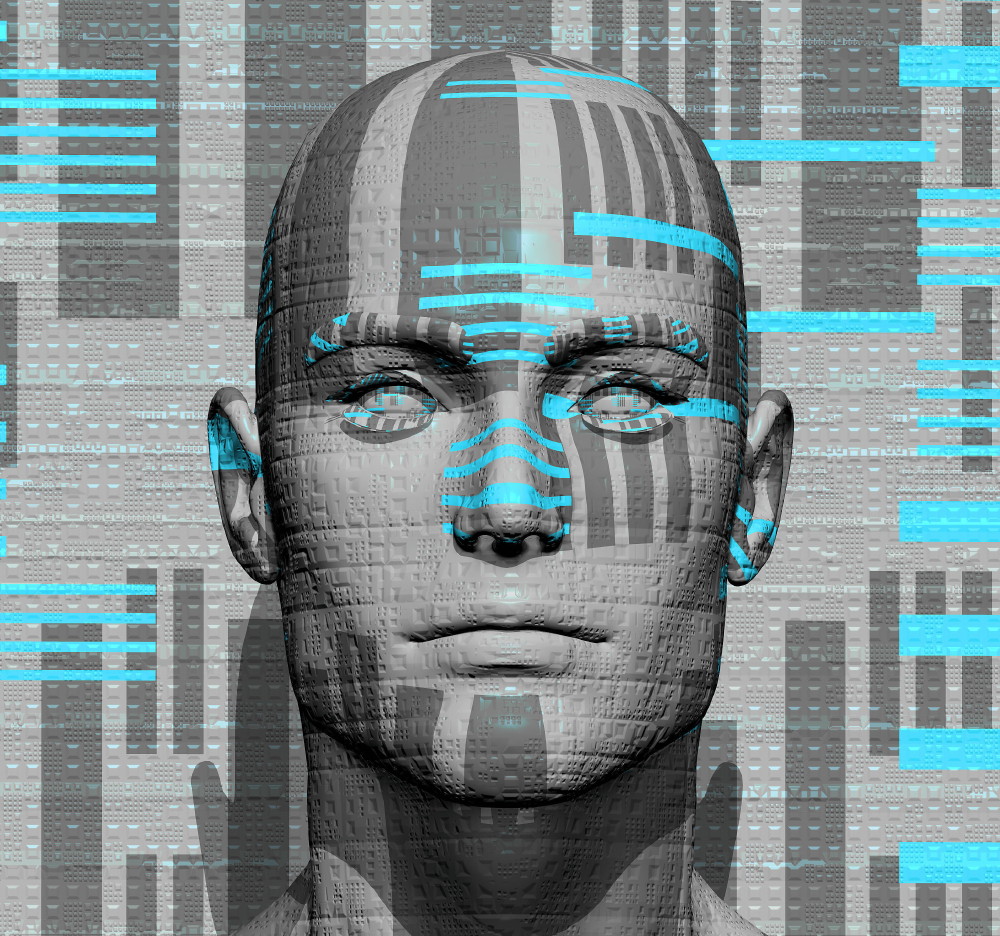
**Visión Por Computador**

**Reconocimiento de Caras**



**Realizado por:**

Francisco Pérez Hernández

Cristina Zuheros Montes

1 Introducción

El objetivo de nuestro trabajo es realizar una técnica con la que podamos reconocer caras en una fotografía.

Para realizar esta tarea lo primero que hemos conseguido es una base de datos de rostros [1]. Estos rostros tienen una limitación, cada fotografía es de un solo rostro en distintas condiciones de iluminación.

Algunos problemas que presuponemos de inicio son:

* Nuestro algoritmo tendrá distintos resultados en función de la iluminación.
* Los rostros no deberían salir en otra posición que no sea vertical.

Por lo tanto, y viendo estos inconvenientes, vamos a proponer una serie de técnicas como son:

* Pasar imágenes en RGB, de color carne a blanco-negro.
* Pasar imágenes en RGB a YCrCb, con lo que obtenemos la piel en las imágenes.

2 Pasar de color carne a blanco-negro

La primera idea que puede saltar cuando piensas como debería actuar el algoritmo, sería sacar el color carne de una persona y a partir de ahí obtendríamos el cuerpo de esta. Esto supone una limitación, y es que las personas con un tono de piel que no sea color carne (dentro de unos rangos),como personas con un tono más amarillo, o marrón, no serán localizadas.

Sacada esta idea, se ha buscado cual podría ser el tono color carne medio, para poder tener un algoritmo que tenga buenos resultados. Para ello también tendremos una tolerancia, para que el rango de colores pueda ser más amplio.

Por lo tanto, nuestro algoritmo será:

Vector<Mat> DetectarRosa(vector<Mat> imágenes\_caras, int tolerancia);

Por lo tanto, nuestro algoritmo recibirá un vector con imágenes y una tolerancia determinada. Este nos devolverá un vector con las imágenes pasadas a blanco-negro.

Tras diferentes pruebas se ha visto que los valores de RGB, que mejor función, junto con la tolerancia, son los siguientes:

R=253, G=221, B=201, tolerancia=70

La tolerancia es lo que nos permite desplazarnos entre los tonos de R, G y B, para que así entre en el rango de valores más tonos de color carne.

Para estos valores, algunos de los resultados obtenidos son:









Como se puede ver en las imágenes, los resultados de este algoritmo son muy malos ya que en función de la iluminación, podría reconocer color carne en zonas donde no las hay.

En conclusión, este algoritmo no es lo suficiente bueno para poder trabajar sobre el. Por lo tanto, en el siguiente apartado vamos a implementar otra técnica, con la cual vamos a convertir la imagen RGB a YCrCb para poder quitar iluminación de las imágenes.

3 Sacar piel de las imágenes

Como se ha visto en el apartado anterior, sacar el color carne en una imagen obtiene muy malos resultados. Por lo tanto, vamos a pasar a detectar la piel de una imagen.

Para ello nos hemos basado en este paper [2], en el cual usa esta idea y vamos a llevarla a cabo.

3.1 Sacar piel

Primeramente, creamos la función:

Mat TransformarDeRGBAYCrCbYPasoABlancoNegro(Mat, valY, minCr, maxCr, minCb, maxCb);

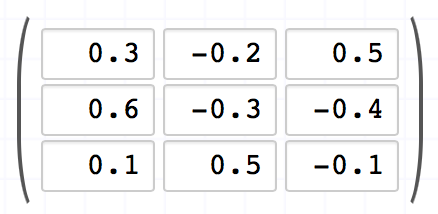
Con esta función, vamos a aplicar un filtro a la imagen en RGB para quitar la iluminación. Como pensamos que podríamos modificar algo de la transformación que hacia OpenCV, la hemos implementado nosotros para poder controlarla en caso de no tener resultados del todo positivos.

Lo primero que hacemos es transformar la imagen de RGB a YCrCb con un método auxiliar que llamamos “RGBtoYCrCb” (lo explicaremos a continuación). Una vez que tenemos la imagen en YCrCb vamos a buscar la piel de acuerdo a los valores Y, Cr, Cb. Establecemos las cotas según hemos ido viendo que los resultados eran mejores o peores. Aquellos pixeles que tengan unos valores de Y, Cr y Cb dentro del umbral que imponemos (tonos de piel) vamos a pasarlos a color blanco y los que estén fuera los pasaremos a color negro. De este modo tendremos la imagen en tonos blancos y negros donde el blanco representa la piel de la imagen.

Veamos el método:

Mat RGBtoYCrCb(Mat im\_original);

Creamos la matriz de transformación :



Para pasar de RGB a YcrCb, multiplicamos el vector RGB por la matriz de transformación. Hemos comparamos los resultados con el método que ofrece OpenCV “ConverTo(RGBtoYCrCb)” y hemos visto que son exactamente iguales a los que obtenemos.

Ahora vamos a ver algunos resultados de este proceso:







Como podemos observar, el segundo ejemplo no reconoceremos la cara debido a que las condiciones de iluminación no nos permiten hacer nada para poder obtener la piel. En el resto de ejemplos que se muestran aquí, se puede observar como se reconoce bastante bien la piel y vamos, a partir de estas imágenes, intentar recortar la cara, que será nuestro paso siguiente.

Por lo tanto, y como acabamos de comentar, vamos a pasar a recortar la cara en las imágenes en blanco y negro, y a eliminar las posibles imágenes que no se hayan podido reconocer la piel.

Hemos creado un método con el cual si no se ha conseguido un determinado número de píxeles blancos en toda la imagen, no se ha reconocido una cara. Por lo tanto en nuestra base de datos de 450 imágenes, no se han reconocido suficiente piel en 196 de ellas, lo que sería un 43,56% de las imágenes totales.

Algunas de las imágenes que no se han reconocido cara es como la mostrada en el segundo ejemplo en la que se ve todo absolutamente negro. Ahora pasaremos a trabajar sobre las que si se ha reconocido cara.

3.2 Recortar piel

Una vez que tenemos la imagen en blanco y negro, vamos a pasar a recortar la cara sobre las 254 imágenes que se ha reconocido suficiente piel.

Para recortar la cara vamos a buscar que haya suficientes blancos concentrados en una zona y nos quedaremos con esa zona. Para ello vamos a recortar primero las zonas laterales, eliminando grandes cantidades de negro, y seguidamente pasaremos a las zonas horizontales.

Como resultado de este recorte obtenemos:







Como podemos observar, se han recortado los rostros con un gran volumen de blanco. Tenemos como inconveniente que de algunas imágenes solo se queda con partes que son los ojos, pero no nos supone demasiado problema ya que en el paso siguiente vamos a pasar a detectar los ojos en estos recortes en blanco y negro.

Si echamos un vistazo sobre todas las imágenes recortadas para ver cuantas imágenes son incorrectas podemos ver que de 254 imágenes, se han recortado mal un total de 16 imágenes, lo que sería un 6,3%.

Por lo tanto, una vez obtenido esto, vamos a pasar a buscar los ojos en estos recortes de imágenes.

3.3 Verificar si hay ojos en la cara

Una vez que tenemos el rostro recortado vamos a pasar a buscar los ojos. Para ello, el primer método que hemos realizado es buscar zonas negras en cada fila. De esta forma recorrerá fila a fila, y buscará franjas en las que haya dos zonas negras seguidas consecutivas. En caso de que no haya dos zonas exactamente, esa fila se desestimará.

Una vez que se tiene un conjunto de filas, pinto un rectángulo donde se presupone que está el ojo. Como resultados podemos ver los siguientes:

Por lo tanto se ve como en la primera, se hace perfectamente. En la segunda coge una zona demasiado amplia empezando en el pelo, y en la tercer solo coge una fila, y esta serían las cejas.

Como resultado del algoritmo, se han producido como salidas incorrectas en las que no se detectan ojos o zonas que podrían ser ojos, 80 imágenes de las 254 que estábamos tratando, es decir, un 31,5%.

Al ver que este método es bastante malo se ha intentado realizar otra búsqueda de ojos con la siguiente idea:

Empezar cada fila desde el principio, haciendo una relación de distancia entre dos ojos. De esta forma comprobaremos el pixel x de cada fila con el x+distancia\_entre\_ojos, para comprobar que ambos son negros y esto nos diría que son exactamente los ojos.

El resultado de este nuevo algoritmo no ha mejorado casi en nada al anterior descrito, por lo que se ha desestimado y se ha trabajado en otro que se explicará más adelante (Punto 5).

4 Aplicar filtro gaussiano

Como idea para reconocer rostros, hemos pensado en sacar la baja frecuencia de una imagen y restarla a la imagen original, obtendríamos una imagen con el rostro en alta frecuencia.

Para ello hemos hecho uso de funciones implementadas en el trabajo uno, realizadas por Cristina. Estas funciones las hemos añadido en el fichero “Header.h” que también se adjunta en la carpeta de entrega.

Como resultado hemos obtenido:

Podemos pensar en buscar los ojos fijándonos en los blancos producidos por el flash, pero no sería un criterio bueno

A la vista de los resultados obtenidos, hemos pensado que seguir trabajando sobre estas imágenes no nos iba a proporcionar mejores resultados. Por lo tanto hemos desechado la idea y vuelto a las imágenes que ya teníamos recortadas para hacer un buscador de ojos que funcionase mejor.

5 Segundo buscador de ojos

Como hemos visto que aplicando el filtro gaussiano no íbamos a conseguir nueva información interesante, hemos procedido a implementar otro método de búsqueda de ojos dentro de las imágenes recortadas.

Este método va a consistir en:

Recibimos, una a una, una imagen original en RGB sin recortes, su imagen en RGB recortada y su imagen en blanco y negro detectando la piel recortada.

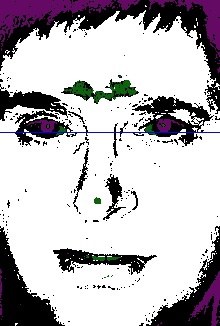
Empezamos considerando la imagen en blanco y negro recortada con la supuesta cara. En las zonas negras, tendremos los ojos, boca, nariz, pelo…., es decir, aquello que no sea piel. Nos vamos a fijar en estas zonas para detectar los ojos. En concreto, vamos a mirar que los valores de los pixeles negros de estas imágenes, sean valores en la imagen a color que correspondan con las pupilas y la zona blanca de los ojos.

Para ello hemos implementado los métodos “ParecePupilaRGB” y “PareceBlancoOjoRGB”, donde vemos que los valores RGB correspondan con tonos “negros” y tonos “blancos”.

Una vez detectadas las zonas de posibles pupilas y posibles blancos de ojos, las pintamos a color morado y verde, respectivamente, en la imagen blanco y negro recortada de la posible cara.

Ahora vamos a buscar los ojos. Para ello comenzamos en la mitad de la imagen, empezaremos buscando hacia arriba la primera coincidencia que podría llegar a ser ojo. Si llegamos hasta el final y no encontramos ojo, empezaríamos desde la mitad de la imagen hacia abajo. En caso de que no se haya encontrado ojo, procederemos a recortar la imagen por los laterales para eliminar zonas que puedan ser pelo que se podrían confundir con las pupilas por el tono. Este proceso eliminará 5 píxeles de cada lateral en un bucle de como máximo 5 iteraciones.

Como resultado se obtiene:

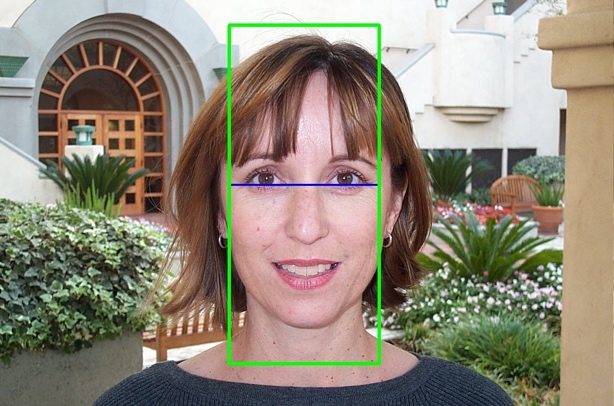
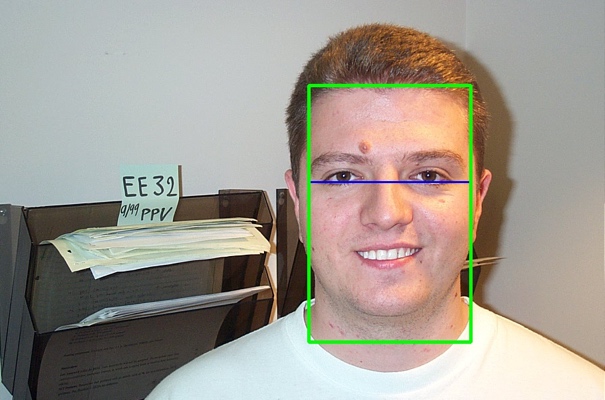
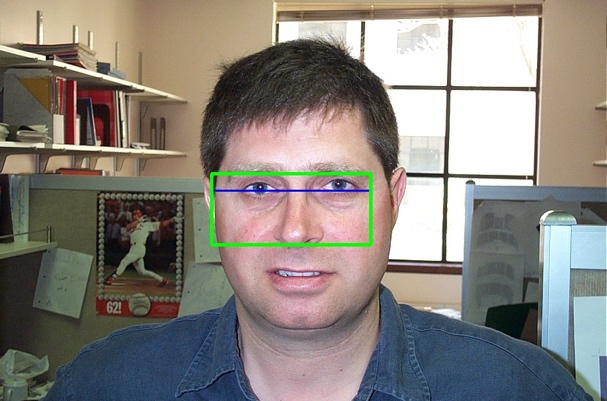
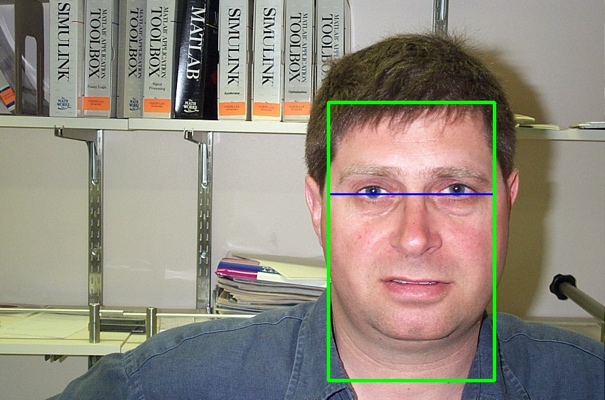
    

Se puede ver como encuentra satisfactoriamente los ojos en las imágenes, y como en caso de no encontrarlas se hace el recorte hasta encontrar los ojos.

Con este algoritmo se puede ver que se han encontrado ojos en 225 imágenes de 254 analizadas, por lo que tenemos un porcentaje del 88,58% de acierto.

Finalmente, marcamos la zona del ojo en la imagen original en RGB. Además, pintaremos un recuadro alrededor de la cara reconocida.

El resultado obtenido es el siguiente:

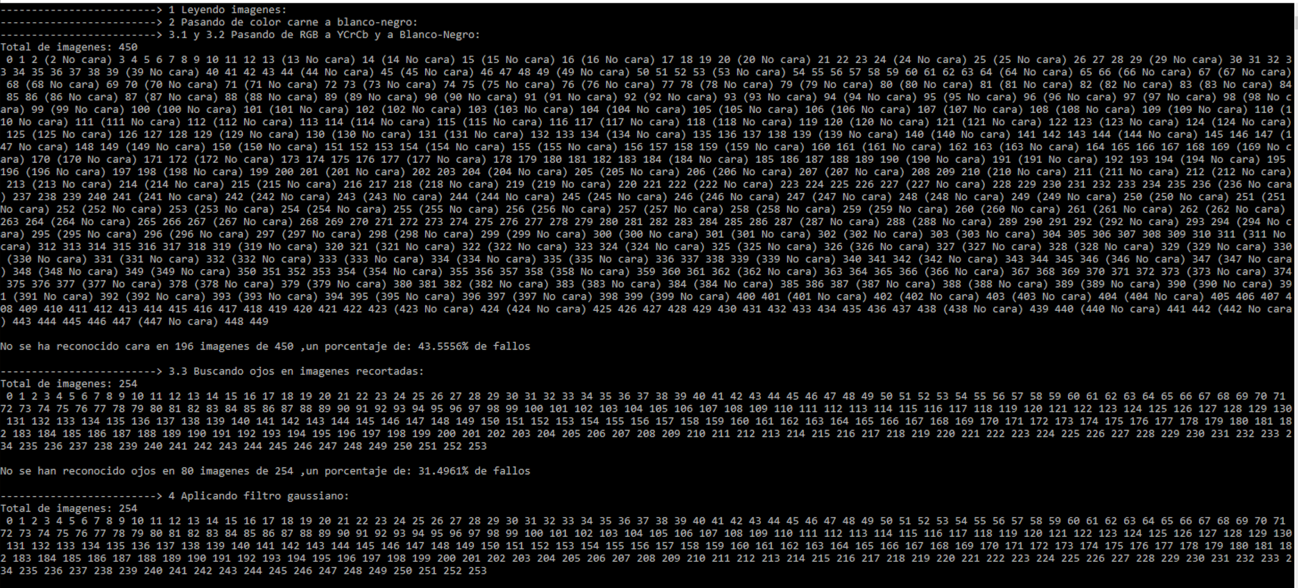


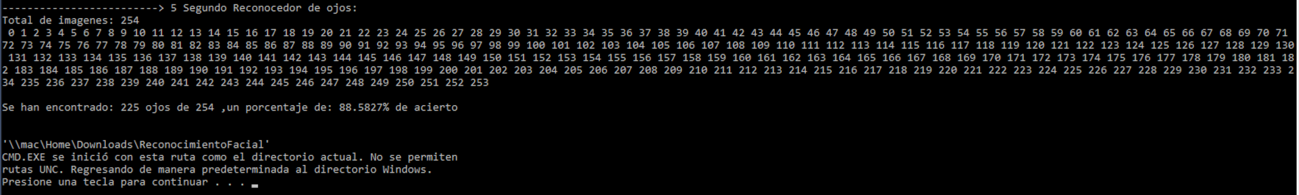
Podemos ver como se obtienen resultados buenos y otros no tan buenos como podría ser la cuarta imagen en la que el hombre esta perfectamente visible a nuestros ojos pero el algoritmo no consigue detectarle la cara.

Como una asunción que nosotros tenemos a la hora de reconocer la cara es que debe tener los ojos abiertos, por lo tanto en la tercera foto, al tener los ojos cerrados, no se dice que se detecta cara y no se identifica como tal.

Otros aspectos

Las salidas que nos proporciona el algoritmo hasta el momento son las siguientes:





6 Buscando rostros en las imágenes “malas”

Con nuestro algoritmo visto en “*Segundo buscador de ojos”* obtenemos bastantes buenos resultados, pero hay imágenes en las que no se reconoce cara. Vamos a intentar tratarlas para detectarlas.

Hemos intentando aplicar filtros Gaussianos, usar gradientes, dejar las bajas y altas frecuencias de la imágenes, aplicar filtros de Canny, incluso detectar puntos en correspondencias tras tratar las imágenes con ciertas combinaciones de estos filtros que hemos ido comentado. Pero en ningún caso hemos obteniendo resultados que nos inspirasen para hacer un nuevo método de búsqueda de ojos que funcionase relativamente bien.

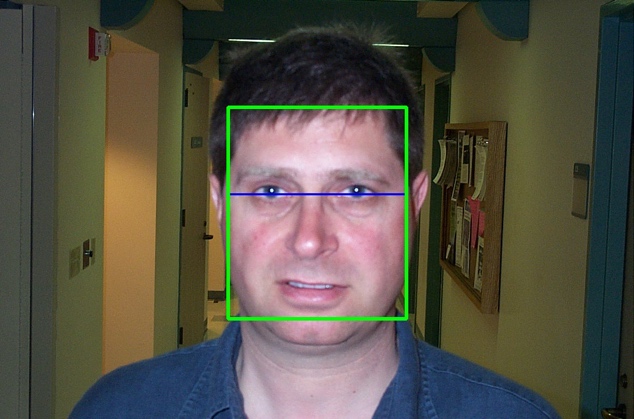
Tras estos intentos fallidos, pero de aprendizaje propio, hemos pensado que lo mejor era volver a usar nuestro algoritmo que vimos en “*Segundo buscador de ojos”.* Esta vez pasamos al método sólo las imágenes malas, con la diferencia de que ahora los valores de la piel que nosotros establecemos como buenos van a suprimir la importancia de la componente Y, y van a establecer (110< Cr <140) y (140<Cb<170).

Podemos ver que se encuentra una gran cantidad de caras. Sin embargo, al darle mayor tolerancia, se producen algunos errores (se consideran como ojos zonas de la imagen que en realidad no son ojos). Al darnos cuenta de esto, hemos ido viendo las salidas y contabilizando las imágenes en las que realmente se obtiene la cara y los ojos. Los resultados obtenidos han sido los siguientes:

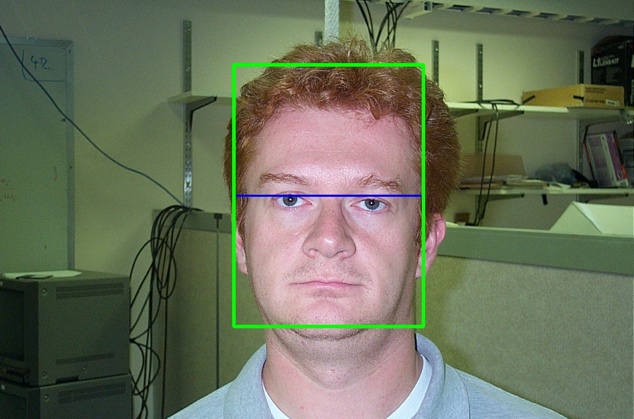
* De 196 imágenes\_malas que le pasamos al método, no se reconoce cara en 63 de ellas, lo que se traduce en un 32,14% de fallos.
* De esas 133 imágenes que obtenemos con, supuestamente, ojos y cara, hemos visto que realmente son 115 las que salen bien. Es decir, hay 18 imágenes en las que se reconoce la cara y los ojos en otro lugar de donde realmente están. Hay un 13,6% de error.

Por lo tanto, si antes teníamos 254 imágenes buenas y le sumamos las 115 actuales tenemos un total de 369, lo que es lo mismo que un 82% del total. Un valor muy bueno.

Algunas de las salidas que obtenemos son:







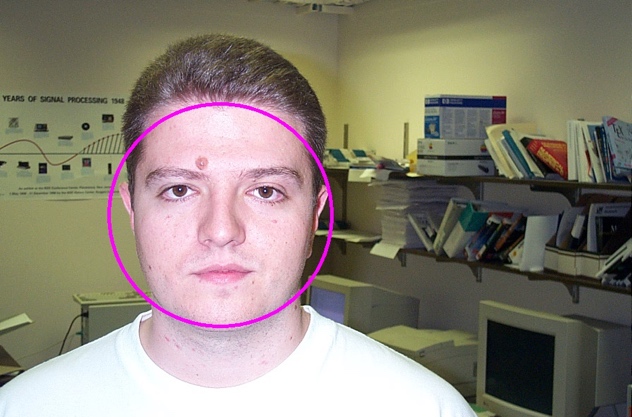
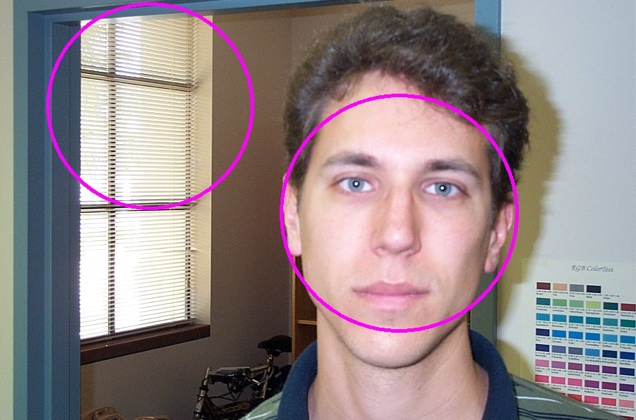
Por lo que vemos que hemos podido recuperar una gran cantidad de caras, aun metiendo más error en nuestro detector, pero hemos considerado que era mejor obtener más caras sacrificando meter malas muestras.

7 Detector propio de OpenCV

Con motivo de tener una comparativa con otro detector, hemos realizado el que trae OpenCV. Este detector nos da como resultado de imágenes buenas:

Pero hay que decir que en algunas de ellas tiene estos resultados, los cuales no son totalmente adecuados según nuestro parecer, ya que debería ser algo más preciso:





OpenCV nos devuelve que de 450 imágenes, nos proporciona buenas 372, lo que es un 82% de acierto. Consigue un resultado igual a nosotros gracias a la segunda pasada sobre imágenes que no se había reconocido cara.

8 Sacar ojos a partir del filtro gaussiano

Con motivo de usar otra técnica para detectar ojos, hemos intentado avanzar con la idea del filtro gaussiano. Lo primero que hacemos es como en el punto 4, pero a toda la imagen, por lo tanto va a ser un proceso bastante lento. Una vez tenemos la imagen con el filtro vamos a pasar a detectar los ojos en esa imagen. Para ello se han implementado dos técnicas, pero como ahora se verá, no funcionan como sería de esperar.

8.1 Primera máscara

La primera técnica que se ha pensado, ha sido crear una máscara, es decir una matriz del tamaño de los ojos, la cual se va a pasar por toda la imagen. La máscara tendría esta pinta:

../../../../Desktop/Captura%20de%20pantalla%202016-02-06%20a%20las%2014.38.0

De forma que la vamos a pasear por toda la imagen y vamos a contar cuantos blancos hay en las zonas blancas, y nos quedaremos con la zona que más blanco tenga, suponiendo que esta será la zona de los ojos.

Como resultados podemos ver:

Los cuales son muy malos.

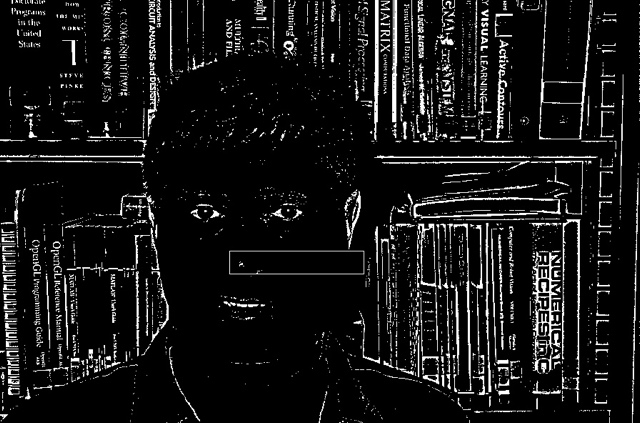
8.2 Segunda máscara

Otra idea ha sido copiar la zona de los ojos de la primera imagen, quedando:

Francisco/documento/8.2/0.jpg

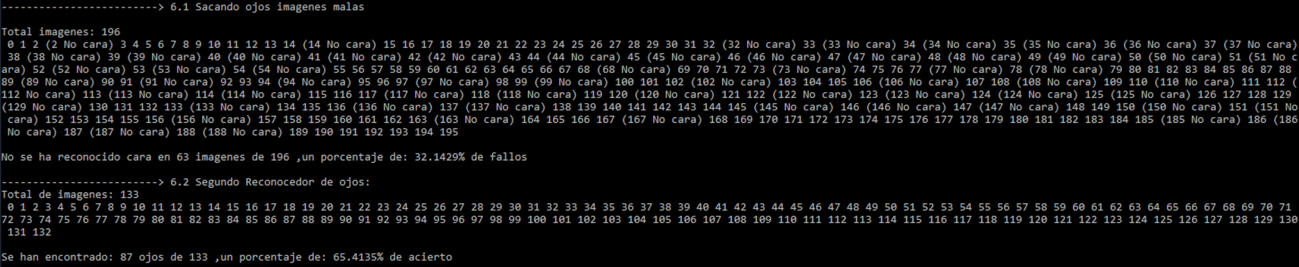
Y una vez que se tiene, pasear esta zona por toda la imagen para ver si en las demás imágenes se tiene como coincidencia los ojos.

Como resultado se pueden ver:

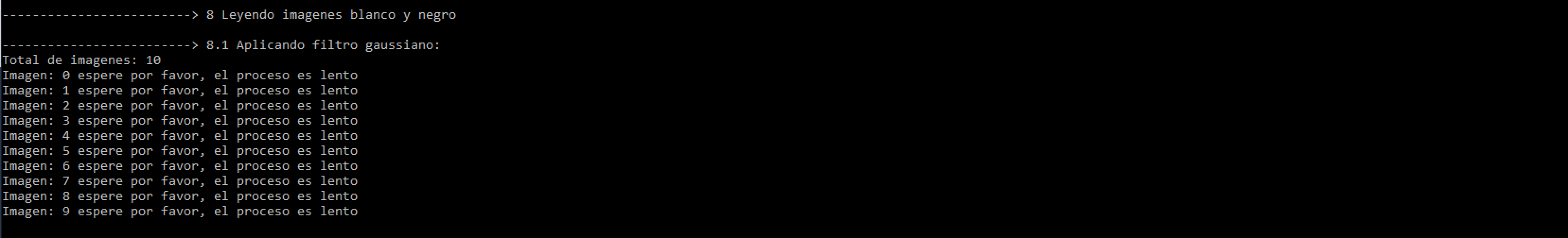


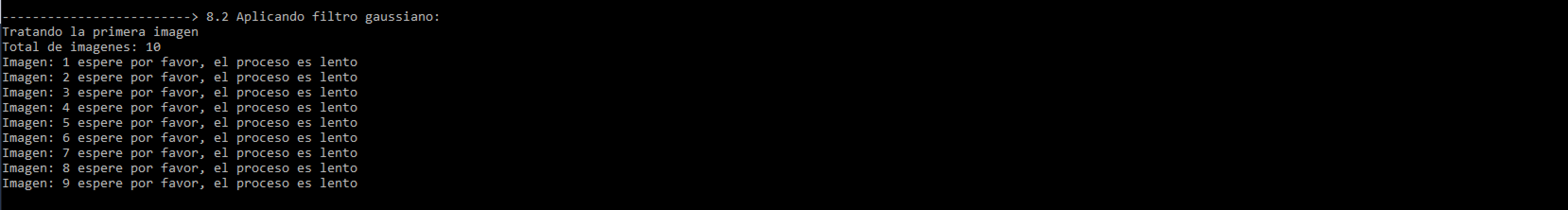
Podemos ver que los resultados son bastante malos, por lo que no creemos que esta técnica nos pueda servir en un futuro.

Otros aspectos



Francisco/documento/Captura%20de%20pantalla%202016-02-06%20a%20las%2014.17.38.png





9 Conclusiones

La detección de un rostro en una imagen no es un problema fácil de resolver. Como hemos visto hay que tratar distintos aspectos para poder hacer una detección correcta. Tenemos que tener en cuenta los valores de los píxeles (tonos de la piel), las formas que estos valores toman en las matrices (forma de las facciones de la cara), la iluminación de las fotografías que hacen que el tono de la piel pueda confundirse, la simetría del rostro….

Hemos conseguido desarrollar varios algoritmos que no dan resultados muy buenos pero nos han permitido encontrar y solventar dificultades. Para ello hemos conseguido desarrollar otro algoritmo, algo más complejo y también de invención propia, que consigue unos resultados prácticamente tan buenos como la técnica Haar que comparte OpenCV. En ambas técnicas, tanto en la usada de OpenCV como la nuestra, se detectan, a veces, como caras zonas de la imagen que en realidad no son caras. En ambos casos, este porcentaje de fallo en mínimo, de modo que interesa permitirlo con el fin de obtener un porcentaje de acierto mayor.

Enfocándonos en este algoritmo que hemos desarrollado, cabe destacar que se podrían hacer mejoras cambiando los valores que establecemos como tonos de piel, incluso cambiando los valores que establecemos como zonas de pupila o zonas blancas del ojo, así como la distancia entre ellas. Para mejorar el porcentaje de acierto de nuestro método, podríamos hacer una tercera pasada sobre las imágenes en las que no se ha detectado cara. No hemos querido realizarlo para mantener un equilibrio entre eficiencia y buenos resultados, pero sería una buena forma de detectar más caras.

X Referencias

[1] – Base de datos de caras:

<http://www.vision.caltech.edu/html-files/archive.html>

[2] – Paper sobre sacar piel:

<http://www.scielo.org.co/scielo.php?pid=S0122-34612011000100009&script=sci_arttext>

<http://web.archive.org/web/20140903080533/http://www.jpeg.org/public/jfif.pdf>

[3] – Paper asef

<http://www.cs.colostate.edu/~vision/publications/Bolme2009Asef.pdf>

[4] – Paper Color Invariant Object Recognition Using Entropic Graphs

<https://ivi.fnwi.uva.nl/isis/publications/2006/vanGemertIJIST2006/vanGemertIJIST2006.pdf>

[5] – Libro Computer Vision: Algorithms and Applications

<http://szeliski.org/Book/drafts/SzeliskiBook_20100903_draft.pdf>

[6] – Repositorio GitHub Personal

<https://github.com/PacoPollos/Vision_Por_Computador.git>

[7] – Enlace a Dropbox con todas las imágenes salidas de los algoritmos, y las imágenes originales <https://www.dropbox.com/sh/pqa9ajchw2kz824/AABt5IbAmXdYhLvOuecp7PuLa?dl=0>