidades para convertirlo en un producto más competitivo en el mercado. En primer lugar se hablará de las posibles mejoras de *hardware* y, en segundo lugar, de las posibles mejoras de *software*.

- Mejoras *Hardware*
    - La primera posible mejora más destacable sería disminuir el tamaño del sistema. Actualmente se trata de un dispositivo relativamente voluminoso, destacando especialmente su altura. Esta altura puede disminuirse notablemente a costa de fotografiar menos cantidad de dedo. Sin embargo, tampoco es conveniente utilizar el dedo entero, por lo que simplemente se podría disminuir la altura de la pieza principal, abaratando además la carcasa al utilizar menos plástico.
    - Otra posible mejora con relación al tamaño del sistema sería diseñar una carcasa completamente distinta en la que se aprovechara mejor el espacio, colocado por ejemplo en una pared de la carcasa el ordenador de a bordo y la cámara lo más abajo posible, reduciendo así altura, anchura y longitud, abaratando aún más la carcasa.
    - Una mejora que podría dirigir a obtener mejores imágenes binarizadas de la anatomía vascular del dedo consiste en aumentar la corriente que circula por los LEDs, ya que éstos son capaces de iluminar de forma pulsada con una intensidad diez veces superior a la utiliza para otorgar una radiancia también diez veces superior aproximadamente. Como contrapartida, esto supondría tener que utilizar resistencias de mayor tamaño y precio.
    - Una de las mejoras más notables sería la de cambiar el ordenador a bordo por uno de mayor potencia, ya que el tiempo de procesamiento en el ordenador elegido es bastante alto. Una buena solución podría ser un NVIDIA Jetson, aunque a costa de aumentar notoriamente el precio del dispositivo.
- Mejoras *Software*
    - Una mejora sería la de implementar la posibilidad de almacenar más de una entrada para cada usuario, disminuyendo de esta manera el FAR y por tanto, aumentando la seguridad a costa de aumentar el tiempo de procesamiento.
    - Otra opción, que sumada a la anterior podría ayudar a aumentar la rapidez y seguridad del sistema es la obligación de que el usuario escriba su nombre de usuario antes de proceder a verificar su identidad. De esta manera, el programa únicamente tendría que cotejar los datos con un número concreto y reducido de entradas, aumentando así la rapidez de procesamiento. Si sumamos esta mejora a la anterior, se obtiene un sistema más rápido, fiable y escalable.
    - Una posible mejora podría haber sido la de fusionar completamente los programas principales y de interfaz de usuario. De esta manera, la interacción entra ambos programas sería notoriamente más simple y efectiva, ya que se encontrarían en el mismo proceso. En este caso, se podría diseñar una interfaz más compleja con la que se pudiera tener un mayor control.
    - Otra mejora sería la de añadir la posibilidad de eliminar entradas de la base de datos o de actualizarlas con nuevas características.

REFERENCIAS

[1] N. Miura, A. Nagasaka and T. Miyatake, "Personal identification device and method". Japan Patent JP2000274987A, 6 September 2000.

[2] J. Rice, "Method and apparatus for the identification of individuals". France Patent WO1985004088A1, 20 March 1984.

[3] Intel Corporation, «OpenCV,» June 2000. [En línea]. Available: https://opencv.org.

[4] P. F. Alcantarilla, A. Bartoli y A. J. Davison, «KAZE Features,» de *European Conference on Computer Vision (ECCV)*, Fiorenze, Italy, 2012.

[5] VISHAY, «Farnell,» 13 March 2014. [En línea]. Available: https://www.farnell.com/datasheets/2049675.pdf.

[6] FAIRCHILD, «Farnell,» February 2016. [En línea]. Available: https://www.farnell.com/datasheets/2304818.pdf.

[7] Texas Instruments, «Texas Instruments,» August 2019. [En línea]. Available: https://www.ti.com/lit/ds/symlink/ulq2004a.pdf?HQS=TI-null-null-alldatasheets-df-pf-SEP-wwe&ts=1627562157184&ref_url=https%253A%252F%252Fpdf1.alldatasheet.es%252F.

[8] NVIDIA, «NVIDIA DEVELOPER,» NVIDIA, [En línea]. Available: https://developer.nvidia.com/embedded/jetson-nano-developer-kit.

[9] Orange Pi, «Orange Pi,» Orange Pi, [En línea]. Available: http://www.orangepi.org/Orange%20Pi%204B/.

[10] Beagleboard Org, «Beagleboard,» 2 February 2018. [En línea]. Available: https://beagleboard.org/x15.

[11] F. J. G. Flores, «Github,» 2021. [En línea]. Available: https://github.com/fraciscoestar/TFG.

[12] Multimedia Signal Processing and Security Lab, «PLUS OpenVein Finger- and Hand-Vein Toolkit,» 21 July 2021. [En línea]. Available: http://wavelab.at/sources/OpenVein-Toolkit/.

[13] B. Nevils, T. Mimbs, N. Naheed y A. Sailesh, «High Frequency Emphasis Filter Instead of Homomorphic Filter,» de *2018 International Conference on Computational Science and Computational Intelligence (CSCI)*, Las Vegas, NV, USA, 2018.

[14] J. Zhao, H. Tian, W. Xu y X. Li, «A New Approach to Hand Vein Image Enhancement,» de *2009 Second International Conference on Intelligent Computation Technology and Automation*, Changsha, China, 2009.

[15] J. Zhang, T. Tan y L. Ma, «Invariant texture segmentation via circular Gabor filters,» de *2002 International Conference on Pattern Recognition*, Quebec City, QC, Canada, 2002.

[16] J. Zhang y J. Yang, «Finger-Vein Image Enhancement Based on Combination of Gray-Level Grouping

and Circular Gabor Filter,» de *2009 International Conference on Information Engineering and Computer Science*, Wuhan, China, 2009.

[17] R. Funayama, H. Yanagihara, L. Van Gool, T. Tuytelaars y H. Bay, «ROBUST INTEREST POINT DETECTOR AND DESCRIPTOR». US Patente US20090238460A1, 24 September 2009.

[18] D. G. Lowe, «Method and apparatus for identifying scale invariant features in an image and use of same for locating an object in an image». US Patente US6711293B1, 23 March 2004.

[19] D. G. Lowe, «Distinctive Image Features from Scale-Invariant Keypoints,» *International Journal of Computer Vision,* nº 60, pp. 91-110, 2004.

[20] «SQLite,» [En línea]. Available: https://www.sqlite.org/index.html. [Último acceso: 3 September 2021].

[21] «Jobted,» [En línea]. Available: https://www.jobted.es/salario/ingeniero#:~:text=Sueldo%20del%20Ingeniero%20en%20Espa%C3%B1a&text=El%20salario%20medio%20de%20un,salario%20medio%20anual%20en%20Espa%C3%B1a.. [Último acceso: 9 September 2021].

[22] S. Y. Sharma, «Techawarey,» 25 April 2020. [En línea]. Available: http://techawarey.com/programming/install-opencv-c-c-in-ubuntu-18-04-lts-step-by-step-guide/.

[23] Q-engineering, «Q-engineering,» 30 July 2021. [En línea]. Available: https://qengineering.eu/install-opencv-4.5-on-raspberry-pi-4.html.

[24] G. Webster, «CNN,» 5 July 2010. [En línea]. Available: https://edition.cnn.com/2010/WORLD/europe/07/05/first.biometric.atm.europe/index.html.

[25] J. Hashimoto, «FInger Vein Authentication Technology and Its Future,» de *2006 Symposium on VLSI Circuits, 2006. Digest of Technical Papers*, Honolulu, HI, USA, 2006.

[26] «Wikipedia,» 4 February 2021. [En línea]. Available: https://es.wikipedia.org/wiki/ARM_big.LITTLE.

[27] «Wikipedia,» 23 August 2021. [En línea]. Available: https://es.wikipedia.org/wiki/SQL.

[28] Microsoft, «Microsoft Docs,» Microsoft, 2 January 2020. [En línea]. Available: https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/concepts/serialization/#:~:text=La%20serializaci%C3%B3n%20es%20el%20proceso,proceso%20inverso%20se%20denomina%20deserializaci%C3%B3n..

[29] Matt, «Raspberry Pi Spy,» [En línea]. Available: https://www.raspberrypi-spy.co.uk/2012/06/simple-guide-to-the-rpi-gpio-header-and-pins/. [Último acceso: 9 September 2021].

ANEXO A: INSTALACIÓN DE OPENCV 4

Para comenzar, en este anexo primero debemos diferenciar entre la instalación realizada en el sistema, es decir, sobre la *Raspberry Pi 4* en un OS Raspbian y la instalación realizada sobre un sistema Ubuntu utilizado para programar de manera más rápida y efectiva. La instalación se realiza de manera distinta dependiendo del sistema. En este apéndice se explicará la instalación de la API en ambos sistemas.

Comenzando por la instalación sobre el sistema Ubuntu [21], empezamos por actualizar el sistema e instalar unas aplicaciones que nos ayudarán más adelante. Esto se realiza con los siguientes comandos:

```
$ sudo apt-get update && sudo apt-get upgrade
$ sudo apt install software-properties-common
$ sudo apt install apt-file
```

A continuación, hay que instalar todas las dependencias de las librerías con los siguientes comandos:

```
$ sudo apt-get install build-essential cmake git libgtk2.0-dev pkg-config
  libavcodec-dev libavformat-dev libswscale-dev
$ sudo apt-get install python3.5-dev python3-numpy libtbb2 libtbb-dev
$ sudo apt-get install libjpeg-dev libpng-dev libtiff5-dev libjasper-dev
  libdc1394-22-dev libeigen3-dev libtheora-dev libvorbis-dev libxvidcore-
  dev libx264-dev sphinx-common libtbb-dev yasm libfaac-dev libopencore-
  amrnb-dev libopencore-amrwb-dev libopenexr-dev libgstreamer-plugins-
  base1.0-dev libavutil-dev libavfilter-dev libavresample-dev
```

Ahora, necesitamos descargar todo el código fuente de *OpenCV* [3] para compilarlo posteriormente, para ello hay varias maneras, pero la más sencilla es desde la consola:

```
$ sudo -s
$ cd /opt
$ wget https://github.com/opencv/opencv/archive/4.5.2.zip
$ unzip 4.5.2.zip
$ mv opencv-4.5.2/ opencv/
$ rm 4.5.2.zip
$ wget https://github.com/opencv/opencv_contrib/archive/4.5.2.zip
$ unzip 4.5.2.zip
$ mv opencv_contrib-4.5.2/ opencv_contrib/
$ rm 4.5.2.zip
```

El siguiente paso es la compilación del código, que se realiza de la siguiente manera:

```
$ cd opencv
$ mkdir release
$ cd release
$ cmake -D BUILD_TIFF=ON -D WITH_CUDA=OFF -D ENABLE_AVX=OFF -D
  WITH_OPENGL=OFF -D WITH_OPENCL=OFF -D WITH_IPP=OFF -D WITH_TBB=ON -D
  BUILD_TBB=ON -D WITH_EIGEN=OFF -D WITH_V4L=OFF -D WITH_VTK=OFF -D
  BUILD_TESTS=OFF -D BUILD_PERF_TESTS=OFF -D OPENCV_ENABLE_NONFREE=ON -D
  OPENCV_GENERATE_PKGCONGIF=ON -D CMAKE_BUILD_TYPE=RELEASE -D
  CMAKE_INSTALL_PREFIX=/usr/local -D OPENCV_EXTRA_MODULES_PATH=
  /opt/opencv_contrib/modules /opt/opencv/
$ make -j8
$ make install
$ ldconfig
$ exit
```

El único paso restante es mover el archivo de *pkg-config* de *OpenCV* a la ubicación correcta. Esto se realiza de la siguiente manera:

```
$ cd /usr/local/lib/pkgconfig/
$ sudo cp opencv4.pc /usr/lib/x86_64-linux-gnu/pkgconfig/opencv.pc
$ pkg-config --modversion opencv
```

Sabremos si la instalación se ha realizado de manera correcta si el último comando nos devuelve el valor de **4.5.2**. Para compilar los programas con *OpenCV* tendremos que añadir a los argumentos de compilación las flags:

```
-std=c++17 `pkg-config --cflags --libs opencv´
```

En este punto, ya hemos terminado con la instalación en un sistema *Ubuntu*. A continuación, se explicará el mismo proceso para un sistema *Raspbian* funcionando sobre una *RP4* [22]. Al igual que antes, es necesario actualizar el sistema e instalar las dependencias descritas anteriormente. Además, tendremos que actualizar el *Firmware* de la EEPROM de la placa de la siguiente manera:

```
$ sudo rpi-eeprom-update -a
$ sudo reboot
```

Antes de compilar las librerías, necesitamos asegurarnos de que hay suficiente memoria para la compilación. Como la *RP4* tiene características reducidas con respecto al sistema anterior, debemos aumentar la memoria *swap* del OS. Para ello, debemos modificar la propiedad CONF_MAXSWAP de un archivo, cambiando su valor a 4096. Para editar este archivo se hace de la siguiente manera:

```
$ sudo nano /sbin/dphys-swapfile
$ sudo reboot
```

Aparte de las dependencias descargadas anteriormente, en este caso, se necesitan descargar algunas dependencias extras:

```
$ sudo apt-get install gfortran libgif-dev libcanberra-gtk* libgtk-3-dev
  libv4l-dev libopenblas-dev libatlas-base-dev libblas-dev liblapack-dev
  libhdf5-dev protobuf-compiler qt5-default
```

Ahora, al igual que antes, descargamos las librerías de *OpenCV*, pero lo haremos en una ubicación distinta. En lugar de descargar el código fuente en */opt*, lo haremos en ~. Entonces, procedemos a la instalación:

```
$ cd opencv
$ mkdir build
$ cd build
$ cmake -D CMAKE_BUILD_TYPE=RELEASE -D CMAKE_INSTALL_PREFIX=/usr/local -D
  OPENCV_EXTRA_MODULES_PATH=~/opencv_contrib/modules -D ENABLE_NEON=ON -D
  ENABLE_VFPV3=ON -D WITH_OPENMP=ON -D WITH_OPENCL=OFF -D BUILD_ZLIB=ON
  -D BUILD_TIFF=ON -D WITH_FFMPEG=ON -D WITH_TBB=ON -D BUILD_TBB=ON -D
  BUILD_TESTS=OFF -D WITH_EIGEN=OFF -D WITH_GSTREAMER=OFF -D WITH_V4L=ON
  -D WITH_LIBV4L=ON -D WITH_VTK=OFF -D BUILD_EXAMPLES=OFF -D WITH_QT=ON
  -D OPENCV_ENABLE_NONFREE=ON -D INSTALL_C_EXAMPLES=OFF -D
  INSTALL_PYTHON_EXAMPLES=OFF -D BUILD_opencv_python3=TRUE -D
  OPENCV_GENERATE_PKGCONFIG=ON ..
$ make -j4
$ sudo make install
$ sudo ldconfig
$ make clean
$ sudo apt-get update
```

De esta manera, completamos la instalación para la *RP4*. El proceso de compilación puede tardar alrededor de una hora y media y una vez esté instalado, los argumentos que necesitamos añadir al compilador son similares. Por último, debemos regresar la configuración CONF_MAXSWAP a su valor original, de la misma manera que fue cambiada en pasos anteriores. El valor por defecto es 100.

ANEXO B: CÓDIGO FUENTE COMÚN

Como ya se ha explicado en el proyecto, el trabajo se ha realizado en dos ramas distintas de trabajo, un sistema funcionando sobre *Ubuntu 21*, y el sistema principal, trabajando sobre *Raspbian* en el ordenador de a bordo, el SBC *Raspberry Pi 4*. En estas dos ramas, hay una gran cantidad de código común, por lo que se decide agruparlo en una clase estática para obtener un código más claro y limpio. El resultado es un archivo de encabezado denominado *DBHandler.hpp*, en el que se define la clase, sus métodos y miembros y un archivo *DBHandler.cpp* en el que se implementa la funcionalidad de todos estos métodos.

DBHandler.hpp

```
#ifndef DBHANDLER_H
#define DBHANDLER_H

#include <iostream>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/features2d.hpp>
#include <opencv2/xfeatures2d.hpp>
#include "sqlite3.h"

#define FILTER_HPF 0
#define FILTER_HFE 1

#define MINMATCHES 65

using namespace std;
using namespace cv;

class DBHandler
{
public:

    static bool Login(string* username);

    static bool Register(string username);

    static tuple<string, vector<KeyPoint>, Mat> FindBestMatch(tuple<vector<KeyPoint>, Mat> features);

    static tuple<string, vector<KeyPoint>, Mat> ReadEntry(int id, sqlite3* e_db = NULL);

    static int WriteEntry(tuple<vector<KeyPoint>, Mat> features, string name);

    static char* EncodeF32Image(Mat& img);

    static Mat DecodeKazeDescriptor(vector<char>& buffer, int nKeypoints);

    static tuple<vector<KeyPoint>, Mat> KAZEDetector(Mat& src);

    static tuple<Mat, int> FLANNMatcher(tuple<vector<KeyPoint>, Mat> m1, tuple<vector<KeyPoint>, Mat> m2, Mat
imgA = Mat(), Mat imgB = Mat());

    static void PreprocessImage(Mat& src, Mat& dst);

    static void CGF(Mat& src, Mat& dst);

    static Mat DFTModule(Mat src[], bool shift);

    static void HPF(Mat& src, Mat& dst, uint8_t filterType);

    static void FFTShift(const Mat& src, Mat& dst);

private:
    static const int d0hfe = 10, d0hpf = 40, k1hfe = 5300, k2hfe = 7327, k1hpf = 3050, k2hpf = 5463, sig = 5, dF
= 436;
    DBHandler() {}
};

#endif
```

DBHandler.cpp

```
#include "DBHandler.hpp"

using namespace std;
using namespace cv;

bool DBHandler::Login(string* username)
{
    Mat img, imgPre;
    img = imread("original4.png", ImreadModes::IMREAD_GRAYSCALE); // Use camera instead
    PreprocessImage(img, imgPre);
    tuple<vector<KeyPoint>, Mat> features = KAZEDetector(imgPre);
    tuple<string, vector<KeyPoint>, Mat> bestMatch = FindBestMatch(features);

    if (get<0>(bestMatch) == "")
        return false;
    else
    {
        *username = get<0>(bestMatch);
        return true;
    }
}

bool DBHandler::Register(string username)
{
    Mat img, imgPre;

    img = imread("original4.png", ImreadModes::IMREAD_GRAYSCALE); // Use camera instead
    PreprocessImage(img, imgPre);
    tuple<vector<KeyPoint>, Mat> features = KAZEDetector(imgPre);
    return !WriteEntry(features, username);
}

tuple<string, vector<KeyPoint>, Mat> DBHandler::FindBestMatch(tuple<vector<KeyPoint>, Mat> features)
{
    sqlite3* db;
    int rc = sqlite3_open_v2("database.db", &db, SQLITE_OPEN_READONLY, NULL);
    if (rc != SQLITE_OK)
    {
        cerr << "DB open failed: " << sqlite3_errmsg(db) << endl;
        throw;
    }

    sqlite3_stmt* stmt = NULL;
    string query = "SELECT * FROM data";

    rc = sqlite3_prepare_v2(db, query.c_str(), -1, &stmt, NULL);
    if (rc != SQLITE_OK)
    {
        cerr << "Prepare failed: " << sqlite3_errmsg(db) << endl;
        sqlite3_finalize(stmt);
        sqlite3_close(db);
        throw;
    }

    tuple<string, vector<KeyPoint>, Mat> bestMatch;
    int maxMatches = 0;
    while (true)
    {
        rc = sqlite3_step(stmt);
        if (rc == SQLITE_ROW)
        {
            int id = sqlite3_column_int(stmt, 0);
            tuple<string, vector<KeyPoint>, Mat> entry = ReadEntry(id, db);

            // Match Features /////////////////
            tuple<Mat, int> matchKAZE = FLANNMatcher(features, make_tuple(get<1>(entry), get<2>(entry)));
            ///////////////////////////////////

            if (get<1>(matchKAZE) > maxMatches)
            {
                maxMatches = get<1>(matchKAZE);
                bestMatch = entry;
            }
        }
        else
```

```
            break;
    }

    if (maxMatches > MINMATCHES) // Match found
    {
        cout << maxMatches << " found for user " << get<0>(bestMatch) << endl;
        return bestMatch;
    }
    else
    {
        return make_tuple(string(""), vector<KeyPoint>(), Mat());
    }
}

tuple<string, vector<KeyPoint>, Mat> DBHandler::ReadEntry(int id, sqlite3* e_db)
{
    sqlite3* db;
    int rc;

    if (e_db != NULL)
    {
        db = e_db;
    }
    else
    {
        rc = sqlite3_open_v2("database.db", &db, SQLITE_OPEN_READONLY, NULL);
        if (rc != SQLITE_OK)
        {
            cerr << "DB open failed: " << sqlite3_errmsg(db) << endl;
            throw;
        }
    }

    sqlite3_stmt* stmt = NULL;
    string query = "SELECT * FROM data WHERE id=?";

    rc = sqlite3_prepare_v2(db, query.c_str(), -1, &stmt, NULL);
    if (rc != SQLITE_OK)
    {
        cerr << "Prepare failed: " << sqlite3_errmsg(db) << endl;
        sqlite3_finalize(stmt);
        sqlite3_close(db);
        throw;
    }

    rc = sqlite3_bind_int(stmt, 1, id);
    if (rc != SQLITE_OK)
    {
        cerr << "bind int failed: " << sqlite3_errmsg(db) << endl;
        sqlite3_finalize(stmt);
        sqlite3_close(db);
        throw;
    }

    rc = sqlite3_step(stmt);
    if (rc == SQLITE_ROW)
    {
        char* name = (char*)sqlite3_column_text(stmt, 1);
        string name_cpp = string(name);

        int nKeypoints = sqlite3_column_int(stmt, 2);
        char* keypointsBinary = (char*)sqlite3_column_blob(stmt, 3);

        vector<KeyPoint> keypoints;
        for (int i = 0; i < nKeypoints; i++)
        {
            vector<char> kp(&keypointsBinary[i * sizeof(KeyPoint)], &keypointsBinary[i * sizeof(KeyPoint)] +
sizeof(KeyPoint));
            KeyPoint* kpPtr = reinterpret_cast<KeyPoint*>(&kp[0]);
            keypoints.push_back(*kpPtr);

            kp.clear();
            kp.shrink_to_fit();
        }
```

```
        int descriptorSize = sqlite3_column_bytes(stmt, 4);
        char* dPtr = (char*)sqlite3_column_blob(stmt, 4);
        vector<char> dData(dPtr, dPtr + descriptorSize);
        Mat descriptor = DecodeKazeDescriptor(dData, nKeypoints);

        sqlite3_finalize(stmt);
        sqlite3_close(db);

        dData.clear();
        dData.shrink_to_fit();

        keypointsBinary = NULL;
        delete[] keypointsBinary;

        return make_tuple(name_cpp, keypoints, descriptor);
    }
    else
    {
        cerr << "No entry with id " << id << " was found." << endl;
        sqlite3_finalize(stmt);
        sqlite3_close(db);
        throw;
    }
}

int DBHandler::WriteEntry(tuple<vector<KeyPoint>, Mat> features, string name)
{
    // First check if user already exists
    ////////////////////////////////////////
    sqlite3* db = NULL;
    int rc = sqlite3_open_v2("database.db", &db, SQLITE_OPEN_READWRITE, NULL);
    if (rc != SQLITE_OK)
    {
        cerr << "DB open failed: " << sqlite3_errmsg(db) << endl;
        throw;
    }

    sqlite3_stmt* stmt = NULL;
    string query = "SELECT * FROM data";

    rc = sqlite3_prepare_v2(db, query.c_str(), -1, &stmt, NULL);
    if (rc != SQLITE_OK)
    {
        cerr << "Prepare failed: " << sqlite3_errmsg(db) << endl;
        sqlite3_finalize(stmt);
        sqlite3_close(db);
        throw;
    }

    while (true)
    {
        rc = sqlite3_step(stmt);
        if (rc == SQLITE_ROW)
        {
            char* name_c_str = (char*)sqlite3_column_text(stmt, 1);
            string name_str = string(name_c_str);

            if (name_str == name)
            {
                return 1;
            }
        }
        else
            break;
    }

    sqlite3_finalize(stmt);

    ////////////////////////////////////////
    char* descriptorBuffer = EncodeF32Image(get<1>(features));
    ////////////////////////////////////////

    ////////////////////////////////////////
    int keypointCount = get<0>(features).size();
```

```
    char* keypointsBuffer = new char[sizeof(KeyPoint) * keypointCount];
    int keypointsByteCounter = 0;
    const void* kpptr = NULL;
    kpptr = keypointsBuffer;

    for (int i = 0; i < keypointCount; i++)
    {
        char* kp = reinterpret_cast<char*>(&get<0>(features)[i]);
        for (int j = 0; j < sizeof(KeyPoint); j++)
        {
            keypointsBuffer[keypointsByteCounter] = kp[j];
            keypointsByteCounter++;
        }
    }
    ////////////////////////////////////////

    ////////////////////////////////////////
    stmt = NULL;
    query = "INSERT INTO data(id, name, nKeypoints, keypoints, descriptor) VALUES(NULL, ?, ?, ?, ?)";

    rc = sqlite3_prepare_v2(db, query.c_str(), -1, &stmt, NULL);
    if (rc != SQLITE_OK)
    {
        cerr << "Prepare failed: " << sqlite3_errmsg(db) << endl;
        sqlite3_finalize(stmt);
        sqlite3_close(db);
        throw;
    }

    rc = sqlite3_bind_text(stmt, 1, name.c_str(), -1, NULL);
    if (rc != SQLITE_OK)
    {
        cerr << "bind text failed: " << sqlite3_errmsg(db) << endl;
        sqlite3_finalize(stmt);
        sqlite3_close(db);
        throw;
    }

    rc = sqlite3_bind_int(stmt, 2, keypointCount);
    if (rc != SQLITE_OK)
    {
        cerr << "bind int failed: " << sqlite3_errmsg(db) << endl;
        sqlite3_finalize(stmt);
        sqlite3_close(db);
        throw;
    }

    rc = sqlite3_bind_blob(stmt, 3, keypointsBuffer, keypointsByteCounter, SQLITE_STATIC);
    if (rc != SQLITE_OK)
    {
        cerr << "bind 1 failed: " << sqlite3_errmsg(db) << endl;
        sqlite3_finalize(stmt);
        sqlite3_close(db);
        throw;
    }

    rc = sqlite3_bind_blob(stmt, 4, descriptorBuffer, get<1>(features).rows * get<1>(features).cols *
sizeof(float), SQLITE_STATIC);
    if (rc != SQLITE_OK)
    {
        cerr << "Prepare failed: " << sqlite3_errmsg(db) << endl;
        sqlite3_finalize(stmt);
        sqlite3_close(db);
        throw;
    }

    rc = sqlite3_step(stmt);
    if (rc != SQLITE_DONE)
    {
        cerr << "Execution failed: " << sqlite3_errmsg(db) << endl;
        sqlite3_finalize(stmt);
        sqlite3_close(db);
        throw;
    }
```

```
    sqlite3_finalize(stmt);
    sqlite3_close(db);
    ////////////////////////////////////////

    free(descriptorBuffer);
    delete[] keypointsBuffer;

    return 0;
}

char* DBHandler::EncodeF32Image(Mat& img)
{
    char* buffer = (char*)malloc(sizeof(float) * img.rows * img.cols);
    int ptr = 0;

    for (int i = 0; i < img.rows; i++)
    {
        for (int j = 0; j < img.cols; j++)
        {
            float val = img.at<float>(Point(j, i));
            char* byteVal = reinterpret_cast<char*>(&val);

            for (int k = 0; k < sizeof(float); k++, ptr++)
            {
                buffer[ptr] = byteVal[k];
            }
        }
    }

    return buffer;
}

Mat DBHandler::DecodeKazeDescriptor(vector<char>& buffer, int nKeypoints)
{
    int cols = 64;
    int rows = nKeypoints;

    Mat img = Mat::zeros(Size(cols, rows), CV_32F);

    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            vector<char> byteVal(&buffer[(j + i * cols) * sizeof(float)], &buffer[(j + i * cols) * sizeof(float)]
+ sizeof(float));
            float* val = reinterpret_cast<float*>(&byteVal[0]);

            img.at<float>(Point(j, i)) = *val;
            byteVal.clear();
            byteVal.shrink_to_fit();
        }
    }

    return img;
}


tuple<vector<KeyPoint>, Mat> DBHandler::KAZEDetector(Mat& src)
{
    Mat img = src.clone();

    // SURF //////////////////////////
    Ptr<KAZE> detector = KAZE::create(false, true);

    vector<KeyPoint> keypoints;
    Mat descriptors;

    detector->detectAndCompute(img, noArray(), keypoints, descriptors);
    cout << "KAZE kp: " << to_string(keypoints.size()) << endl;
    return make_tuple(keypoints, descriptors);
    ////////////////////////////////////
}

tuple<Mat, int> DBHandler::FLANNMatcher(tuple<vector<KeyPoint>, Mat> m1, tuple<vector<KeyPoint>, Mat> m2, Mat
imgA, Mat imgB)
```

```
{
    Ptr<DescriptorMatcher> matcher = DescriptorMatcher::create(DescriptorMatcher::FLANNBASED);
    std::vector< std::vector<DMatch> > knn_matches;
    matcher->knnMatch(get<1>(m1), get<1>(m2), knn_matches, 2);

    // Filter matches using the Lowe's ratio test.
    const float ratio_thresh = 0.72f;
    std::vector<DMatch> good_matches;
    for (size_t i = 0; i < knn_matches.size(); i++)
    {
        if (knn_matches[i][0].distance < ratio_thresh * knn_matches[i][1].distance)
        {
            good_matches.push_back(knn_matches[i][0]);
        }
    }

    /*float maxInc = 0.34f;
    std::vector<DMatch> goodest_matches;

    for (size_t i = 0; i < good_matches.size(); i++)
    {
        int idx1 = good_matches[i].trainIdx;
        int idx2 = good_matches[i].queryIdx;

        const KeyPoint& kp1 = get<0>(m2)[idx1], & kp2 = get<0>(m1)[idx2];
        Point2f p1 = kp1.pt;
        Point2f p2 = kp2.pt;
        Point2f triangle = Point2f(std::abs(p2.x - p1.x), std::abs(p2.y - p1.y));

        float angle = std::atan2(triangle.y, triangle.x);

        if (std::abs(angle) < maxInc)
        {
            goodest_matches.push_back(good_matches[i]);
        }
    }*/


    cout << "Good matches: " << to_string(good_matches.size()) << endl;
    //cout << "Goodest matches: " << to_string(goodest_matches.size()) << endl;

    if (imgA.rows > 1 && imgB.rows > 1)
    {
        // Draw matches.
        Mat img_matches;
        drawMatches(imgA.clone(), get<0>(m1), imgB.clone(), get<0>(m2), good_matches, img_matches, Sca -
lar::all(-1),
    Scalar::all(-1), std::vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS);

        // Return mathes image.
        return make_tuple(img_matches.clone(), good_matches.size());
    }
    else
    {
        return make_tuple(Mat(), good_matches.size());
    }
}

void DBHandler::PreprocessImage(Mat& src, Mat& dst)
{
    Mat img1, img2, img3, img4, img5, img6, img7, img8;

    // Umbralizar ////////////////////
    threshold(src, img1, 60, 255, THRESH_BINARY);

    Mat framed = Mat::zeros(Size2d(img1.cols + 2, img1.rows + 2), CV_8U);
    Rect r = Rect2d(1, 1, img1.cols, img1.rows);
    img1.copyTo(framed(r));

    Canny(framed, img2, 10, 50);

    Mat kernel = getStructuringElement(MORPH_RECT, Size2d(5, 5));
    dilate(img2, img2, kernel);

    vector<vector<Point>> contours;
```

```
    findContours(img2, contours, RETR_TREE, CHAIN_APPROX_NONE);

    Mat contoursImg = Mat::zeros(img2.size(), CV_8U);
    double maxArea = 0;
    int mIdx = -1;
    for (int i = 0; i < contours.size(); i++)
    {
        double area = contourArea(contours[i]);
        if (area > maxArea)
        {
            maxArea = area;
            mIdx = i;
        }
    }

    if (mIdx >= 0)
        drawContours(contoursImg, contours, mIdx, Scalar::all(255), FILLED);


    bitwise_and(src, src, img3, contoursImg(r));
    ////////////////////////////////////

    // CLAHE ///////////////////////////
    Ptr<CLAHE> clahe = createCLAHE();
    clahe->setClipLimit(6);

    clahe->apply(img3, img3);
    ////////////////////////////////////

    // HFE + HPF ///////////////////////
    HPF(img3, img4, FILTER_HFE);
    HPF(img4, img5, FILTER_HPF);
    resize(img5, img6, img3.size());
    bitwise_and(img6, img6, img7, contoursImg(r));
    clahe->apply(img7, img7);
    ////////////////////////////////////

    // CGF /////////////////////////////
    CGF(img7, img8);
    Mat mask;
    bitwise_not(contoursImg(r), mask);
    normalize(img8, img8, 1.0, 0.0, 4, -1, mask);
    img8.convertTo(img8, CV_8U);
    threshold(img8, img8, 0, 255, THRESH_BINARY);

    kernel = getStructuringElement(MORPH_ELLIPSE, Size(5, 5));
    dilate(img8, img8, kernel);
    erode(img8, img8, kernel);

    kernel = getStructuringElement(MORPH_ELLIPSE, Size(3, 3));
    erode(img8, img8, kernel);
    dilate(img8, img8, kernel);
    ////////////////////////////////////

    Mat cropMask = Mat::zeros(img8.size(), CV_8U);
    Rect cropROI = Rect2d(22, 35, img8.cols - 60, img8.rows - 35 - 70);
    cropMask(cropROI).setTo(255);
    bitwise_and(img8, cropMask, img8);

    dst = img8.clone();
}

void DBHandler::CGF(Mat& src, Mat& dst)
{
    float sigma = 5; //5-pixel width
    float DeltaF = 1.12; //[0.5, 2,5]

    sigma = sig;
    DeltaF = dF / 100.0f;

    float fc = ((1 / M_PI) * sqrt(log(2) / 2) * ((pow(2, DeltaF) + 1) / (pow(2, DeltaF) - 1))) / sigma;

    Mat g = Mat::zeros(Size2d(30, 30), CV_32F);
    Mat G = Mat::zeros(Size2d(30, 30), CV_32F);
    Mat Gim = Mat::zeros(Size2d(30, 30), CV_32F);
```

```
    for (int i = 0; i < g.cols; i++)
    {
        for (int j = 0; j < g.rows; j++)
        {
            g.at<float>(Point(i, j)) = (1 / (2 * M_PI * pow(sigma, 2))) * exp(-(pow(i - g.cols / 2, 2) + pow(j -
g.rows / 2, 2)) / (2 * pow(sigma, 2)));
            G.at<float>(Point(i, j)) = g.at<float>(Point(i, j)) * cos(2 * M_PI * fc * sqrt(pow(i - g.cols / 2, 2)
+ pow(j - g.rows / 2, 2)));
            Gim.at<float>(Point(i, j)) = g.at<float>(Point(i, j)) * sin(2 * M_PI * fc * sqrt(pow(i - g.cols / 2,
2) + pow(j - g.rows / 2, 2)));
        }
    }

    Mat res;
    filter2D(src, res, CV_32F, G);
    dst = res.clone();
}

Mat DBHandler::DFTModule(Mat src[], bool shift)
{
    Mat magImg;
    magnitude(src[0], src[1], magImg); // Magnitud = planes[0]

    magImg += Scalar::all(1); // Cambia a escala log
    log(magImg, magImg);

    // Recorta el espectro si tiene rows o cols impares
    magImg = magImg(Rect(0, 0, magImg.cols & -2, magImg.rows & -2));

    if (shift)
        FFTShift(magImg, magImg);

    normalize(magImg, magImg, 0, 1, NORM_MINMAX);

    return magImg;
}

void DBHandler::HPF(Mat& src, Mat& dst, uint8_t filterType)
{
    Mat padded; // Expande la imagen al tamaño óptimo
    int m = getOptimalDFTSize(src.rows);
    int n = getOptimalDFTSize(src.cols); // Añade valores cero en el borde
    copyMakeBorder(src, padded, 0, m - src.rows, 0, n - src.cols, BORDER_CONSTANT, Scalar::all(0));

    Mat planes[] = { Mat_<float>(padded), Mat::zeros(padded.size(), CV_32F) };
    Mat complexImg;
    merge(planes, 2, complexImg); // Añade un plano complejo

    /// DFT
    dft(complexImg, complexImg);
    split(complexImg, planes);
    ///

    /// FILTER
    Mat H = Mat::zeros(src.size(), CV_32F);
    Mat filt = Mat::zeros(src.size(), CV_32F);
    Mat HF = complexImg.clone();

    // float D0 = 40.0f;
    // float k1 = filterType ? 0.5f : -4;
    // float k2 = filterType ? 0.75f : 10.9;
    float D0;
    float k1, k2;

    if (filterType)
    {
        D0 = (d0hfe > 0) ? d0hfe : 1;
        k1 = (k1hfe - 5000) / 100.0f;
        k2 = (k2hfe - 5000) / 100.0f;
    }
    else
    {
        D0 = (d0hpf > 0) ? d0hpf : 1;
        k1 = (k1hpf - 5000) / 1000.0f;
        k2 = (k2hpf - 5000) / 1000.0f;
```

```
    }

    // float a = 10.9, b = -4;

    for (int i = 0; i < H.cols; i++)
    {
        for (int j = 0; j < H.rows; j++)
        {
            H.at<float>(Point(i, j)) = 1.0 - exp(-(pow(i - H.cols / 2, 2) + pow(j - H.rows / 2, 2)) / (2 *
pow(D0, 2)));
        }

    }

    filt = k1 + k2 * H;

    FFTShift(HF, HF);
    split(HF, planes);
    for (int i = 0; i < filt.cols; i++)
    {
        for (int j = 0; j < filt.rows; j++)
        {
            planes[0].at<float>(Point(i, j)) *= filt.at<float>(Point(i, j));
            planes[1].at<float>(Point(i, j)) *= filt.at<float>(Point(i, j));
        }
    }

    Mat filteredImg;
    merge(planes, 2, filteredImg);

    FFTShift(filteredImg, filteredImg);
    ///

    /// IDFT
    Mat result;
    dft(filteredImg, result, DFT_INVERSE | DFT_REAL_OUTPUT);
    normalize(result, result, 0, 1, NORM_MINMAX);
    result.convertTo(result, CV_8U, 255);
    equalizeHist(result, result);
    ///

    dst = result.clone();
}

void DBHandler::FFTShift(const Mat& src, Mat& dst)
{
    dst = src.clone();
    int cx = dst.cols / 2;
    int cy = dst.rows / 2;

    Mat q1(dst, Rect(0, 0, cx, cy));
    Mat q2(dst, Rect(cx, 0, cx, cy));
    Mat q3(dst, Rect(0, cy, cx, cy));
    Mat q4(dst, Rect(cx, cy, cx, cy));

    Mat temp;
    q1.copyTo(temp);
    q4.copyTo(q1);
    temp.copyTo(q4);
    q2.copyTo(temp);
    q3.copyTo(q2);
    temp.copyTo(q3);
}
```

El código fuente desarrollado para crear la aplicación UI también es común para ambos sistemas operativos, por lo que el código incluido a continuación contiene las cabeceras y demás archivos pertenecientes al diseño creado en QT Creator.

mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QProcess>

QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget* parent = nullptr);
    ~MainWindow();

private slots:
    void on_pushButton_2_clicked();

    void on_pushButton_clicked();

private:
    Ui::MainWindow* ui;
};
#endif // MAINWINDOW_H
```

mainwindow.cpp

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <iostream>
#include <unistd.h>
#include <mqueue.h>
#include <signal.h>

#define TAM_MSG 128

qint64 pid;

struct sigaction sa;
siginfo_t info;
sigset_t ss;

void handler(int signal, siginfo_t* data, void*)
{

}

MainWindow::MainWindow(QWidget* parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    // Signals ////////////////////////
    sa.sa_sigaction = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_SIGINFO;
    sigaction(SIGRTMIN, &sa, NULL);

    sigemptyset(&ss);
    sigaddset(&ss, SIGRTMIN);
    sigprocmask(SIG_BLOCK, &ss, NULL);
    ////////////////////////////////////

    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
    delete ui;
}
```

```
void MainWindow::on_pushButton_2_clicked()
{
    // Login
    int uiPid = getpid();
    if (!QProcess::startDetached("cvprog", { "0", QString::number(uiPid) }, "", &pid))
    {
        throw;
    }

    int ret = sigwaitinfo(&ss, &info);
    if (ret == SIGRTMIN)
    {
        int len = info.si_value.sival_int;
        if (len > 0)
        {
            char* username = (char*)malloc(sizeof(char) * (len + 1));

            int i;
            for (i = 0; i < len; i++)
            {
                kill(pid, SIGRTMIN);
                sigwaitinfo(&ss, &info);
                username[i] = (char)info.si_value.sival_int;
            }
            username[i] = '\0';

            QString output = QString("User ") + QString(username) + QString(" successfully logged in.");
            ui->lineEdit->setText(output);

            free(username);
        }
        else
        {
            ui->lineEdit->setText(QString("User is not registered."));
        }
    }

    QProcess::startDetached("kill", { QString::number(pid) });
}

void MainWindow::on_pushButton_clicked()
{
    // Register
    QString username = ui->lineEdit->text();
    if (username.isEmpty() || username == "Insert a valid user name." ||
        username == "User registered successfully." ||
        username == "User is already registered." ||
        username == "User is not registered.")
    {
        ui->lineEdit->setText(QString("Insert a valid user name."));
        return;
    }

    int uiPid = getpid();
    if (!QProcess::startDetached("cvprog", { "1", QString::number(uiPid), username }, "", &pid))
    {
        throw;
    }

    int ret = sigwaitinfo(&ss, &info);
    if (ret == SIGRTMIN)
    {
        if (info.si_value.sival_int == 1)
        {
            ui->lineEdit->setText(QString("User registered successfully."));
        }
        else
        {
            ui->lineEdit->setText(QString("User is already registered."));
        }
    }

    QProcess::startDetached("kill", { QString::number(pid) });
}
```

main.cpp

```
#include "mainwindow.h"

#include <QApplication>

int main(int argc, char* argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```

mainwindow.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
  <class>MainWindow</class>
  <widget class="QMainWindow" name="MainWindow">
    <property name="geometry">
      <rect>
        <x>0</x>
        <y>0</y>
        <width>400</width>
        <height>120</height>
      </rect>
    </property>
    <property name="minimumSize">
      <size>
        <width>400</width>
        <height>120</height>
      </size>
    </property>
    <property name="maximumSize">
      <size>
        <width>400</width>
        <height>120</height>
      </size>
    </property>
    <property name="windowTitle">
      <string>Security System</string>
    </property>
    <widget class="QWidget" name="centralwidget">
      <widget class="QLabel" name="label">
        <property name="geometry">
          <rect>
            <x>15</x>
            <y>20</y>
            <width>40</width>
            <height>17</height>
          </rect>
        </property>
        <property name="text">
          <string>User:</string>
        </property>
      </widget>
      <widget class="QLineEdit" name="lineEdit">
        <property name="geometry">
          <rect>
            <x>70</x>
            <y>17</y>
            <width>321</width>
            <height>25</height>
          </rect>
        </property>
        <property name="text">
          <string/>
        </property>
      </widget>
      <widget class="QPushButton" name="pushButton">
        <property name="geometry">
          <rect>
            <x>100</x>
```

```
            <y>60</y>
            <width>89</width>
            <height>25</height>
          </rect>
        </property>
        <property name="text">
          <string>Register</string>
        </property>
      </widget>
      <widget class="QPushButton" name="pushButton_2">
        <property name="geometry">
          <rect>
            <x>200</x>
            <y>60</y>
            <width>89</width>
            <height>25</height>
          </rect>
        </property>
        <property name="text">
          <string>Login</string>
        </property>
      </widget>
    </widget>
    <widget class="QStatusBar" name="statusbar"/>
  </widget>
  <resources/>
  <connections/>
</ui>
```

ANEXO C: CÓDIGO FUENTE UBUNTU

cvprog.cpp

```
#include <iostream>
#include <fstream>
#include <iterator>
#include <signal.h>
#include <unistd.h>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/features2d.hpp>
#include <opencv2/xfeatures2d.hpp>
#include "sqlite3.h"
#include "DBHandler.hpp"

using namespace std;
using namespace cv;

void test();
void PerformTest(Mat& src);
tuple<vector<KeyPoint>, Mat> SURFDetector(Mat& src);
tuple<vector<KeyPoint>, Mat> SIFTDetector(Mat& src);
tuple<vector<KeyPoint>, Mat> ORBDetector(Mat& src);
tuple<Mat, int> BruteForceMatcher(tuple<vector<KeyPoint>, Mat> m1, tuple<vector<KeyPoint>, Mat> m2, Mat imgA =
Mat(), Mat imgB = Mat());
void TestMatchers();

void handler(int signal, siginfo_t* data, void*) { }

int main(int argc, char const* argv[])
{
    // Signals ////////////////////////
    struct sigaction sa;
    siginfo_t info;
    sigset_t ss;

    sa.sa_sigaction = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_SIGINFO;
    sigaction(SIGRTMIN, &sa, NULL);

    sigemptyset(&ss);
    sigaddset(&ss, SIGRTMIN);
    sigprocmask(SIG_BLOCK, &ss, NULL);
    ////////////////////////////////////

    if (argc > 1)
    {
        int action = atoi(argv[1]);

        switch (action)
        {
        case 0: // Login
            if (argc > 2)
            {
                int pid = atoi(argv[2]);
                union sigval value;
                string username;

                value.sival_int = (int)DBHandler::Login(&username);

                if (value.sival_int != 0)
                {
                    value.sival_int = username.length();
                }

                sigqueue(pid, SIGRTMIN, value); // Sends 0 if user is not found in db, size of name if success-
ful.
                sigwaitinfo(&ss, &info);

                for (int i = 0; i < value.sival_int; i++) // Send characters of username
                {
```

```
                    value.sival_int = (int)username.at(i);
                    sigqueue(pid, SIGRTMIN, value);
                    sigwaitinfo(&ss, &info);
                }

                exit(0);
            }
            break;

        case 1: // Register
            if (argc > 3)
            {
                int pid = atoi(argv[2]);
                string username = string(argv[3]);
                union sigval value;

                value.sival_int = (int)DBHandler::Register(username);
                sigqueue(pid, SIGRTMIN, value); // Sends 0 if user already registered, 1 if successful.
                exit(0);
            }
            break;

        default:
            break;
        }

        exit(1);
    }

    // No arguments -> perform test

    Mat img;

    // Take images ////////////////////
    img = imread("original4.png", ImreadModes::IMREAD_GRAYSCALE);
    imshow("Img", img);
    ////////////////////////////////////

    PerformTest(img);

    waitKey(0);
    return 0;
}

void TestMatchers()
{
    Mat img, img2, imga, imgb, resA, resB;

    // Take images ////////////////////
    img = imread("original.png", ImreadModes::IMREAD_GRAYSCALE);
    img2 = imread("original4.png", ImreadModes::IMREAD_GRAYSCALE);
    imshow("Img", img);
    imshow("Img2", img2);
    ////////////////////////////////////

    // Preprocess /////////////////////
    DBHandler::PreprocessImage(img, imga);
    DBHandler::PreprocessImage(img2, imgb);
    ////////////////////////////////////

    // SURF ///////////////////////////
    tuple<vector<KeyPoint>, Mat> surf1 = SURFDetector(imga);
    tuple<vector<KeyPoint>, Mat> surf2 = SURFDetector(imgb);
    ////////////////////////////////////

    // SIFT ///////////////////////////
    tuple<vector<KeyPoint>, Mat> sift1 = SURFDetector(imga);
    tuple<vector<KeyPoint>, Mat> sift2 = SURFDetector(imgb);
    ////////////////////////////////////

    // ORB ////////////////////////////
    tuple<vector<KeyPoint>, Mat> orb1 = ORBDetector(imga);
    tuple<vector<KeyPoint>, Mat> orb2 = ORBDetector(imgb);
    ////////////////////////////////////
```

```
    // KAZE ////////////////////////////
    tuple<vector<KeyPoint>, Mat> kaze1 = DBHandler::KAZEDetector(imga);
    tuple<vector<KeyPoint>, Mat> kaze2 = DBHandler::KAZEDetector(imgb);
    ////////////////////////////////////

    // Match Features /////////////////
    tuple<Mat, int> matchesSURF = DBHandler::FLANNMatcher(surf1, surf2, imga, imgb);
    imshow("SURF Matches", get<0>(matchesSURF));

    tuple<Mat, int> matchesSIFT = DBHandler::FLANNMatcher(sift1, sift2, imga, imgb);
    imshow("SIFT Matches", get<0>(matchesSIFT));

    tuple<Mat, int> matchesORB = BruteForceMatcher(orb1, orb2, imga, imgb);
    imshow("ORB Matches", get<0>(matchesORB));

    tuple<Mat, int> matchesKAZE = DBHandler::FLANNMatcher(kaze1, kaze2, imga, imgb);
    imshow("KAZE Matches", get<0>(matchesKAZE));
    ////////////////////////////////////
}

tuple<vector<KeyPoint>, Mat> SURFDetector(Mat& src)
{
    Mat img = src.clone();

    // SURF ////////////////////////////
    Ptr<xfeatures2d::SURF> detector = xfeatures2d::SURF::create(5000);

    vector<KeyPoint> keypoints;
    Mat descriptors;

    detector->detectAndCompute(img, noArray(), keypoints, descriptors);
    return make_tuple(keypoints, descriptors);
    ////////////////////////////////////
}

tuple<vector<KeyPoint>, Mat> SIFTDetector(Mat& src)
{
    Mat img = src.clone();

    // SURF ////////////////////////////
    Ptr<SIFT> detector = SIFT::create(1000);

    vector<KeyPoint> keypoints;
    Mat descriptors;

    detector->detectAndCompute(img, noArray(), keypoints, descriptors);
    return make_tuple(keypoints, descriptors);
    ////////////////////////////////////
}

tuple<vector<KeyPoint>, Mat> ORBDetector(Mat& src)
{
    Mat img = src.clone();

    // SURF ////////////////////////////
    Ptr<ORB> detector = ORB::create(1000);

    vector<KeyPoint> keypoints;
    Mat descriptors;

    detector->detectAndCompute(img, noArray(), keypoints, descriptors);
    return make_tuple(keypoints, descriptors);
    ////////////////////////////////////
}

tuple<Mat, int> BruteForceMatcher(tuple<vector<KeyPoint>, Mat> m1, tuple<vector<KeyPoint>, Mat> m2, Mat imgA, Mat
imgB)
{
    Ptr<BFMatcher> matcher = BFMatcher::create(NORM_HAMMING);
    vector<vector<DMatch>> knn_matches;
    matcher->knnMatch(get<1>(m1), get<1>(m2), knn_matches, 2);

    //-- Filter matches using the Lowe's ratio test
    const float ratio_thresh = 0.75f;
    std::vector<DMatch> good_matches;
```

```
    for (size_t i = 0; i < knn_matches.size(); i++)
    {
        if (knn_matches[i][0].distance < ratio_thresh * knn_matches[i][1].distance)
        {
            good_matches.push_back(knn_matches[i][0]);
        }
    }

    float maxInc = 0.34f;
    std::vector<DMatch> goodest_matches;

    for (size_t i = 0; i < good_matches.size(); i++)
    {
        int idx1 = good_matches[i].trainIdx;
        int idx2 = good_matches[i].queryIdx;

        const KeyPoint& kp1 = get<0>(m2)[idx1], & kp2 = get<0>(m1)[idx2];
        Point2f p1 = kp1.pt;
        Point2f p2 = kp2.pt;
        Point2f triangle = Point2f(std::abs(p2.x - p1.x), std::abs(p2.y - p1.y));

        float angle = std::atan2(triangle.y, triangle.x);

        if (std::abs(angle) < maxInc)
        {
            goodest_matches.push_back(good_matches[i]);
        }
    }


    cout << "Good matches: " << to_string(good_matches.size()) << endl;
    cout << "Goodest matches: " << to_string(goodest_matches.size()) << endl;

    if (imgA.rows > 1 && imgB.rows > 1)
    {
        Mat img_matches;
        drawMatches(imgA.clone(), get<0>(m1), imgB.clone(), get<0>(m2), goodest_matches, img_matches, Sca-
lar::all(-1),
            Scalar::all(-1), std::vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS);

        return make_tuple(img_matches.clone(), good_matches.size());
    }
    else
    {
        return make_tuple(Mat(), good_matches.size());
    }
}

void PerformTest(Mat& src)
{
    Mat img, img2, img3, img4, img5, img6, img7, img8, img9, img10;

    img2 = src.clone();
    // Umbralizar ////////////////////
    threshold(img2, img3, 60, 255, THRESH_BINARY);

    Mat framed = Mat::zeros(Size2d(img3.cols + 2, img3.rows + 2), CV_8U);
    Rect r = Rect2d(1, 1, img3.cols, img3.rows);
    img3.copyTo(framed(r));

    Canny(framed, img4, 10, 50);

    Mat kernel = getStructuringElement(MORPH_RECT, Size2d(5, 5));
    dilate(img4, img4, kernel);

    vector<vector<Point>> contours;
    findContours(img4, contours, RETR_TREE, CHAIN_APPROX_NONE);

    Mat contoursImg = Mat::zeros(img4.size(), CV_8U);
    double maxArea = 0;
    int mIdx = -1;
    for (int i = 0; i < contours.size(); i++)
    {
        double area = contourArea(contours[i]);
        if (area > maxArea)
```

```
        {
            maxArea = area;
            mIdx = i;
        }
    }

    if (mIdx >= 0)
        drawContours(contoursImg, contours, mIdx, Scalar::all(255), FILLED);


    bitwise_and(img2, img2, img5, contoursImg(r));
    ////////////////////////////////////

    // CLAHE ////////////////////////////
    Ptr<CLAHE> clahe = createCLAHE();
    clahe->setClipLimit(6);

    clahe->apply(img5, img5);
    imshow("CLAHE", img5);
    ////////////////////////////////////

    Mat imga, imgb, imgc, imgd, imge;

    // HFE //////////////////////
    DBHandler::HPF(img5, img6, FILTER_HFE);
    resize(img6, img8, img5.size());
    bitwise_and(img8, img8, imga, contoursImg(r));
    clahe->apply(imga, imga);
    imshow("HFE", imga);
    ////////////////////////////////////

    // HPF //////////////////////
    DBHandler::HPF(img5, img6, FILTER_HPF);
    resize(img6, img8, img5.size());
    bitwise_and(img8, img8, imgb, contoursImg(r));
    clahe->apply(imgb, imgb);
    imshow("HPF", imgb);
    ////////////////////////////////////

    // HPF + HFE ////////////////////////
    DBHandler::HPF(img5, img6, FILTER_HPF);
    DBHandler::HPF(img6, img7, FILTER_HFE);
    resize(img7, img8, img5.size());
    bitwise_and(img8, img8, imgc, contoursImg(r));
    clahe->apply(imgc, imgc);
    imshow("HPF + HFE", imgc);
    ////////////////////////////////////

    // HFE + HPF ////////////////////////
    DBHandler::HPF(img5, img6, FILTER_HFE);
    DBHandler::HPF(img6, img7, FILTER_HPF);
    resize(img7, img8, img5.size());
    bitwise_and(img8, img8, imgd, contoursImg(r));
    clahe->apply(imgd, imgd);
    imshow("HFE + HPF", imgd);
    ////////////////////////////////////

    // CGF //////////////////////////////
    DBHandler::CGF(imgd, imge);
    Mat mask;
    bitwise_not(contoursImg(r), mask);
    normalize(imge, imge, 1.0, 0.0, 4, -1, mask);
    imge.convertTo(imge, CV_8U);
    threshold(imge, imge, 0, 255, THRESH_BINARY);

    kernel = getStructuringElement(MORPH_ELLIPSE, Size(5, 5));
    dilate(imge, imge, kernel);
    erode(imge, imge, kernel);

    kernel = getStructuringElement(MORPH_ELLIPSE, Size(3, 3));
    erode(imge, imge, kernel);
    dilate(imge, imge, kernel);

    imshow("gabor filtered", imge);
    ////////////////////////////////////
```

```
    // SURF ////////////////////////////
    tuple<vector<KeyPoint>, Mat> surf = SURFDetector(imge);
    Mat surfMat;
    drawKeypoints(imge, get<0>(surf), surfMat, Scalar::all(-1), DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
    imshow("SURF", surfMat);
    ////////////////////////////////////

    // SIFT ////////////////////////////
    tuple<vector<KeyPoint>, Mat> sift = SURFDetector(imge);
    Mat siftMat;
    drawKeypoints(imge, get<0>(surf), siftMat, Scalar::all(-1), DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
    imshow("SIFT", siftMat);
    ////////////////////////////////////

    // ORB /////////////////////////////
    tuple<vector<KeyPoint>, Mat> orb = ORBDetector(imge);
    Mat orbMat;
    drawKeypoints(imge, get<0>(surf), orbMat, Scalar::all(-1), DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
    imshow("ORB", orbMat);
    ////////////////////////////////////

    // KAZE ////////////////////////////
    tuple<vector<KeyPoint>, Mat> kaze = DBHandler::KAZEDetector(imge);
    Mat kazeMat;
    drawKeypoints(imge, get<0>(surf), kazeMat, Scalar::all(-1), DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
    imshow("KAZE", kazeMat);
    ////////////////////////////////////


}
```

ANEXO D: CÓDIGO FUENTE RASPBIAN

Webcam.cpp

```
#include <iostream>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/imgcodecs.hpp>
#include <wiringPi.h>

using namespace std;
using namespace cv;

int main(int argc, char const *argv[])
{
    Mat img;
    VideoCapture vc(0);

    while (true)
    {
        // Activate Lights ////////////////
        wiringPiSetup();
        pinMode(4, OUTPUT);
        pinMode(5, OUTPUT);

        digitalWrite(4, HIGH); //980nm
        digitalWrite(5, HIGH); //850nm
        ////////////////////////////////////

        // Take image ////////////////////
        vc.read(img);
        imshow("Img", img);
        ////////////////////////////////////

        // Deactivate Lights //////////////
        digitalWrite(4, LOW);
        digitalWrite(5, LOW);
        ////////////////////////////////////

        waitKey(1);
    }

    return 0;
}
```

cvprog.cpp

```
#include <iostream>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/xfeatures2d.hpp>
#include <wiringPi.h>
#include <unistd.h>
#include <signal.h>
#include "sqlite3.h"
#include "DBHandler.hpp"
```

```
using namespace std;
using namespace cv;

Mat TakeImage();
bool LoginRP4(string* username);
bool RegisterRP4(string username);

void handler(int signal, siginfo_t* data, void*) { }

int main(int argc, char const* argv[])
{
    wiringPiSetup();
    pinMode(4, OUTPUT);
    pinMode(5, OUTPUT);

    // Signals //////////////////////
    struct sigaction sa;
    siginfo_t info;
    sigset_t ss;

    sa.sa_sigaction = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_SIGINFO;
    sigaction(SIGRTMIN, &sa, NULL);

    sigemptyset(&ss);
    sigaddset(&ss, SIGRTMIN);
    sigprocmask(SIG_BLOCK, &ss, NULL);
    ////////////////////////////////

    if (argc > 1)
    {
        int action = atoi(argv[1]);

        switch (action)
        {
        case 0: // Login
            if (argc > 2)
            {
                int pid = atoi(argv[2]);
                union sigval value;
                string username;

                value.sival_int = (int)LoginRP4(&username);

                if (value.sival_int != 0)
                {
                    value.sival_int = username.length();
                }

                sigqueue(pid, SIGRTMIN, value); // Sends 0 if user is not found
in db, size of name if successful.
                sigwaitinfo(&ss, &info);

                for (int i = 0; i < value.sival_int; i++) // Send characters of
username
                {
                    value.sival_int = (int)username.at(i);
```

```
                    sigqueue(pid, SIGRTMIN, value);
                    sigwaitinfo(&ss, &info);
                }

                return 0;
            }
            break;

        case 1: // Register
            if (argc > 3)
            {
                int pid = atoi(argv[2]);
                string username = string(argv[3]);
                union sigval value;

                value.sival_int = (int)RegisterRP4(username);
                sigqueue(pid, SIGRTMIN, value); // Sends 0 if user already
registered, 1 if successful.
                return 0;
            }
            break;

        default:
            break;
        }

        return 1;
    }

    // No arguments -> show vascularity
    Mat img, img2;
    img = TakeImage();
    DBHandler::PreprocessImage(img, img2);
    imshow("Vascularity", img2);

    waitKey(0);
    destroyAllWindows();

    return 0;
}

bool LoginRP4(string* username)
{
    Mat img, imgPre;
    img = TakeImage();
    DBHandler::PreprocessImage(img, imgPre);
    tuple<vector<KeyPoint>, Mat> features = DBHandler::KAZEDetector(imgPre);
    tuple<string, vector<KeyPoint>, Mat> bestMatch =
DBHandler::FindBestMatch(features);

    if (get<0>(bestMatch) == "")
        return false;
    else
    {
        *username = get<0>(bestMatch);
        return true;
    }
}

bool RegisterRP4(string username)
{
    Mat img, imgPre;
```

```
    img = TakeImage();
    DBHandler::PreprocessImage(img, imgPre);
    tuple<vector<KeyPoint>, Mat> features = DBHandler::KAZEDetector(imgPre);
    return !DBHandler::WriteEntry(features, username);
}

Mat TakeImage()
{
    VideoCapture vc(0);
    Mat img, img2;

    digitalWrite(4, HIGH);
    digitalWrite(5, HIGH);

    vc.read(img);
    cvtColor(img, img, COLOR_BGR2GRAY);

    Rect fingerRegion = Rect(255, 0, 465 - 255, img.rows - 40);
    img2 = img(fingerRegion);

    digitalWrite(4, LOW);
    digitalWrite(5, LOW);

    return img2.clone();
}
```

ANEXO E: CÓDIGO FUENTE ARDUINO

tfg.ino

```
void setup() {
    // put your setup code here, to run once:
    pinMode(2, OUTPUT);
    pinMode(3, OUTPUT);
    pinMode(7, INPUT);
    pinMode(8, INPUT);

    digitalWrite(2, LOW);
    digitalWrite(3, LOW);
}

void loop() {
    // put your main code here, to run repeatedly:
    if (digitalRead(7))
    {
        digitalWrite(2, HIGH);
    }
    else
    {
        digitalWrite(2, LOW);
    }

    if (digitalRead(8))
    {
        digitalWrite(3, HIGH);
    }
    else
    {
        digitalWrite(3, LOW);
    }
}
```