

# Visión Artificial

Jose Antonio Lorencio Abril



Profesor: Alberto Ruiz García

Curso: 2021/2022

Convocatoria: Junio

e-mail del alumno: joseantonio.lorencioa@um.es

## Contents

1 Ejercicio 1 - Calibración	4
2 Ejercicio 2 - Actividad	12
3 Ejercicio 3 - Color	16
4 Ejercicio 4 - SIFT	19
5 Ejercicio 5 - RECTIF	21
6 Ejercicio 6 - RA	29
7 Opcional 1 - Filtros	32
8 Opcional 5 - RC	35
9 Opcional 6 - Swap	37
10 Opcional 9: SUDOKU	40
A Notebook findcam.ipynb	44
B Notebook segmentacionDensa.ipynb	51

## **Abstract**

En esta memoria se recogen los distintos problemas realizados para la asignatura de Visión Artificial. Cada ejercicio se recoge en una sección, con el desarrollo teórico de los mismos, ejemplos de uso y las explicaciones que he considerado pertinentes para la correcta comprensión del código correspondiente a cada uno, que se podrá encontrar en las carpetas acompañantes de este documento. Los notebooks desarrollados pueden verse también en los anexos de esta memoria, sin necesidad de ejecutarlos aparte.

En la última versión el documento ha sufrido los siguientes cambios:

- Añadidos los enunciados de los ejercicios.
- Añadida la sección 4, correspondiente al ejercicio SIFT.
- Añadida la sección 5, correspondiente al ejercicio RECTIF.
- Añadida la sección 6, correspondiente al ejercicio RA.
- Añadida la sección 8, correspondiente al ejercicio opcional RC.
- Añadida la sección 9, correspondiente al ejercicio opcional Swap.
- Añadido la sección 10, correspondiente al ejercicio opcional SUDOKU

# 1 Ejercicio 1 - Calibración

## Enunciado

- a) Realiza una calibración precisa de tu cámara mediante múltiples imágenes de un chessboard.
- b) Haz una calibración aproximada con un objeto de tamaño conocido y compara con el resultado anterior.
- c) Determina a qué altura hay que poner la cámara para obtener una vista cenital completa de un campo de baloncesto.
- d) Haz una aplicación para medir el ángulo que definen dos puntos marcados con el ratón en el imagen.
- e) Opcional: determina la posición aproximada desde la que se ha tomado una foto a partir ángulos observados respecto a puntos de referencia conocidos.

## Apartado A

Se nos pide calibrar la cámara de nuestro móvil. Es decir, encontrar el valor de la **distancia 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 imprimirmos 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 ( $\pm 30\%$ ).

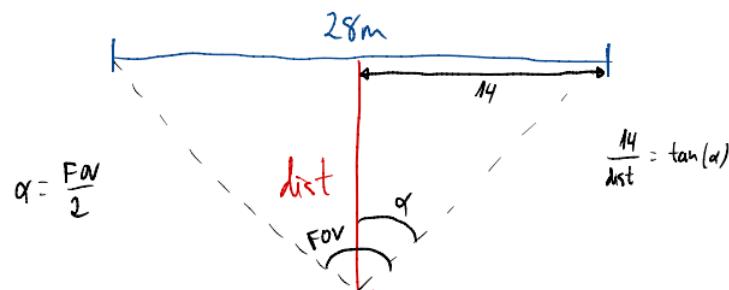
### Apartado C

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

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

Otra forma de hacerlo es usar la fórmula

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

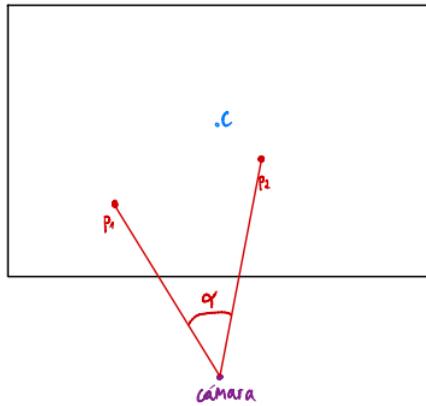
obteniendo

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

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:

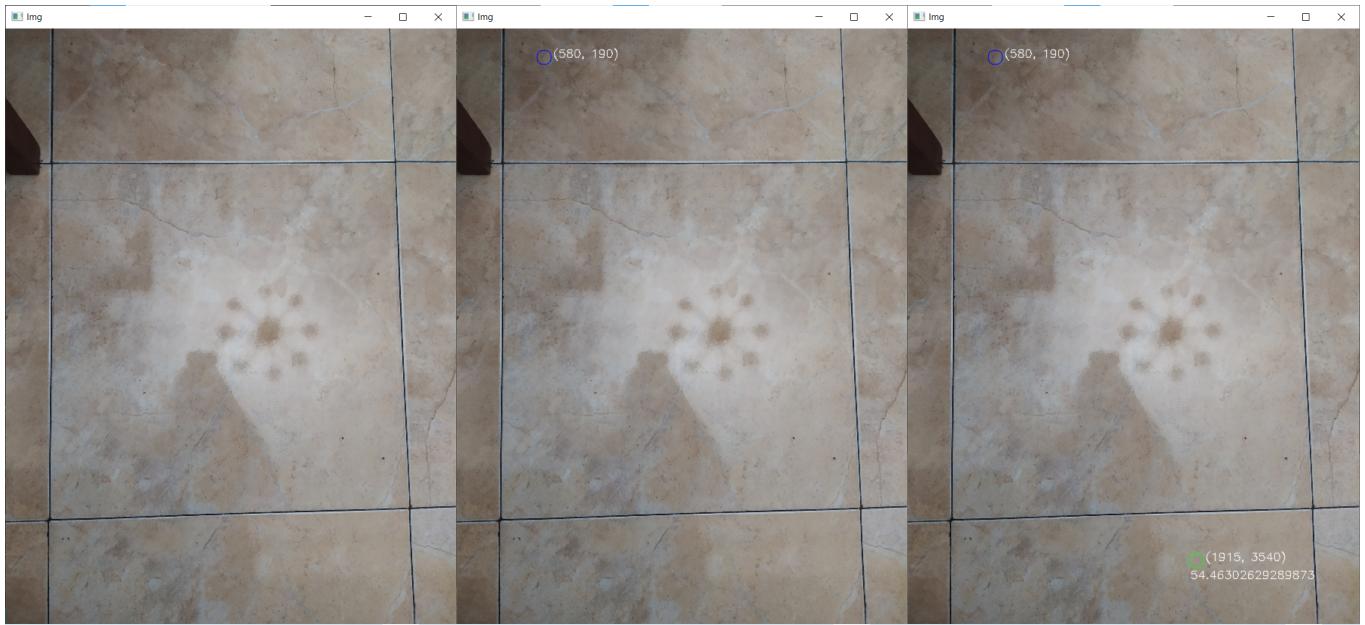


Figure 1: Ejemplo de ejecución

### 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.

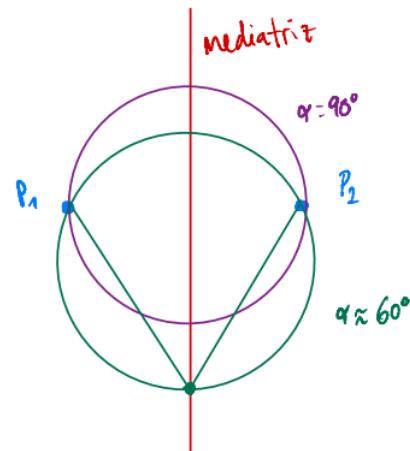
Por otro lado, el lugar geométrico de los puntos del plano de aquellas circunferencias que contienen a dos puntos dados es su mediatrix.

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  $P_3$ .

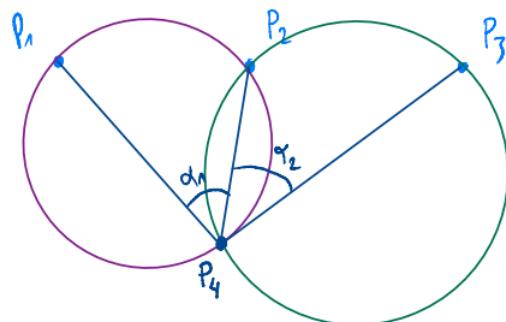
Esta situación queda ilustrada en el siguiente diagrama:



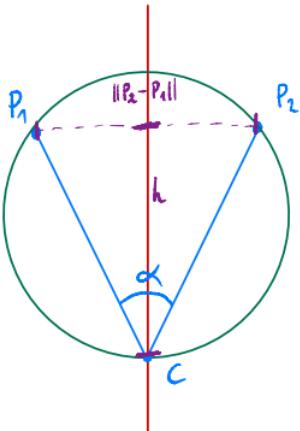
La localización se puede hacer de forma precisa si conocemos un punto adicional de la imagen, y el ángulo que forma entre la cámara y uno de los puntos anteriores. Es decir, ahora estamos en la situación de conocer  $P_1, P_2, P_3$  puntos de la imagen,  $P_4$  la cámara y  $\alpha_1 = \angle(\overrightarrow{P_4P_1},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



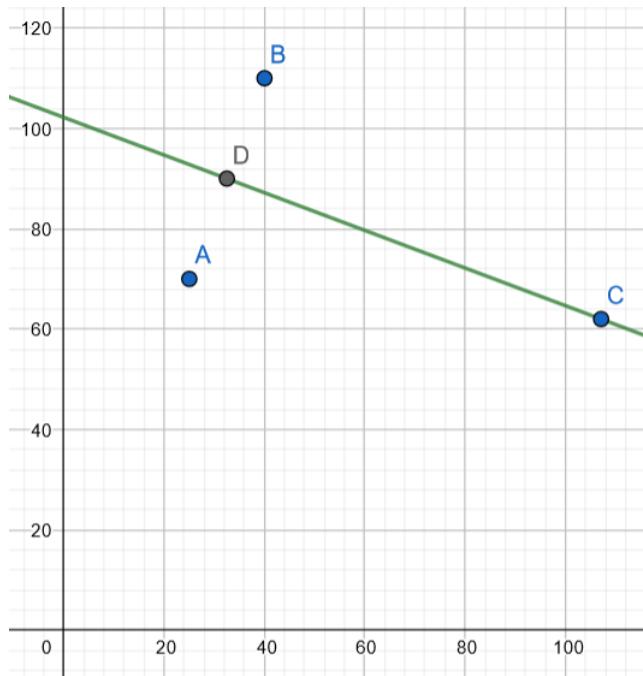
Tenemos que  $\tan\left(\frac{\alpha}{2}\right) = \frac{\|P_2 - P_1\|}{h}$ , por lo que

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

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

### Enunciado

Construye un detector de movimiento en una región de interés de la imagen marcada manualmente. Guarda 2 ó 3 segundos de la secuencia detectada en un archivo de vídeo.

Opcional: muestra el objeto seleccionado anulando el fondo.

### Solución

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:

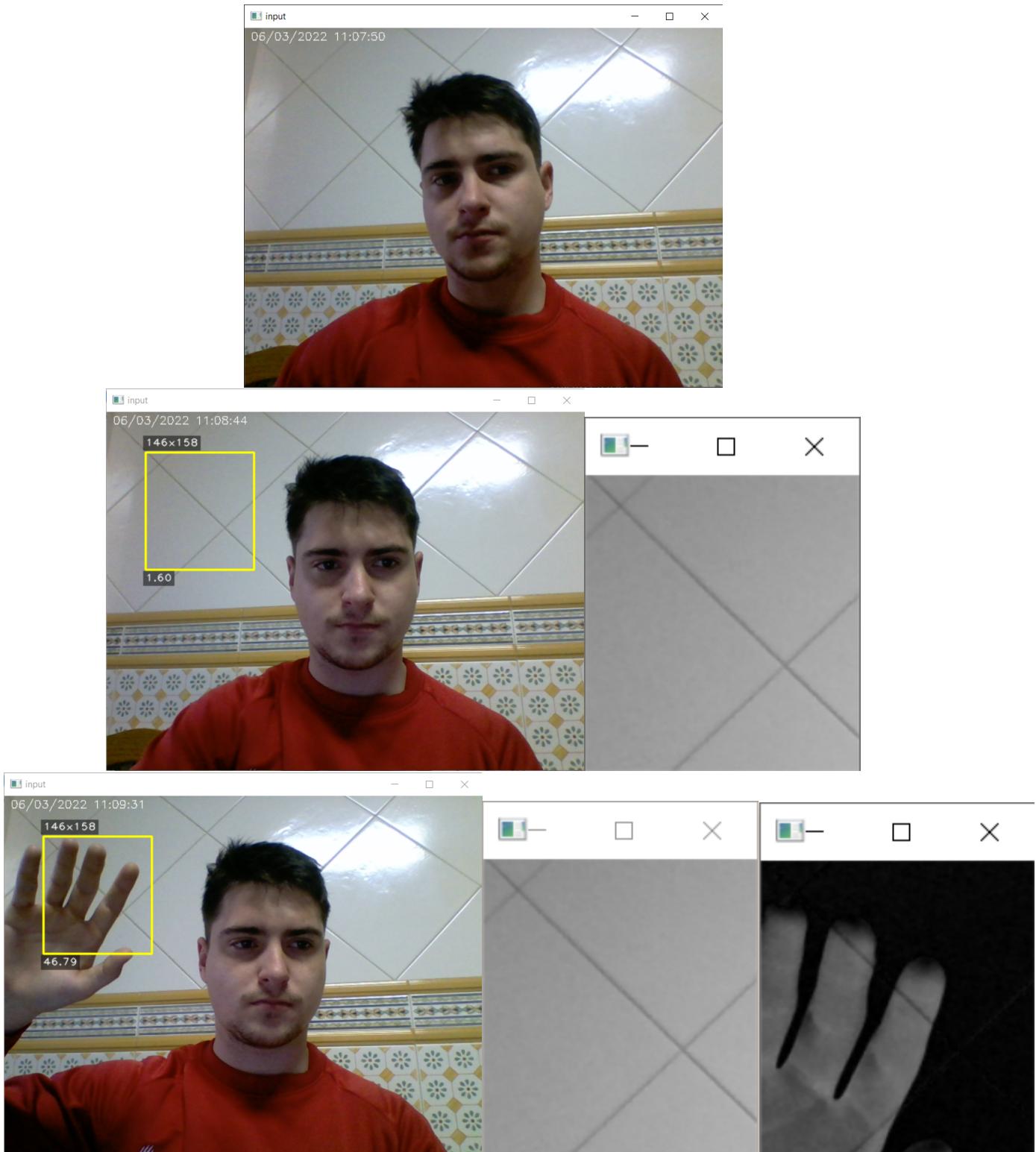


Figure 2: Ejemplo de funcionamiento del detector de movimiento

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:

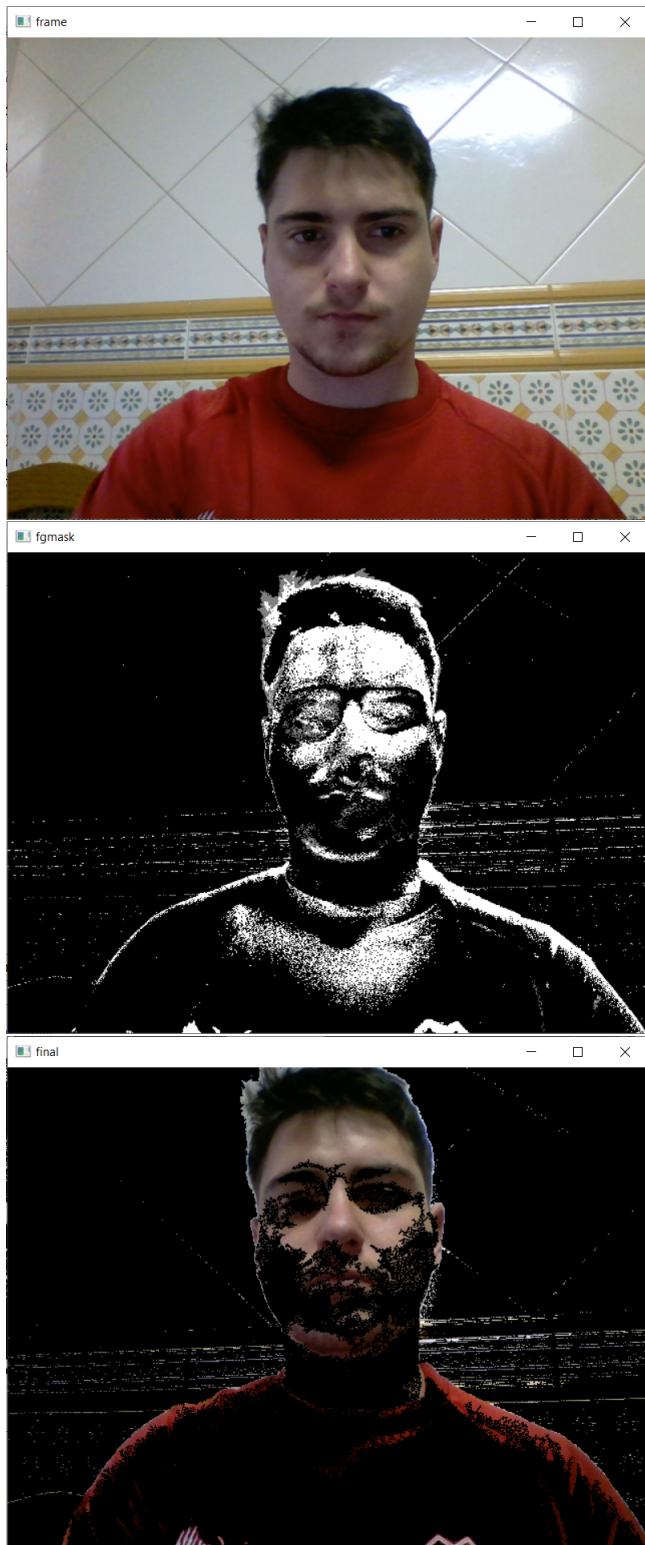


Figure 3: Ejemplo de funcionamiento del eliminador del fondo

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

#### Enunciado

Construye un clasificador de objetos en base a la similitud de los histogramas de color del ROI (de los 3 canales por separado).

Opcional: Segmentación densa por reprojeción de histograma.

#### Solución

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 ( $h_1, h_2$ )* 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 resizo 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 empeoren 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:

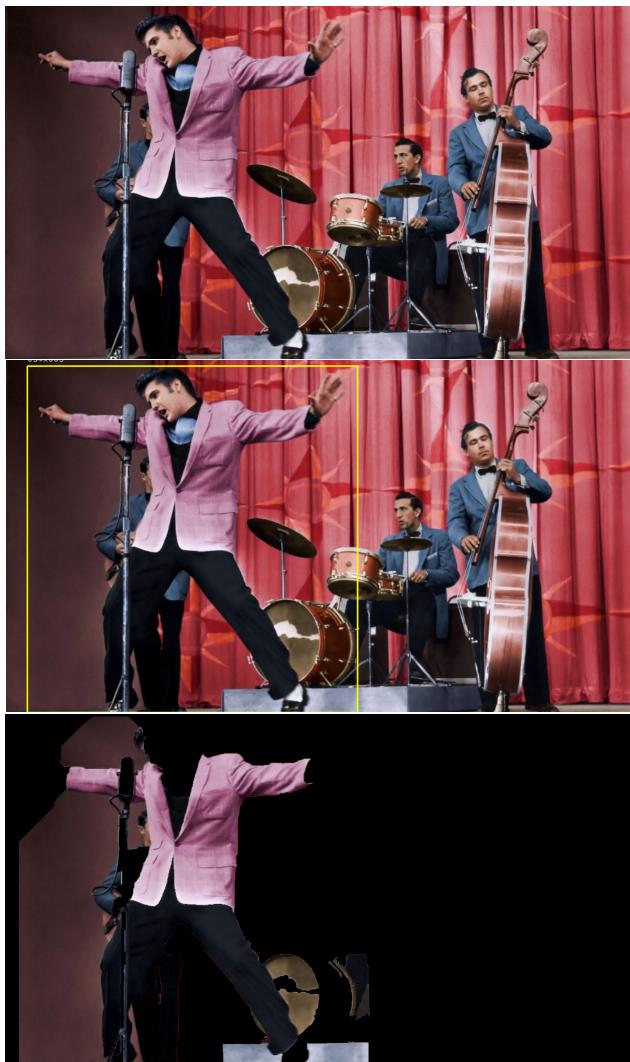


Figure 4: Ejemplo de mi programa que aplica grabcut

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 Ejercicio 4 - SIFT

### Enunciado

Escribe una aplicación de reconocimiento de objetos (p. ej. carátulas de CD, portadas de libros, cuadros de pintores, etc.) con la webcam basada en el número de coincidencias de keypoints.

### Solución

Para realizar el reconocimiento de objetos, he modificado el programa del profesor *sift.py* para que coja todas las imágenes ubicadas en la carpeta *modelos*, ubicada en el mismo directorio que el programa. Calculo los keypoints y los descriptores de todos ellos y los guardo, para no tener que recalcularlos constantemente.

Tengo dos variables de estado que me ayudan con la lógica del programa:

- *state*: está a 0 si no hemos estamos comparando la imagen actual con los modelos. Si pulsamos *c* se pone a 1 y entramos en modo de comparación. Para comparar, usamos *knnMatch* para cada modelo y el frame actual, y hacemos un ratio test, quedándonos solo con aquellos matches en los que la distancia en el mejor match sea menor que 0.75 por la del segundo. Entre todos los modelos, nos quedemos con aquel que obtenga mayor cantidad de matches sobre el total. Consideraremos que un modelo es adecuado si el porcentaje de matches buenos sobre el total es mayor a 20%, basándome en las pruebas que he realizado. Este porcentaje es pequeño porque los modelos han sido imágenes de carátulas de libros sacadas de internet, que enseñaba a la cámara con el móvil, por lo que hay muchos factores que pueden hacer que se pierdan keypoints, como los reflejos en la pantalla del móvil, la posible inclinación de mi mano, etc.
- *matched*: está a *False* si no hemos encontrado un modelo suficientemente bueno para el frame actual, o si no estamos comparando. Está a *True* en caso contrario. Este estado nos ayuda a identificar si la ventana que muestra el modelo correcto ha sido creada, y a destruirla cuando corresponda.

En la figura

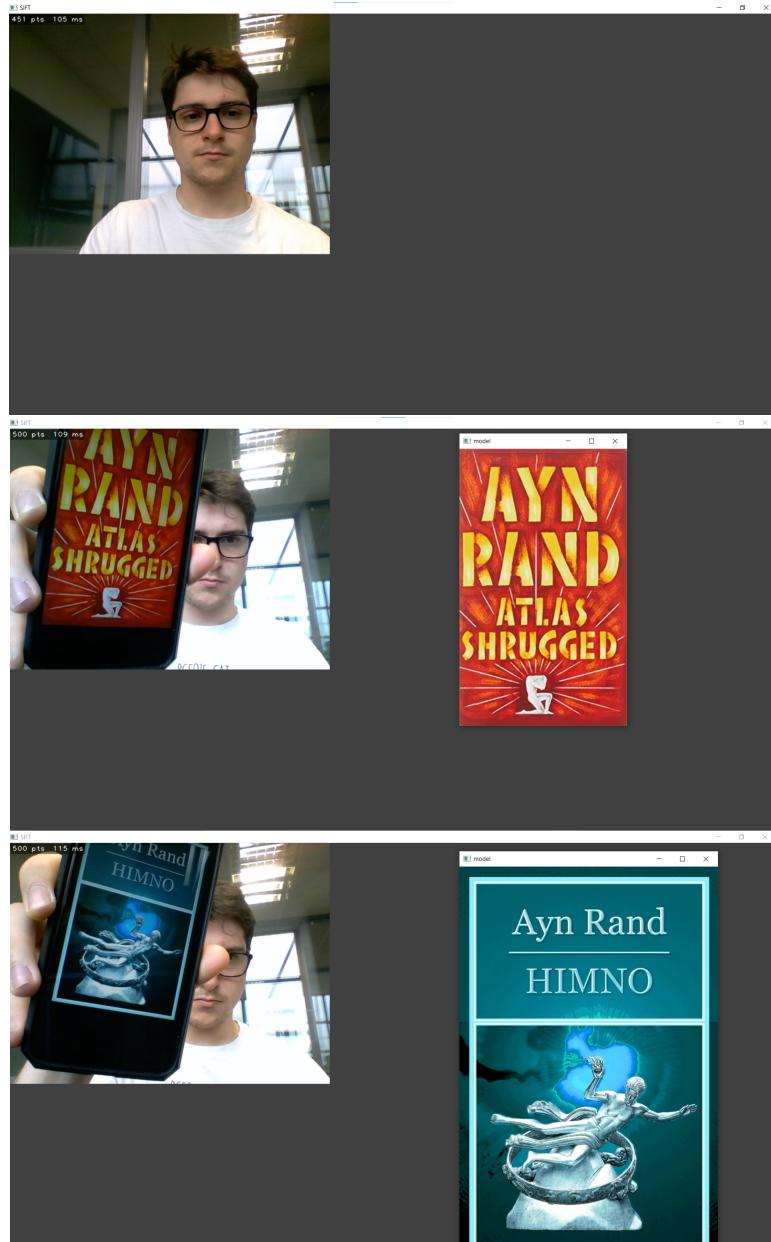


Figure 5: Ejemplo de funcionamiento de SIFT. La primera imagen se corresponde con el inicio del programa. La segunda ocurre al poner el teléfono con esa imagen en pantalla y pulsar *c*, la tercer al cambiar a la segunda imagen y pulsar *c*.



Figure 6: gol-eder.png

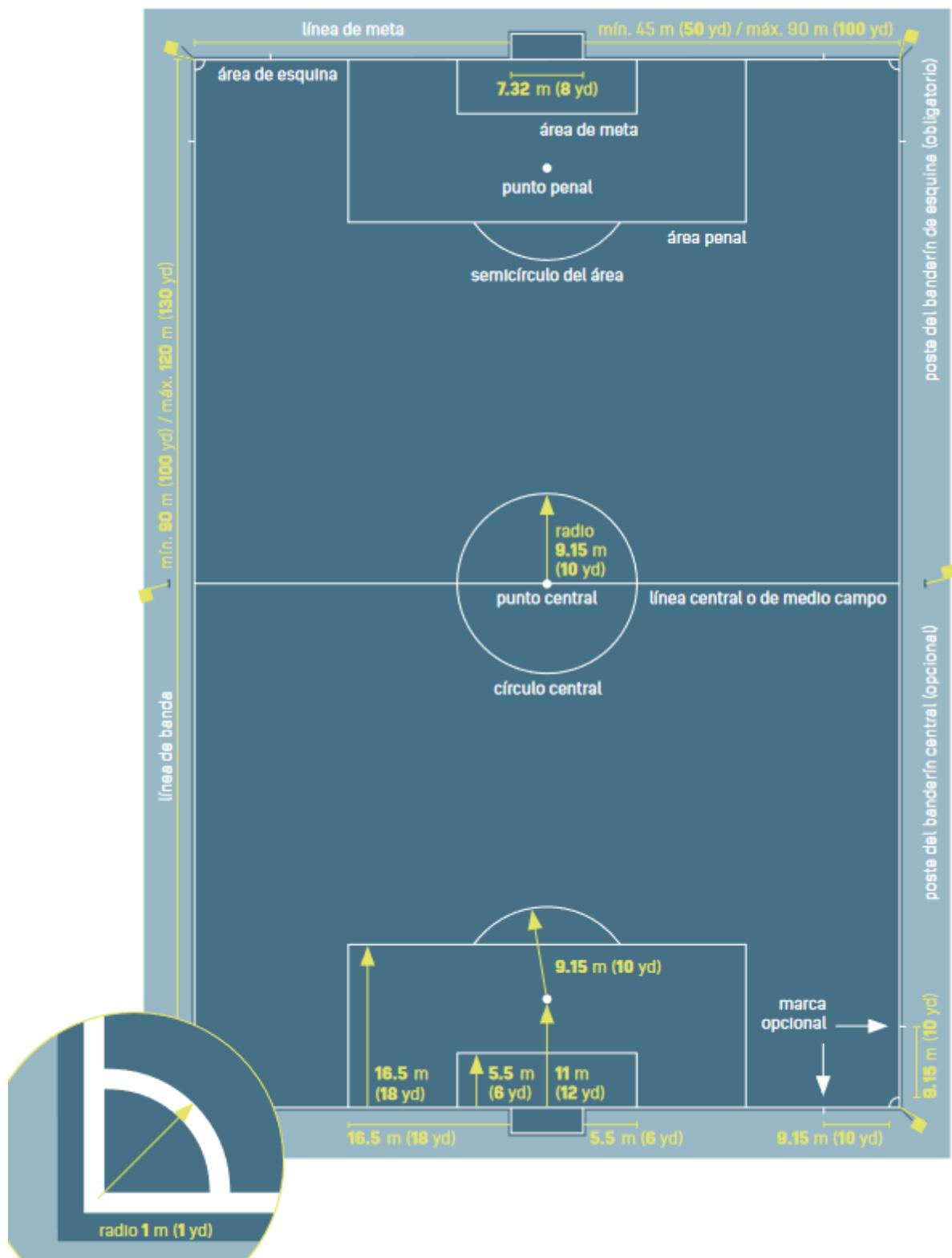
## 5 Ejercicio 5 - RECTIF

### Enunciado

Rectifica la imagen de un plano para medir distancias (tomando manualmente referencias conocidas). Por ejemplo, mide la distancia entre las monedas en coins.png o la distancia a la que se realiza el disparo en gol-eder.png. Las coordenadas reales de los puntos de referencia y sus posiciones en la imagen deben pasarse como parámetro en un archivo de texto. Aunque puedes mostrar la imagen rectificada para comprobar las operaciones, debes marcar los puntos y mostrar el resultado sobre la imagen original. Verifica los resultados con imágenes originales tomadas por ti.

### Solución

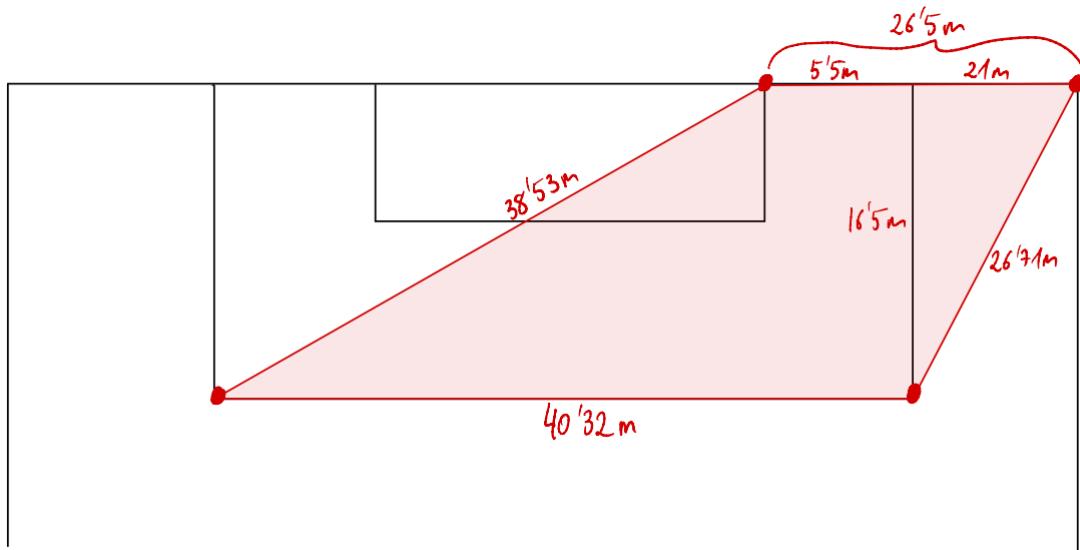
Primero, vamos a hacer el desarrollo teórico para el caso de la imagen *gol – eder.png*:  
Según el reglamento oficial de fútbol [7], un campo de fútbol tiene las siguientes medidas



por lo que podemos tomar los 4 puntos marcados en la siguiente imagen:  
Y tenemos la siguiente situación



Figure 7: *gol – eder.png* con los puntos importantes marcados en rojo.



donde, dado que la línea de meta puede medir entre 45 y 90 metros, he buscado el partido y se jugó en el *Stade de France*, que tiene 75 metros de ancho. Así obtenemos las distancias del esquema. Podemos tomar, por ejemplo, los puntos  $\{(100, 150), (262, 150), (344, 84), (238, 84)\}$  verifican de forma bastante aproximada las proporciones requeridas.

En el programa *rectif.py* le vamos a pedir al usuario que marque los puntos en este orden, y vamos a aplicar *cv.findHomography* entre los puntos marcados y los pasados mediante un fichero de texto. Los ficheros de texto tienen que tener el formato:

```
d12=N
(p1x, p1y)
...
(pKx, pKy)
```

siendo  $N$  la distancia entre  $p1$  y  $p2$  en la realidad, en la medida que deseemos (debe ser un entero) y

lo demás las coordenadas de cada uno de los puntos que pasamos. Esto se usa para poder transformar la distancia en píxeles a la distancia en la medida pasada como parámetro.

El usuario debe entonces seleccionar  $K$  puntos que se correspondan con los pasados como parámetros, en el mismo orden.

Tras esto, al seleccionar puntos en la imagen original, se dibujarán en la imagen transformada y se mostrará su distancia en el centro de la imagen.

Se muestran, paralelamente, las posiciones de los polígonos de puntos tal y como quedan en la imagen real y cómo deben quedar en la imagen reconstruida.

En la figura 8 podemos ver lo que nos devuelve el programa con la entrada

```
d12=40
(100,150)
(262,150)
(344,84)
(238,84)
```

es decir, que la distancia entre los dos primeros puntos es de 40 (la unidad de medida es metros, y estamos aproximando los 40.3 m que mide el borde del área de un campo de fútbol).

Nótese que señalamos que mida la distancia entre el pie de Eder y el punto más cercano a la portería que hay en la foto. Lo he hecho de esta manera porque no me parece que tenga mucho sentido tratar de medir distancias hacia objetos que quedan fuera de la imagen. Como puede observarse, Eder disparó aproximadamente desde una distancia de 27 metros.

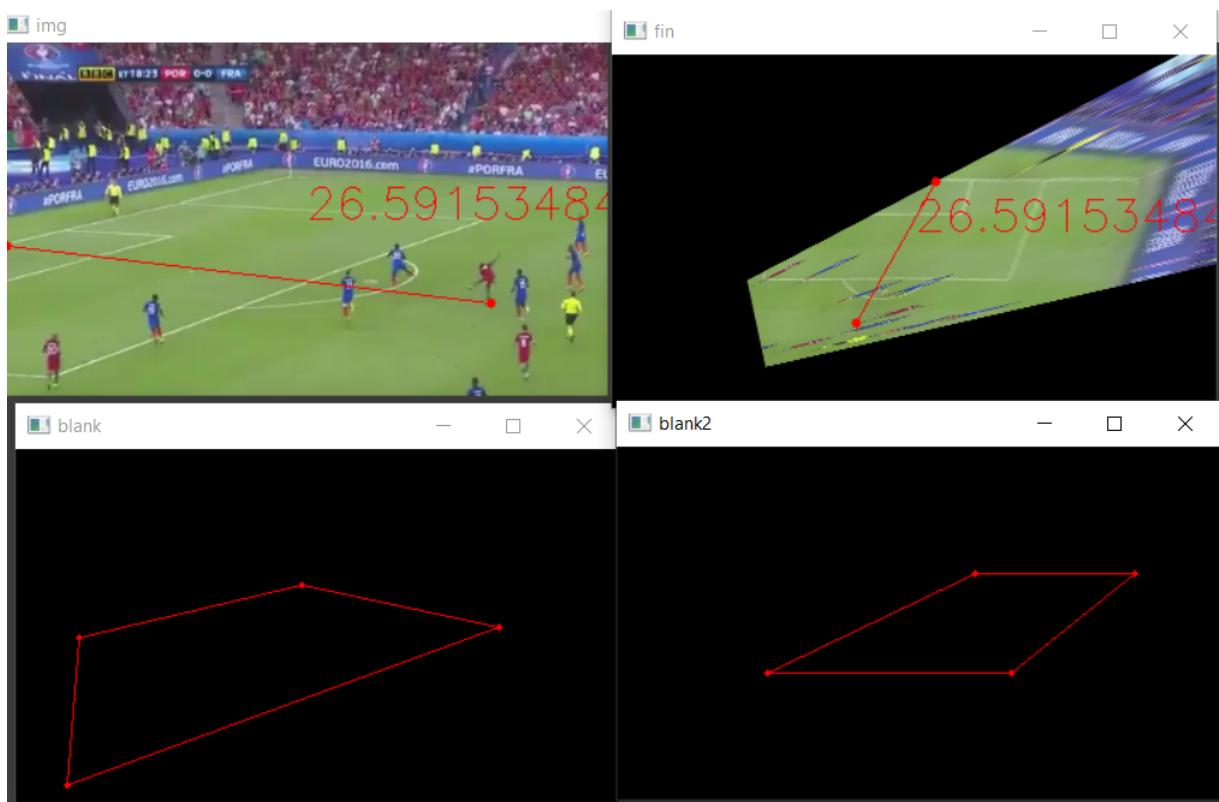


Figure 8: Ejemplo final para *gol – eder.png*

Para el caso de *coins.png*, la salida la podemos ver en la figura 9. Como observamos, la distancia entre las monedas es de aproximadamente 111, y la medida está en mm, ya que el archivo de entrada es:

```
d12=85
```

(100,200)  
 (185,200)  
 (185,255)  
 (100,255)

donde estoy marcando las esquinas de una tarjeta de medidas  $8.5 \times 5.5\text{cm}$ , luego  $d12 = 85\text{mm}$ . Es decir, la distancia entre las monedas es de  $11.1\text{cm}$ . Además, las aproximaciones son bastante buenas, ya que usando la regla para comparar el resultado obtenido con la realidad, nos encontramos que desde la marca en el 0 hasta el 10, el programa nos dice que hay  $100.3\text{ mm}$ , lo cual es bastante buen resultado. Más aún si tenemos en cuenta que estamos señalando los puntos a mano.

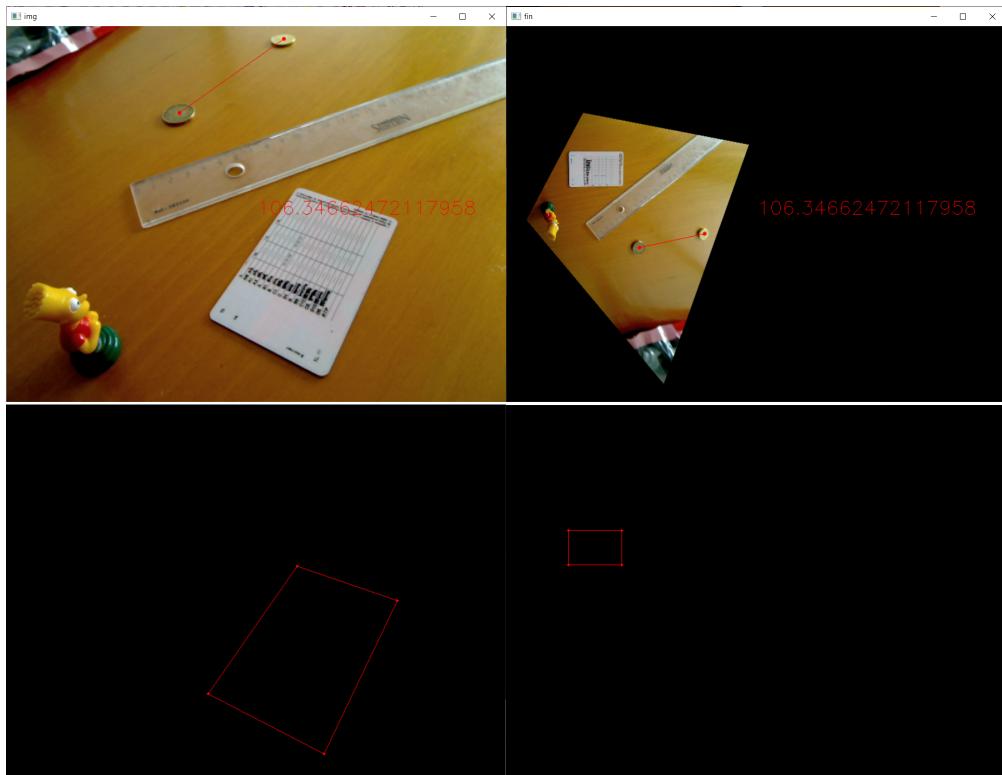
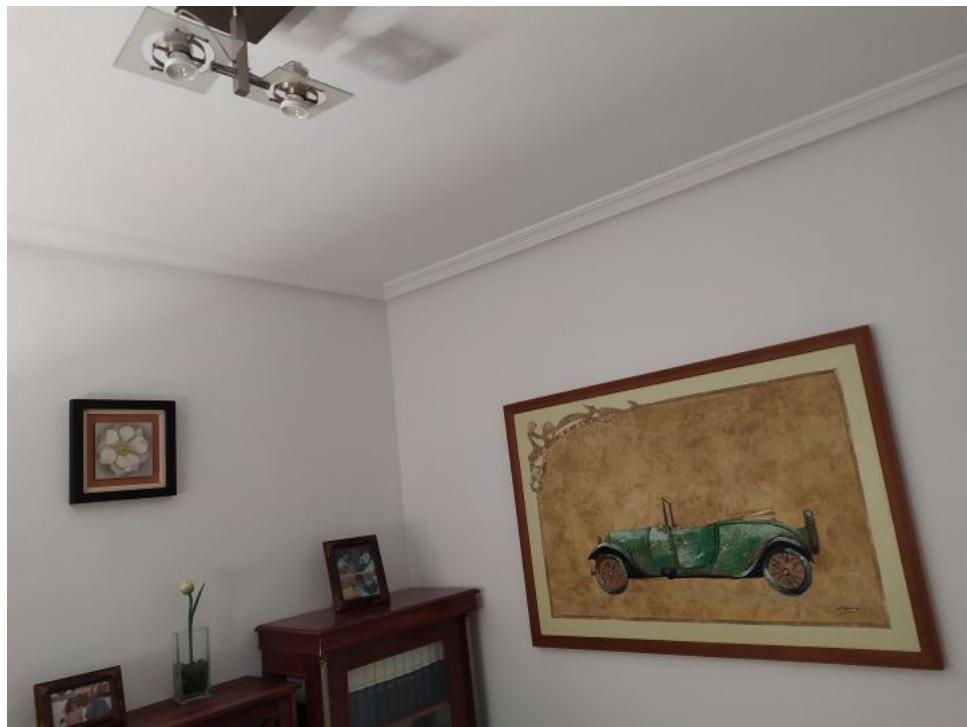


Figure 9: Ejemplo para *coins.png*

Vamos ahora a ver un ejemplo propio. Consistirá en la siguiente imagen:



Vamos a usar el cuadro del coche. Las dimensiones del marco son  $127 \times 87\text{cm}$ , por lo que el texto de información es

- $d_{12}=127$   
 $(400, 200)$   
 $(654, 200)$   
 $(654, 374)$   
 $(400, 374)$

Por otro lado, voy a medir la distancia entre los centros de las ruedas del coche del cuadro. En la realidad esta distancia es de 56.5 cm. El programa arroja los siguientes resultados:

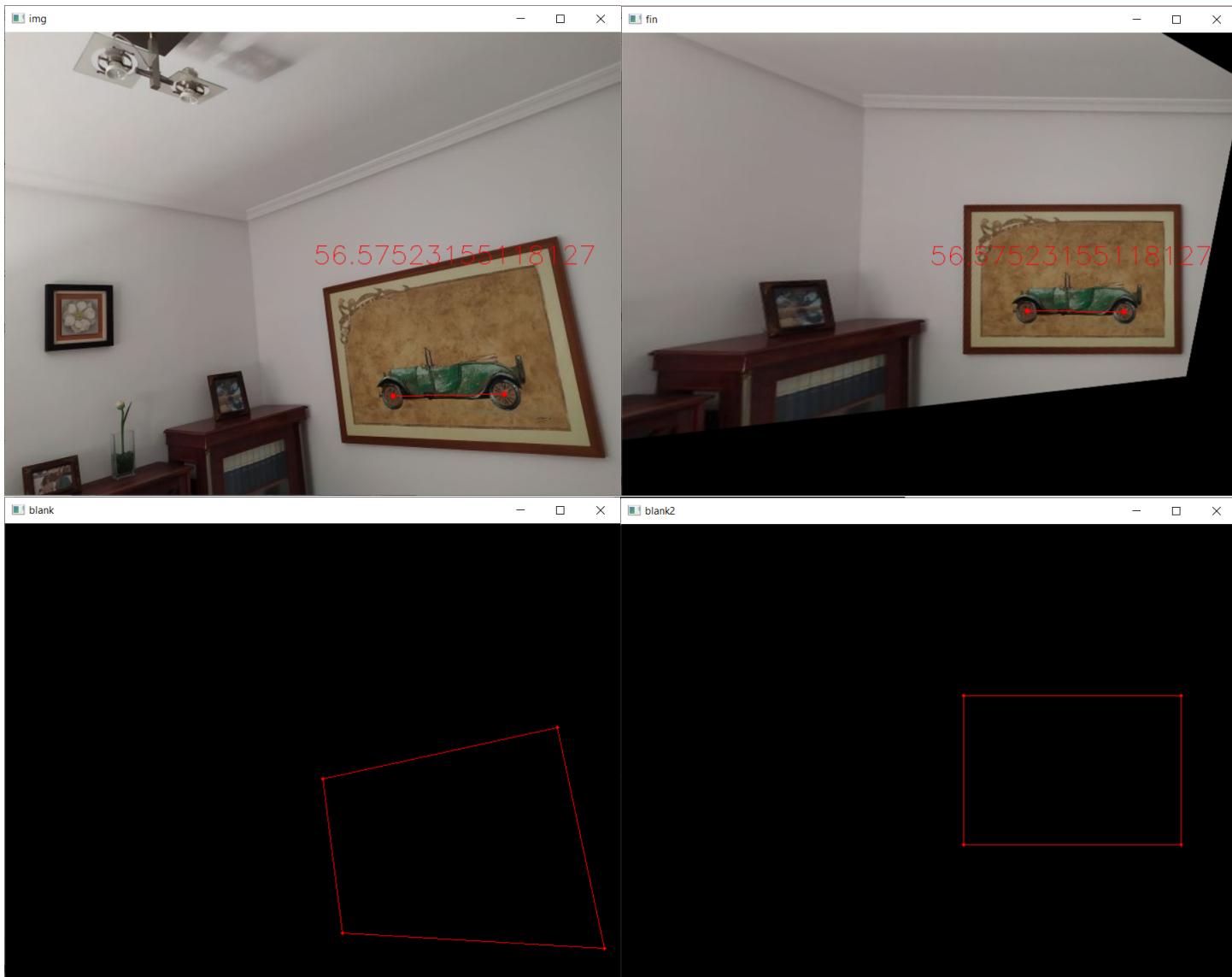


Figure 10: Ejemplo propio con *cuadro.jpg*.

Como podemos observar, el resultado obtenido es bastante preciso.

### Casos de error y consideraciones de uso

Debemos hacer una puntuación al respecto del funcionamiento de este programa. Al pasarle los puntos, debemos tener en cuenta las siguientes consideraciones:

- Debe haber, al menos, 4 puntos, de tal forma que tengamos un polígono de, al menos, 4 lados. Es decir, si un punto está contenido en el segmento que une otros dos puntos, entonces no obtendremos el resultado deseado.
- Los puntos seleccionados es deseable que abarquen la mayor cantidad de imagen posible, esto hace que los resultados sean más precisos.
- Los puntos pasados como parámetro deben tener unas dimensiones similares a las que tiene el mismo área en la imagen. Esto podría solucionarse calculando el factor  $f = \frac{\text{areaZonaImagen}}{\text{areaTotalImagen}}$  y haciendo que la imagen reconstruida tenga un área de  $f \cdot \text{areaZonaReal}$ .

El caso de error podemos verlo, por ejemplo, usando *gol – eder.png* con el archivo:

```
d12=85  
(100,200)  
(185,200)  
(185,255)  
(100,255)
```

Y obtenemos los resultados de la figura 11, en los que observamos como la imagen reconstruida no nos es útil, y es claro que algo no está funcionando correctamente.

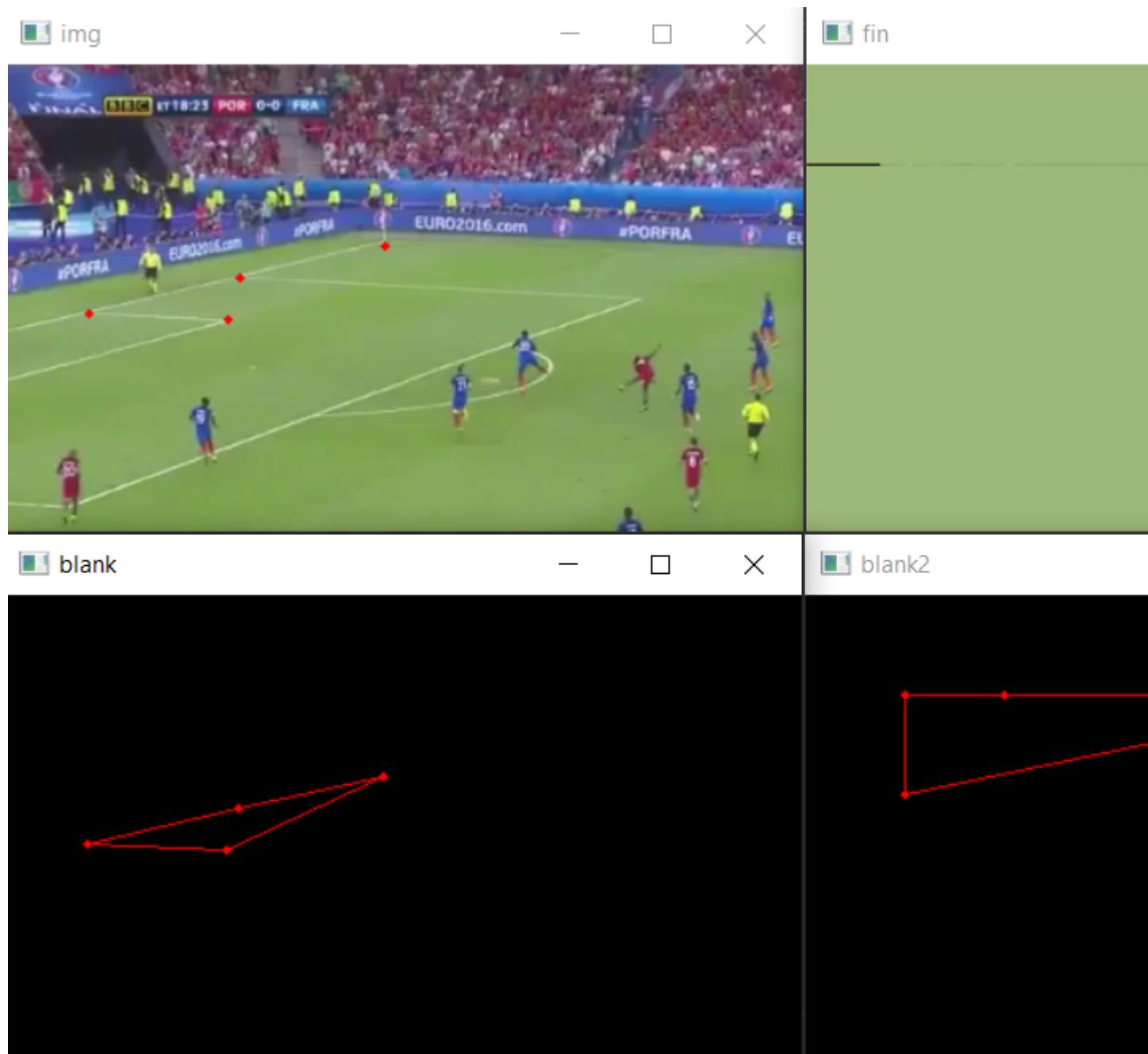


Figure 11: Caso de error

## 6 Ejercicio 6 - RA

### Enunciado

Crea un efecto de realidad aumentada interactivo: esto significa que

- a) los objetos virtuales deben cambiar de forma, posición o tamaño siguiendo alguna lógica;
- b) el usuario puede observar la escena cambiante desde cualquier punto de vista moviendo la cámara alrededor del marcador; y
- c) el usuario puede marcar con el ratón en la imagen puntos del plano de la escena para interactuar con los objetos virtuales.

### Solución

Para resolver este ejercicio, he seguido las indicaciones del notebook de clase *camera.ipynb* ([8]).

Los apartados a) y b) vienen prácticamente desarrollados en dicho notebook. El proceso, a grandes rasgos, es el siguiente:

1. Definimos el marcador, así como la figura que deseemos dibujar. Nótese que la figura es un cubo, y que lo defino mediante un vértice  $v$ , un lado de la base  $l$  y una altura  $h$ , esto permite realizar las operaciones para su modificación de forma muy sencilla. Debemos también conocer la matriz de cámara, para lo que podemos seguir el proceso descrito en 1.
2. Pasamos la imagen a escala de grises, para poder realizar una búsqueda de la forma del marcador de forma más sencilla.
3. Buscamos los contornos que hay en la imagen.
4. De entre los contornos encontrados, filtramos aquellos con la misma cantidad de lados que nuestro marcador. En nuestro caso tiene 6 lados.
5. De entre los contornos de 6 lados, nos quedamos con aquellos que no superen cierto umbral de error al ajustarlos a nuestro modelo. Obtenemos la homografía que lleva el modelo a la imagen real.
6. Transformamos la figura que deseamos dibujar mediante la homografía obtenida.
7. Solo queda dibujar la figura en la imagen.

Respecto al apartado c), podemos añadir diversas opciones, de forma no excesivamente compleja, permitiendo al usuario pulsar teclas que produzcan distintos efectos, como desplazar el modelo en diferentes direcciones (relativas al marcador), aumentar su tamaño o cambiar su color.

El código puede consultarse en el archivo *ra.py*. Si se ejecuta este programa, veremos una ventana de ayuda que nos indicará todos los comandos que podemos ejecutar.

La interacción más interesante que he añadido ha sido la posibilidad de eliminar el marcador, haciendo parecer que no hay ninguno. Para ello, defino un punto cercano al marcador, y relleno un cuadrado que cubra completamente el marcador de un color igual al del punto guía. Nótese que esto solo sirve si sabemos que los alrededores del marcador son constantes. No obstante, es una primera aproximación interesante al problema de eliminar el marcador.

También he hecho lo que se menciona en el enunciado: el usuario puede marcar un punto con el ratón y el cubo se mueve a esa posición. Esto no es tan sencillo como puede parecer en un primer momento, ya que estamos trabajando con figuras tridimensionales, y entonces la matriz  $H$  que obtenemos no nos sirve para conseguir el punto inverso de un punto en la pantalla. Tampoco podemos utilizar el polígono que obtenemos para conseguir esa  $H$  y usarlo para obtener una nueva  $H_2$  con *findHomography*, ya

que el polígono puede tener un orden distinto a nuestro marcador. La solución que se me ha ocurrido es obtener la homografía mediante la siguiente línea de código:

```
H2 = cv.findHomography(np.int32([htrans(H, marker)]), marker)
```

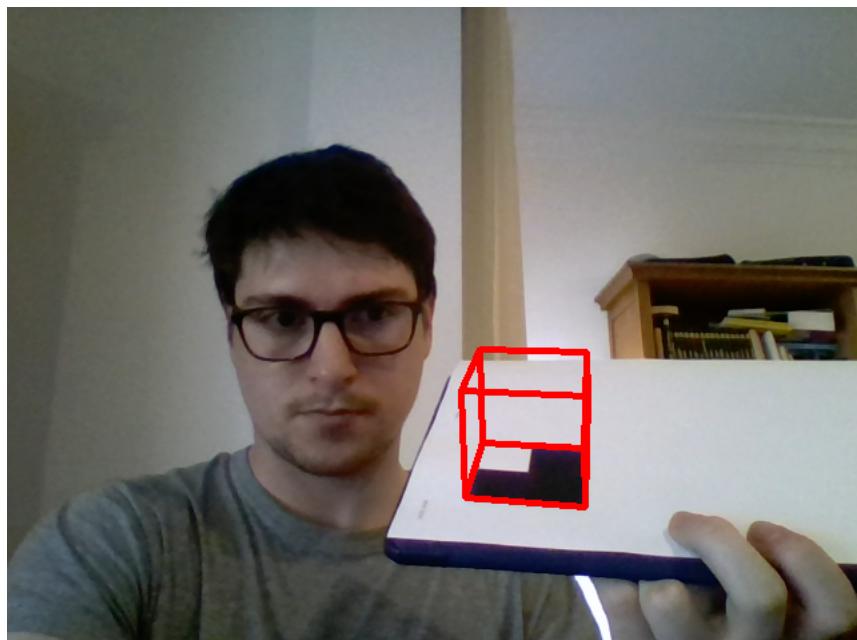
Así, transformo el punto en la imagen a su correspondiente punto en el plano, y muevo el cubo a esa posición.

Vamos a ver una secuencia de comandos ejecutados en nuestro programa.

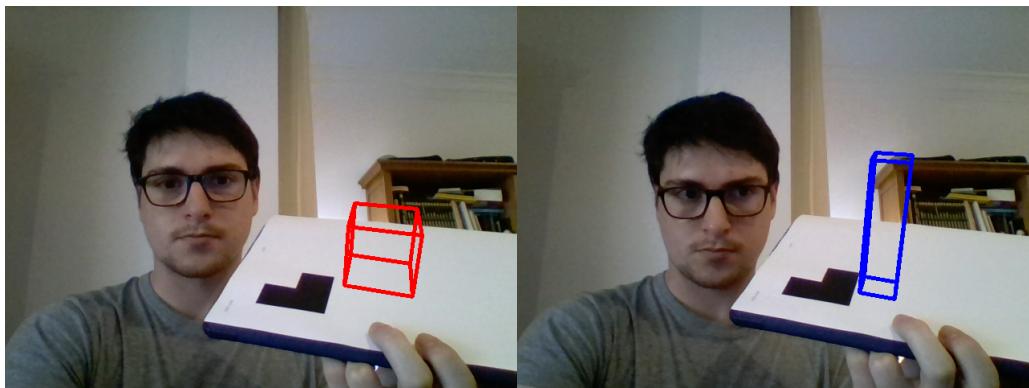
Al comenzar el programa, estamos en esta situación:



Si pulsamos la *r*, el cubo cambia de color:



Podemos desplazarlo, y volver a cambiar su color:



Si ahora pulsamos  $o$ , podemos eliminar el marcador:



En esta última imagen, si nos fijamos podemos ver el cuadrado blanco, que difiere ligeramente de sus alrededores, pero el efecto es suficientemente bueno como para que no se note a simple vista. Nótese, no obstante, que hay fluctuaciones, y el marcador en ocasiones parpadea.

## 7 Opcional 1 - Filtros

### Enunciado

Amplía el código de la práctica 4 para mostrar en vivo el efecto de diferentes filtros, seleccionando con el teclado el filtro deseado y modificando sus parámetros (p.ej. el nivel de suavizado) con trackbars.

a) Aplica el filtro en un ROI para comparar el resultado con el resto de la imagen

### 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).

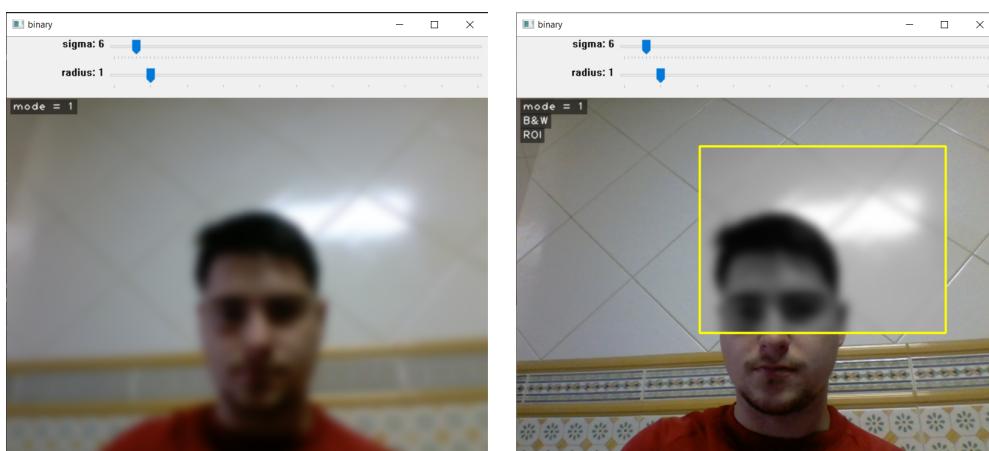
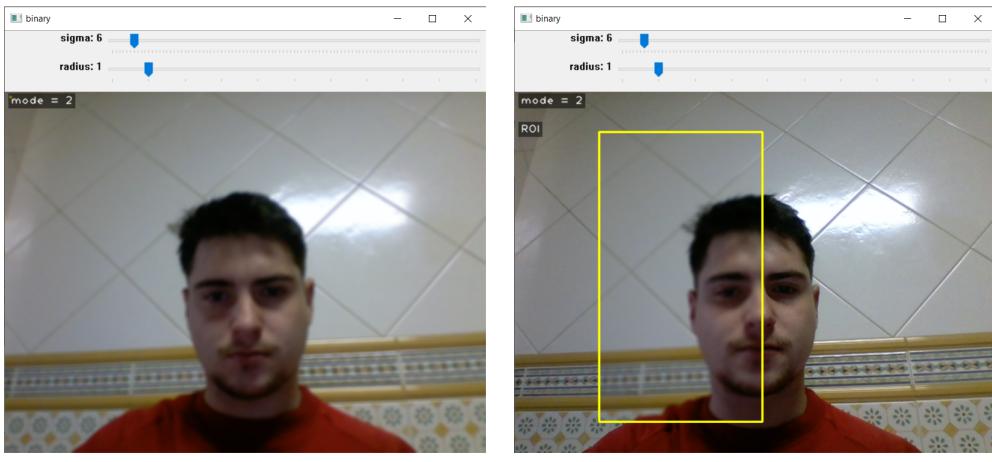


Figure 12: Filtro gaussiano.

- 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.

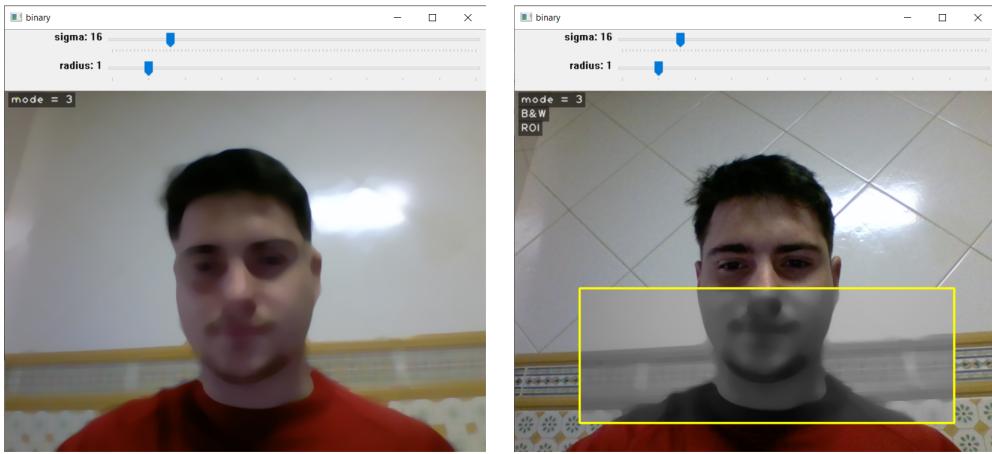


(a) A color. A toda la imagen.

(b) A color. En un ROI.

Figure 13: 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).

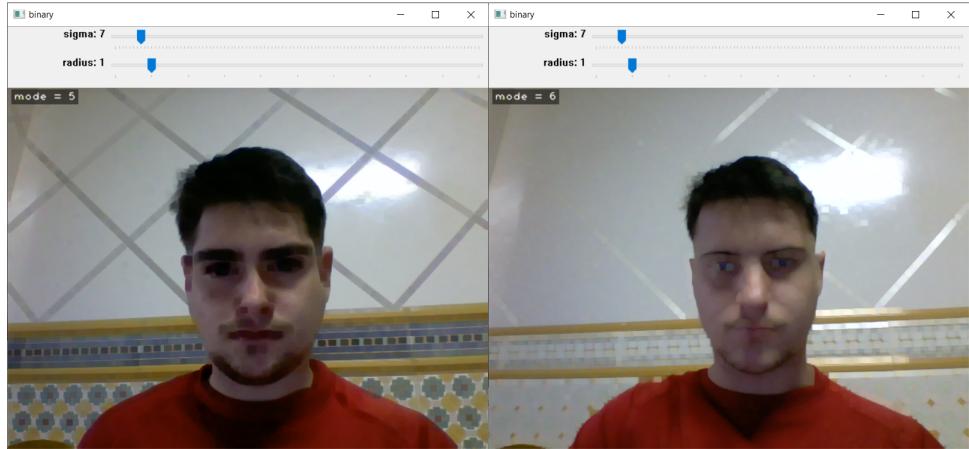


(a) A color. A toda la imagen.

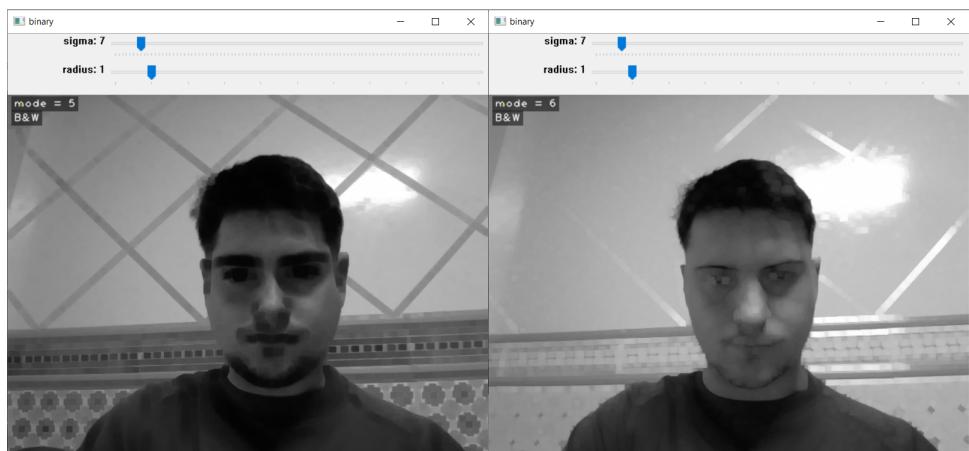
(b) A B&W. En un ROI.

Figure 14: 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 15: Filtros del mínimo (izq) y del máximo (dcha).

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

## 8 Opcional 5 - RC

### Enunciado

Reproduce la demostración del cross-ratio de 4 puntos en una recta del tema de transformaciones del plano. Marca tres puntos con el ratón y añade automáticamente tres más y el punto de fuga, suponiendo que en el mundo real los puntos están alineados y a la misma distancia.

### Solución

Este ejercicio lo voy a realizar usando la siguiente imagen:



Es una imagen tomada en el casco antiguo de mi pueblo, Cehegín. Concretamente son los Soportales, que tienen cierta historia, como puede leerse en <https://latitudesinfinitas.com/que-ver-en-cehegin-en-un-dia/>. “Construidos en el año 1725, en su día fueron palcos privados de familias adineradas que usaban para ver los espectáculos que se ofrecían en la plaza. Y es que, en esta plaza se han dado todo tipo de espectáculos, desde corridas de toros a ceremonias religiosas pasando por supuesto por los mercados callejeros. Visitando los soportales de Cehegín. Con el paso del tiempo, en el siglo XIX, los huecos que quedan bajo los soportales se cerraron para crear espacios en los que se vendía carne y pescado.” Pues bien, vemos que las tejas del techado están alineadas (aunque dada la antigüedad del edificio es plausible que no lo estén perfectamente), por lo que podemos usarlas para nuestros propósitos. Voy a tomar cuatro tejas, con una separación de dos tejas (una entre medias). De esta manera, tenemos 4 puntos equiespaciados y alineados,  $A, B, C, D$  que podemos expresar como  $A, B = A+t, C = A+2t, D = A+3t$ , por lo que, según la notación usada en la asignatura, es

$$\begin{aligned}a &= \text{dist}(A, B) = t, \\b &= \text{dist}(B, C) = t, \\c &= \text{dist}(C, D) = t.\end{aligned}$$

Y entonces, el cross-ratio queda

$$CR = \frac{a}{a+b} \frac{c}{c+b} = \frac{t}{2t} \frac{t}{2t} = \frac{1}{4}.$$

Ahora lo hacemos en la imagen, para calcular los puntos en los que están las tejas he utilizado un manejador de ratón similar al usado en el Ejercicio 1 para otro asunto, pero que me servía ahora. Una vez conocemos los puntos, basta calcular los nuevos  $a, b, c$  y el  $CR$ . Esto puede verse en el programa *CR.py*, que es muy sencillo. Si lo ejecutamos podremos ver que obtenemos un  $CR = 0.242$ , suficientemente cercano a  $\frac{1}{4}$  como para quedarnos agusto, ya que la aproximación al tomar los puntos en la imagen pasa factura.

Solo falta añadir ahora tres puntos más.

Para añadir otro punto,  $p$ , tenemos que tener en cuenta que  $P = D + t$  en la realidad, y que debe cumplirse el CR. Llamemos a los puntos en la imagen  $A_p, B_p, C_p, D_p$  y  $P$ . Entonces es

$$\frac{1}{4} = \frac{\text{dist}(B_p, C_p)}{\text{dist}(B_p, C_p) + \text{dist}(C_p, D_p)} \frac{\text{dist}(D_p, P)}{\text{dist}(D_p, P) + \text{dist}(C_p, D_p)},$$

que, con la nomenclatura del programa, y llamando  $d = \text{dist}(D_p, P)$  (nuestra incógnita) queda

$$\frac{1}{4} = \frac{b}{b+c} \frac{d}{d+c} \iff (b+c)(d+c) = 4bd \iff bc + c^2 = d(3b - c) \iff d = \frac{bc + c^2}{3b - c}.$$

Y entonces

$$p = D_p + d \cdot \frac{\overrightarrow{C_p D_p}}{\|\overrightarrow{C_p D_p}\|},$$

esto puede verse también en el programa *CR.py*, con el proceso repetido para otros tres puntos.

Para obtener el punto de fuga, podemos iterar este proceso hasta alcanzarlo (paro cuando avanzamos menos de un pixel en la siguiente iteración,  $d < 1$ ). El punto de fuga puede observarse como un punto rojo relleno. Parece que se desvía ligeramente de la recta formada por las tejas, pero es bastante poco y lo hemos hecho iterando a partir de los tres primeros puntos de la misma, por lo que el resultado es aceptable.

El resultado final puede verse en la figura 16.



Figure 16: Resultado final del ejercicio CR

## 9 Opcional 6 - Swap

### Enunciado

Intercambia dos cuadriláteros en una escena marcando manualmente los puntos de referencia.

### Solución

La solución a este problema está basada en la del ejercicio RECTIF (5).

El programa realizado, que puede consultarse en el archivo *swap.py* permite al usuario marcar dos cuadriláteros, que serán intercambiados, manteniendo las posiciones originales.

Para ello, se sigue el siguiente proceso:

1. El usuario introduce la ruta de la imagen.
2. El usuario marca 8 puntos, los 4 primeros se almacenan en *pImagen* y los 4 siguientes en *pImagen2*.
3. Buscamos la homografía entre el primer cuadrilátero y el segundo, y viceversa (también podríamos calcular la inversa de la primera).
4. Calculamos dos nuevas imágenes, mediante *cv.warpPerspective*, aplicado a la imagen original y con cada una de las homografías obtenidas. Supongamos que se llaman *i1* e *i2*.
5. Creamos dos máscaras que marcan el interior de cada cuadrilátero mediante *cv.fillConvexPoly*. Supongamos que se llaman *m1* y *m2*.
6. Creamos una nueva imagen, que no es más que una copia de la primera. En esta nueva imagen, *i*, hacemos

$$\begin{aligned} i[m1] &= i1[m1] \\ i[m2] &= i2[m2], \end{aligned}$$

de tal forma que los cuadriláteros quedan intercambiados y encajados correctamente.

Como ejemplo, podemos ver la figura 17. Puede observarse que el resultado es bastante bueno, y si se marcan bien las esquinas queda bastante bien. En la figura 18, podemos ver un ejemplo en el que el resultado no es tan bueno. Básicamente el problema es que la luminosidad es distinta en las distintas caras del cubo, por lo que el resultado queda raro. Además, como las piezas tienen relieve, es complicado saber dónde poner bien los puntos que marcan los vértices del cuadrilátero.

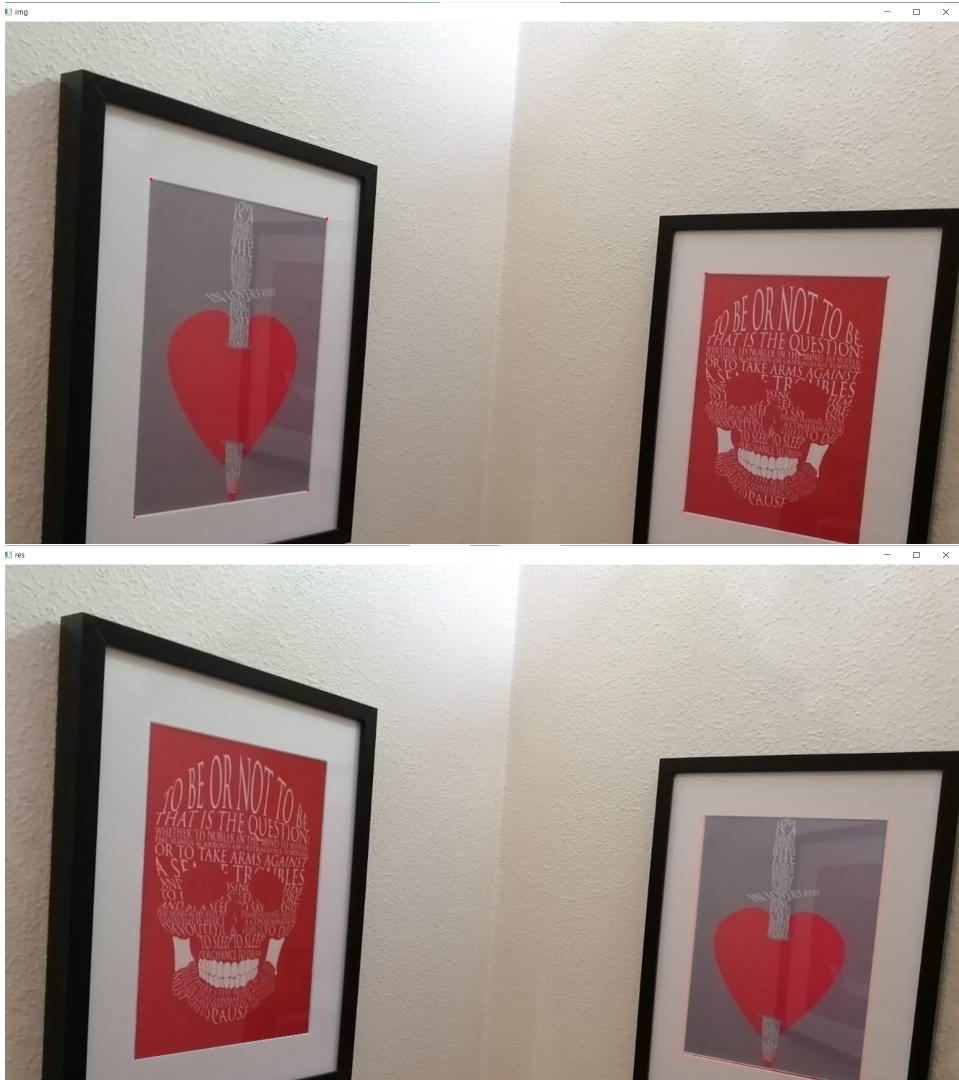


Figure 17: Ejemplo de *swap.py* para la imagen *cuadros.jpg*. Foto original (arriba) y resultado (abajo).

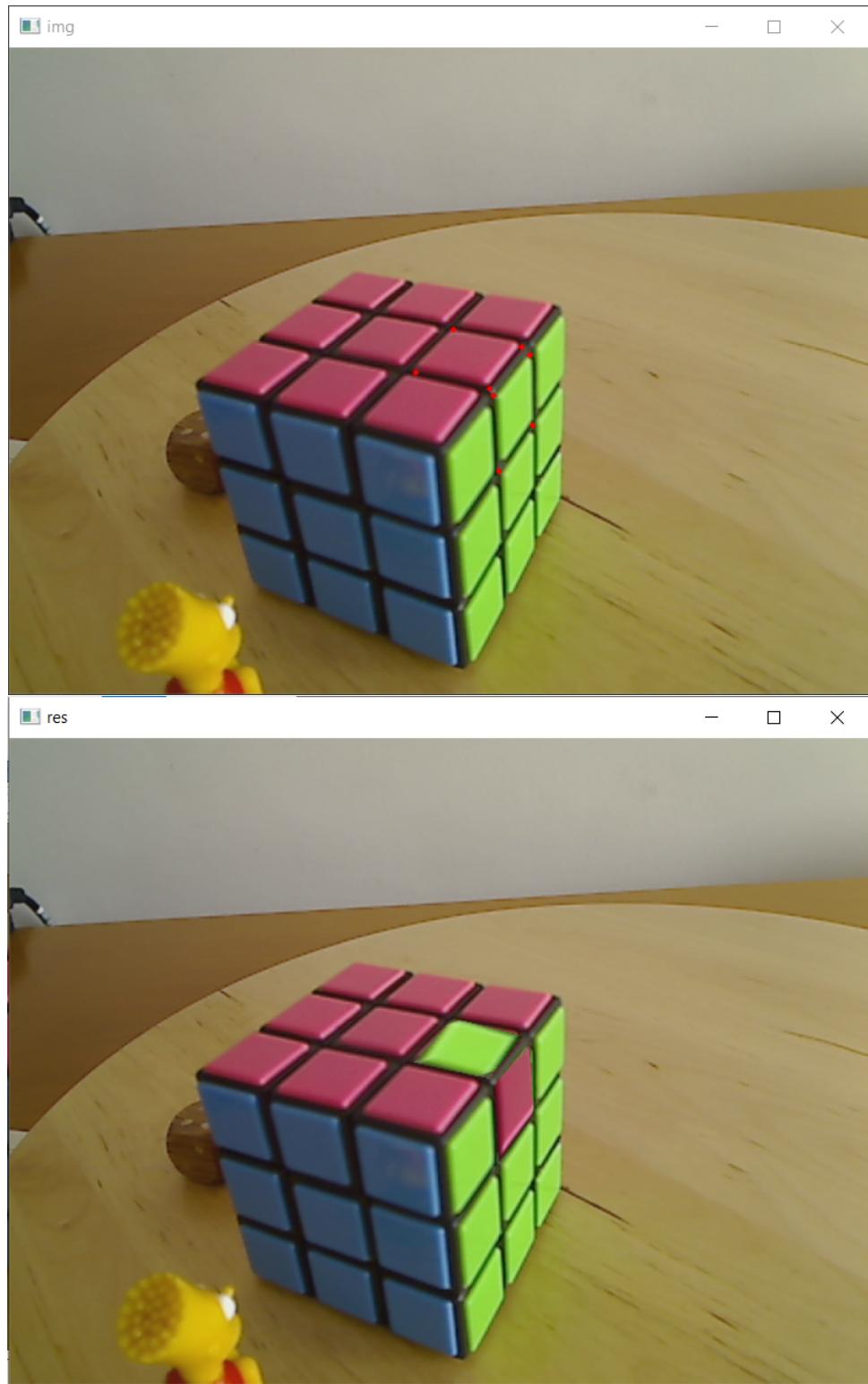


Figure 18: Ejemplo de *swap.py* para la imagen *cube4.png*. Foto original (arriba) y resultado (abajo).

## 10 Opcional 9: SUDOKU

### Enunciado

Haz un programa capaz de llenar con realidad aumentada un sudoku que se observa en una imagen en vivo.

### Solución

Tras ver diferentes fuentes, he desarrollado el siguiente proceso:

1. Detectamos el tablero, como el mayor contorno de la imagen.
2. Medimos su ancho y alto. Creamos un tablero artificial con estas dimensiones.
3. Buscamos la homografía entre el tablero artificial y el obtenido en la imagen. De esta forma rectificamos el tablero.
4. En este punto, debemos detectar las casillas del tablero. Tenemos, hasta donde yo sé, dos opciones:
  - (a) Como el tablero tiene dimensión 9x9, en principio podríamos dividir el ancho y el alto entre 9, e ir recortando trozos de la imagen con estas dimensiones. Esto presenta un **problema** de difícil solución. El problema es que es normal que el tablero esté doblado en alguna medida, porque se dobla el papel, por ejemplo. Además, las líneas del tablero suelen tener distintos tamaños, siendo más gruesas las de cada región de 3x3. Esto hace que las zonas no salgan exactamente como queremos, topándonos con partes de otras casillas en el trozo cogido. Esta opción la descarté porque no conseguía arreglar esto. Quizás con un análisis más profundo se pueda hacer así.
  - (b) Podemos detectar las líneas en el tablero rectificado y eliminarlas, poniéndolas completamente blancas. Esto hace que las casillas queden como un cuadrado grisáceo rodeado de blanco, por lo que podemos encontrarlos todos como contornos. Es una asunción razonable que el papel no se va a ver tan blanco como para ser indistinguible del blanco puro. Con este enfoque, tenemos una dificultad, y es que *findContours* devuelve los contornos en un orden indeseado. No se mantiene el orden que vemos en el tablero. Esto se puede arreglar ordenándolos a mano.
5. Una vez tenemos las casillas, cualquiera que sea la alternativa usada para esto, debemos detectar los números. Lo que he hecho yo es aplicar un blur a cada casilla, aumentando la diferencia entre el número y el fondo, si es que hay número. Tratamos de determinar ahora si hay algún número o no, para lo que aplicamos un threshold que trunca cualquier valor mayor de 110 a este valor. Es decir, si algo es suficientemente blanco, lo ponemos a 110. Ahora hacemos la media y todo lo que sea más oscuro que la media se pone negro, y lo que sea más claro se pone blanco. resizeamos a 28x28. Si la casilla tiene más de 30 píxeles negros, asumimos que tiene un número, si no, asumimos que está vacía. El valor de 30 ha sido obtenido mediante prueba y error. He visto que las casillas con número tienen más de 40 píxeles negros, y las vacías menos de 20.
6. Creamos la matriz que codificará el puzzle.
7. Para la casilla  $(i, j)$ :
  - (a) Si está vacía, ponemos un 0.
  - (b) Si no está vacía:

- i. Pasamos la casilla a un reconocedor de dígitos.
  - ii. Obtenemos el dígito que el reconocedor cree que es el más apropiado.
  - iii. Comprobamos que el dígito introducido es un valor posible del sudoku: no se repite en casilla, ni fila ni zona. Si se repite, comparamos las puntuaciones dadas por el reconocedor para las dos casillas. En la casilla con mayor puntuación, dejamos este dígito. En la otra ponemos el segundo dígito según la puntuación. **Nota:** el algoritmo programado no funciona si algo en el proceso anterior está muy mal, o si el sudoku de base tiene errores. No obstante, en condiciones normales debe ser suficiente.
8. Una vez llena la matriz, pasamos a la resolución del sudoku y a mostrarlo sobre el sudoku real.
9. La solución planteada es un backtracking con ramificación y poda:
- (a) Nos ponemos en la primera casilla vacía. O sea, la primera casilla con valor 0.
  - (b) Aumentamos el valor de la casilla actual, tomamos módulo 10.
    - i. Si el valor de la casilla actual es 0, retrocedemos a la casilla anterior (entre las que no están inicialmente llenas). Si estamos detrás de la primera casilla: el sudoku no tiene solución. Mi programa no contempla esta opción, asumo que el sudoku tiene solución.
    - ii. Si es distinto de 0, comprobamos que es compatible con el tablero actual. Si es compatible, avanzamos a la siguiente casilla vacía.
  - (c) Escribimos los números actuales en el tablero rectificado. Aplicamos la homografía inversa a este y lo colocamos sobre el original (como en *swap*). Así mostramos los números sobre el sudoku original.
  - (d) Si sobrepasamos la última casilla, hemos resuelto el sudoku. Si no, volver al paso (b).

He añadido otra forma de solucionar el sudoku, que es más rápida, pero solo muestra la solución final en pantalla.

Los resultados obtenidos no son óptimos, pero sí que estoy muy satisfecho, ya que he dedicado mucho tiempo a la resolución de este problema, consultando muchas fuentes distintas e introduciendo mis ideas propias a los problemas que he encontrado.

### Problemas:

- Es muy lento. Backtracking tiene que comprobar muchísimas posibilidades hasta encontrar la solución. Además, para cada dígito introducido, tenemos que ejecutar muchas operaciones.
- No funciona con la webcam, solo con un sudoku pasado como imagen. Con la webcam habría que calcular la homografía en cada frame, aumentando el ya elevado tiempo que tarda el programa en ejecutarse.
- Si la iluminación de la imagen es buena, no funciona bien.

Un ejemplo de funcionamiento es el siguiente. Ejecutamos `python sudoku.py --dev = dir : sudoku1.png`, que es el sudoku

Tras esto, pulsamos “a”. Veremos varias pantallas informativas. Una de ellas es esta

(0,0)	(0,1)	<b>1</b>	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)	(0,8)
(1,0)	(1,1)	<b>2</b>	(1,3)	<b>3</b>	(1,5)	(1,6)	(1,7)	<b>4</b>
(2,0)	(2,1)	(2,2)	<b>5</b>	(2,4)	(2,5)	<b>6</b>	(2,7)	(2,8)
<b>5</b>	(3,1)	(3,2)	(3,3)	<b>4</b>	(3,5)	(3,6)	(3,7)	(3,8)
(4,0)	<b>7</b>	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	<b>2</b>	(4,8)
(5,0)	(5,1)	(5,2)	(5,3)	<b>7</b>	<b>8</b>	(5,6)	(5,7)	<b>9</b>
<b>8</b>	(6,1)	<b>7</b>	(6,3)	(6,4)	<b>9</b>	(6,6)	(6,7)	(6,8)
<b>4</b>	(7,1)	(7,2)	(7,3)	<b>6</b>	(7,5)	<b>3</b>	(7,7)	(7,8)
(8,0)	(8,1)	(8,2)	(8,3)	(8,4)	(8,5)	<b>5</b>	(8,7)	(8,8)

en la que podemos ver las casillas detectadas y en qué orden han quedado.

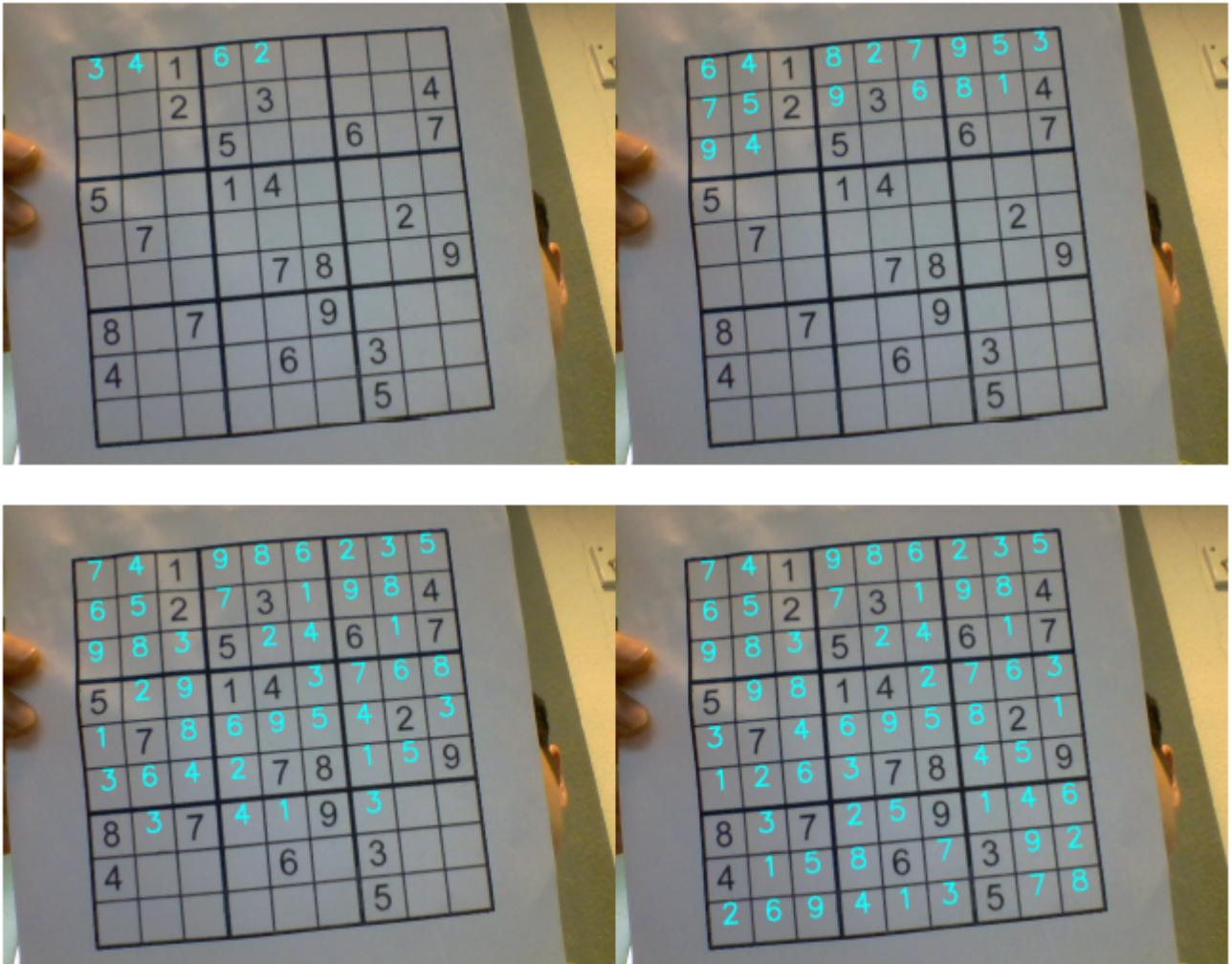
Otra es

$$\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & & 2 & 3 & & & 4 \\
 & & & & & 5 & 6 & 7 & \\
 & & & & 5 & 1 & 4 & & \\
 & & & & & 7 & & 2 & \\
 & & & & & & 7 & 8 & 9 \\
 & & & & 8 & 7 & 9 & & \\
 & & & & 4 & & 6 & 3 & \\
 & & & & & & 5 & & \\
 \end{array}$$

en la que vemos cómo queda cada casilla tras aplicar las distintas transformaciones indicadas. Sobre estas casillas haremos el reconocimiento de dígitos. También veremos el tablero rectificado

3	4	1	6	8	2	9		
			2	3			4	
				5		6	7	
5			1	4				
		7				2		
				7	8		9	
8		7		9				
4			6	3				
					5			

sobre el que se van escribiendo los números. Por último, veremos que en la imagen original se ponen también estos números sobre el tablero. Una secuencia de ejecución sería la siguiente:



La última imagen puede comprobarse que proporciona la solución al Sudoku. No obstante, ha tardado 36 horas en ejecutarse. Esto es claramente demasiado.  
 El código puede verse en el programa *sudoku.py*. En el mismo se encontrarán las distintas referencias y explicaciones detalladas de cada función.

## 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)
```

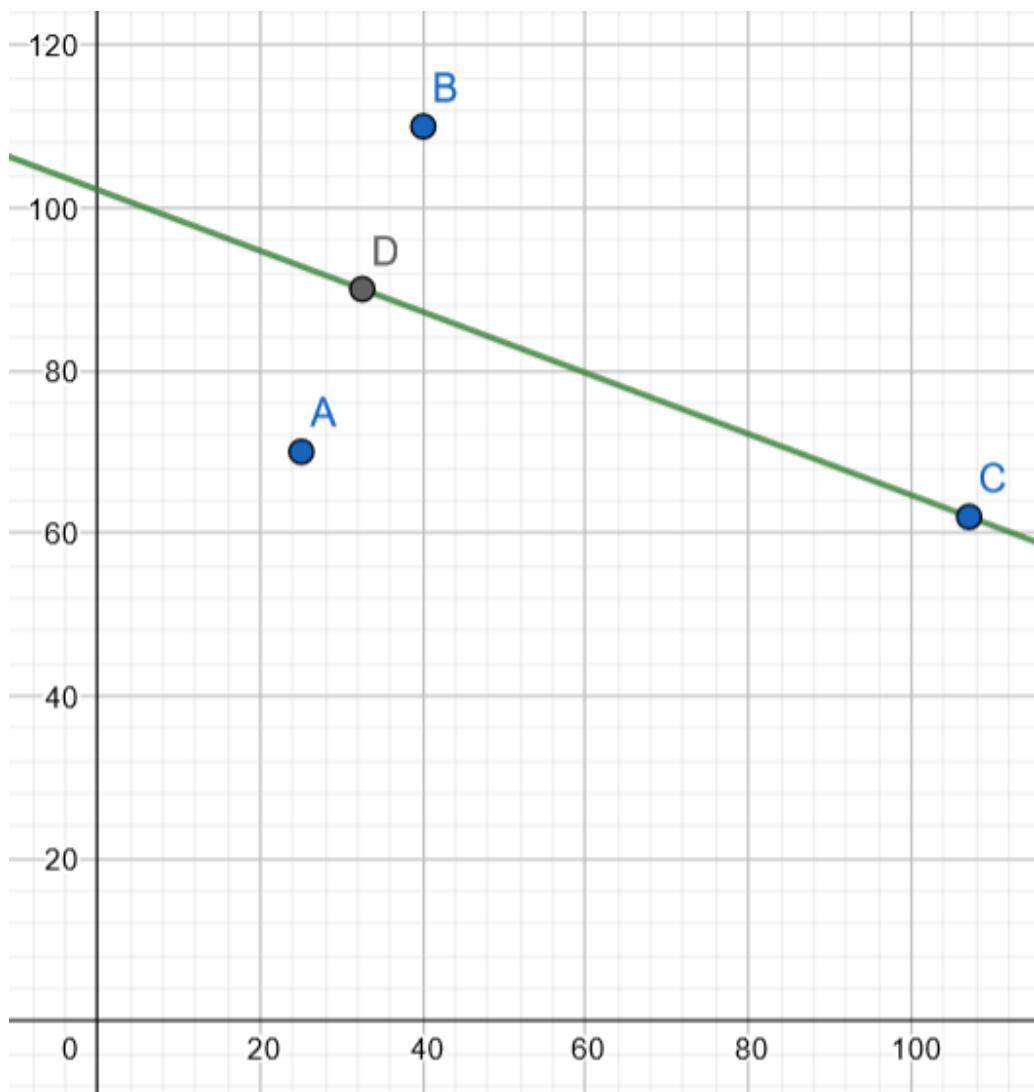
79.7166401939595

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)
```

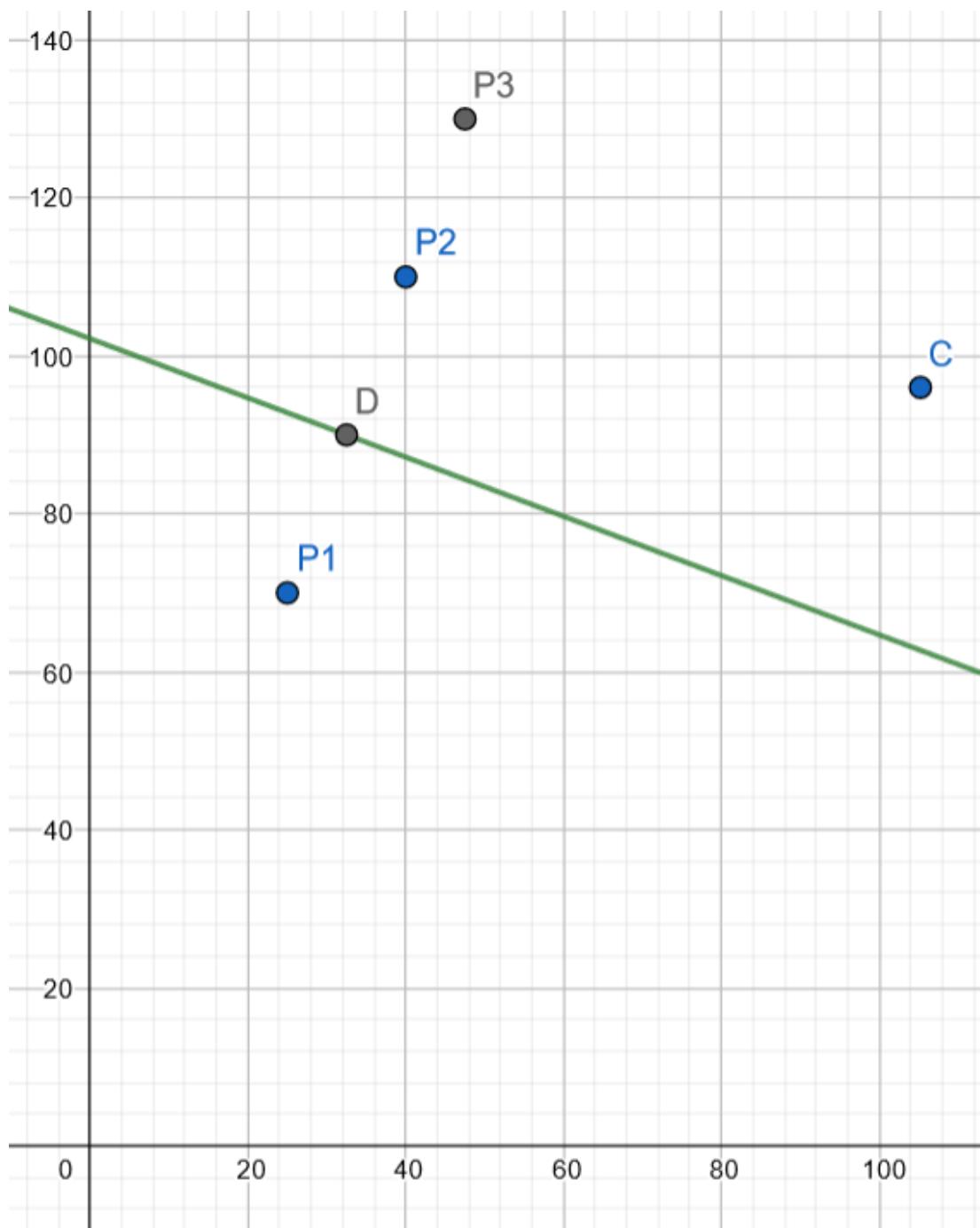
```
[52.86877085946814 array([-65.84446279])]
```

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

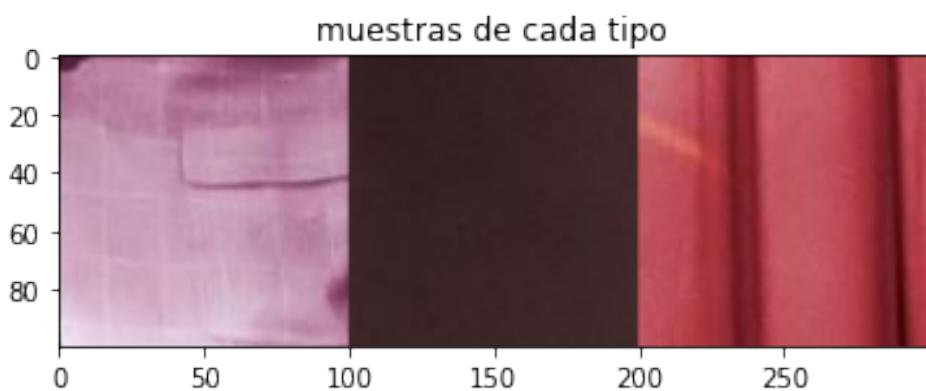
[ ]: img = readrgb("elvis.jpg")
      imshow(img); title('original');
      rows,cols,d = img.shape
      print(img.shape)
```

(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

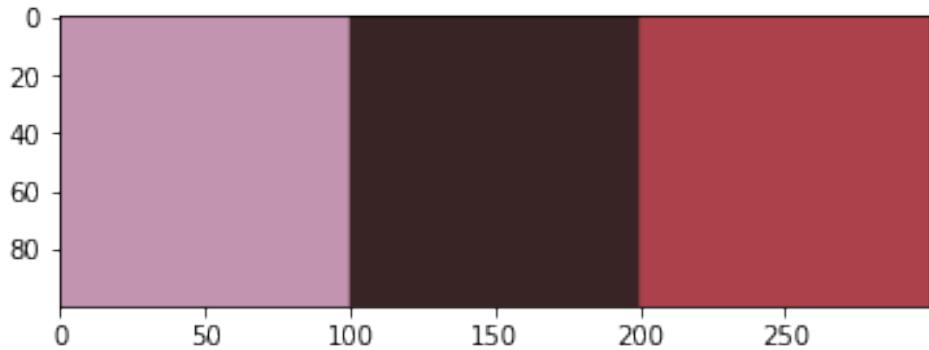
```
[ ]: med = [ np.mean(r,(0,1)) for r in models ]
muestras = []
```

```

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

imshow(np.hstack(muestras));

```

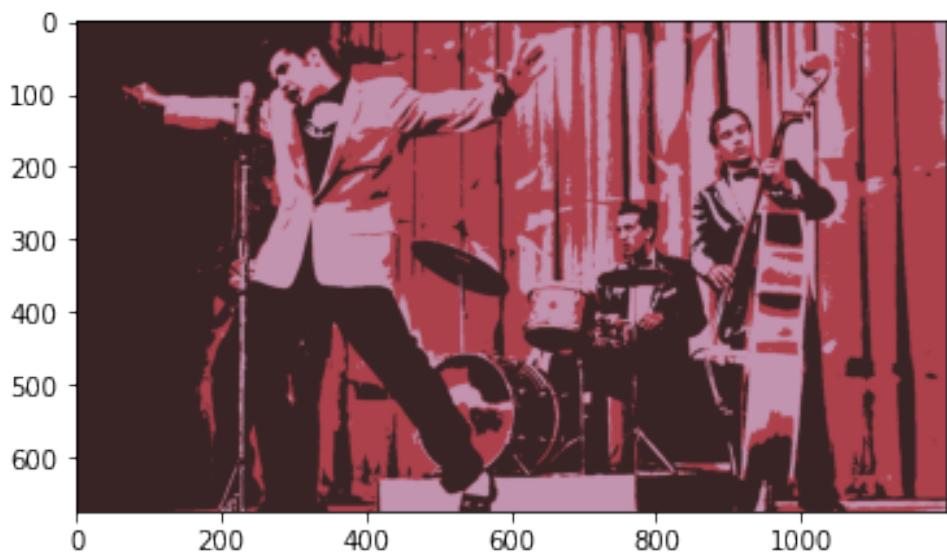


```

[ ]: 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]

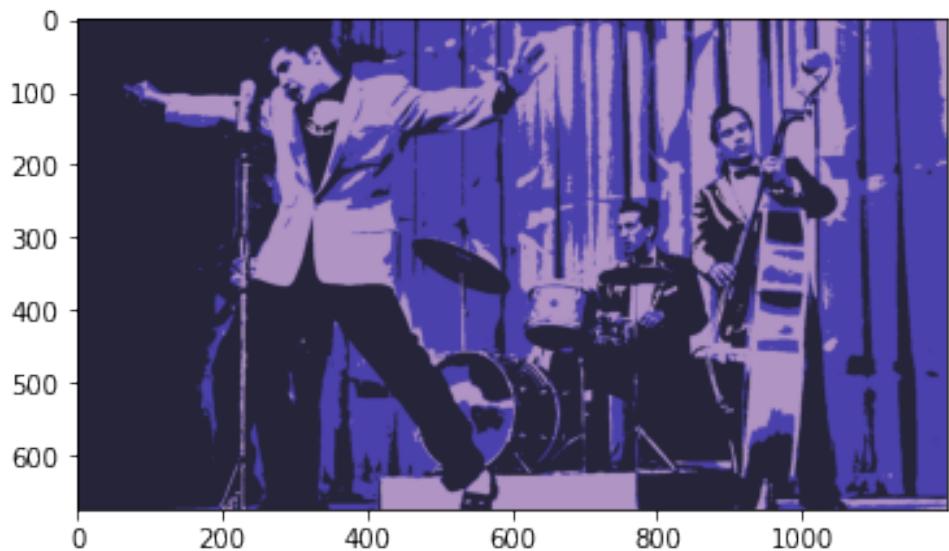
imshow(res);

```



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);
```



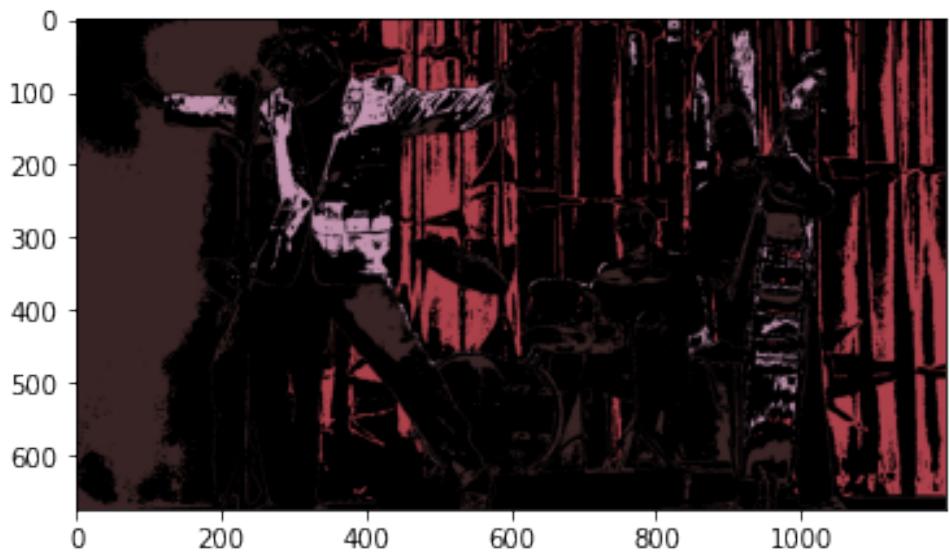
Entonces podríamos tomar, por ejemplo, tres tonos característicos de una imagen. Aplicar la reproyección tal y como hemos hecho, y cambiar los colores en función del resultado buscado.

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

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

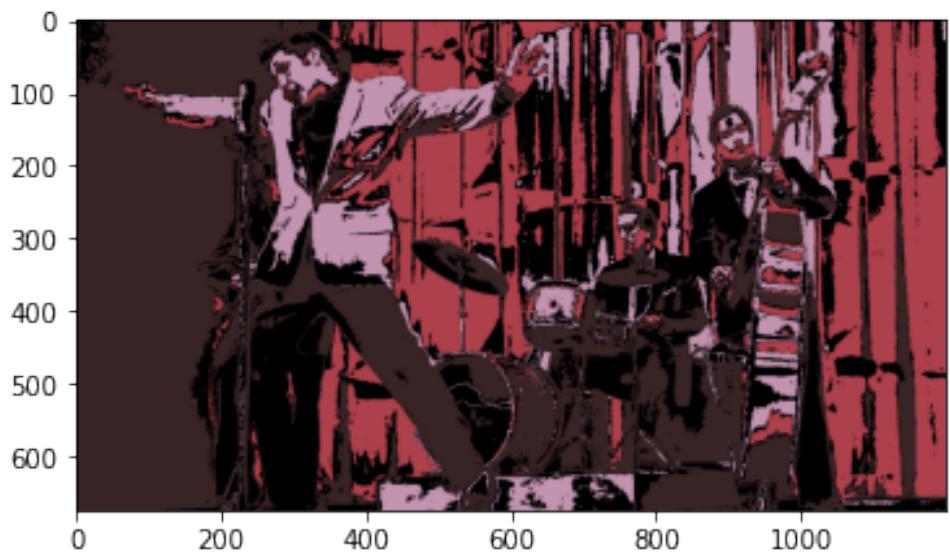
res [md > 40] = 0,0,0

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)
```

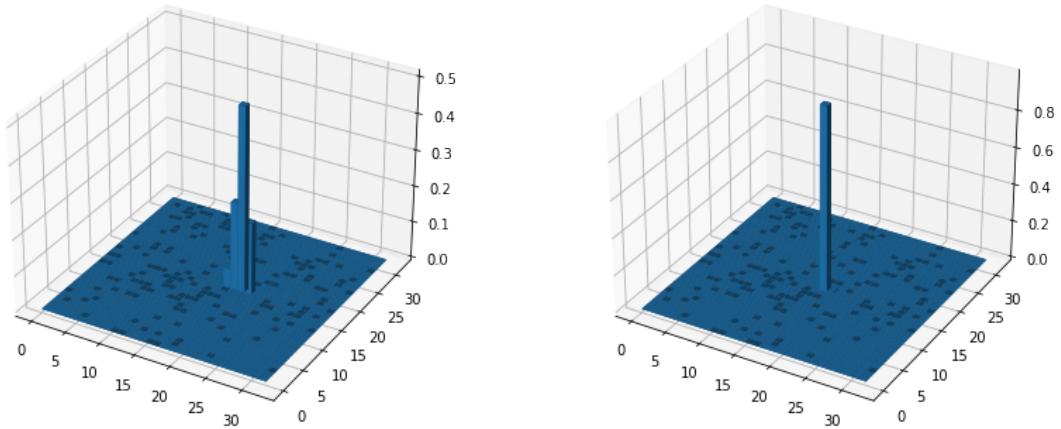
```
[ ]: hist = [uvh(r) for r in models]
```

```
[ ]: 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);

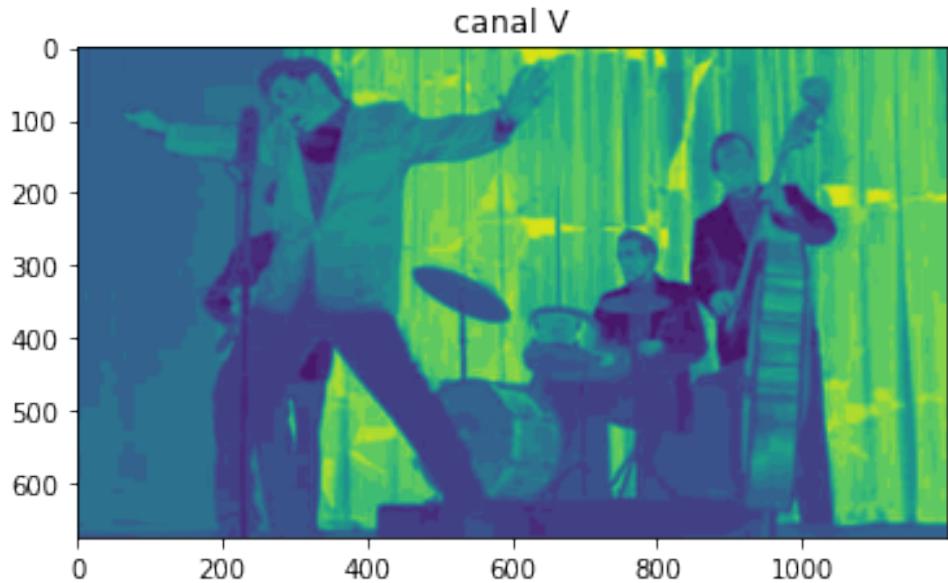
ax1 = fg.add_subplot(122, projection='3d')
top = hist[1].ravel()
ax1.bar3d(x, y, bottom, width, depth, top, shade=True);
```



```
[ ]: 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');
```



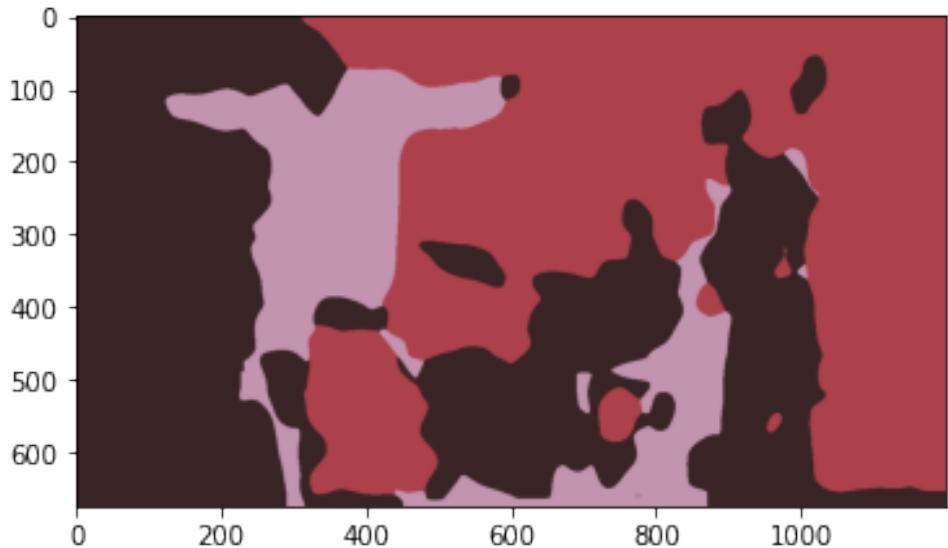
```
[ ]: u = uvr[:, :, 0]
v = uvr[:, :, 1]
```

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

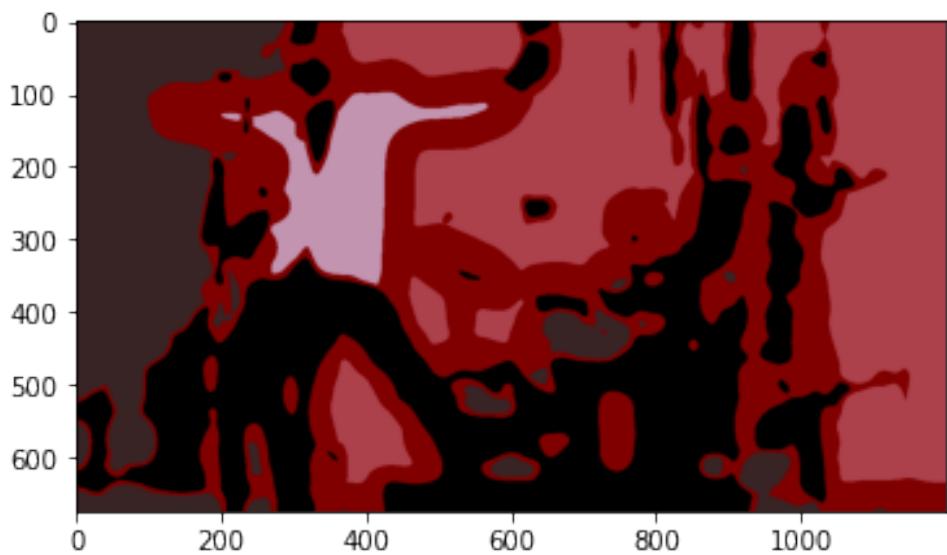
```
[ ]: E = np.sum(lik, axis=0)
p = np.array(lik) / E
c = np.argmax(p, axis=0)
mp = np.max(p, axis=0)
mp[E < 0.1] = 0
res = np.zeros(img.shape, np.uint8)
for k in range(len(models)):
    res[c==k] = med[k]

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. Ruiz 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. Ruiz 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)
- [7] IFAB, *Reglas de juego 21/22*, IFAB.
- [8] A. Ruiz Garcia, “camera.ipynb,” apuntes de la asignatura de vision artificial. Matriz de camara, busqueda de marcadores, realidad aumentada. [Online]. Available: <https://github.com/albertoruiz/umucv/blob/1cc00a490202aeb25af1af7277b8db0003e94e9b/notebooks/camera.ipynb>