

# VIA - Primera Entrega

Jose Antonio Lorencio Abril



Profesor: Alberto Ruiz García

Curso: 2021/2022

Convocatoria: Junio

## Contents

<b>1 Ejercicio 1 - Calibración</b>	<b>3</b>
<b>2 Ejercicio 2 - Actividad</b>	<b>11</b>
<b>3 Ejercicio 3 - Color</b>	<b>15</b>
<b>4 Opcional: Ejercicio 4 - Filtros</b>	<b>18</b>
<b>A Notebook findcam.ipynb</b>	<b>21</b>
<b>B Notebook segmentacionDensa.ipynb</b>	<b>28</b>

# 1 Ejercicio 1 - Calibración

## Apartado A

Se nos pide calibrar la cámara de nuestro móvil. Es decir, encontrar el valor de la **distanzia focal**,  $f$ , de forma precisa y de forma aproximada.

Para hacerlo de **forma precisa**, disponemos del programa *calibrate.py* (como se explica en [1]), al que le pasamos un conjunto de imágenes de un chessboard diseñado para este propósito. Este programa, básicamente, busca en las imágenes las 4 esquinas del tablero y cuando ha analizado todas ellas, llama a la función de opencv *calibrateCamera*, pasándole estos puntos, así como los puntos de referencia del tablero. Esta función nos devuelve la información que buscamos, y algunas medidas relativas al error cometido estimado.

Así, lo primero que hacemos es imprimirnos una copia de *pattern.png* y realizar varias fotos desde distintos ángulos de la misma.

Tras esto, ejecutamos el comando

```
python .\calibrate.py "imgs/*.jpg"
```

Y obtenemos los siguientes resultados:

$$RMS = 5.3252464724981525$$

$$\text{camera matrix} = \begin{pmatrix} 3487.81 & 0 & 1798.26 \\ 0 & 3483.37 & 1548.94 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\text{distorsion coefficients} = (-0.338 \quad 1.73 \quad -0.00385 \quad -0.00025 \quad -2.8)$$

Por lo que  $f_P \simeq 3488$ .

## Apartado B

Para la **calibración aproximada**, he fotografiado una losa del suelo de mi piso, que mide 40cm, desde una distancia de 55cm. La imagen es esta:



En la imagen, la losa mide unos 2280 píxeles. Así, utilizando la ecuación

$$u = f \frac{X}{Z}$$

y sabiendo que  $u = 2280$ ,  $X = 40$ ,  $Z = 55$ , tenemos que

$$f_A \simeq \frac{uZ}{X} = 3135$$

Que es ligeramente distinto del valor preciso. Concretamente, un 10.13% menor. Este error entra dentro del rango mencionado en el notebook explicativo ( $\pm 30\%$ ).

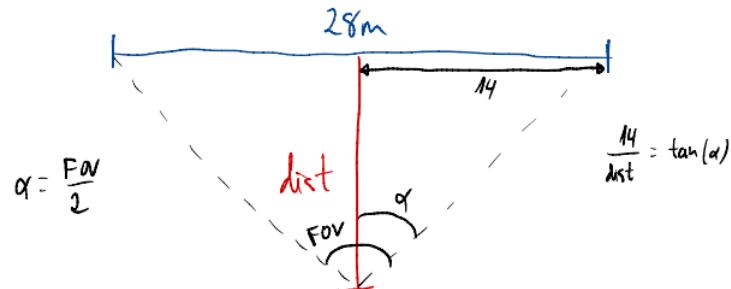
### Apartado C

Según el reglamento oficial español ([2]) los campos de baloncesto deben medir  $28m \times 15m$ . Por tanto, nuestra cámara debe ponerse suficientemente alejada para que tanto el ancho como el largo del campo queden dentro de su campo de visión. Como mi cámara es de  $3000 \times 4000px$ , y  $\frac{3}{4} > \frac{15}{28}$ , entonces basta con hacer que los 28 metros de largo de la pista entren en los 4000 px de mi cámara.

Calculamos el FOV vertical

$$\tan\left(\frac{FOV}{2}\right) = \frac{\frac{h}{2}}{f} = \frac{2000}{3488} = 0.5734 \implies \frac{FOV}{2} = 0.5206 \implies FOV = 1.041rad \simeq 60^\circ$$

Observemos ahora el siguiente diagrama



mediante el que vemos que es

$$dist = \frac{14}{\tan\left(\frac{FOV}{2}\right)} = \frac{14}{\sqrt{3}/3} \simeq 24.25m$$

Otra forma de hacerlo es usar la fórmula

$$px = f \frac{tam}{dist}$$

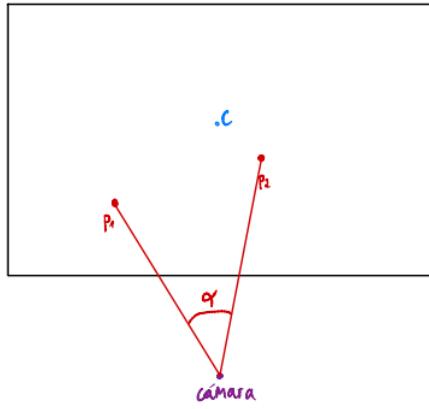
obteniendo

$$dist = f \frac{tam}{px} = 3488 \frac{28}{4000} = 24.4m$$

que son resultados suficientemente cercanos como para fiarnos de ellos.

### Apartado D

En el siguiente diagrama se ilustra qué queremos que haga el programa:



En esta situación, conocemos:

- $cam = (\frac{w}{2}, \frac{h}{2}, 0)$
- $C = (\frac{w}{2}, \frac{h}{2}, dist')$
- $p_1 = (x_1, y_1, dist') , p_2 = (x_2, y_2, dist')$

donde  $dist'$  es la distancia expresada en px. Esto coincide con  $f$ .

Y podemos obtener:

- $\overrightarrow{camp_1} = (x_1 - \frac{w}{2}, y_1 - \frac{h}{2}, f)$
- $\overrightarrow{camp_2} = (x_2 - \frac{w}{2}, y_2 - \frac{h}{2}, f)$

Haciendo el el producto escalar entre estos dos vectores tenemos

$$\langle \overrightarrow{camp_1}, \overrightarrow{camp_2} \rangle = \left( x_1 - \frac{w}{2} \right) \left( x_2 - \frac{w}{2} \right) + \left( y_1 - \frac{h}{2} \right) \left( y_2 - \frac{h}{2} \right) + f^2 = \| \overrightarrow{camp_1} \| \| \overrightarrow{camp_2} \| \cos \alpha$$

Por tanto

$$\alpha = \arccos \left\{ \frac{\left( x_1 - \frac{w}{2} \right) \left( x_2 - \frac{w}{2} \right) + \left( y_1 - \frac{h}{2} \right) \left( y_2 - \frac{h}{2} \right) + f^2}{\| \overrightarrow{camp_1} \| \| \overrightarrow{camp_2} \|} \right\} \quad (1)$$

Esto queda implementado en *angulos.py*. Aquí voy a comentar las partes más importantes del mismo:

- Básicamente, guardo un estado, indicando si hay uno, dos o ningún punto seleccionado. Se cambia de estado al hacer doble click sobre algún lugar de la imagen.
- En el estado 0 simplemente se muestra la imagen, y en el estado 1 la imagen con el primer punto seleccionado marcado. En el estado 2 se muestran los dos puntos y el ángulo que forman con la cámara.
- Para calcular el ángulo se utiliza la fórmula (1), para lo que he implementado un producto escalar y una norma (esto no era necesario pues *numpy* ya las tiene implementadas).

Como ejemplo, ejecutamos *python angulos.py "losa.jpg"*. Y observamos la siguiente secuencia de sucesos:

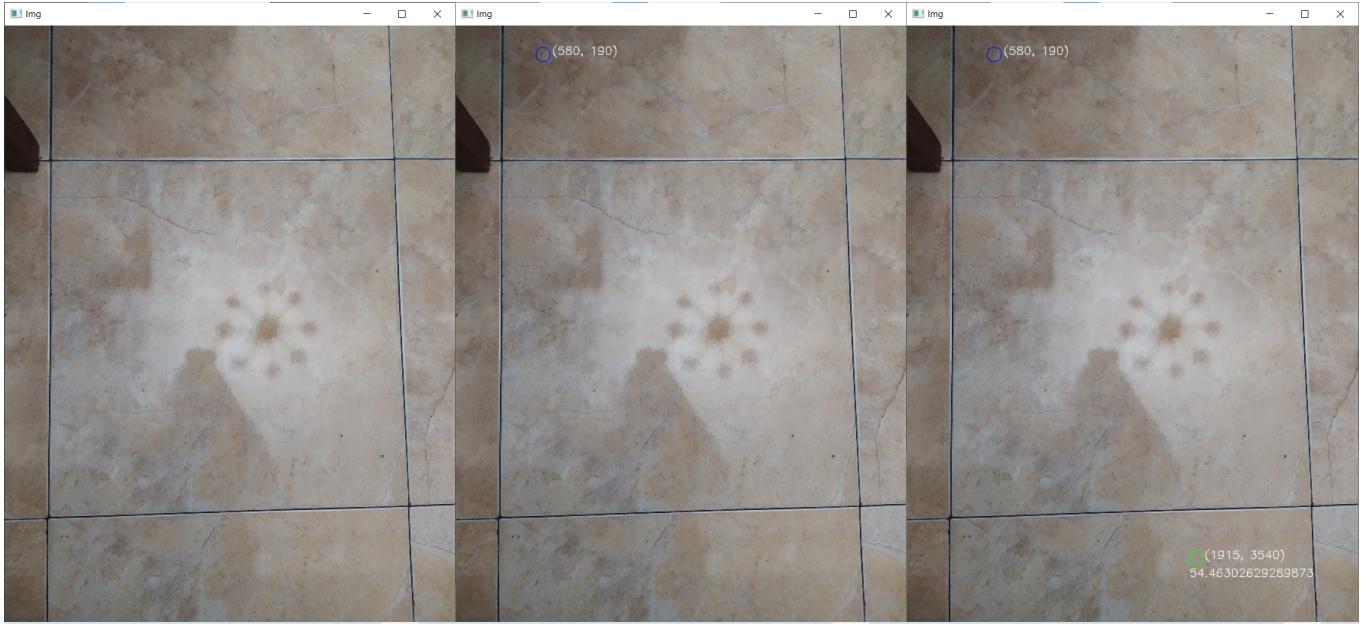


Figure 1: Ejemplo de ejecución

### Apartado E: Opcional

Suponemos que proyectamos la imagen sobre una recta, y obviamos la altura de la cámara, para poder utilizar geometría del plano y algunas propiedades básicas de las circunferencias.

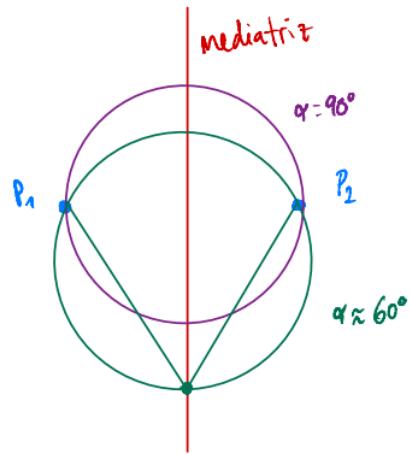
Por un lado, sabemos que dada una circunferencia y dos puntos,  $P_1, P_2$ , fijados de la misma, si tomamos un tercer punto,  $P_3$ , el ángulo  $\alpha = \angle(\overrightarrow{P_3P_1}, \overrightarrow{P_3P_2})$  es el mismo (esta propiedad es muy conocida, y puede ser consultada en [3], por ejemplo), independientemente del  $P_3$  escogido.

Por otro lado, el lugar geométrico de los puntos del plano de aquellas circunferencias que contienen a dos puntos dados es su mediatrix.

De esta forma, si conocemos dos puntos  $P_1$  y  $P_2$  en la imagen, y consideramos que  $P_3$  es el punto donde se encuentra la cámara, dado un ángulo  $\alpha$ , podemos estimar  $P_3$  de la siguiente forma:

1. Trazamos la mediatrix de  $P_1$  y  $P_2$ .
2. Cada punto de la mediatrix, determina una única circunferencia, y el ángulo  $\beta$  de sus puntos con  $P_1$  y  $P_2$  será fijo para cada punto. Por tanto, basta encontrar un punto de la mediatrix que nos dé una circunferencia con  $\beta = \alpha$ .
3. Nótese que habrá dos puntos de la mediatrix que verificarán esta propiedad, pero si consideramos que la imagen está orientada, podemos seleccionar uno de ellos.
4. Por último, debemos seleccionar un punto de la circunferencia. Pero todos verifican las propiedades del enunciado, por lo que no podemos acertar de forma segura. Una opción razonable parece tomar el punto de la mediatrix que corte a la circunferencia en la dirección de la imagen. Así obtendremos  $P'_3$ , una aproximación de  $P_3$ .

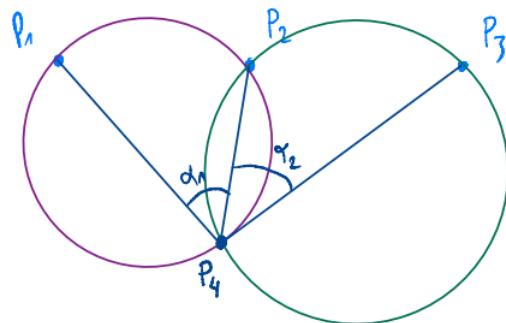
Esta situación queda ilustrada en el siguiente diagrama:



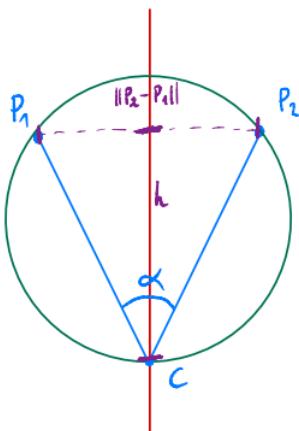
La localización se puede hacer de forma precisa si conocemos un punto adicional de la imagen, y el ángulo que forma entre la cámara y uno de los puntos anteriores. Es decir, ahora estamos en la situación de conocer  $P_1, P_2, P_3$  puntos de la imagen,  $P_4$  la cámara y  $\alpha_1 = \angle(\overrightarrow{P_4P_1}, \overrightarrow{P_4P_2})$ ,  $\alpha_2 = \angle(\overrightarrow{P_4P_2}, \overrightarrow{P_4P_3})$ . Y hacemos:

1. Con el procedimiento anterior, obtenemos las circunferencias  $C_1, C_2$  tales que  $\beta_1 = \alpha_1$  y  $\beta_2 = \alpha_2$ .
2. En esta situación, se tendrá  $C_1 \cap C_2 = \{P_2, P_4\}$ . Por lo que calculamos esta intersección, y el punto obtenido distinto de  $P_2$ , será  $P_4$ .

Esta situación queda ilustrada en el siguiente diagrama:



Toda esta explicación intuitiva está bien, pero ahora debemos conseguir hacerlo bien. Comenzamos con el caso de la localización aproximada a partir de dos puntos y un ángulo. En el siguiente diagrama vemos la solución:



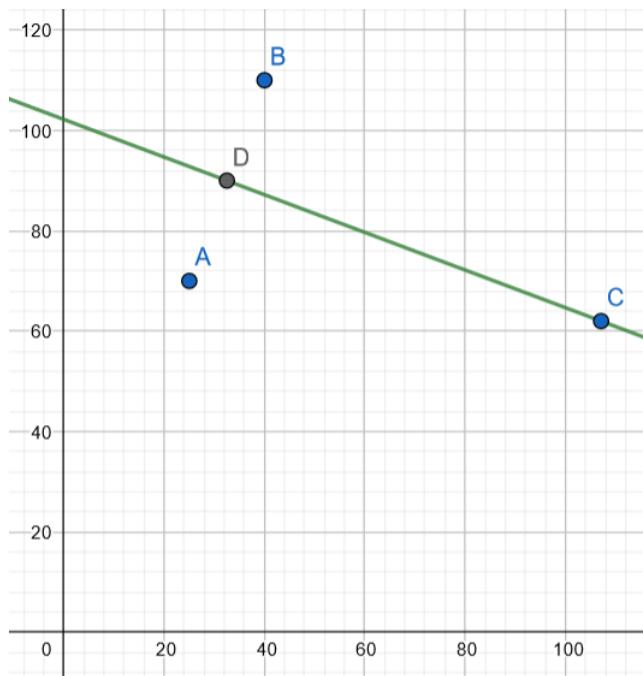
Tenemos que  $\tan\left(\frac{\alpha}{2}\right) = \frac{\|P_2 - P_1\|}{h}$ , por lo que

$$h = \frac{\|P_2 - P_1\|}{2 \cdot \tan\left(\frac{\alpha}{2}\right)}$$

y entonces nuestra aproximación de la localización de la cámara será el punto

$$C = \frac{P_2 + P_1}{2} + h \cdot \frac{\overrightarrow{P_1 P_2}^\perp}{\|P_2 - P_1\|}$$

El código queda desarrollado y explicado en el notebook *findCam.ipynb*, que puede consultarse en A, donde se desarrolla el ejemplo siguiente:



Para el caso de la posición precisa, no es tan sencillo.

Conocemos  $P_1, P_2, P_3, \alpha_1 = \angle CP_1P_2$  y  $\alpha_2 = \angle CP_2P_3$ .

Tenemos tres puntos  $P_1, P_2$  y  $P_3$  alineados (porque son una proyección de una misma imagen sobre el suelo) que mediante una translación y una rotación podemos dejar como  $P_1 = (0, 0)$ ,  $P_2 = (x_2, 0)$  y

$P_3 = (x_3, 0)$  con  $0 < x_2 < x_3$ . Tomamos el vector normal a la recta que los une  $v = (0, -1)$  y definimos sus rectas mediatrices

$$B_{12}(\lambda) = \left( \frac{x_2}{2}, -\lambda \right)$$

$$B_{23}(\lambda) = \left( \frac{x_2 + x_3}{2}, -\lambda \right)$$

Y las distancias de cada punto de las mediatrices al punto respectivo. Estas distancias nos darán los radios de las circunferencias:

$$d_1(B_{12}(\lambda), P_1) = \sqrt{\frac{x_2^2}{4} + \lambda^2} = d_1(\lambda)$$

$$d_2(B_{23}(\lambda), P_3) = \sqrt{\left( \frac{x_2 - x_3}{2} \right)^2 + \lambda^2} = d_2(\lambda)$$

Y las circunferencias respectivas

$$C_{12}(\lambda, \theta_1) = B_{12}(\lambda) + (d_1(\lambda) \cos \theta_1, d_1(\lambda) \sin \theta_1)$$

$$C_{23}(\lambda, \theta_2) = B_{23}(\lambda) + (d_2(\lambda) \cos \theta_2, d_2(\lambda) \sin \theta_2)$$

Buscamos ahora  $\lambda_1$  y  $\lambda_2$  tales que los ángulos de todos los puntos definidos por cada circunferencia son, respectivamente,  $\alpha_1$  y  $\alpha_2$ . Esto lo hacemos de forma numérica, de la siguiente forma:

```
cos1 = np.cos(alpha1)
eq1 = lambda t :
    np.dot(pp1-C12(t,0), ppp2-C12(t,0))
    /
    (np.linalg.norm(pp1-C12(t,0))*np.linalg.norm(ppp2-C12(t,0)))
    - cos1
t1 = fsolve(eq1, 37)
```

Tomamos  $\theta_1 = 0$  porque nos sirve cualquier punto de la circunferencia. Para obtener  $\lambda_2$  hacemos exactamente lo mismo.

Una vez conocemos  $\lambda_1$  y  $\lambda_2$ , buscamos la intersección de  $C_{12}(\lambda_1, \cdot)$  con  $C_{23}(\lambda_2, \cdot)$ . Esto podemos hacerlo buscando un punto de  $C_{12}$  que diste de  $B_{23}$  lo mismo que  $P_3$ :

$$d(C_{12}(\lambda_1, \theta), B_{23}(\lambda_2))^2 = d_2(\lambda_2)^2$$

Tras algunas operaciones llegamos a que esto es equivalente a que se satisfaga

$$\lambda_1^2 + \sqrt{\frac{x_2^2}{4} + \lambda_1^2} \cdot (2\lambda_2 \sin \theta - x_3 \cos \theta) + x_2 x_3 = 0$$

Por lo que resolviendo esto de forma numérica para  $\theta$ , obtendremos la posición de la cámara buscada. Será  $C_{12}(\lambda_1, \theta)$ .

El resultado completo se puede ver en *findCam.ipynb*, donde se ha desarrollado el siguiente ejemplo:



## 2 Ejercicio 2 - Actividad

Para realizar este ejercicio me he basado en *roi.py* para obtener un ROI. Además, mantengo una serie de estados que me ayudan a que el programa siga la secuencia deseada. El programa puede verse en el archivo *movementDetection.py*. La imagen se procesa en blanco y negro.

- Comenzamos con  $trozoFix = []$  y  $X_1 = X_2 = Y_1 = Y_2 = -1$ . Esto indica que ninguna zona de la imagen ha sido seleccionada para detectar movimiento. Si seleccionamos un ROI, y pulsamos “c”, automáticamente *trozoFix* será la zona seleccionada y  $X_i, Y_i i = 1, 2$  serán sus delimitadores. Este *trozoFix* será la imagen de referencia que usaremos para detectar movimiento, por eso es útil guardar estas variables.
  - Una vez que hemos seleccionado un *trozoFix*, lo cual sabremos porque tendremos  $X_1 \geq 0$ , entonces querremos comparar cada nuevo frame capturado en esa misma zona con *trozoFix*. Para este cometido he decidido usar la media entre la diferencia absoluta (*MAE*) entre el frame actual y *trozoFix*. Tras diferentes pruebas he determinado que, en condiciones de luminosidad estable, un buen umbral para decidir que ha habido movimiento es  $media \geq 20$ .
- Tenemos, por otro lado, las variables de control de secuencia *frames :: int* y *detected :: Bool*.
  - *frames* se utiliza para saber cuántos frames hemos guardado en el vídeo hasta ahora. Se inicializa a 0. Cada vez que se detecta movimiento se pone a 1 y comienza la grabación. Guardamos cada frame procesado en el vídeo hasta tener  $secs \cdot fps$  frames guardados, siendo *secs* los segundos que queremos guardar y *fps* los fps del vídeo. Una vez esto sucede, volvemos a poner *frames* = 0.
  - *detected* se utiliza para saber si hemos detectado nuevo movimiento. Se inicializa a False. Cada vez que se detecta movimiento se pone a True. Cuando la grabación de esa secuencia de movimientos termina y ya no se detecta más movimiento, se pone de nuevo a False y estamos como en el estado inicial.
  - Notar que cuando detectamos movimiento creamos una nueva ventana, donde mostraremos la diferencia entre *trozoFix* y la zona de interés del frame actual.
- Si tenemos un ROI seleccionado y pulsamos la “x”, se resetean todos los parámetros y es como reiniciar el programa.
- Si estando en detección de una zona, seleccionamos un nuevo ROI y pulsamos “c”, cambia la zona de detección.
- Nótese que los diferentes movimientos detectados se guardan en el mismo vídeo, que se graba al terminar el programa. Por tanto, el vídeo consiste en diferentes trozos de movimiento de 3 segundos de duración cada uno.

Como ejemplo de funcionamiento, vemos la siguiente secuencia de capturas:

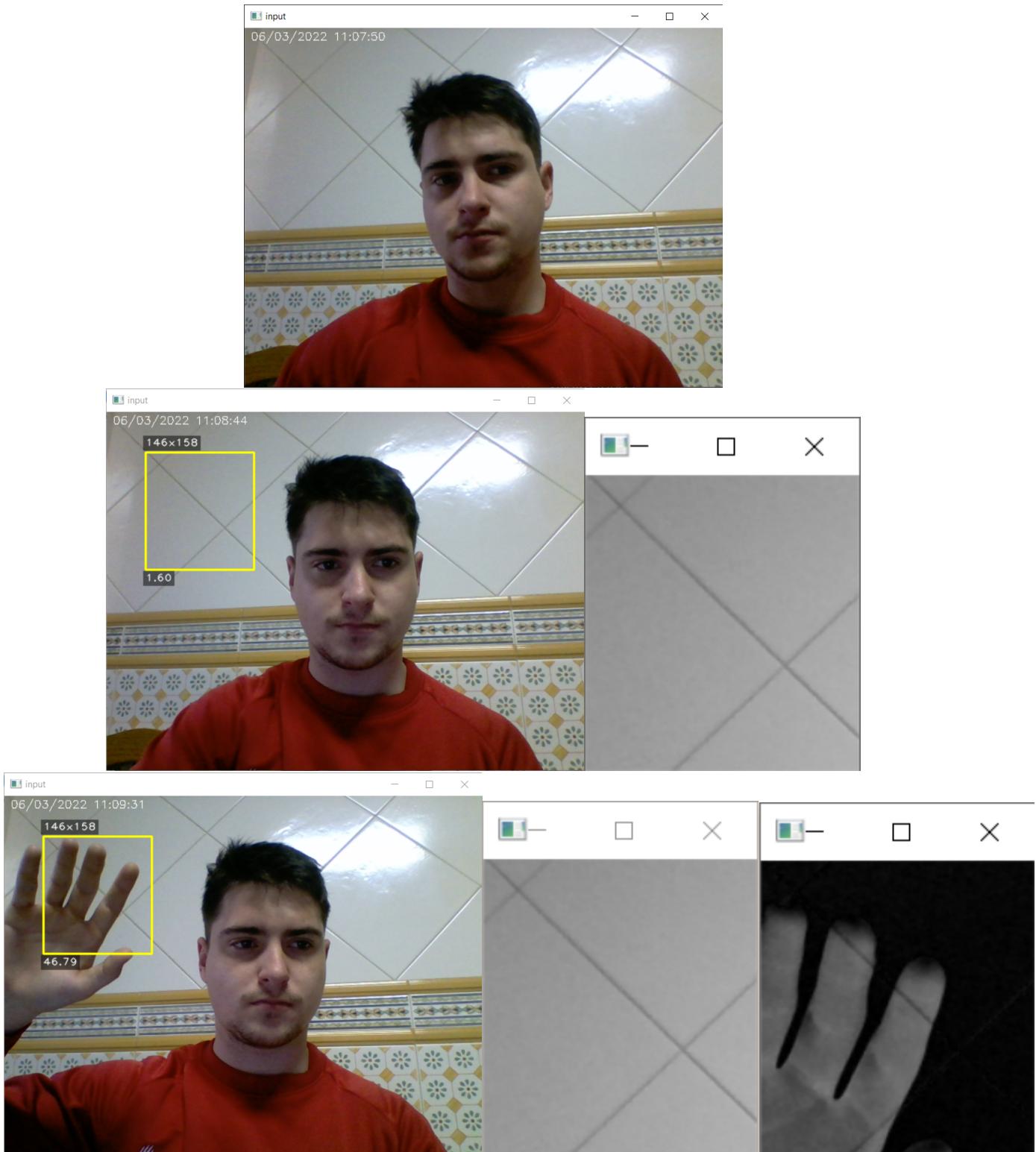


Figure 2: Ejemplo de funcionamiento del detector de movimiento

Observamos como se detecta un gran cambio al poner la mano dentro del recuadro, lo que se identifica como movimiento. Al suceder esto aparece la ventana con la diferencia entre lo que está sucediendo en la zona de interés y la imagen de referencia. Al finalizar el programa se graba el vídeo de aquellos momentos en los que se ha detectado movimiento.

## Apartado Opcional

Para este apartado me he inspirado en [4]. OpenCV tiene implementados filtros de eliminación del background de una imagen. Esto se hace con las funciones `createBackgroundSubtractorX()`, donde  $X$  es el método utilizado para la creación del filtro. Hay varios métodos, como:

- **MOG:** usa una mezcla de filtros gaussianos (**Mixture Of Gaussians**).
- **MOG2:** usa también un filtro gaussiano, aunque tiene en cuenta los cambios de luminosidad de la imagen.
- **KNN:** utiliza K-nearest neighbours para la obtención del filtro.

Yo he utilizado KNN porque ha sido el que mejores resultados me ha proporcionado tras realizar diferentes pruebas.

El funcionamiento es simple:

1. Creamos  $fgbg = cv.createBackgroundSubtractorKNN()$ , un filtro para sustracción del fondo.
2. Obtenemos la máscara  $fgmask = fgbg.apply(frame)$ , para cada frame procesado.
3. Obtenemos el contorno que conforma el foreground  
 $cont = cv.findContours(fgmask, cv.RETR_EXTERNAL, cv.CHAIN_APPROX_SIMPLE)[-2]$
4. Y ahora creamos una nueva máscara,  $mask$ , con el mismo tamaño que  $frame$  y usando la máscara anterior (tengamos en cuenta que la anterior solo es una matriz  $N \times M \times 1$ ).
5. Obtenemos el frame nuevo sin el fondo, multiplicando  $mask \cdot frame$ .

El ejemplo de funcionamiento es el siguiente:

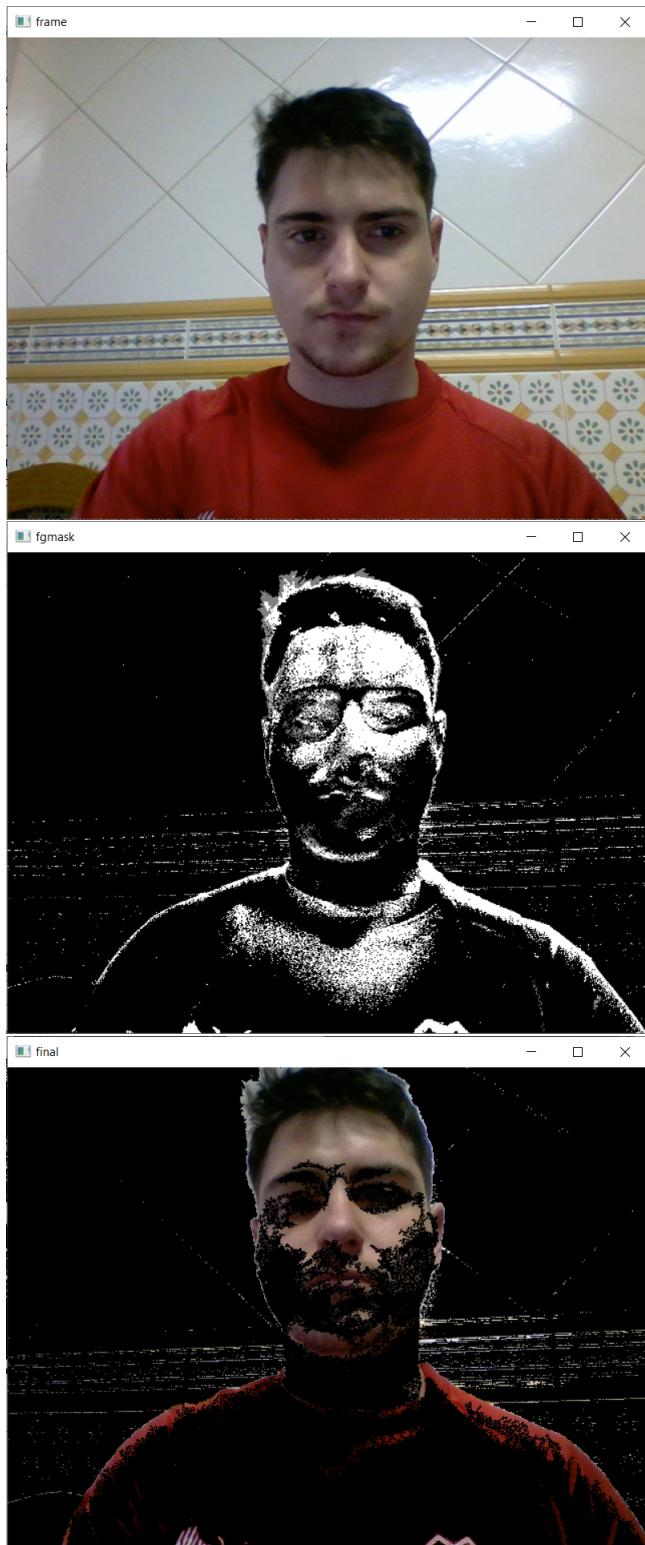


Figure 3: Ejemplo de funcionamiento del eliminador del fondo

La primera imagen es la original obtenida con la cámara. La segunda es la máscara obtenida y la tercera es el resultado final. Vemos como el resultado no es ideal, pero esto en realidad es esperable, pues esto está pensado para detectar objetos en movimiento respecto de un fondo fijo, que elimina. Pero se ve cómo detecta los bordes principales y elimina

### 3 Ejercicio 3 - Color

Para hacer este ejercicio, me he basado en *histogram.py* para obtener el histograma de una imagen. La lógica de la solución desarrollada es sencilla: tenemos una deque de tres elementos en la que introduciremos las zonas seleccionadas por el usuario usando un ROI y pulsando la “c”. Cuando se selecciona un ROI para ser una referencia a utilizar para la comparación, se calculan sus histogramas en los 3 canales de color, y estos se guardan en otro deque usado para guardarlos. Continuamente, se comparan las imágenes seleccionadas con el ROI actual, simplemente mediante el cálculo del error medio absoluto (MAE) entre los histogramas de los 3 canales de color del ROI actual y los de las imágenes guardadas, que tenemos almacenados. Para cada uno de los histogramas almacenados, calculamos el máximo entre los MAE de los tres canales. Finalmente, consideramos que la imagen más parecida es la que nos arroja un menor valor al hacer esta operación.

Todo esto puede verse en *color.py*, pero voy a comentar algunas partes importantes del código:

- *histogramas(fr)* calcula los histogramas en los 3 canales de una imagen GBR. Además, los normaliza dividiendo por el máximo valor obtenido, permitiendo la comparación entre imágenes de diferentes tamaños. Esto es así puesto que, de no hacerlo, el tamaño de la foto haría que las frecuencias de las fotos más grandes fuesen mayores, simplemente por tamaño. De esta forma, nos estamos fijando, en cierto sentido, más en la forma del histograma, que en los valores concretos que nos arroja.
- *compareHist(h<sub>1</sub>, h<sub>2</sub>)* compara dos histogramas aplicándoles el MAE en cada uno de sus 3 componentes (una por canal)
- En *imgsOrig* guardo los trozos de imagen originales (aunque resizeados) y en *imgs* tengo los trozos tal cual los quiero mostrar por pantalla, con alguna información adicional. En *hists* guardo los histogramas de las capturas de referencia, para no tener que recalcularlos, pues son constantes. Nótese que las imágenes las resizeo a un mismo tamaño común para poder stackearlas y mostrarlas conjuntamente. Nótese, además, que los histogramas los cálculo en el trozo antes de resizearlo, porque de otra forma se introducen distorsiones que emprobezcan la comparación posterior.
- El resto del código es muy sencillo, buscar el mínimo entre los MAE y mostrar la imagen de referencia que más se asemeja al ROI actual.

#### Opcional: Segmentación densa por reprojeción del histograma

Esta vez me baso en la explicación de [5]. El desarrollo del ejercicio puede verse en el notebook *segmentacionDensa.ipynb*, donde básicamente se ha seguido la explicación del notebook de teoría, aunque se han añadido algunas consideraciones y pruebas adicionales. El notebook puede verse en el anexo B.

Ahora voy a realizar una versión simplificada del programa *grabcut.py*, que es un programa de extracción de foreground de imágenes utilizando el algoritmo grabcut. Una explicación detallada de este algoritmo puede consultarse en [6], pero, básicamente, consiste en lo siguiente:

- El usuario, en una imagen, recuadra la zona en la que se encuentra el foreground de la imagen. Todo lo que quede fuera del rectángulo será considerado background con seguridad.
- Se etiquetan los píxeles de foreground y background mediante ese rectángulo.
- Un modelo de varias gaussianas (GMM) se usa para modelar el foreground y el background y crea una nueva distribución de píxeles.

- Se construye un grafo a partir de esta distribución. Se añaden un nodo fuente (source) y un nodo sumidero (sink). Cada píxel del foreground se conecta al nodo fuente, y cada uno de background al sumidero. Los pesos del grafo se obtienen por la información de los bordes o por parecido entre píxeles.
- Se aplica un algoritmo mincut para segmentar el grafo. O sea, se separa el grafo en dos partes, separando el nodo fuente y el nodo sumidero.
- Se considera que los nodos conectados al fuente son foreground y los conectados al sumidero son background.
- Se repite hasta alcanzar convergencia.

En el programa *grabcut.py* puede verse la solución parcial que he proporcionado. En ella, he realizado únicamente la parte de seleccionar con un rectángulo la zona en la que está el foreground, y a esta le aplico el algoritmo de grabcut, ya implementado en OpenCV. Como ejemplo vemos la siguiente imagen de input y el resultado obtenido:



Figure 4: Ejemplo de mi programa que aplica grabcut

Como vemos, el resultado es mediocre, pero se aprecia también que varios detalles los capta bien. Si se

aumentase e incorporase las siguientes fases para poder indicar correcciones al algoritmo, obtendríamos mucho mejores resultados.

## 4 Opcional: Ejercicio 4 - Filtros

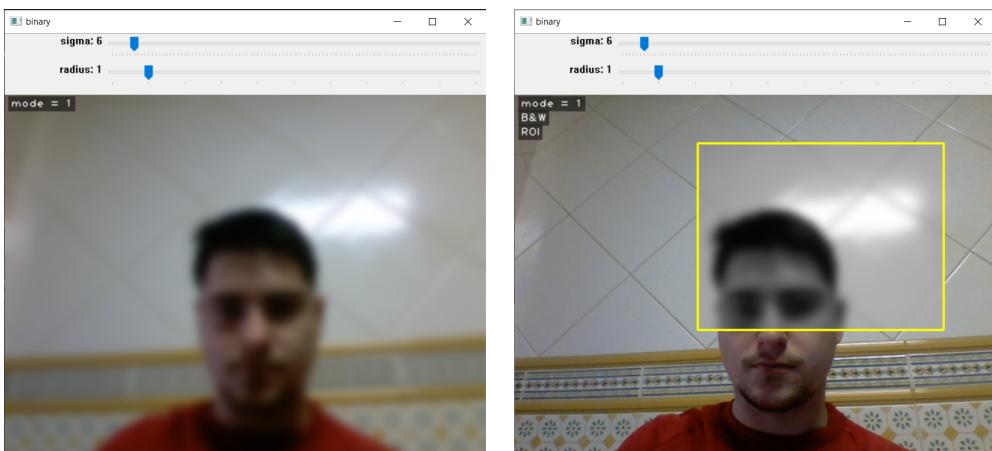
### Apartado a)

Para este ejercicio el problema simplemente consiste en tratar de controlar el diagrama de estados de la forma más sencilla posible, y aplicar las distintas transformaciones en función del estado en que estamos. Mis ideas para conseguir esto han sido las siguientes:

- Para decidir qué filtro aplicar, simplemente llevo una variable entera que lo indica.
- Para mostrar la imagen en color o en escala de grises, llevo una variable booleana *color*. Si es true, proceso la imagen tal cual me llega. Si es false, la paso a escala de grises, y la proceso así en todo el programa.
- Para poder filtrar la imagen completa o un ROI, llevo otra variable booleana *roi* y cuatro variables  $X_1, X_2, Y_1, Y_2$ , que indican el rectángulo a procesar. Si *roi == False*, entonces el rectángulo indica que sea toda la imagen. Si es true, hago que el rectángulo sea el ROI seleccionado. Durante todo el programa proceso *frame*  $[Y_1 : Y_2, X_1 : X_2]$ , en lugar de frame entero. Esto permite hacer lo que queremos.

Por otro lado, he incluido los siguientes filtros:

- Filtro gaussiano: se le pasa el valor obtenido en el trackbar para que calcule el *ksize*, al que le paso (0,0).

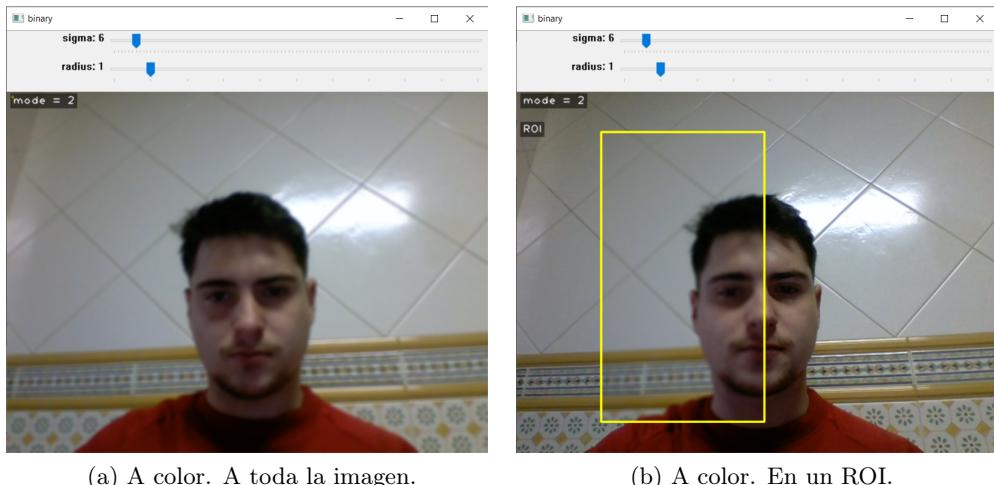


(a) A color. A toda la imagen.

(b) En B&W. En un ROI.

Figure 5: Filtro gaussiano.

- Filtro box: le paso un *ddepth* de -1 para que utilice *src.depth*, y un *ksize = (h, h)*, siendo *h* el sigma obtenido del trackbar.

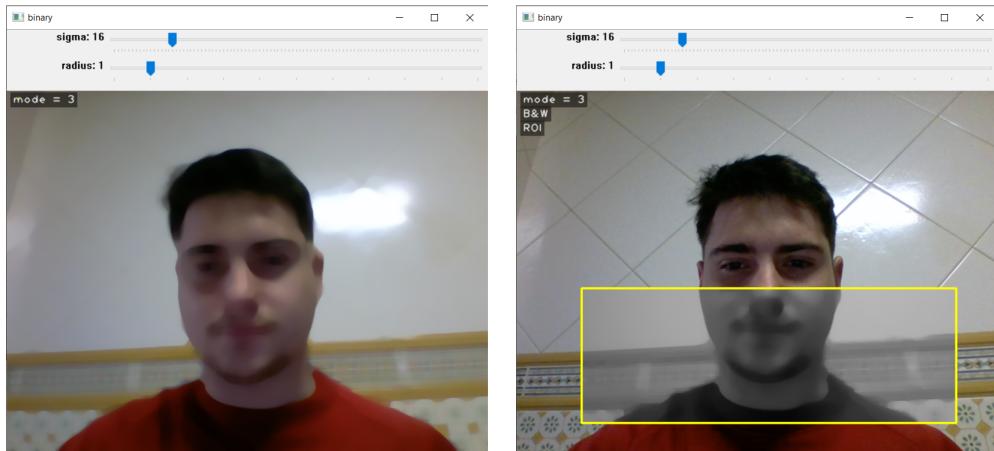


(a) A color. A toda la imagen.

(b) A color. En un ROI.

Figure 6: Filtro box.

- Difuminación de la mediana: le pasamos un *ksize* de  $h$  si es impar o, si es par, le pasamos  $h + 1$  (*medianBlur* pide un *ksize* impar).

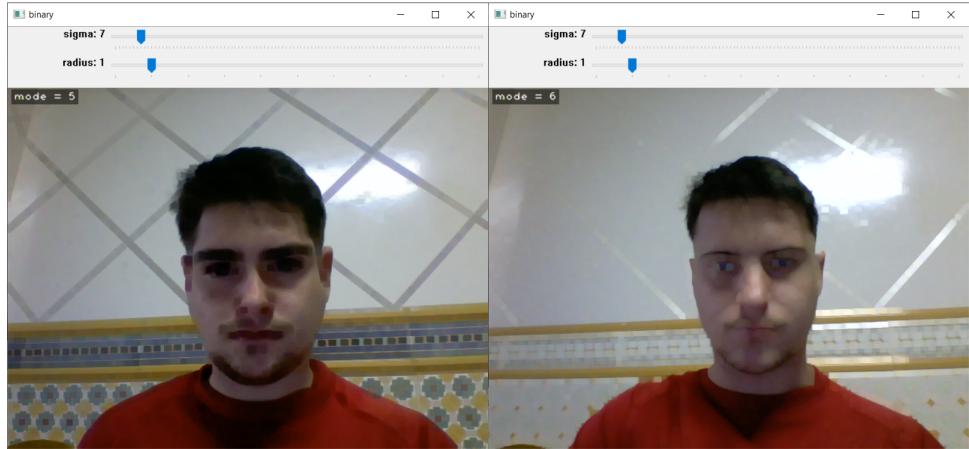


(a) A color. A toda la imagen.

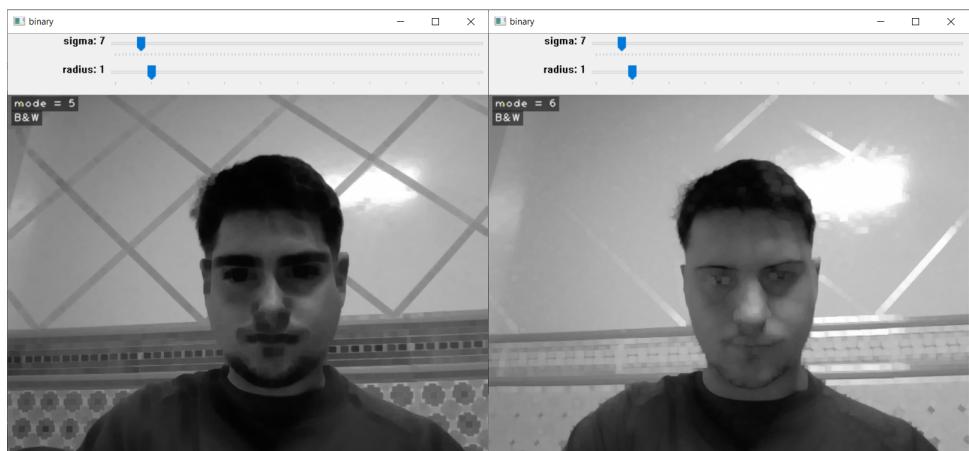
(b) A B&W. En un ROI.

Figure 7: Difuminación de la mediana.

- Filtro bilateral: este filtro no consigo que funcione correctamente. Se ralentiza mucho el programa si lo uso, y no se aprecia ningún cambio significativo en la imagen.
- Filtros del mínimo y del máximo: estos filtros al aplicarlos a la imagen en color, me daban la imagen en blanco y negro. Por tanto, lo que he hecho ha sido aplicarlos a cada componente por separado.



(a) A color. A toda la imagen.



(b) A B&W. En toda la imagen.

Figure 8: Filtros del mínimo (izq) y del máximo (dcha).

Los siguientes apartados, de momento, los dejo para próximas entregas.

## A Notebook findcam.ipynb

# findCam

March 6, 2022

```
[1]: import math
import numpy as np
import pandoc
```

## Localización estimada a partir de dos puntos y un ángulo

Introducimos los puntos de nuestra imagen y el ángulo que forman con la cámara.

```
[2]: p1 = np.array([25,70])
p2 = np.array([40,110])
alpha = 30
alpha = alpha*2*np.pi/360
```

Calculamos ahora el vector  $v = \overrightarrow{P_1 P_2} = P_2 - P_1$

```
[3]: v = p2-p1
print(v)
```

[15 40]

Y calculamos el vector ortogonal normalizado.

```
[4]: vv = np.array([v[1],-v[0]])
print(vv)
norm = np.linalg.norm(v)
print(norm)
vv = vv / norm
print(vv)
```

[ 40 -15]  
42.720018726587654  
[ 0.93632918 -0.35112344]

Ahora calculamos  $\tan(\frac{\alpha}{2})$ :

```
[5]: tan = np.tan(alpha/2)
print(tan)
```

0.2679491924311227

Calculamos  $h = \frac{\|P_2 - P_1\|}{2 \cdot \tan(\frac{\alpha}{2})}$ :

```
[6]: h = norm/(2*tan)  
print(h)
```

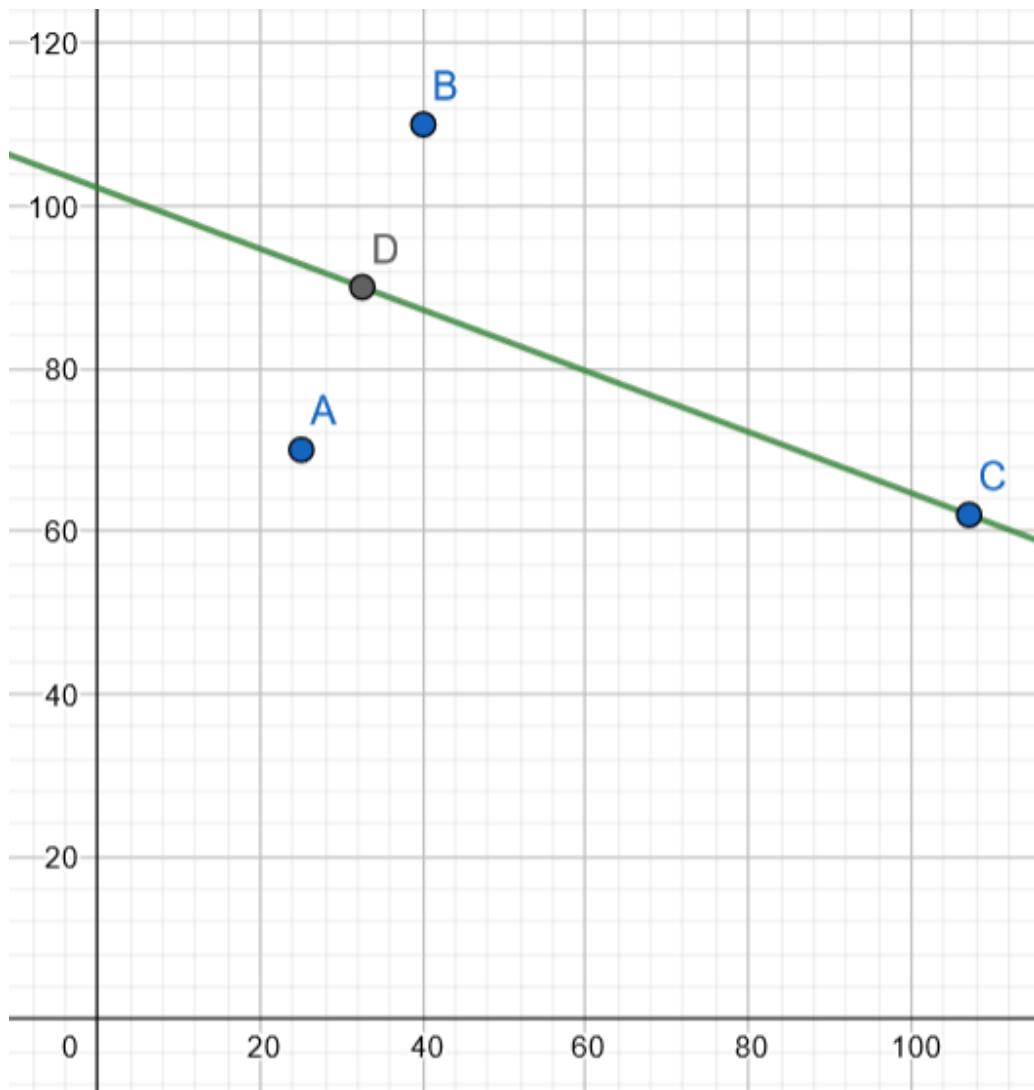
79.7166401939595

Y ya podemos calcular  $C$ :

```
[7]: C = (p1+p2)/2 + h * vv  
print(C)
```

[107.14101615 62.00961894]

Como podemos ver en la siguiente imagen, ¡funciona correctamente!



### Localización exacta a partir de tres puntos y dos ángulos conocidos

Ahora conocemos 3 puntos y dos ángulos. Nótese que deben estar alineados, puesto que están en la misma foto y los estamos viendo desde arriba.

```
[8]: p1 = np.array([25,70])
p2 = np.array([40,110])
p3 = np.array([47.5,130])
alpha1 = 30
alpha1 = alpha1*2*np.pi/360
alpha2 = 20
alpha2 = alpha2*2*np.pi/360
```

Trasladamos para que sea  $p_1 = (0, 0)$ :

```
[9]: pp1 = np.array([0,0])
pp2 = p2-p1
pp3 = p3-p1
```

Y ahora rotamos para que los tres puntos caigan en el eje X:

```
[10]: e1 = np.array([1,0])
angle = -np.arccos(np.dot(pp2,e1)/np.linalg.norm(pp2))
print(angle)

rotMat = np.matrix([[np.cos(angle), -np.sin(angle)], [np.sin(angle), np.
    ↵cos(angle)]])
print(rotMat)

ppp2 = np.squeeze(np.asarray(rotMat@pp2))
print(ppp2)
ppp3 = np.squeeze(np.asarray(rotMat@pp3))
print(ppp3)
```

```
-1.2120256565243244
[[ 0.35112344  0.93632918]
 [-0.93632918  0.35112344]]
[42.72001873  0.          ]
[ 6.40800281e+01 -8.88178420e-16]
```

Como vemos, estos puntos ya tienen a 0 (o prácticamente 0) su coordenada  $y$ , y podemos continuar. Tomamos el vector normal unitario  $v = (0, -1)$  y calculamos las bisectrices y las circunferencias respectivas.

```
[11]: from scipy.optimize import fsolve, least_squares
```

```
[12]: B12 = lambda t : np.array([ppp2[0]/2, -t], dtype = object)
B23 = lambda t : np.array([(ppp2[0]+ppp3[0])/2, -t], dtype = object)

d1 = lambda t : np.linalg.norm(np.sqrt(ppp2[0]**2/4+t**2))
d2 = lambda t : np.linalg.norm(np.sqrt((ppp2[0]-ppp3[0])**2/4+t**2))

C12 = lambda t, x : B12(t) + np.array([d1(t)*np.cos(x), d1(t)*np.sin(x)], dtype=object)
```

```
C23 = lambda t, x : B23(t) + np.array([d2(t)*np.cos(x), d2(t)*np.sin(x)], dtype=object)
```

Ahora buscamos  $t_1$  y  $t_2$  tales que los ángulos de los puntos de  $C_{12}(t_1, 0) = \alpha_1$  y  $C_{23}(t_2, 0) = \alpha_2$ , y por tanto lo serán para cualquier ángulo:

```
[13]: cos1 = np.cos(alpha1)
eq1 = lambda t : np.dot(pp1-C12(t,0), ppp2-C12(t,0)) / (np.linalg.
    ↪norm(pp1-C12(t,0))*np.linalg.norm(ppp2-C12(t,0))) - cos1
t1 = fsolve(eq1, 37)

cos2 = np.cos(alpha2)
eq2 = lambda t : np.dot(ppp2-C23(t,0), ppp3-C23(t,0)) / (np.linalg.
    ↪norm(ppp2-C23(t,0))*np.linalg.norm(ppp3-C23(t,0))) - cos2
t2 = fsolve(eq2, 0)
print(t1)
print(t2)
```

[36.99662147]

[29.3430717]

```
[14]: print(B12(t1))
print(B23(t2))
print(C12(t1,0))
print(C23(t2,0))
```

[21.36000936329383 array([-36.99662147])]

[53.400023408234574 array([-29.3430717])]

[64.08002808988151 array([-36.99662147])]

[84.6262680901962 array([-29.3430717])]

Ahora tenemos que resolver la ecuación que hemos derivado en la memoria:

$$t_1^2 + \sqrt{\frac{x_2^2}{4} + t_1^2} \cdot (2t_2 \cdot \sin \theta - x_3 \cos \theta) + x_2x_3 = 0$$

```
[15]: eq3 = lambda w : t1**2 + np.linalg.norm(np.sqrt(ppp2[0]**2/4+t1**2))*(2*t2*np.
```

```
    ↪sin(w) - ppp3[0]*np.cos(w)) + ppp2[0]*ppp3[0]
```

```
w0 = fsolve(eq3, 0)
```

```
print(w0)
```

[-0.74134005]

```
C:\Users\Jose\AppData\Local\r-miniconda\envs\via\lib\site-
packages\scipy\optimize\minpack.py:175: RuntimeWarning: The iteration is not
making good progress, as measured by the
    improvement from the last ten iterations.
    warnings.warn(msg, RuntimeWarning)
```

Por lo tanto, ya tenemos el resultado, que no es más que  $C_{12}(t_1, w_0)$

```
[16]: C = C12(t1, w0[0])
print(C)
```

```
[52.86877085946814 array([-65.84446279])]
```

Por último, invertimos la transformación de traslación y rotación:

```
[17]: rotMatInv = np.matrix([[np.cos(-angle), -np.sin(-angle)], [np.sin(-angle), np.
    ↪cos(-angle)]])
Cfinal = p1 + np.squeeze(np.asarray(rotMatInv@C))
print(Cfinal)
```

```
[array([105.21555647]) array([96.38303835])]
```

Y ya tenemos (jal fin!) el resultado buscado. Comprobamos en la siguiente imagen que es verosímil:



## B Notebook segmentacionDensa.ipynb

# segmentacion

March 6, 2022

```
[ ]: import numpy           as np
      import cv2            as cv
      import matplotlib.pyplot as plt
      import ipywidgets
      from matplotlib.pyplot import imshow, subplot, title

[ ]: def fig(w,h):
      return plt.figure(figsize=(w,h))

def readrgb(file):
    return cv.cvtColor( cv.imread(file), cv.COLOR_BGR2RGB)

def rgb2yuv(x):
    return cv.cvtColor(x,cv.COLOR_RGB2YUV)

def yuv2rgb(x):
    return cv.cvtColor(x,cv.COLOR_YUV2RGB)

byte = np.uint8

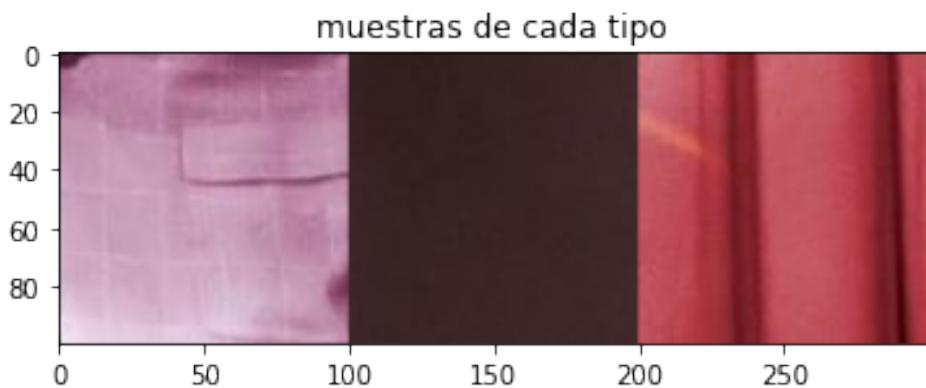
[ ]: img = readrgb("elvis.jpg")
      imshow(img); title('original');
      rows,cols,d = img.shape
      print(img.shape)
```

(675, 1200, 3)



```
[ ]: r1 = img[250:350,330:430]
r2 = img[250:350,50:150]
r3 = img[250:350,1050:1150]
models = [r1,r2,r3]

imshow(np.hstack(models)); title('muestras de cada tipo');
```



Primero vamos a reproducir el modelo simple

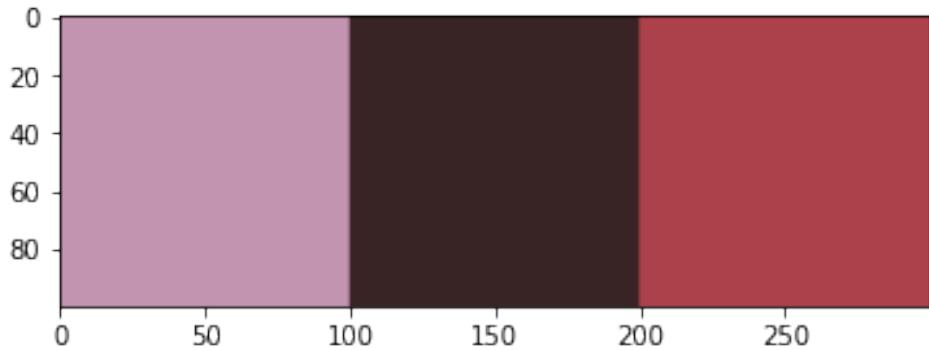
```
[ ]: med = [ np.mean(r,(0,1)) for r in models ]
muestras = []
```

```

for color in med:
    x = np.zeros([100,100,3],byte)
    x[:, :] = color
    muestras.append(x)

imshow(np.hstack(muestras));

```

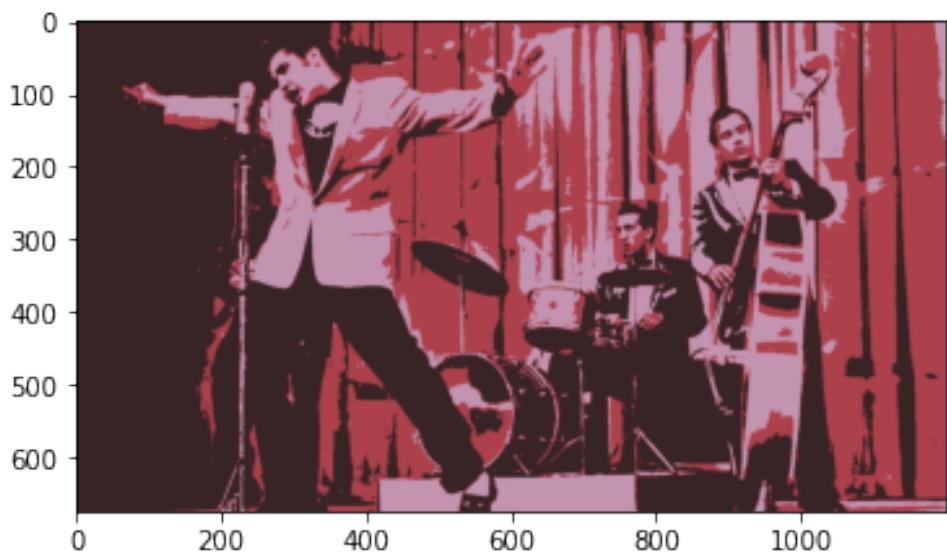


```

[ ]: d = [ np.sum(abs(img - m), axis=2) for m in med ]
c = np.argmin(d, axis=0)
res = np.zeros(img.shape, byte)
for k in range(len(models)):
    res[c==k] = med[k]

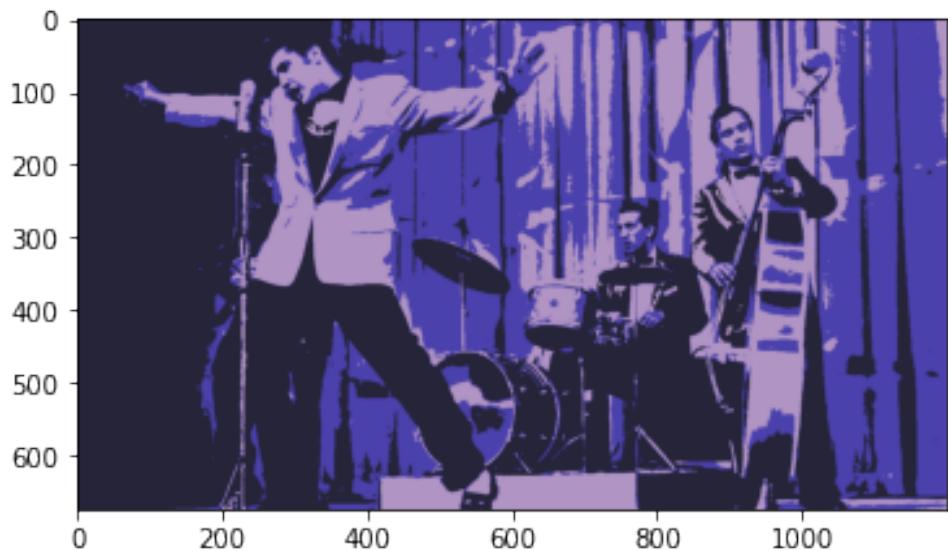
imshow(res);

```



Vemos como necesitaríamos utilizar más colores para reproducir fielmente la imagen original. Pero el resultado obtenido es bastante bonito, y podemos utilizar esta idea para aplicar algunos filtros de colores a las imágenes. Por ejemplo, podríamos hacer algo así:

```
[ ]: nwIm = res.copy()
nwIm[:, :, 0] = res[:, :, 2]
nwIm[:, :, 2] = res[:, :, 0]
imshow(nwIm);
```



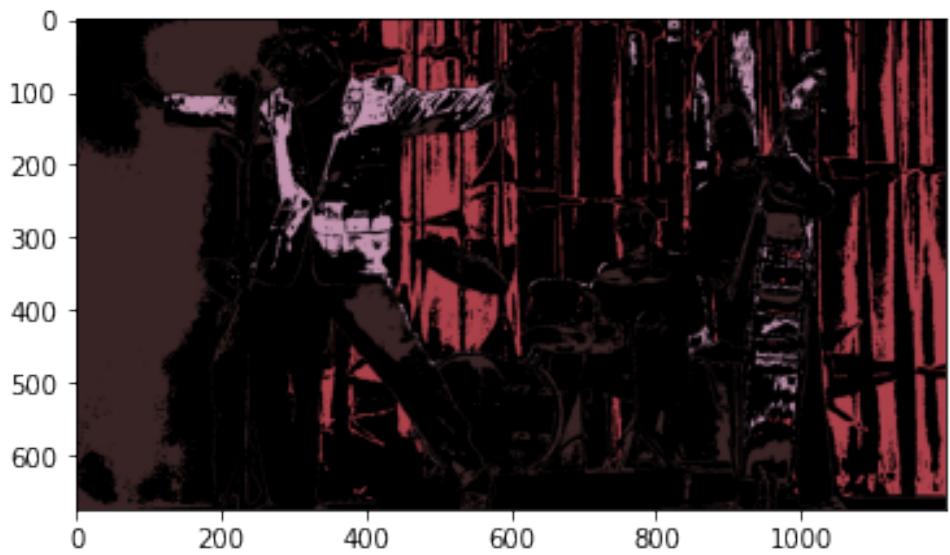
Entonces podríamos tomar, por ejemplo, tres tonos característicos de una imagen. Aplicar la reproyección tal y como hemos hecho, y cambiar los colores en función del resultado buscado.

Vamos a seguir tal y como se hace en el notebook explicativo.

```
[ ]: md = np.min(d, axis=0)

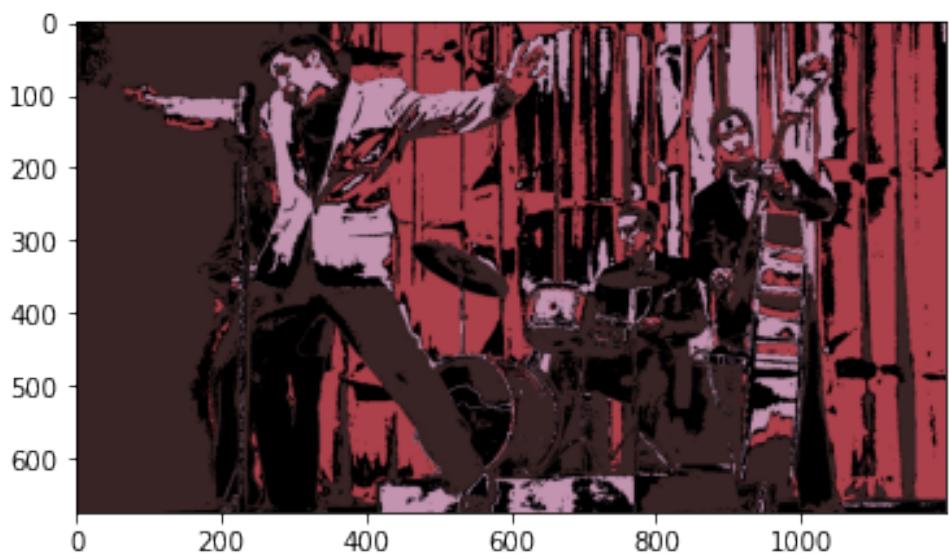
res [md > 40] = 0,0,0

imshow(res);
```



El resultado al hacer esto es muy pobre, ya que la imagen presenta colores muy diversos y hemos elegido una gama estrecha de colores (hemos tomado tres colores rojizos). Esto era esperable. Podemos ajustar el threshold tomado:

```
[ ]: md = np.min(d, axis=0)  
res [md > 80] = 0,0,0  
imshow(res);
```



Y ya se distinguen los objetos de la imagen. Además, da un toque de cartoon, que hace que la imagen parezca un dibujo.

Ahora lo hacemos probabilísticamente:

```
[ ]: # calcula el histograma (normalizado) de los canales conjuntos UV
def uvh(x):

    # normalizar un histograma
    # para tener frecuencias (suman 1)
    # en vez de número de elementos
    def normhist(x): return x / np.sum(x)

    yuv = rgb2yuv(x)
    h = cv.calcHist([yuv])      # necesario ponerlo en una lista aunque solo u
    ↪admite un elemento
        ,[1,2]      # elegimos los canales U y V
        ,None       # posible máscara
        ,[32,32]    # las cajitas en cada dimensión
        ,[0,256]+[0,256] # rango de interés (todo)
    )
    return normhist(h)
```

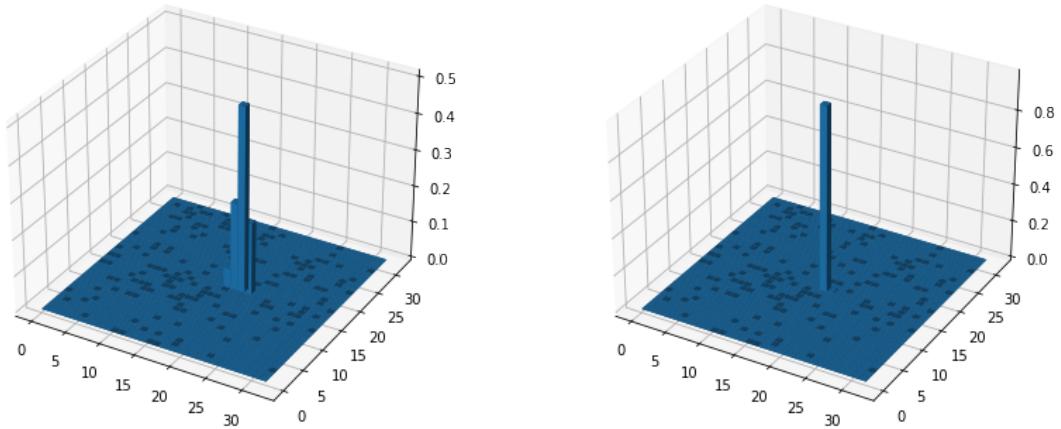
```
[ ]: hist = [uvh(r) for r in models]
```

```
[ ]: from mpl_toolkits.mplot3d import Axes3D
fg = plt.figure(figsize=(14, 6))

_xx, _yy = np.meshgrid(np.arange(32), np.arange(32))
x, y = _xx.ravel(), _yy.ravel()
bottom = 0
width = depth = 1

ax1 = fg.add_subplot(121, projection='3d')
top = hist[0].ravel()
ax1.bar3d(x, y, bottom, width, depth, top, shade=True);

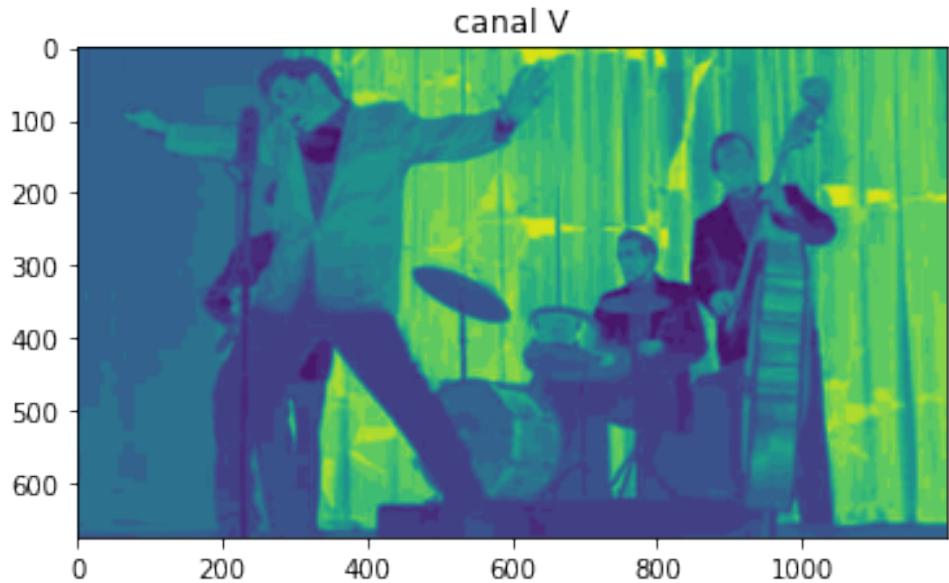
ax1 = fg.add_subplot(122, projection='3d')
top = hist[1].ravel()
ax1.bar3d(x, y, bottom, width, depth, top, shade=True);
```



```
[ ]: uvr = np.floor_divide( cv.cvtColor(img, cv.COLOR_RGB2YUV) [:,:, [1,2]], 8)
print(uvr.shape, uvr.dtype)
```

(675, 1200, 2) uint8

```
[ ]: imshow(uvr[:, :, 1]); title('canal V');
```



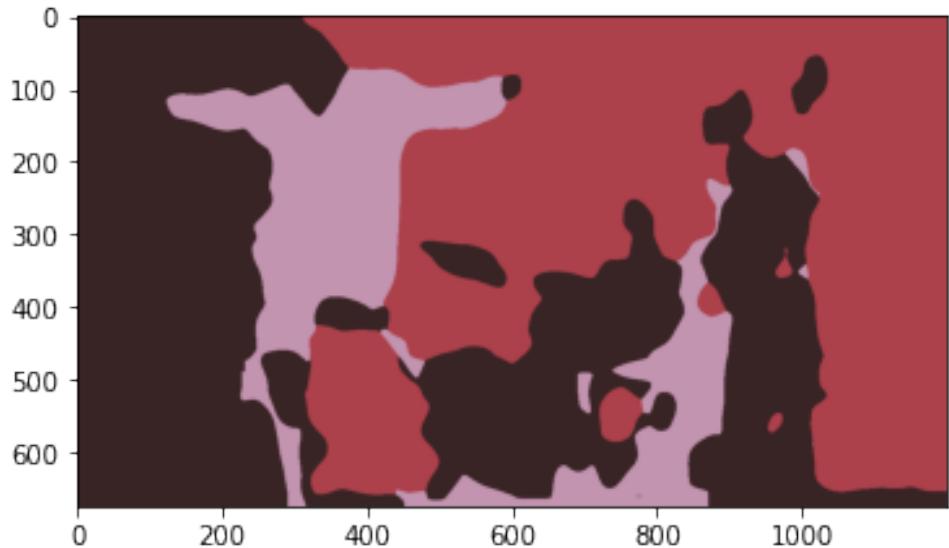
```
[ ]: u = uvr[:, :, 0]
v = uvr[:, :, 1]
```

```
lik = [ h[u,v] for h in hist ]
lik = [ cv.GaussianBlur(l, (0,0), 10) for l in lik ]
```

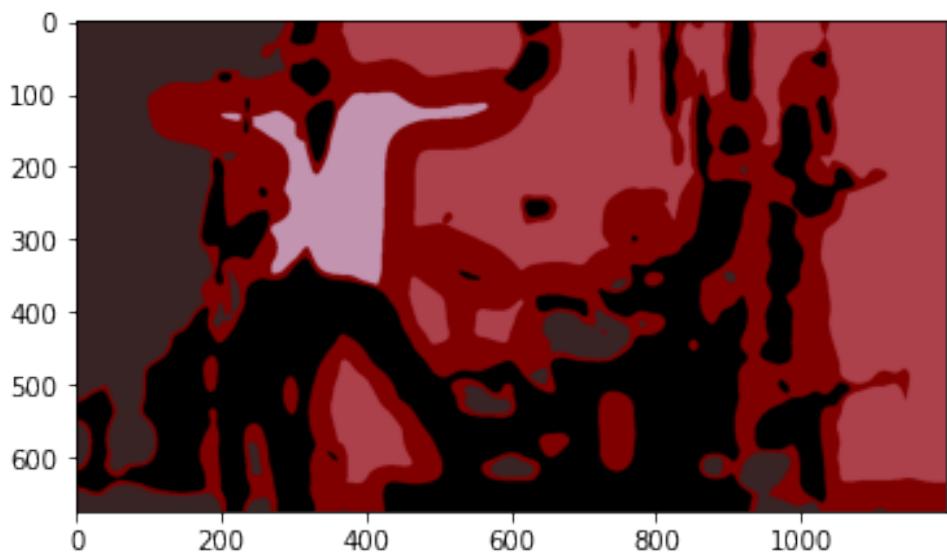
```
[ ]: E = np.sum(lik, axis=0)
p = np.array(lik) / E
c = np.argmax(p, axis=0)
mp = np.max(p, axis=0)
mp[E < 0.1] = 0
res = np.zeros(img.shape, np.uint8)
for k in range(len(models)):
    res[c==k] = med[k]

imshow(res);
```

```
C:\Users\Jose\AppData\Local\Temp\ipykernel_4828\937867013.py:2: RuntimeWarning:
invalid value encountered in true_divide
  p = np.array(lik) / E
```



```
[ ]: res[mp < 0.99] = 128,0,0
res[E < 0.05] = 0,0,0
imshow(res);
```



Y obviamente obtenemos una imagen compleja de entender si no conoces la original. Pero se observan algunas de las características principales de la misma. Si usásemos más colores más variados, seguramente los resultados serían considerablemente mejores.

## References

- [1] A. R. Garcia, “Imagen.ipynb,” apuntes de la asignatura de vision artificial. Explicacion del FOV, la distancia focal y diferentes formas de calcularlos. [Online]. Available: <https://github.com/albertoruiz/umucv/blob/775b9e3ebb157b0a648d53649b7b2576ecce1633/notebooks/imagen.ipynb>
- [2] C. S. de Deportes, “Normativa de instalaciones deportivas y esparcimiento: Normas reglamentarias de baloncesto.” [Online]. Available: [https://www.csd.gob.es/sites/default/files/media/files/2018-10/blc\\_baloncesto\\_2015.pdf](https://www.csd.gob.es/sites/default/files/media/files/2018-10/blc_baloncesto_2015.pdf)
- [3] Wikipedia, “Inscribed angle.” [Online]. Available: [https://en.wikipedia.org/wiki/Inscribed\\_angle](https://en.wikipedia.org/wiki/Inscribed_angle)
- [4] S. Sinha, “Python | background subtraction using opencv.” [Online]. Available: <https://www.geeksforgeeks.org/python-background-subtraction-using-opencv/>
- [5] A. R. Garcia, “Segmentacion por color.” apuntes de la asignatura de vision artificial. Modelos de segmentacion por color. Segmentacion probabilistica. Tracking. [Online]. Available: <https://github.com/albertoruiz/umucv/blob/955c5ca55e76f84e226e3462013a3823d2a995e9/notebooks/colorseg.ipynb>
- [6] OpenCV, “Interactive foreground extraction using grabcut algorithm.” [Online]. Available: [https://docs.opencv.org/3.2.0/d8/d83/tutorial\\_py\\_grabcut.html](https://docs.opencv.org/3.2.0/d8/d83/tutorial_py_grabcut.html)