

Universidad de Murcia

Grado en Ingeniería Informática

Visión Artificial

Ejercicios - Entrega Final

Javier Polo Gambín
javier.polog@um.es
48753080A
Grupo PCEO

Profesor: ALBERTO RUIZ GARCÍA

Índice

Índice de figuras	3
1 Introducción:	5
2 CALIBRACIÓN:	6
2.a Realiza una calibración precisa de tu cámara mediante múltiples imágenes de un <i>chessboard</i>	6
2.b Haz una calibración aproximada con un objeto de tamaño conocido y compara con el resultado anterior	7
2.b.1 Comparación de la calibración precisa y la calibración aproximada	8
2.c Determina a qué altura hay que poner la cámara para obtener una vista cenital completa de un campo de baloncesto	8
2.d Haz una aplicación para medir el ángulo que definen dos puntos marcados con el ratón en la imagen	10
2.e Opcional: determina la posición aproximada desde la que se ha tomado una foto a partir ángulos observados respecto a puntos de referencia conocidos.	10
3 ACTIVIDAD	20
3.a Utilizando un substractor de fondo de opencv como en los ejemplos <i>backsub0.py</i> y <i>backsub.py</i>	20
3.b Mediante un procedimiento sencillo que construya un modelo de fondo con frames anteriores y compare con el actual.	22
4 COLOR	25
4.a Construye un contador de objetos que tengan un color característico en la escena, simplemente pinchando con el ratón en dos o tres de ellos	25
5 FILTROS	27
5.a Comprueba la propiedad de “cascading” del filtro gaussiano	29
5.b Comprueba la propiedad de “separabilidad” del filtro gaussiano	31
5.c Implementa en Python desde cero (usando bucles) el algoritmo de convolución con una máscara general y compara su eficiencia con la versión de OpenCV	32
5.d Añade la posibilidad de seleccionar varios filtros para aplicarlos sucesivamente	33
6 SIFT	35
6.a 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.	35
7 RECTIF	38
7.a Rectifica la imagen de un plano para medir distancias (tomando manualmente referencias conocidas). Las coordenadas reales de los puntos de referencia y sus posiciones en la imagen deben pasarse como parámetro en un archivo de texto.	38
8 RA	41

8.a Crea un efecto de realidad aumentada en el que el usuario interactúe con los objetos virtuales	41
9 Opcional: VROT	47
9.a Determinar en qué dirección se mueve la cámara (UP,DOWN,LEFT,RIGHT, [FORWARD, BACKWARD])	47
9.b Estimar de forma aproximada la velocidad angular de rotación de la cámara (grados/segundo)	47
10 Opcional: SILU	48
10.a Amplía el reconocedor de siluetas para que admita múltiples prototipos. Intenta leer placas de matrícula vistas de frente.	48
11 Opcional: SWAP	51
11.a Intercambia dos cuadriláteros en una escena marcando manualmente los puntos de referencia.	51
12 Opcional: SUDOKU	52
12.a Detección del contorno exterior del sudoku	52
12.b Rectificación de perspectiva y detección de los contornos interiores del sudoku	53
12.c Identificación de los números	54
12.d Resolución del sudoku y muestra de resultados	56
13 Conclusión	59

Índice de figuras

1	Resultados de la calibración precisa.	6
2	Objeto de referencia para la calibración aproximada.	7
3	Resultados de la calibración aproximada.	8
4	Diagrama del ejercicio, pista de baloncesto y cámara con vista cenital	9
5	Ejecución del programa <i>pista.py</i> .	9
6	Ejemplo de ejecución en una imagen. Se seleccionan los dos puntos y se calcula el ángulo entre ellos.	10
7	Planteamiento del problema de localización.	11
8	Concepto de ángulo inscrito en una circunferencia.	12
9	Reducción del problema al plano bidimensional. Las proyecciones de los puntos A, B, C, P son A', B', C', P' respectivamente.	12
10	Cada par de puntos y el ángulo que forman con la cámara determinan una circunferencia.	13
11	Puntos marcados sobre la imagen.	15
12	Diagrama de posición estimada de la cámara respecto a los puntos de la imagen.	15
13	Resultados numéricos de la ejecución.	16
14	Ejemplo con puntos en distinta profundidad	17
15	Resultado de la ejecución	18
16	Resultados numéricos de la ejecución.	19
17	Detector de actividad con OpenCV. Antes de entrar al ROI.	21
18	Detector de actividad con OpenCV. En el ROI, detecta cambios.	21
19	Detector de actividad con OpenCV. Al salir del ROI.	22
20	Detector de actividad manual. Antes de entrar al ROI.	23
21	Detector de actividad manual. En el ROI, detecta cambios.	24
22	Detector de actividad manual. Al salir del ROI.	24
23	Imagen empleada para las pruebas del código.	25
24	Prueba 1. Seleccionamos las piezas de color morado oscuro.	26
25	Prueba 2. Seleccionamos las piezas de color marrón.	26
26	Ejemplo de ejecución con el filtro <i>gaussian</i> con $\sigma = 3$.	28
27	Ejemplo de ejecución con el filtro <i>maximum (max)</i> .	28
28	Imagen base para comprobar las propiedades del filtro gaussiano.	29
29	Resultados de aplicar el filtro gaussiano en cascading y en 1 solo paso.	30
30	Diferencias entre cascading y aplicación en 1 paso.	30
31	Aplicación de un filtro gaussiano primero en una dirección y luego en otra y aplicación del mismo filtro 2D.	31
32	Diferencias entre los dos métodos anteriores.	32
33	Comparación entre la convolución “manual” y la función de convolución de librería. Imagen de ejemplo.	33
34	Comparación entre la convolución “manual” y la función de convolución de librería. Imagen de gran tamaño.	33
35	Aplicación de varios filtros.	34
36	Imágenes empleadas para generar los modelos de las portadas de los libros.	36
37	Ejemplo de ejecución. Detecta correctamente de qué libro se trata y se muestran las coincidencias entre <i>keypoints</i> .	36

38	Coincidencias con cada uno de los modelos.	36
39	Otro ejemplo.	37
40	Otro ejemplo. Reconoce el libro en una imagen con más elementos.	37
41	Ejemplo RECTIF: Cálculo de distancia hasta la pelota.	39
42	Posición inicial, se pulsa “C” para poder definir el ROI. He definido el ROI donde se seguirá la punta del dedo.	42
43	Cuando el dedo se introduce en el ROI empieza el seguimiento. Al mover el dedo se va definiendo un camino.	42
44	Cuando el dedo se deja quieto, la ruta se fija (se vuelve verde y se transforma al plano del marcador).	43
45	Al pulsar la tecla “M”, el cubo empieza a moverse desde el primer punto de la ruta.	43
46	Movimiento siguiendo la ruta.	44
47	Movimiento siguiendo la ruta.	44
48	Pulsando la tecla “C” se puede ver la ruta. Si se vuelve a pulsar la tecla “M” se inicia de nuevo el movimiento.	45
49	Formato de una placa de matrícula en España.	48
50	Ejemplo SILU.	49
51	Modo avanzado del programa SILU.	50
52	Ejemplo de ejecución de SWAP.	51
53	Rectificación del sudoku una vez detectados los bordes.	53
54	Detección de contornos de las casillas y los números del sudoku.	54
55	Modelos empleados para el reconocimiento de números mediante análisis frecuencial.	55
56	Función para distinguir entre el número 6 y el número 9 a partir de sus contornos.	56
58	Sudoku resuelto tras deshacer la transformación homográfica de rectificación.	57
57	Sudoku resuelto.	57

1. Introducción:

En este informe se describe en detalle la resolución de los ejercicios de la entrega final de la asignatura Visión Artificial.

Cada ejercicio va acompañado de sus correspondientes imágenes como ejemplos de ejecución.

Los ficheros de cada ejercicio, tanto los códigos como las imágenes y vídeos de prueba se pueden encontrar en el directorio correspondiente a cada ejercicio, dentro del directorio */ejercicios*.

Se han añadido las explicaciones que se consideran oportunas, buscando un equilibrio entre los conceptos técnicos y rigurosos y el planteamiento intuitivo e ideas fundamentales detrás de cada ejercicio.

En cada momento se ha tratado de documentar las dificultades encontradas, aprendizaje adquirido, observaciones interesantes o curiosidades y similares.

En general, se trata de un estudio a fondo del contenido de la materia y su aplicación práctica a ejercicios más o menos realistas en situaciones variadas.

2. CALIBRACIÓN:

Observación: En este ejercicio es importante emplear una misma cámara en todos los apartados, para poder comparar resultados y aplicar la correlación entre los apartados. He decidido emplear la cámara de mi teléfono móvil.

2.a. Realiza una calibración precisa de tu cámara mediante múltiples imágenes de un *chessboard*.

Para generar las imágenes del *chessboard* he usado como base la imagen *chessboard.png* proporcionada en el material de la asignatura, tomando fotos del *chessboard* desde distintas posiciones y ángulos. La colección de imágenes resultante se puede encontrar en */Ejercicios/calibracion/fotoschessboard/*.

Una vez generadas las imágenes, he ejecutado el código *calibrate.py* pasándole como argumento dicha colección de imágenes para obtener una calibración precisa de la cámara. Esta calibración nos proporciona la matriz de cámara que denominaremos K y los coeficientes de distorsión. He modificado ligeramente el código para que estos valores se almacenen en los ficheros *camera_matrix.txt* y *dist_coefs.txt* para poder utilizarlos en los siguientes apartados.

Finalmente, se calculan la *distancia focal*, *FOV horizontal* y *FOV vertical* a partir de lo obtenido anteriormente.

```
camera matrix:
[[728.44463795  0.          522.65726783]
 [ 0.           727.36639158 381.88079057]
 [ 0.           0.           1.          ]]

distortion coefficients: [ 0.13624072 -0.70937847 -0.00271643
                           0.00482159  0.355815693]

Focal length: 728.444637949495 pix
```

```
FOV vertical:
70.20426802628951
FOV horizontal:
55.59196319541317
```

Figura 1: Resultados de la calibración precisa.

La *distancia focal* se corresponde con la entrada $K[0, 0]$ de la matriz de cámara. A partir de esto se calcula el FOV horizontal como

$$\text{FOV horizontal} = 2 \cdot \arctan\left(\frac{w}{2 \cdot f}\right)$$

Donde w = ancho de la imagen (píxeles) y f = distancia focal (píxeles).
y análogamente

$$\text{FOV vertical} = 2 \cdot \arctan\left(\frac{h}{2 \cdot f}\right)$$

2.b. Haz una calibración aproximada con un objeto de tamaño conocido y compara con el resultado anterior

Para este apartado he empleado como objeto de referencia una figura de tamaño conocido (una lámpara que mide 168cm de alto por 27cm de ancho (base)). La foto se ha realizado a 150cm del objeto, se muestra a continuación.



Figura 2: Objeto de referencia para la calibración aproximada.

Para realizar la calibración aproximada, empleamos el código de la herramienta *medidor.py* para obtener las medidas en píxeles de la figura referencia. He integrado la herramienta en el propio código de la función *calibrate_approx.py*. Con esto, calculamos los valores buscados:

$$f = \frac{I_h \cdot d}{R_h}$$

Donde: I_h = Altura del objeto en la imagen (*pixeles*)
 R_h = Altura del objeto en la realidad (*cm*)
 d = Distancia de la cámara al objeto *cm*

Y el FOV vertical y horizontal se calculan como antes.

Obtenemos los siguientes resultados:

```
Focal length: 675.0714484105987 pix
FOV vertical: 74.35614146178689°
FOV horizontal: 59.26492087713244°
```

Figura 3: Resultados de la calibración aproximada.

2.b.1. Comparación de la calibración precisa y la calibración aproximada

Vemos que, aunque los resultados no son iguales, sí que son muy similares. Los datos de la calibración aproximada están dentro de un rango del 30 % de error respecto a la calibración precisa, de forma que la consideraremos una aproximación aceptable.

2.c. Determina a qué altura hay que poner la cámara para obtener una vista cenital completa de un campo de baloncesto

Según el Consejo superior de deportes, las medidas de una pista de baloncesto reglamentaria en España es de $28 \times 15m$ [1]. Sabiendo esto, y con los datos de la cámara obtenidos de los apartados anteriores, calcularemos a qué altura h hay que colocar la cámara desde el centro de la pista para poder ver la pista en su totalidad.

En la Figura 4 se muestra un diagrama del ejercicio, donde α es el FOV vertical y β el FOV horizontal calculados en el ejercicio 2.a.

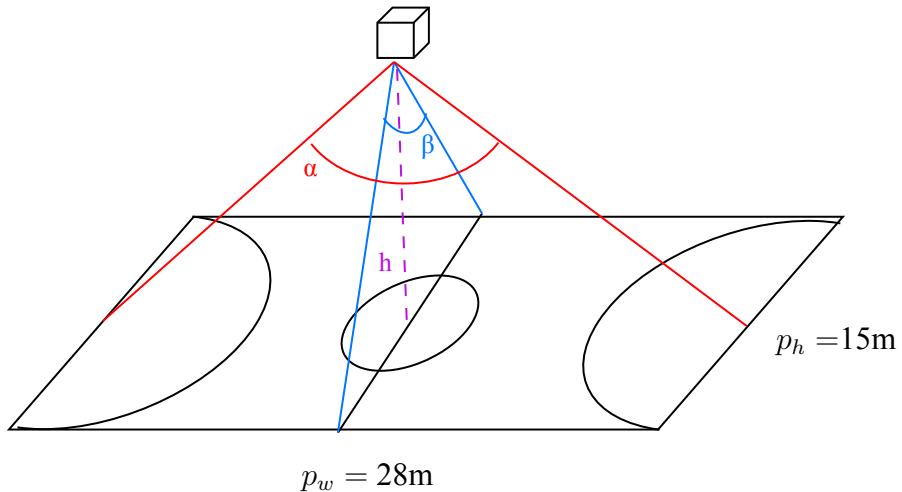


Figura 4: Diagrama del ejercicio, pista de baloncesto y cámara con vista cenital

Para calcular h obtenemos la altura h_1 del triángulo T_1 determinado por el ángulo α y el largo de la pista (en **rojo**), y la altura h_2 del triángulo T_2 , determinado por el ángulo β y el ancho de la pista (en **azul**).

Vemos cómo obtener h_1 , la otra es análoga:

$$h_1 = \frac{p_w}{2} \cdot \tan\left(\frac{\alpha}{2}\right)$$

La altura a la que habrá que colocar la cámara será:

$$h = \max(h_1, h_2) = 19,91m$$

Lo comprobamos con la ejecución del código *pista.py*:

```
Altura mínima en vertical: 14.227434334951074 m
Altura mínima en horizontal: 19.9184080689315 m
Altura mínima: 19.9184080689315 m
```

Figura 5: Ejecución del programa *pista.py*.

El resultado parece razonable, lo consideraremos bueno (teniendo en cuenta que la calibración de la que partimos no es perfecta, y pueden comentarse errores internos en los cálculos o aproximaciones de valores).

```
X:178, Y:365
X:399, Y:371
p1: [-334.           -19.           728.44463795], p2: [-113.           -13.           728.44463795]
Angle: 15.814658350707882°
```

Figura 6: Ejemplo de ejecución en una imagen. Se seleccionan los dos puntos y se calcula el ángulo entre ellos.

2.d. Haz una aplicación para medir el ángulo que definen dos puntos marcados con el ratón en la imagen

La idea del ejercicio es que, una vez obtenidos los parámetros de la cámara calculados en los apartados anteriores, podemos medir los ángulos de dos puntos en una imagen (espacio vectorial bidimensional) viéndolos como puntos en un espacio vectorial tridimensional basándonos en el modelo de cámara pinhole.

La cámara la suponemos situada en el centro de la imagen, con tercera coordenada 0:

$$C = (w/2, h/2, 0)$$

Vemos la imagen como un plano que se sitúa a una distancia f = distancia focal (px) de la cámara. Por tanto, los puntos de la imagen tienen coordenadas:

$$P = (w_p, h_p, f)$$

Ahora basta tomar los vectores que van desde la cámara hasta cada uno de los puntos y operando con el producto escalar obtenemos el ángulo entre los dos puntos.

Como observación, lo he implementado como función para poder utilizarlo en sucesivos ejercicios.

2.e. Opcional: determina la posición aproximada desde la que se ha tomado una foto a partir ángulos observados respecto a puntos de referencia conocidos.

El objetivo que se busca en este ejercicio es desarrollar un sistema rudimentario de localización. A partir de una imagen en la que se puedan observar puntos conocidos, queremos determinar desde qué punto fue tomada la imagen. Para ello necesitaremos conocer únicamente al menos tres puntos de referencia, las posiciones reales de dos de ellos y los parámetros intrínsecos de la cámara con la que se ha tomado la foto (trabajaremos con los datos de la matriz de cámara obtenidos en los apartados anteriores).

El planteamiento es el siguiente:

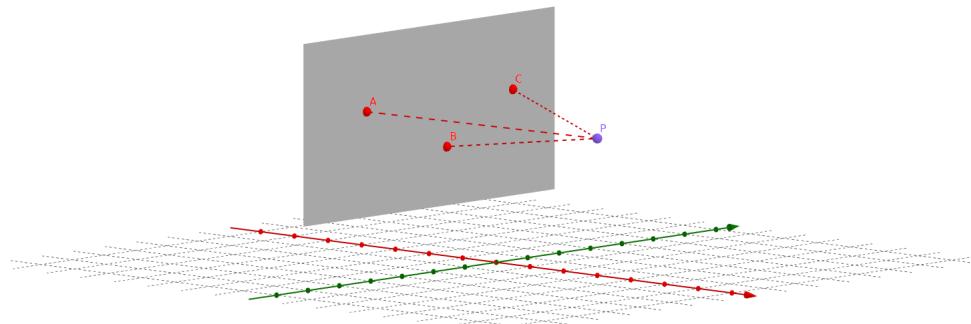


Figura 7: Planteamiento del problema en el que se emplea la representación basada en el modelo *Pinhole* de cámara. En imagen se han marcado los puntos conocidos A, B, C y el punto P en el que se sitúa la cámara (siguiendo el modelo, se han tomado como coordenadas de la cámara las del centro de proyección). Se toman los ejes X y Y habituales en la imagen, siendo el eje Z el de profundidad.

Vemos que se está empleando el modelo *pinhole* de cámara y que los puntos conocidos no están necesariamente alineados. La idea será emplear un método trigonométrico similar a la trilateración para determinar la posición del punto P .

Como observación inicial, nuestro objetivo será localizar las coordenadas del punto P desde el que se tomó la imagen en el plano horizontal, es decir, conoceremos su posición en el plano pero no su altura. Esto se debe a que emplearemos como concepto fundamental para la resolución del problema el **ángulo inscrito en una circunferencia** [8]. Este resultado solamente es aplicable en un espacio bidimensional. Para generalizar el resultado a 3 dimensiones deberíamos conocer más puntos y probablemente emplear conceptos de profundidad dentro de la imagen, por eso reduciremos el problema a 2 dimensiones. Supondremos sin pérdida de generalidad que la proyección de los 3 puntos se realiza sobre la recta del eje X , y el punto P se proyecta en la mitad negativa del eje Z , esto nos facilitará los cálculos más tarde.

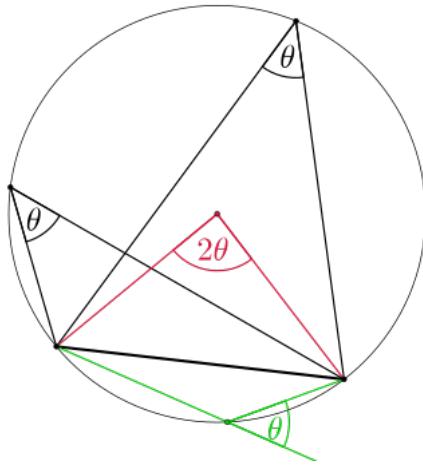


Figura 8: Concepto de ángulo inscrito en una circunferencia. Dos puntos y un ángulo θ entre ellos determinan una circunferencia. Cualquier punto arbitrario de la circunferencia formará el mismo ángulo θ tomando las cuerdas que lo unen con cada uno de los puntos iniciales. Fuente: Wikipedia.

Para reducir el problema a 2 dimensiones tomamos la proyección de los puntos sobre el eje Y , y obtenemos el nuevo problema:

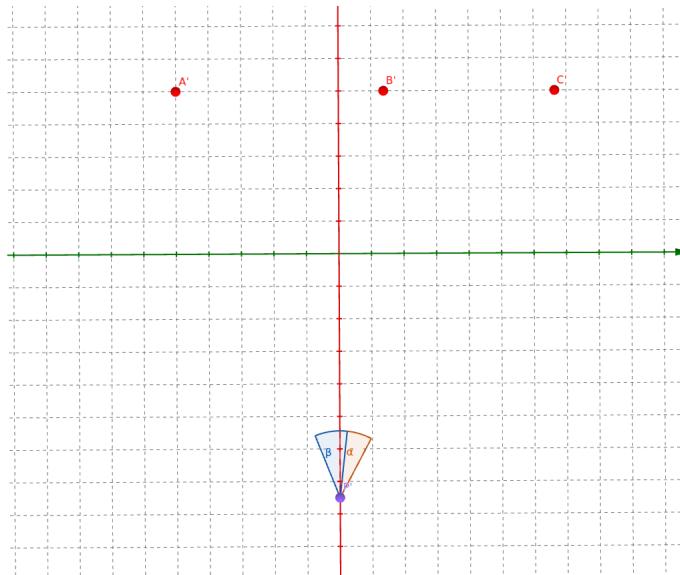


Figura 9: Reducción del problema al plano bidimensional. Las proyecciones de los puntos A, B, C, P son A', B', C', P' respectivamente.

En esta situación, aplicamos la utilidad *medidor.py* desarrollada en el apartado anterior (2.d) con algunas modificaciones para suprimir la dimensión sobrante. Con esto obtenemos 2 ángulos α entre los puntos A', P', B' y B', P', C' . Por medio del *teorema del ángulo inscrito* de Euclides [4], se concluye que existen 2 circunferencias C_1 y C_2 determinadas por cada par de puntos y el ángulo correspondiente. El ángulo entre el centro de cada circunferencia y los dos puntos será 2α y 2β respectivamente.

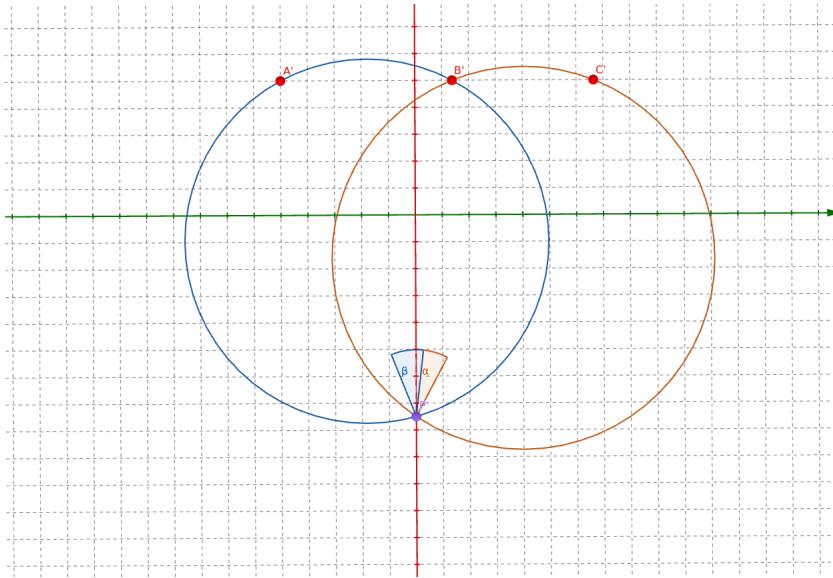


Figura 10: Cada par de puntos y el ángulo que forman con la cámara determinan una circunferencia.

La idea es calcular la intersección entre las circunferencias C_1 y C_2 para obtener las coordenadas del punto P' (realmente obtendremos 2 puntos de intersección, pero descartamos el que queda detrás del plano de la imagen, claramente no es el que buscamos).

Una vez realizada la aproximación intuitiva, explicaré los detalles del cálculo:

Para calcular una circunferencia C a partir de dos puntos P_1, P_2 y el ángulo θ del arco que forman en dicha circunferencia, emplearemos el siguiente método trigonométrico.

El primer paso será calcular el segmento $\overline{P_1P_2}$ que une ambos puntos (será la cuerda de la circunferencia) y su punto medio Q . El centro de la circunferencia, que llamaremos O , se encuentra en la recta perpendicular al segmento $\overline{P_1P_2}$ y contiene al punto Q .

Ahora queda determinar la altura a la que se encuentra el centro en la recta. Como se ha visto anteriormente, el ángulo formado entre el centro de la circunferencia y los dos puntos será 2θ . Consideramos el triángulo rectángulo

$$\widehat{P'_1OQ'}$$

Con ángulo θ entre P'_1, O, Q .

Un sencillo razonamiento por el *Teorema del seno* nos permite calcular la altura h del triángulo.

$$\begin{aligned} \frac{||\overline{QO}||}{\sin(\pi - \pi/2 - \theta)} &= \frac{||\overline{P_1Q}||}{\sin \theta} \\ \rightarrow h = ||\overline{QO}|| &= \frac{||\overline{P_1Q}||}{\sin \theta} \times \sin(\pi - \pi/2 - \theta) \end{aligned}$$

Y con esto se obtiene O :

$$O = Q - (0, h)$$

Conocido el centro de la circunferencia, podemos calcular el radio como la norma del vector que une el centro con uno de los puntos:

$$r = \|\overline{OP_1}\|$$

De esta manera se determina la circunferencia C .

Para calcular la intersección de dos circunferencias C_1 y C_2 :

Primero se comprueba si las circunferencias tienen intersección o no. En caso de que haya intersección, se calcula utilizando trigonometría.

Calculamos el punto medio M entre los centros de C_1 y C_2 y la distancia d entre el centro de C_1 y M . Después, consideramos el triángulo $\widehat{O_1, M, P}$ entre el centro de C_1 , M y el punto de intersección buscado P . Resolviendo el triángulo mediante el *Teorema de Pitágoras* obtenemos fácilmente el punto de intersección.

$$P = \sqrt{r_1^2 - d^2}$$

De hecho, al resolver la raíz cuadrada podemos tener 2 puntos de intersección (uno delante de la proyección de la imagen y otro detrás), nos quedamos con el que nos interesa. En este caso, como siempre que se corten en más de un punto, uno de ellos será el punto B en común entre ambos pares de puntos, tomaremos como solución el punto que no coincida con B .

Finalmente, mostramos un ejemplo de ejecución. Utilizaremos una imagen tomada con la misma cámara que en los apartados anteriores para poder utilizar la información que teníamos sobre la matriz de cámara:

En este ejemplo práctico marcamos los 3 puntos que se ven sobre la imagen, que se corresponden con puntos de referencia de los que podemos conocer la posición.

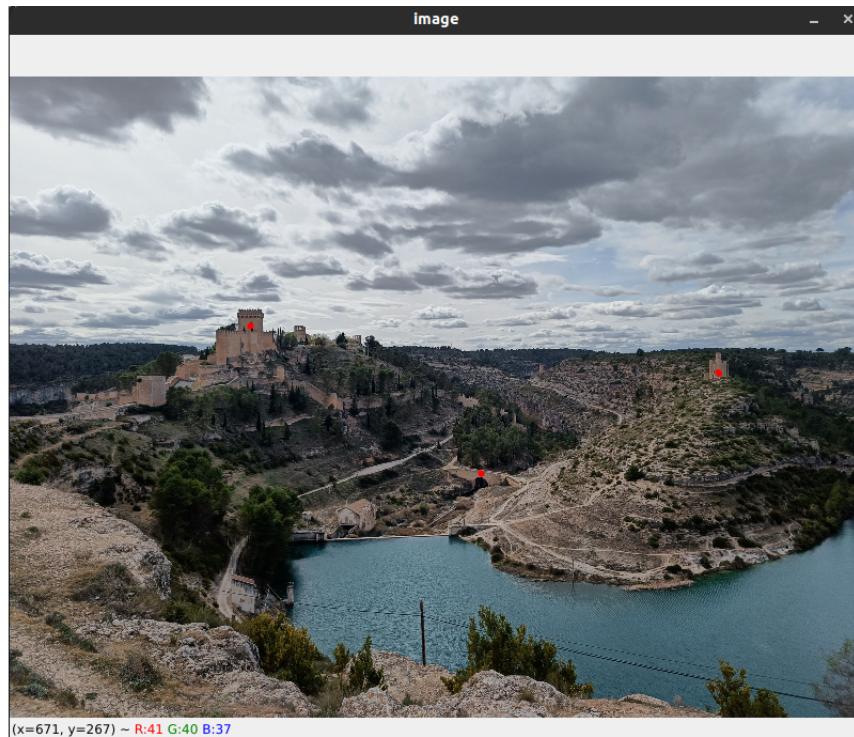


Figura 11: Puntos marcados sobre la imagen.

Y obtenemos los siguientes resultados:

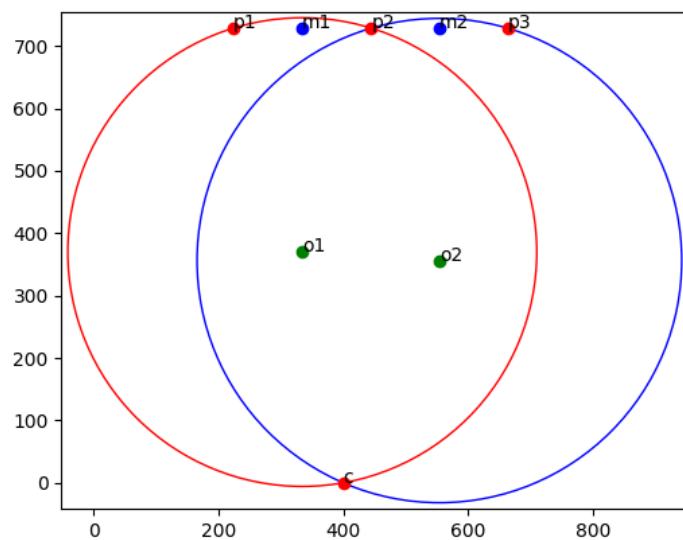


Figura 12: Diagrama de posición estimada de la cámara respecto a los puntos de la imagen.

Numéricamente, estos son los valores calculados:

Puntos marcados en la imagen

X:225, Y:233

X:441, Y:371

X:664, Y:275

Proyección sobre Y=0

p1: [225. 728.44463795]

p2: [441. 728.44463795]

p3: [664. 728.44463795]

Posición REAL de la cámara (se usa para comprobar)

cam: [399.9999999999982, 1.7053025658242404e-13]

Valores para calcular las circunferencias

m1: [333. 728.44463795]

m2: [552.5 728.44463795]

d1: 216.0

d2: 223.0

a1 (deg): 16.730074345973474

a2 (deg): 16.699873033460175

o1: [333. 369.14719564]

o2: [552.5 356.79279075]

circle1: (333.0, 369.1471956408709, 375.1781604111826)

circle2: (552.5, 356.7927907469957, 388.01719746556324)

Intersecciones

intersections: ([399.9999999999982, 1.7053025658242404e-13], [441.0, 728.444637949495])

Posición ESTIMADA de la cámara

camera position (c): [399.9999999999982, 1.7053025658242404e-13]

Figura 13: Resultados numéricos de la ejecución.

Vemos cómo a partir de los puntos marcados en la imagen y siguiendo el procedimiento explicado anteriormente, se calculan las 2 circunferencias determinadas por cada pareja de puntos $[p_1, p_2]$ y $[p_2, p_3]$ y los ángulos que estos forman con la cámara. El punto de intersección c coincide con el valor real de la posición de la cámara, por lo que hemos podido localizarla correctamente en el espacio de coordenadas en el que trabajamos.

Atendiendo al diagrama [7], podemos ver que este caso es relativamente sencillo puesto que los 3 puntos siempre se encuentran situados sobre el plano de imagen (como hemos dicho, no consideramos nociones de profundidad sobre la imagen en este caso) y por tanto, al reducir el problema a 2 dimensiones, siempre se encontrarán alineados. Eliminando esta restricción y ciñéndonos al plano bidimensional obtenemos un ejemplo más interesante:

Nos encontramos en la siguiente situación:

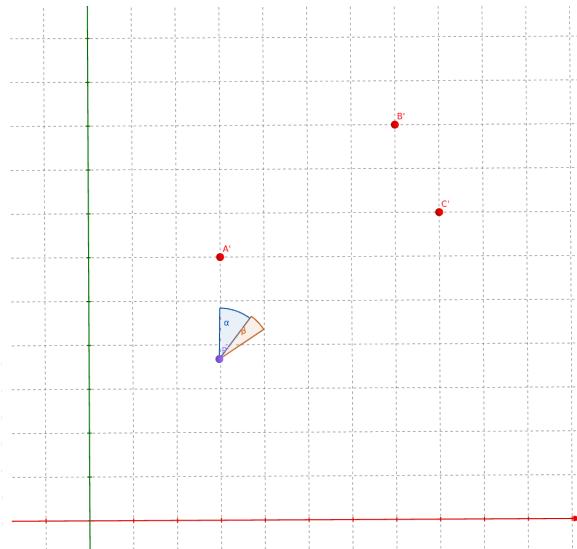


Figura 14: Ejemplo con puntos en distinta profundidad. Coordenadas: $A = (15, 30)$, $B = (35, 45)$, $C = (40, 35)$, $P = (14, 85, 18, 37)$, $\alpha = 36,39$, $\beta = 19,42$.

En este caso, no es posible alinear los 3 puntos A, B, C , de forma que habrá que tener en cuenta consideraciones extra.

El procedimiento es igual al explicado anteriormente, pero para calcular el centro de cada circunferencia no basta desplazarse desde el punto medio de los dos puntos que la determinan “hacia abajo”, sino que debemos desplazarnos en la dirección de la recta perpendicular que une los dos puntos. Para ello calculamos el ángulo de esta recta y el eje X , sumamos $\pi/2$ radianes para obtener el perpendicular y en esa dirección (tomando coseno y seno en cada coordenada) avanzamos la distancia al centro de la circunferencia, calculada como antes (para más detalles sobre esto consultar el código [/ejercicios/CALIBRACION/localizar/locateCam.py](#), no quería sobrecargar el informe con otra explicación geométrica sencilla).

Es importante destacar que lo que estamos haciendo es generalizar el método original para un caso en el que los 3 puntos no se encuentren alineados. El ejemplo anterior podría resolverse apli-

cando este método y los resultados serían equivalentes.

De esta forma, se calculan las dos circunferencias como antes y obtenemos el siguiente resultado:

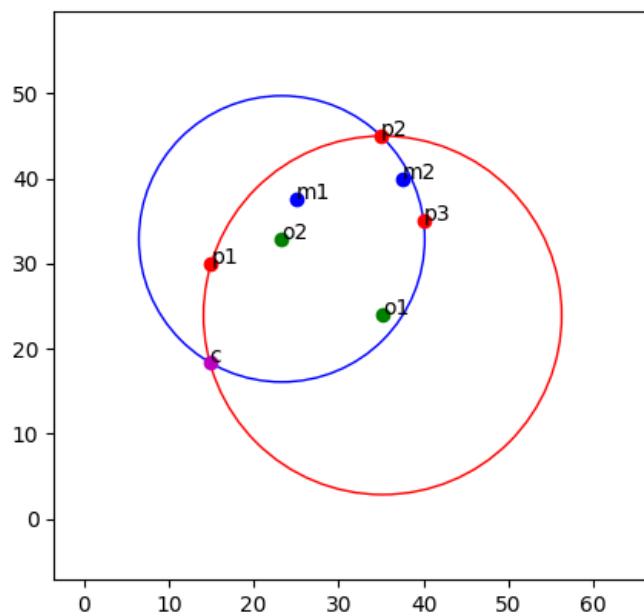


Figura 15: Resultado de la ejecución. Los puntos p_1, p_2, p_3 son los puntos introducidos en el programa. El punto C es la posición aproximada de la cámara que se calcula.

```

p1: [15. 30.]
p2: [35. 45.]
p3: [40. 35.]
c: [14.85071216386631, 18.383242261722025]
m1: [25. 37.5]
m2: [37.5 40. ]
d1: 25.0
d2: 11.180339887498949
a1 (deg): 36.39
a2 (deg): 19.419999999999998
o1: [35.17647061 23.93137252]
o2: [23.3175338 32.9087669]
circle1: (35.17647061020175, 23.931372519731, 21.069366525327975)
circle2: (23.317533796373866, 32.90876689818693, 16.81302871350807)
intersections: ([35.0, 45.0], [14.85071216386631, 18.383242261722025])
camera position: [14.85071216386631, 18.383242261722025]

```

Figura 16: Resultados numéricos de la ejecución.

La localización aproximada de la cámara que se ha calculado coincide casi exactamente con la posición real, de manera que el funcionamiento es correcto.

Este último ejemplo podría ser un caso realista si disponemos, por ejemplo, de un método para estimar la profundidad en una imagen. Podríamos utilizar entonces este segundo método para obtener el resultado de forma más precisa. Por esta razón consideraba interesante desarrollarlo.

Finalmente, algunos detalles de implementación:

He implementado funciones para calcular cada uno de los valores necesarios para realizar los cálculos. Para ello me he basado en la explicación trigonométrica que he dado anteriormente, la implementación no supone ninguna dificultad más allá de aplicar las fórmulas correctamente. La función para detectar la intersección de dos circunferencias es sencilla y no la explicaré puesto que no es relevante.

Como he explicado anteriormente, este ejercicio se puede realizar de dos formas, dependiendo de si consideramos o no la noción de profundidad. Por esta razón en el código he implementado dos modos de funcionamiento, en función de los argumentos que se le suministren al programa.

- En el primero, sin profundidad, se le suministra como argumento al programa la ruta de un fichero de imagen. Permitirá marcar los puntos en ella para obtener los resultados. En este modo se utiliza la información de la matriz de cámara (concretamente la distancia focal) para poder calcular los ángulos que forman los puntos con la cámara. *ejemplo: locateCam.py /path/to/image*
- En el segundo, el programa recibe como argumentos los pares de coordenadas de cada punto y los dos ángulos que forman con respecto a la cámara. En este modo no necesitamos información sobre la matriz de cámara porque ya conocemos los ángulos.
ejemplo: locateCamp.py x1,y1 x2,y2 x3,y3 a1 a2

3. ACTIVIDAD

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. Muestra el objeto seleccionado anulando el fondo. Implementalo de dos maneras.

El objetivo de los programas es implementar un detector de movimiento sencillo. Una vez iniciado el programa, se deja funcionando un par de segundos y se selecciona una región de la imagen en la que se quiere detectar el movimiento. A partir de este momento, cuando se detecte “movimiento” en la región seleccionada, se empieza a grabar un vídeo mientras persista el movimiento, y hasta un par de segundos después. Como añadido, también se incluye en el vídeo el segundo anterior al inicio del movimiento, que tenemos almacenado en un buffer de frames, para poder apreciar bien toda la acción.

Para detectar el “movimiento”, extraemos el fondo de la imagen (entendiéndolo como lo que permanece estático, que no cambia) y obtenemos el frame donde el fondo toma el valor 0. Con esto, calculamos una máscara con el complementario (lo que no es fondo), esto es lo que varía en la imagen, el “movimiento”.

Para afinar la detección y evitar los “falsos positivos” (por ejemplo, por variaciones puntuales de luz o por movimientos de autoenfoque de la cámara, ambas situaciones con las que me he encontrado haciendo pruebas), solo consideraremos movimiento si el cambio detectado es superior al 1 % del área del ROI.

Los dos apartados siguientes tienen la misma funcionalidad, solamente difieren en la manera en la que se realiza la “extracción del fondo”.

Como observación, estamos considerando una fuente de vídeo con un framerate de 60 fps, y mantenemos un buffer de frames cuya longitud es $3 \cdot 60\text{fps} = 180$ frames, que nos permite almacenar 3 segundos de vídeo.

3.a. Utilizando un substractor de fondo de opencv como en los ejemplos *backsub0.py* y *backsub.py*

El código sigue el esquema que se ha explicado, y se emplea la función de opencv, *cv.createBackgroundSubtractorMOG2()*, para crear un extractor de fondo (*bg_substractor*). Posteriormente, una vez que tenemos definido el ROI, se aplica el extractor de fondo *bg_substractor.apply()* para eliminar el fondo del ROI, y emplearlo como una máscara para tomar los píxeles distintos del fondo en el ROI.

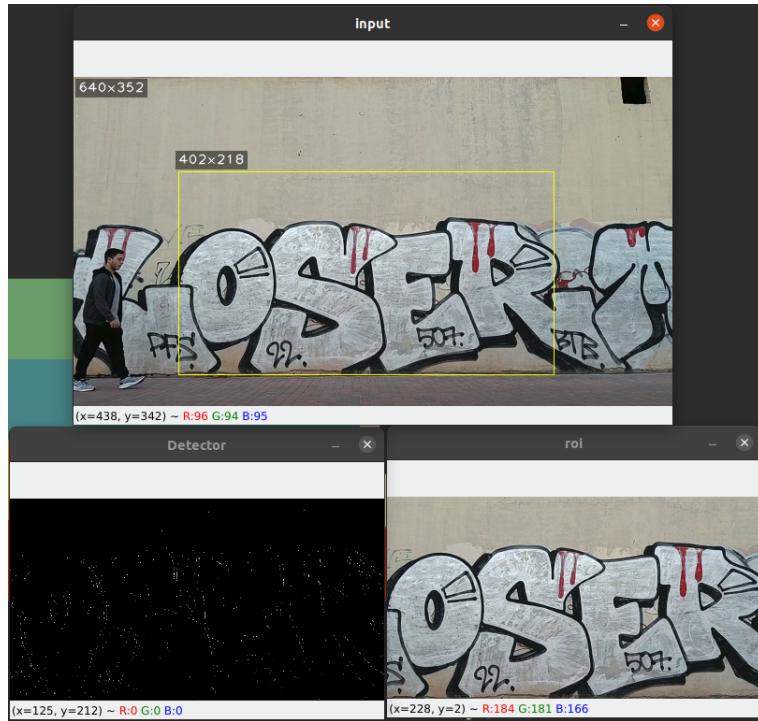


Figura 17: Detector de actividad con OpenCV. Antes de entrar al ROI.

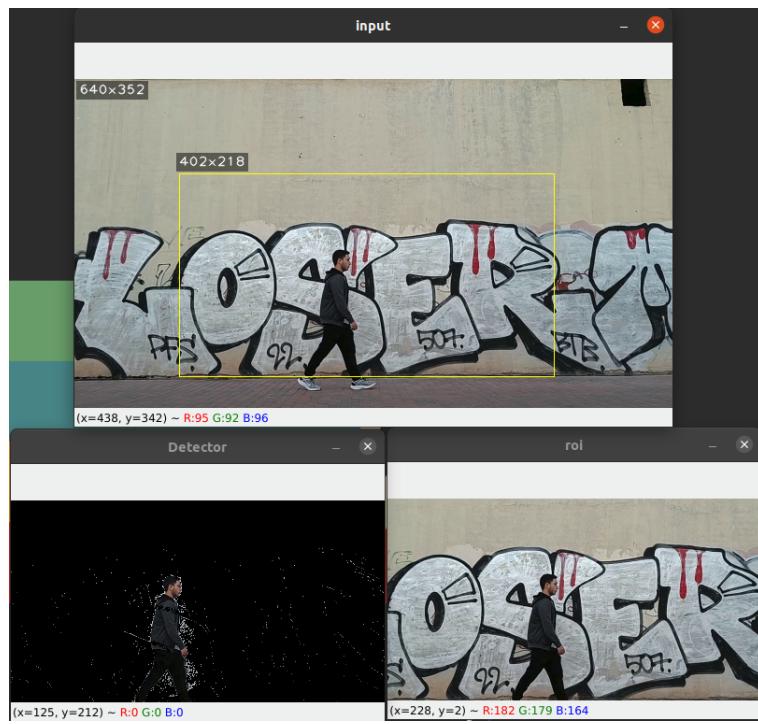


Figura 18: Detector de actividad con OpenCV. En el ROI, detecta cambios.

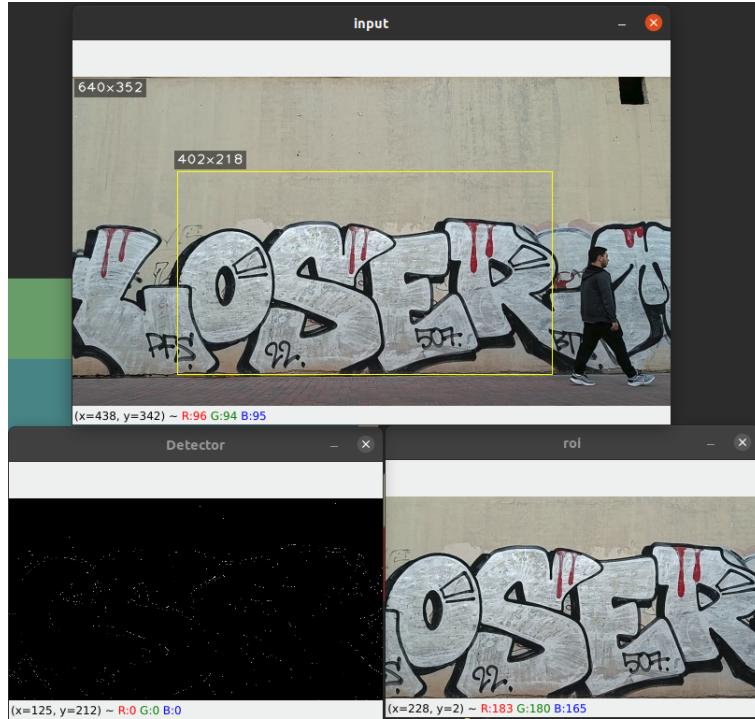


Figura 19: Detector de actividad con OpenCV. Al salir del ROI.

Vemos que los resultados son bastante buenos en condiciones buenas y estables de luz, movimiento, número de elementos en el fondo, etc.

Como observación, encontré ciertas dificultades a la hora de crear el fichero de vídeo con los frames almacenados empleando la clase *Video* de la librería *umucv.util*, puesto que únicamente se grababa en el fichero 1 de cada 2 frames. No conseguí solucionar el problema, así que implementé la mecánica de creación y escritura del fichero de vídeo directamente con las funciones de *Open cv.VideoCapture_fourcc()* para escoger el codec y *cv.VideoWriter()* para escribir los frames al fichero.

3.b. Mediante un procedimiento sencillo que construya un modelo de fondo con frames anteriores y compare con el actual.

Nuevamente se sigue el esquema general explicado antes, pero ahora además del buffer de frames para almacenar la grabación, mantenemos un buffer de frames en escala de grises *listFrames_gray* que nos permitirá construir un modelo del fondo y comparar con el frame actual para detectar los cambios. Aunque el buffer almacena $3 \cdot \text{fps}$, para computar el fondo emplearemos únicamente los últimos de la lista (los más recientes, que se acaban de insertar).

La manera de hacer esto es sencilla: cogemos el frame anterior del buffer (*old_roi_frame*), lo comparamos con el frame actual en la región del ROI y calculamos la diferencia absoluta (*cv.absdiff*) entre ambos como una máscara. Finalmente aplicamos esta máscara al frame actual *roi_frame* y obtenemos qué parte del mismo no es fondo (es “movimiento”, “actividad”).

Además, puesto que almacenamos un buffer con varios frames, he considerado que una manera de refinar la detección de actividad es no solo calcular las diferencias entre el frame actual y el frame anterior (el instante inmediatamente anterior), sino comparar con varios de los frames anteriores (5 frames anteriores, es decir, varios instantes antes del frame actual), ponderando más a los frames más alejados. De esta manera estamos dando más importancia a los píxeles que han cambiado a lo largo de varios instantes, puesto que probablemente se trate de píxeles de fondo, en los que de repente sucede una variación que dura varios frames. Este método lo he obtenido de forma empírica, a base de probar con diferente número de frames y diferentes ponderaciones, y me he quedado con uno que considero ofrece los mejores resultados con una cantidad de cálculos razonable. Con esto consigo no marcar como actividad los movimientos tan rápidos que solamente duran un par de frames (al tener en cuenta varios, las oscilaciones puntuales se descartan porque las diferencias se suavizan), y al ponderar más los frames más lejanos, se detecta mejor los objetos que se mueven más lentamente (de lo contrario, un movimiento muy lento haría que dos frames consecutivos comparten muchos píxeles y la diferencia fuera casi nula).

A continuación se muestran los resultados:

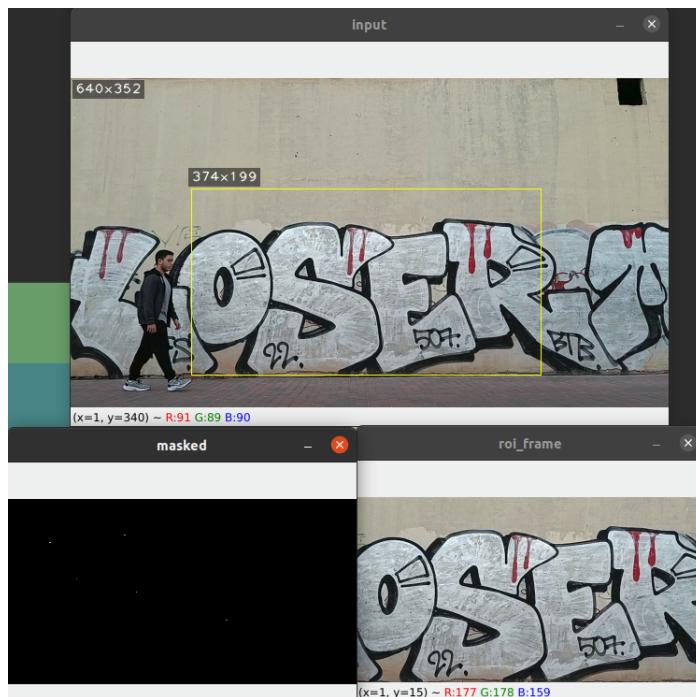


Figura 20: Detector de actividad manual. Antes de entrar al ROI.

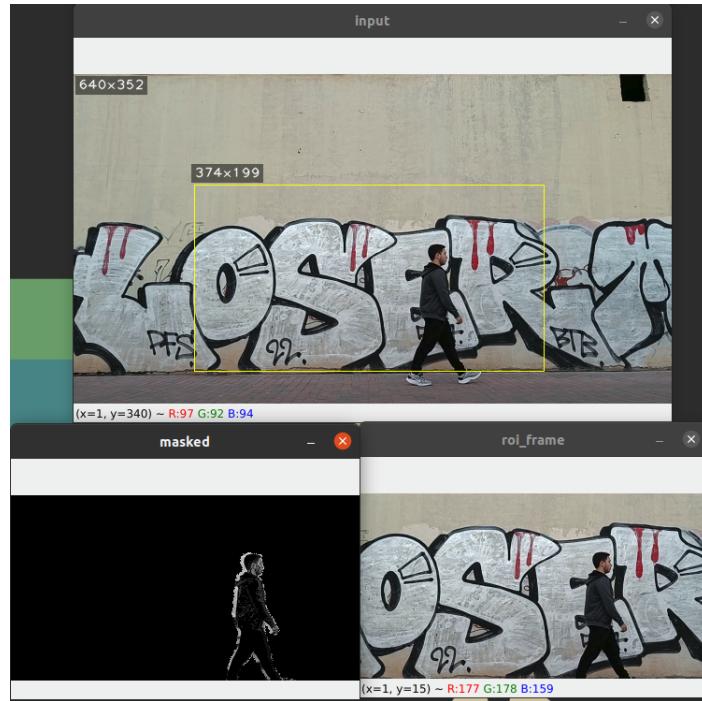


Figura 21: Detector de actividad manual. En el ROI, detecta cambios.

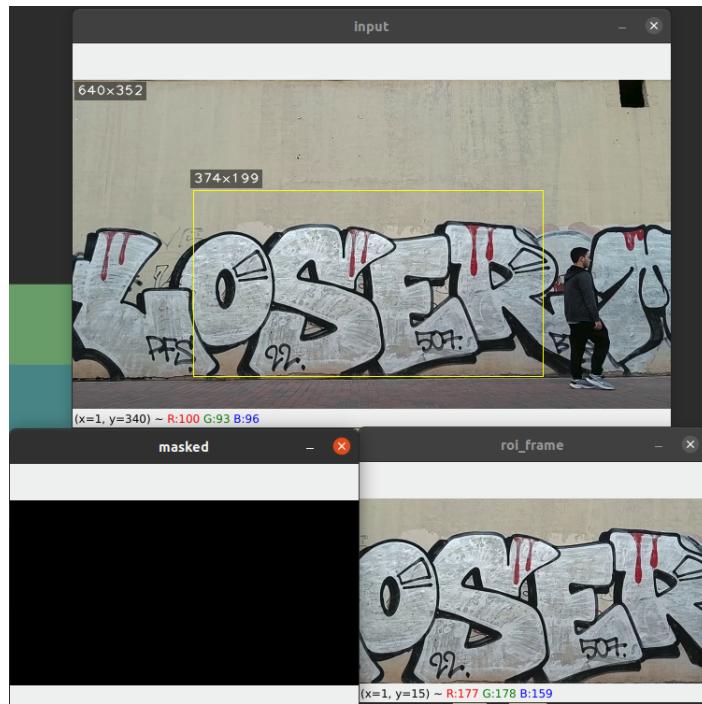


Figura 22: Detector de actividad manual. Al salir del ROI.

Vemos que el resultado es ciertamente peor que el obtenido con la función de librería de *OpenCV*, pero podemos considerarlo adecuado si se cumplen las condiciones favorables explicadas anteriormente.

4. COLOR

4.a. Construye un contador de objetos que tengan un color característico en la escena, simplemente pinchando con el ratón en dos o tres de ellos

Primero se calcula el color medio de los píxeles seleccionados. Para procesar la imagen, convertimos tanto la propia imagen como el color medio al espacio de color **HSV** (más adecuado para realizar cálculos con el color).

La idea es obtener un rango de color a partir del color medio seleccionado, en el que se encuentren todos los objetos del mismo color en la imagen. Para ajustar este rango y contar de forma precisa, el usuario dispone de *trackbars* para ajustar los valores *Hue*, *Saturation* y *Value*. Una vez obtenido el rango, se calcula una máscara de los píxeles de la imagen original que entran dentro de ese rango de color.

A continuación, se extraen los contornos (*extractContours()*, que hace uso de la función *cv.findContours* de la librería de *OpenCV*) y se aplican varios criterios vistos en clase para quedarnos únicamente con los contornos razonables (eliminamos contornos demasiado pequeños, y tenemos en cuenta la orientación en la que se recorre cada contorno).

Contando el número de contornos obtenemos el número de objetos del color seleccionado que hay en la imagen.

A continuación se muestran algunas pruebas de ejecución (el número de objetos contados aparece en la parte del centro abajo. Se muestra por terminal, pero lo he situado en esa posición para poder hacer bien las capturas).



Figura 23: Imagen empleada para las pruebas del código.

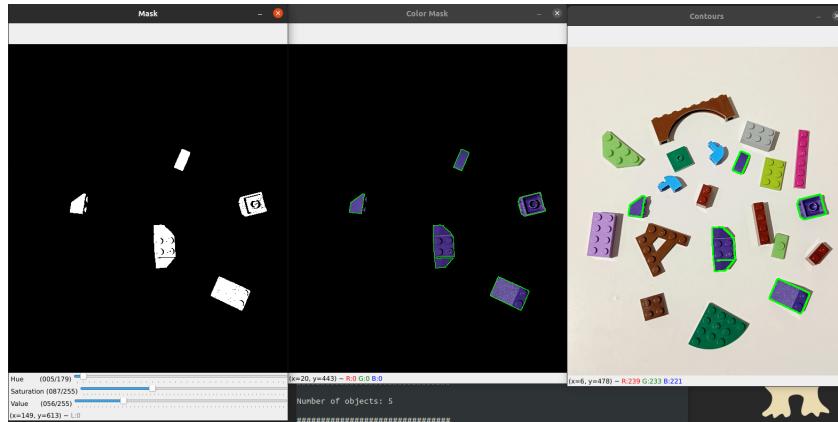


Figura 24: Prueba 1. Seleccionamos las piezas de color morado oscuro.

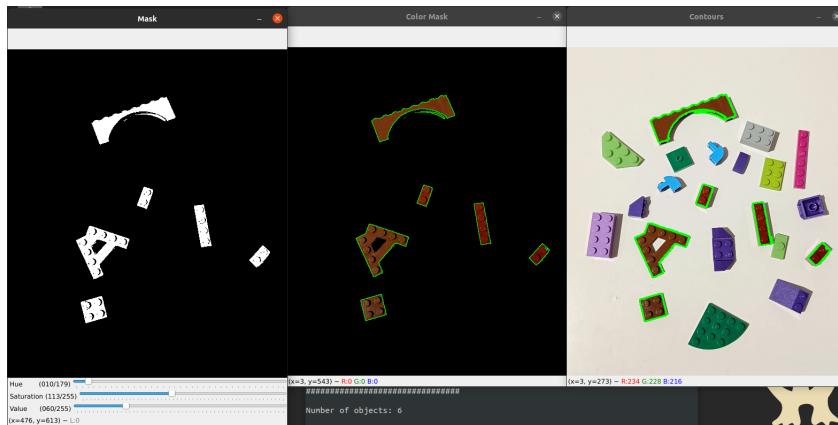


Figura 25: Prueba 2. Seleccionamos las piezas de color marrón.

Vemos que los resultados conseguidos son bastante buenos. Para el caso de las piezas marrones, hemos tenido que regular un poco el umbral considerado para el valor *HUE* y para el valor *Saturation* para que la máscara captase bien todas las piezas.

Aunque es cierto que el ejemplo es sencillo y las condiciones de ejecución son buenas (fondo plano, objetos relativamente grandes, buena iluminación, casi ausencia de sombras, etc.), podemos decir que los resultados son aceptables.

En caso de querer afinar el programa para detectar objetos en una situación particular, es posible hacerlo. Para ello bastaría especificar más criterios de selección de contornos, como por ejemplo el grado de convexidad de un objeto (mejora la detección de contornos con forma cuadrangular), etc.

5. FILTROS

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

En este ejercicio he escrito un programa para poder aplicar diferentes filtros a la imagen, en particular el *box filter*, *gaussian filter*, *median*, *bilateral*, *minimum and maximum*. Cada filtro podrá seleccionarse con el teclado y sus parámetros se podrán configurar con trackbars.

Lo primero que he hecho ha sido crear un trackbar para cada filtro, donde se almacena el parámetro para configurar dicho filtro. Además, utilizo la clase *ROI* de la librería *umucv.util* para definir la región en la que aplicarlo. Como observación, antes de aplicar el filtro compruebo que la región sea válida (área > 0, ninguna coordenada negativa o mayor que el tamaño del frame, etc.), como una medida de seguridad para prevenir posibles errores.

Para seleccionar el filtro en cuestión, se emplean las teclas numéricas (se almacena la tecla pulsada en una variable, para identificar el filtro que hay que aplicar). Cada número 1 . . . 6 tiene asociado un filtro, como se puede consultar en la ventana de ayuda.

Finalmente, se procesa la imagen, aplicando el filtro seleccionado en el frame con sus valores de configuración obtenidos del trackbar. Si se selecciona otro filtro, se aplicará a partir del siguiente frame.

Como el código del último apartado de esta sección amplía este (es igual, pero permite componer varios filtros), solamente se incluye el código de este último.

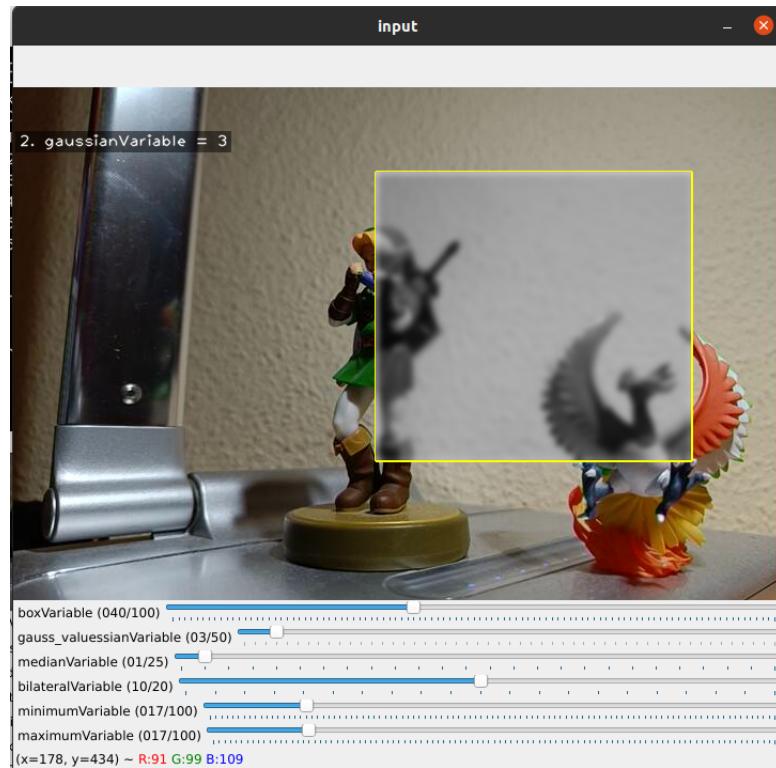


Figura 26: Ejemplo de ejecución con el filtro *gaussian* con $\sigma = 3$.

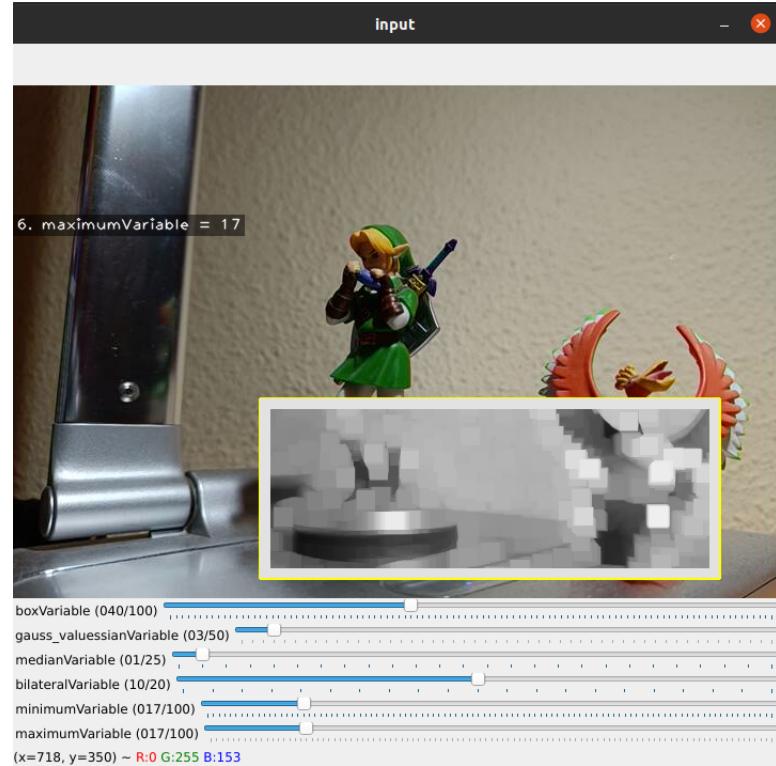


Figura 27: Ejemplo de ejecución con el filtro *maximum (max)*.

5.a. Comprueba la propiedad de “cascading” del filtro gaussiano

Para comprobar la propiedad de *cascading* del filtro gaussiano, se comprueba que aplicando sucesivos filtros de valor σ_1, σ_2 es equivalente a aplicar el filtro con un valor $\sigma_3 = \sqrt{\sigma_1^2 + \sigma_2^2}$ ([5] p.4).

Para ello aplicamos primero un filtro gaussiano mediante la función `cv.gaussianblur()` con $\sigma = 3$, y sobre el resultado aplicamos otro filtro gaussiano con $\sigma = 4$. Finalmente, aplicamos un filtro gaussiano con $\sigma = 5$ sobre la imagen original. Comparamos el resultado de aplicar en cascading y en un solo paso realizando la diferencia entre las dos imágenes.

Como observación, se han tomado como valores de σ una *terna pitagórica*, para minimizar errores en el resultado derivados de la aproximación del cálculo de una raíz cuadrada no exacta.

A continuación se muestra el resultado de la ejecución:



Figura 28: Imagen base para comprobar las propiedades del filtro gaussiano.

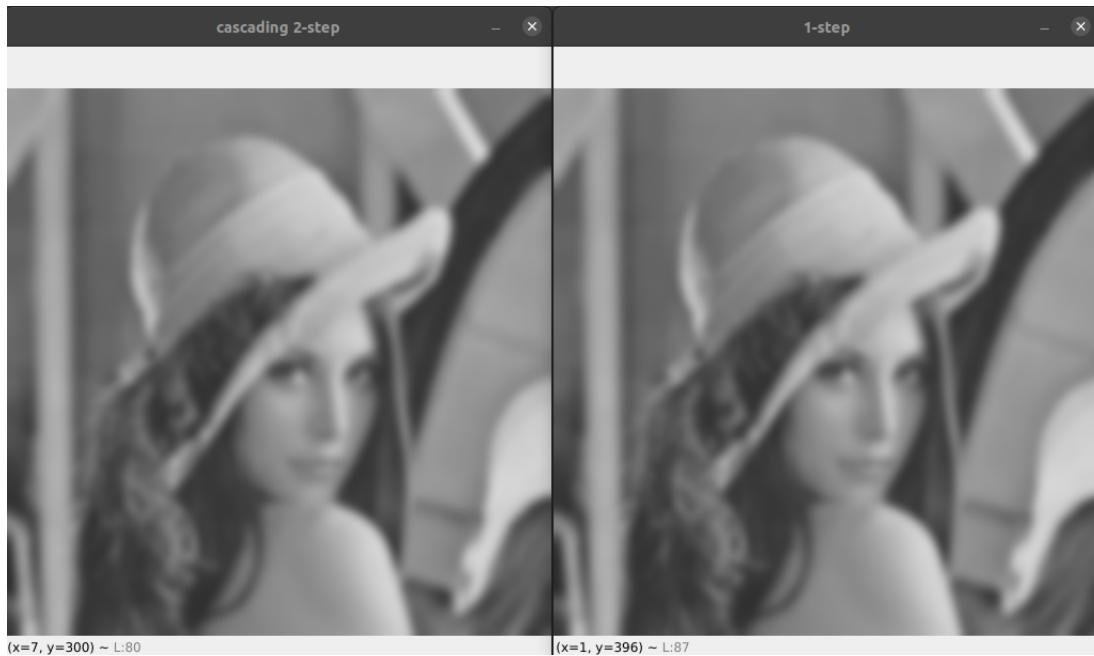


Figura 29: Resultados de aplicar el filtro gaussiano en cascading y en 1 solo paso.

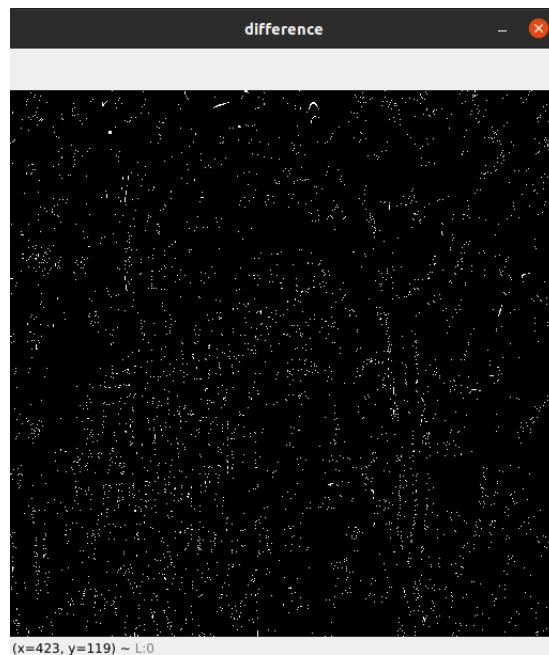


Figura 30: Diferencias entre cascading y aplicación en 1 paso.

Se pueden apreciar pequeñas diferencias en ambos resultados. Si lo calculamos con precisión, vemos que el método cascading comete un error del 2,9 % respecto a la aplicación del filtro en un paso.

El error puede deberse al hecho de que la función gaussiana es continua en la teoría, pero en la implementación se está trabajando con una discretización de la misma. Esto puede provocar que la

convolución de las gaussianas no sea exactamente una gaussiana, sino una aproximación. En este caso, la aproximación es bastante buena y la consideraremos admisible.

5.b. Comprueba la propiedad de “separabilidad” del filtro gaussiano

La propiedad de separabilidad garantiza que la aplicación de un filtro bidimensional es equivalente a aplicar el filtro en una dimensión y después en la otra ([5] p.8).

Para comprobar la propiedad de separabilidad, no podemos emplear la función *gaussianblur()* de *openCV*, puesto que la desviación estándar σ debe ser positiva en ambas direcciones ([6]).

Por eso usaremos la función *cv.sepFilter2D()* de *openCV*, que aplica un filtro lineal separable a una imagen, primero se aplica el filtro sobre las filas con el kernel unidimensional *kernelX* y después las columnas con el kernel unidimensional *kernelY*.

Finalmente aplicamos un filtro gaussiano en un paso con un kernel tamaño *kernelX* \times *kernelY*, y comparamos los resultados.

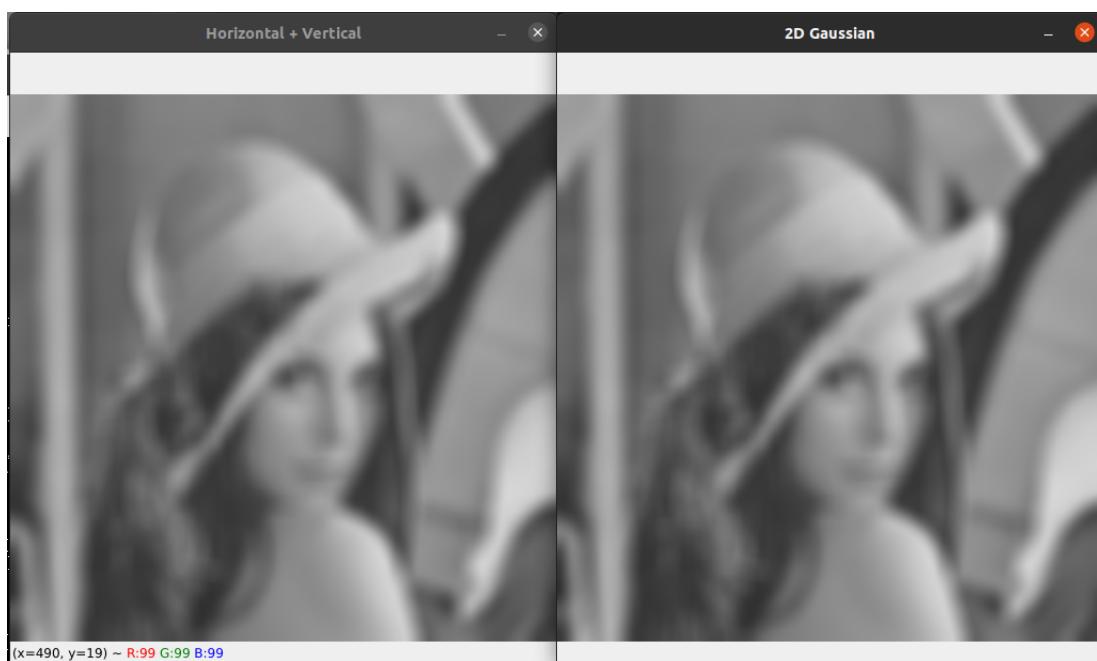


Figura 31: Aplicación de un filtro gaussiano primero en una dirección y luego en otra y aplicación del mismo filtro 2D.



Figura 32: Diferencias entre los dos métodos anteriores.

Como se puede observar, hay pequeñas diferencias entre ambos resultados, debidos probablemente a las razones expuestas en el apartado anterior. Por tanto, podemos afirmar la equivalencia y así la propiedad de separabilidad del filtro gaussiano.

5.c. Implementa en Python desde cero (usando bucles) el algoritmo de convolución con una máscara general y compara su eficiencia con la versión de OpenCV

Para implementar la convolución con bucles, necesitamos la máscara (kernel), y la imagen sobre la que aplicarla. Empezamos definiendo una matriz con las mismas dimensiones que la imagen original, donde se almacena el resultado de la convolución. La idea es recorrer elemento por elemento de la imagen y por cada elemento aplicar la máscara.

Un problema surge en los extremos (bordes) de la imagen, puesto que el kernel no se encuentra completamente dentro de la imagen. Para solucionar esto, hemos empleado el método de extender los bordes de la imagen original, asignando el valor del píxel más cercano. De esta forma, podemos aplicar la convolución sin que los resultados, se vean afectados en extremo (suponemos que en pocos píxeles, los valores de la imagen varían poco).

Para ver la eficiencia hacemos además del ejemplo, una ejecución con una imagen de gran tamaño ($10000 \times 10000\text{px}$, fichero /filtros/large.jpg). El resultado obtenido es:

```
convolution time: 0.8149206638336182
opencv time: 0.015822410583496094
opencv is 51.504204 times faster than convolution
```

Figura 33: Comparación entre la convolución “manual” y la función de convolución de librería. Imagen de ejemplo.

```
convolution time: 412.7373208999634
opencv time: 0.3654313087463379
opencv is 1129.452543 times faster than convolution
```

Figura 34: Comparación entre la convolución “manual” y la función de convolución de librería. Imagen de gran tamaño.

Vemos que la función de la librería *OpenCV* tiene una complejidad temporal mucho menor, y que la diferencia crece conforme aumenta el tamaño de la entrada. Esto se debe probablemente a que la función de librería emplea técnicas como la transformada de Fourier (FFT) para optimizar mucho el cálculo.

5.d. Añade la posibilidad de seleccionar varios filtros para aplicarlos sucesivamente

Es una extensión del primer apartado del ejercicio. Partiendo del código anterior, en lugar de almacenar el filtro seleccionado en una variable, se mantiene un array con tantas posiciones como filtros. Cuando se selecciona un filtro, se pone a 1 la posición del array, y cuando se procesa la imagen se recorre el array, aplicando en orden todos los filtros que se han seleccionado.

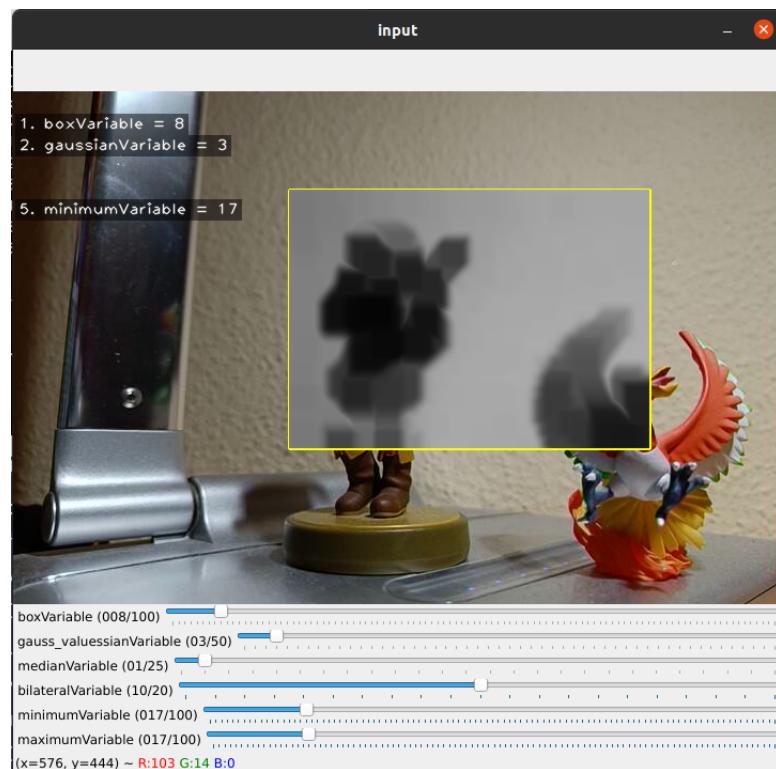


Figura 35: Aplicación de varios filtros.

6. SIFT

6.a. 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.

La idea del ejercicio es detectar las coincidencias de puntos clave (keypoints) entre la imagen de la cámara y una serie de modelos de imágenes previamente almacenadas y utilizar estas coincidencias para poder identificar de qué modelo se trata. Para ello, se utilizará el algoritmo *scale-invariant feature transform (SIFT)* para detectar los puntos de interés y establecer las asociaciones entre puntos.

Para implementar este ejercicio partimos del código del programa “*code/sift.py*” visto en clase de prácticas. En este código se captura una imagen y se calculan los keypoints, para después calcular los keypoints de la imagen actual y realizar asociaciones entre ambos conjuntos de puntos clave (*matcher.knnMatch()*). Lo más destacable es la implementación de un *ratio test*, de manera que solamente se toman en cuenta las coincidencias que son “mucho mejores que la segunda opción”. De esta forma se consigue evitar coincidencias erróneas e identificar de forma más precisa los objetos.

Partiendo de esta base, he añadido los modelos que se emplearán para identificar los objetos (directorio *SIFT/models*). Estos modelos se preprocesan para calcular los keypoints antes de iniciar el bucle de captura de imagen. Al pulsar la tecla “c”, se captura la imagen actual, se calculan sus keypoints y se emplea a comparar con los distintos modelos de la forma explicada anteriormente. Si se encuentra un porcentaje determinado (lo he establecido a 0,2) de asociaciones suficientemente buenas, se considera que se ha encontrado un *match* y se ha identificado correctamente el objeto.

Como observación, he empleado un sistema de estados (*IDLE*, *READ*, *MATCH*) para gestionar cuándo capturar, procesar y eliminar los keypoints que queremos comparar con los modelos. Cuando comienza el programa, el estado es *IDLE*, donde no hay ninguna imagen que comparar con los modelos. Cuando se pulsa la tecla “c”, se produce la captura de keypoints y descriptores del frame actual, y se pasa al estado *READ*. Si nos encontramos en este estado, se realiza la comparación de keypoints explicada anteriormente para encontrar un “*match*” con los modelos. Si se encuentra, se pasa al estado *MATCH*, donde ya no se procesan los keypoints y se muestran las coincidencias encontradas con el modelo. Finalmente, si se pulsa la tecla “x” o si el programa se encuentra en estado *READ* pero no se ha encontrado un match suficientemente bueno, se regresa al estado *IDLE*.

Por último, algunos ejemplos de ejecución. En mi caso, se trata de detectar portadas de algunos libros.

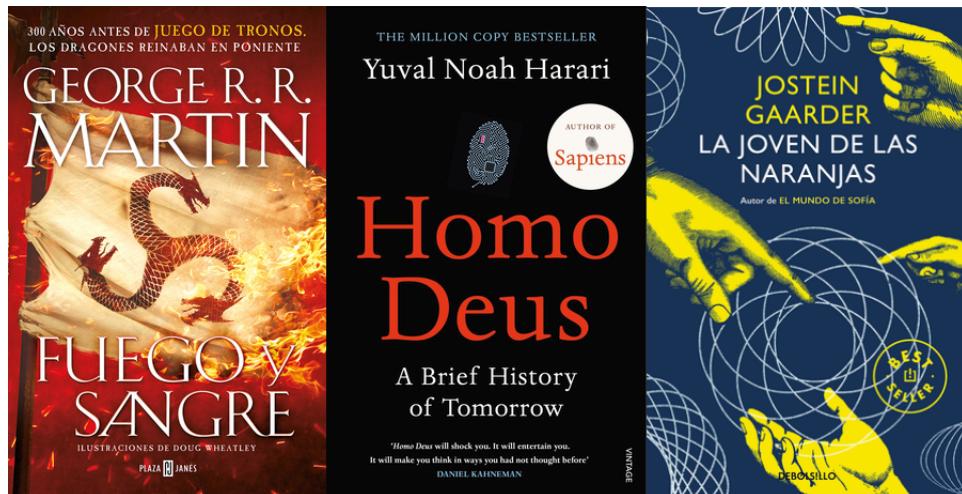


Figura 36: Imágenes empleadas para generar los modelos de las portadas de los libros.

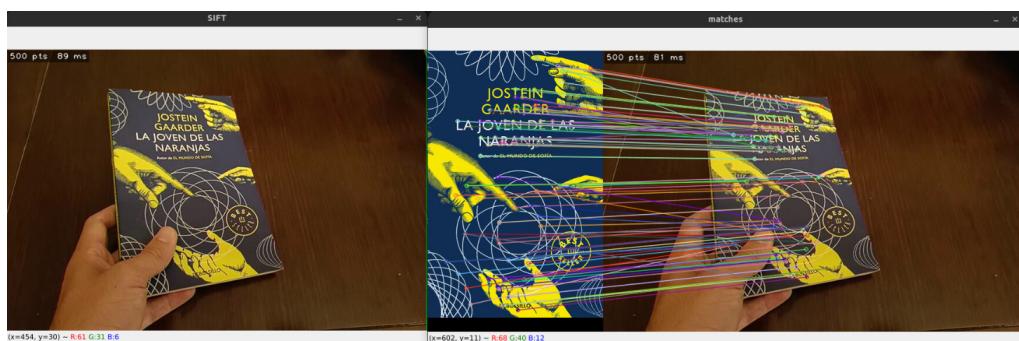


Figura 37: Ejemplo de ejecución. Detecta correctamente de qué libro se trata y se muestran las coincidencias entre *keypoints*.

En la salida del programa vemos el número de coincidencias de keypoints con cada modelo:

```
15 matches with model 0
108 matches with model 1
8 matches with model 2
model 1 with 108 matches
```

Figura 38: Coincidencias con cada uno de los modelos.

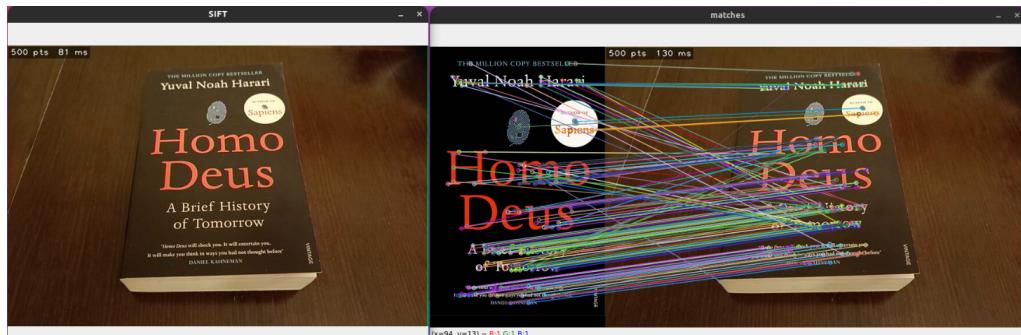


Figura 39: Otro ejemplo.

También considero interesante probar otro ejemplo en el que el libro se ve con dificultad en una imagen en la que también hay otros objetos. Utilizando otros sistemas podría ser complicado identificarlo, pero empleando coincidencias de keypoints podemos conseguirlo. Esto nos muestra que se trata de una técnica muy potente con muchas potenciales aplicaciones y usos.

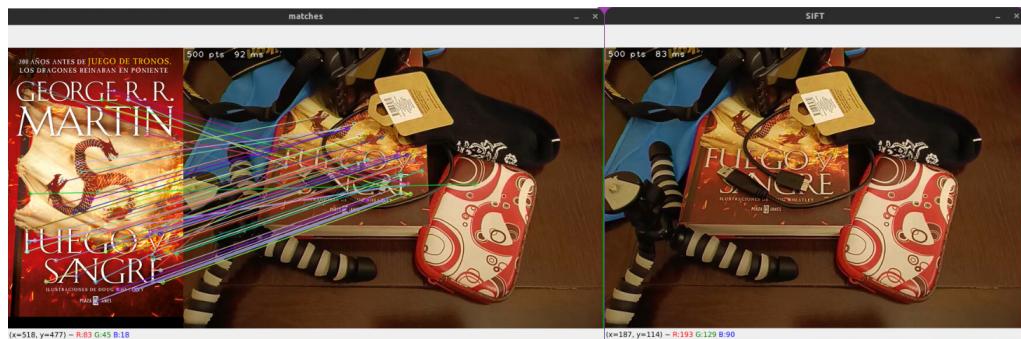


Figura 40: Otro ejemplo. Reconoce el libro en una imagen con más elementos.

7. RECTIF

- 7.a. Rectifica la imagen de un plano para medir distancias (tomando manualmente referencias conocidas). Las coordenadas reales de los puntos de referencia y sus posiciones en la imagen deben pasarse como parámetro en un archivo de texto.**

El objetivo del ejercicio es que, conocidos una serie de puntos de referencia en la imagen y sus coordenadas reales, así como la distancia real entre dos de los puntos (la llamaremos d), seamos capaces de medir distancias entre dos puntos de la imagen en el mundo real. Esto lo conseguiremos empleando la noción de *homografía*,

Primero utilizamos los puntos de referencia para rectificar la imagen mediante una transformación homográfica (pasar a las coordenadas “reales”). Con esta transformación, calculamos las coordenadas “reales” de los puntos marcados. En el espacio rectificado de las coordenadas “reales”, medimos la distancia en **píxeles** entre los puntos de referencia (la llamaremos pix_r) y la distancia entre los puntos marcados (pix_m). Entonces, empleamos la distancia real conocida entre puntos de referencia para calcular la distancia real entre los puntos marcados, siguiendo la **proporcionalidad** existente entre distancia en píxeles en el plano rectificado y distancia real. Esta proporción es válida debido a que el plano rectificado tiene la misma perspectiva que las mediciones en el mundo real (es decir, es una representación a escala del mundo real) y por tanto sus medidas son proporcionales.

$$d_{\text{marcados}} = d * \frac{\text{pix}_m}{\text{pix}_r}$$

Y este es el resultado que buscábamos.

Como observación, esta distancia se mide en las mismas unidades que la distancia d .

A continuación, muestro algunos detalles de implementación que considero relevantes:

Para realizar el código usaré como apoyo la herramienta *medidor* de la librería de códigos de la asignatura (*code/medidor.py*) para poder medir distancias en píxeles en una imagen. Por otra parte, definiré una función *mouse_callback()* similar a las que se han visto en clase de prácticas, que detecta los eventos de click del ratón y almacena las coordenadas de los puntos marcados.

Para introducir en el programa las posiciones de los puntos de referencia conocidos, así como la distancia real entre los dos primeros puntos de referencia, se leen del fichero *RECTIF/ref.txt*, con el siguiente formato:

```

d
Rx1, Ry1
Ix1, Iy1
Rx2, Ry2
Ix2, Iy2
...
Rxn, Ryn
Ixn, Iyn

```

Donde:

- d es la distancia **real** entre los dos primeros puntos de referencia.
- Rxi, Ryi son las coordenadas del punto de referencia i -ésimo en la realidad.
- Ixi, Iyi son las coordenadas del punto de referencia i -ésimo en la imagen.

Es interesante destacar que en los ejemplos hemos seleccionado unos puntos de referencia lo más alejados posible, para que pequeños errores en la precisión de las medidas no se traduzcan en un gran error al calcular la transformación homográfica.

Para calcular la transformación homográfica y rectificar la imagen se emplean las funciones de *openCV*, *cv.warpPerspective()* y *cv.findHomography()*. Para aplicar la transformación sobre puntos se utiliza la función de la librería *umucv.htrans()*.

Por último, mostraré un ejemplo de ejecución:

En las semifinales del Madrid Open del pasado año 2022, el tenista murciano y joven estrella del tenis mundial Carlos Alcaraz se impuso al serbio Novak Djokovic en uno de los partidos más destacados de la historia reciente del tenis. Uno de los momentos clave del partido fue la jugada con la que Alcaraz consiguió la victoria del segundo set al salvar con una gran carrera un punto que parecía imposible de ganar. La situación en cuestión era la siguiente (imágenes superpuestas para ver mejor la situación):



Figura 41: Asombrosa carrera de Carlos Alcaraz para golpear la pelota. Sobre la imagen (ampliar) se puede ver la distancia que queremos medir (marcada en rojo).

Nuestro objetivo es medir cuánta distancia recorrió en esa carrera hasta golpear la pelota.

Siguiendo los mismos pasos que en el ejemplo anterior, tomamos esta vez las 4 esquinas de la pista de tenis como referencias para la rectificación del plano. Considerando las medidas oficiales del campo de tenis [2], introducimos los puntos reales para que guarden la proporción. Como observación, hemos tomado las medidas del campo multiplicadas varios órdenes de magnitud para poder trabajar con enteros en el cálculo de la homografía. Esto es irrelevante (la matriz de homografía se multiplica por un escalar, y la inversa se divide por dicho escalar), porque lo importante es que se mantengan las **proporciones**.

Para medir, marcaremos el pie izquierdo en los dos instantes (el menos adelantado en ambos), y obtenemos como resultado que el tenista recorrió **9,54 metros**.

8. RA

8.a. Crea un efecto de realidad aumentada en el que el usuario interactúe con los objetos virtuales.

Para este ejercicio he decidido hacer un programa que detecte un marcador en la imagen, muestre un cubo en realidad aumentada sobre el plano del marcador y permita trazar una ruta con el dedo para que el cubo la siga desplazándose por ella.

Para realizar el ejercicio me basaré en los ejercicios sobre RA vistos en clase de prácticas. En concreto, me basaré en el código *code/pose/pose2.py* para implementar el sistema de detección y estimación de pose, por tanto no entrará en detalles aquí. Como observación, emplearé el marcador que puede encontrarse en */RA/marker.png*

Primero explicaré el sistema de seguimiento del dedo para trazar la ruta. He aplicado el método de Lucas–Kanade para obtener un tracker de puntos. Mi objetivo es conseguir un tracker de la posición de la punta del dedo para poder almacenar los puntos que conforman el tracker y utilizarlos como ruta para que el cubo la siga en RA.

Para ello he empleado la funcionalidad *ROI* de la librería proporcionada en las prácticas para definir un ROI para definir la región en la que se deben seguir los puntos con el tracker. Es decir, el ROI define una máscara y dentro de esa región se empezarán a seguir puntos con el tracker.

La idea aquí será definir una región para hacer el seguimiento, situar el dedo dentro de esa región para que lo detecte y ya mover el dedo fuera de la región mientras el algoritmo va siguiendo su posición en un track.

En este punto se mueve el dedo libremente definiendo una ruta para trazar en la imagen. Cuando el dedo deja de moverse durante un par de segundos, esa ruta se fija (se vuelve de color verde, y se dibuja sobre el plano del marcador (usando la estimación de pose)). Será la ruta que seguirá el cubo.

Finalmente, pulsando la tecla “M”, el cubo iniciará su desplazamiento por la ruta marcada, siguiendo todos los puntos desde el principio hasta el final.

A continuación, muestro un ejemplo de ejecución:

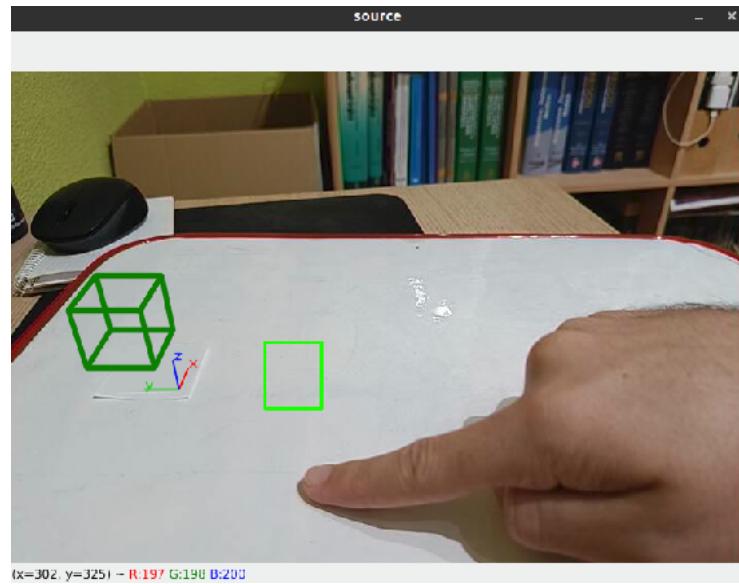


Figura 42: Posición inicial, se pulsa “C” para poder definir el ROI. He definido el ROI donde se seguirá la punta del dedo.

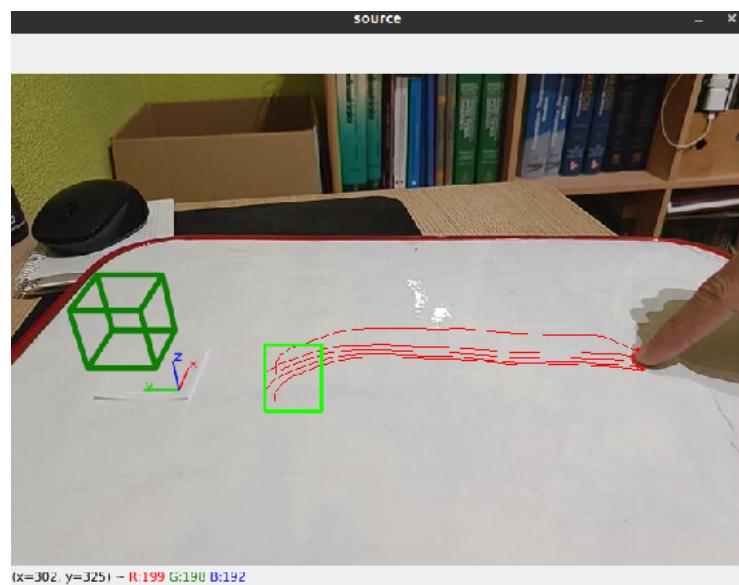


Figura 43: Cuando el dedo se introduce en el ROI empieza el seguimiento. Al mover el dedo se va definiendo un camino.

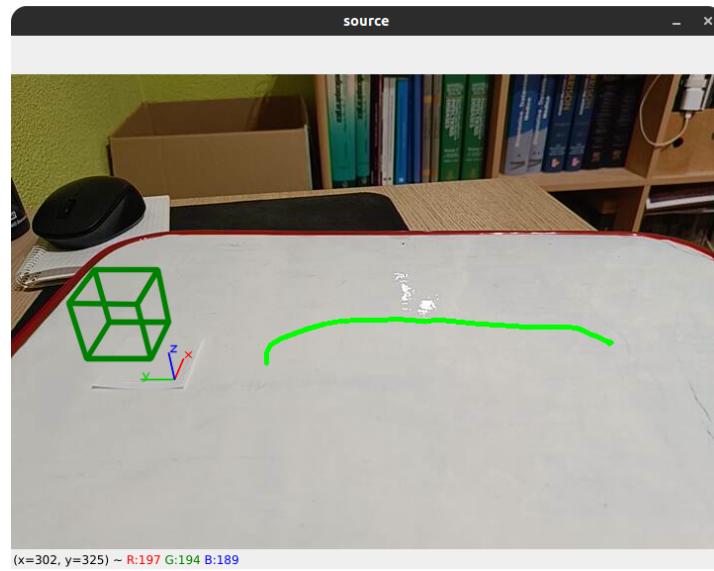


Figura 44: Cuando el dedo se deja quieto, la ruta se fija (se vuelve verde y se transforma al plano del marcador).

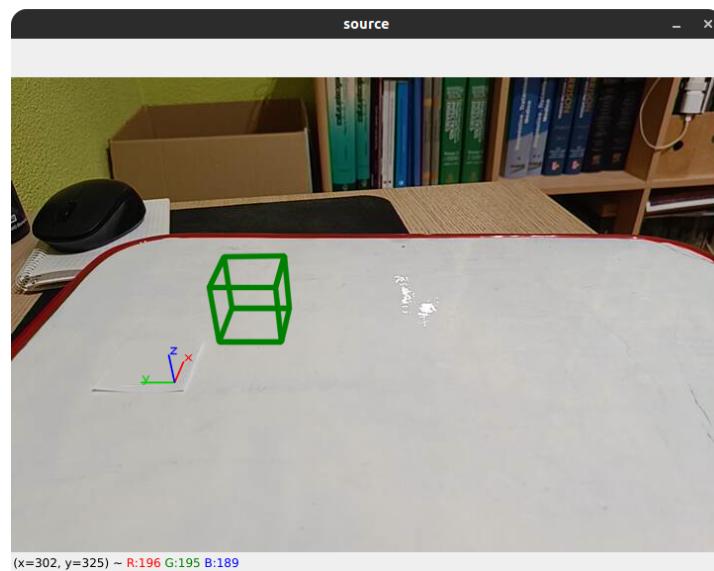


Figura 45: Al pulsar la tecla “M”, el cubo empieza a moverse desde el primer punto de la ruta.

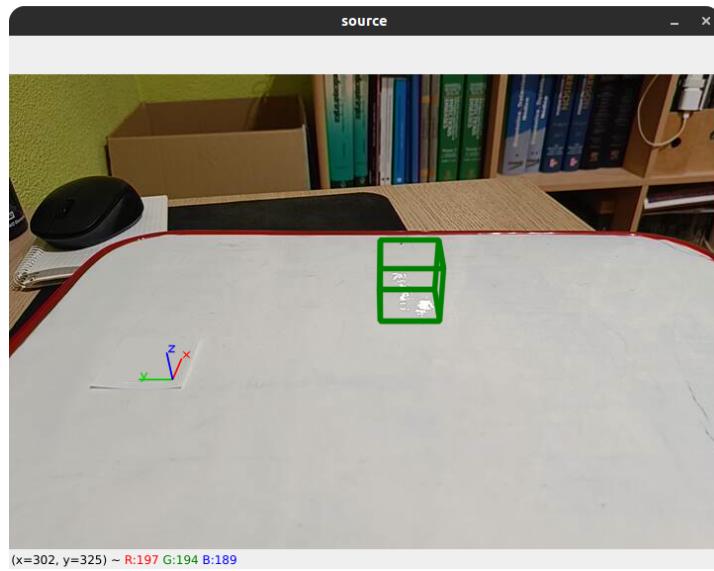


Figura 46: Movimiento siguiendo la ruta.

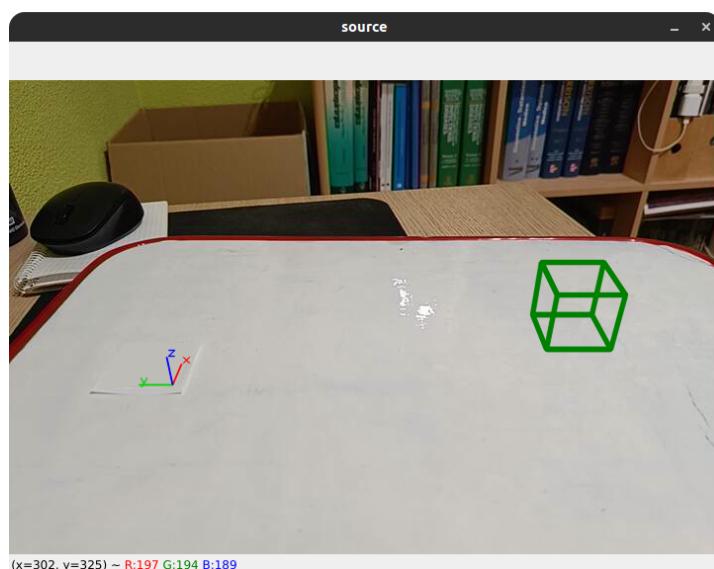


Figura 47: Movimiento siguiendo la ruta.

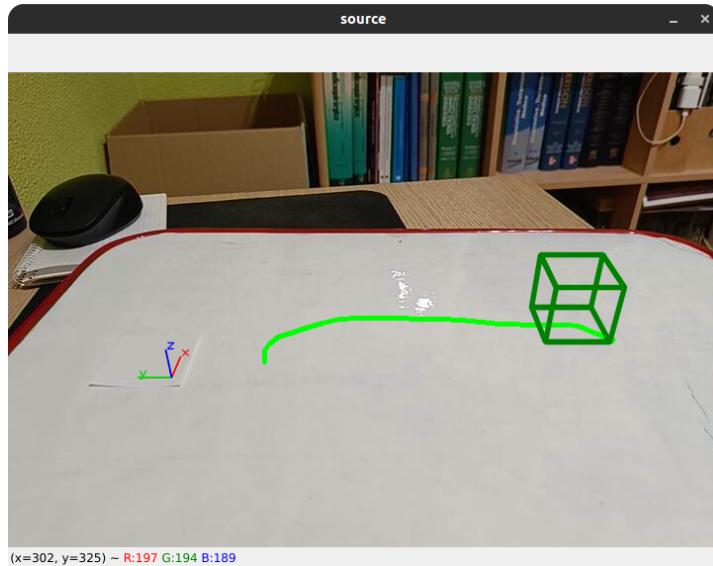


Figura 48: Pulsando la tecla “C” se puede ver la ruta. Si se vuelve a pulsar la tecla “M” se inicia de nuevo el movimiento.

En cuanto a la implementación, he creado una clase *Cube* para poder gestionar algunos parámetros internos del cubo como el estado en el que se encuentra, la posición y ruta que debe seguir, la velocidad, etc. El cubo tiene un sistema de estados (STILL, FINDING, FOUND, MOVING). Lo importante es que con la tecla “C” se entra a estado FINDING en el que podemos definir un ROI y con la tecla “M” empieza el movimiento. No es muy relevante desde el punto de vista de la visión artificial así que no lo explicaré.

Es fascinante cómo, a partir de la matriz de pose estimada, podemos lograr la transformación de posiciones en la imagen a posiciones en el plano del marcador. A primera vista, puede parecer un proceso complejo, ya que no se trata de una simple transformación homográfica. Estamos trabajando con la matriz de pose (matriz extrínseca), que define la transformación entre el sistema de coordenadas real y el sistema de coordenadas de la cámara. Esta matriz tiene dimensiones de 3x4, a diferencia de la matriz de homografía que debería tener dimensiones de 3x3.

Sin embargo, la homografía se puede considerar como un caso especial de esta transformación, en el cual las coordenadas del mundo real se proyectan en el plano Z=0. En otras palabras, si ignoramos la coordenada Z, la matriz de pose representa precisamente la transformación de homografía entre el mundo real y el plano del marcador. Por lo tanto, al eliminar la columna correspondiente a la tercera dimensión de la matriz de pose, obtendremos la matriz H de homografía, que podemos aplicar de manera convencional como hemos hecho en otros ejercicios. **Podemos pasar de las coordenadas 2D en la imagen a las coordenadas “reales” en el plano del marcador:** Para ello bastará calcular la inversa de la matriz de homografía y aplicar la transformación sobre los puntos de la imagen.

En resumen, al suprimir la columna correspondiente a la tercera dimensión de la matriz de pose, obtenemos la matriz H de homografía, lo cual nos permite realizar la transformación de manera adecuada. Se puede encontrar una prueba formal de esta equivalencia, así como información más

detallada en [7].

Una vez entendido cómo transformar de coordenadas de la imagen a coordenadas del mundo real, es sencillo mostrar los elementos en RA. Para dibujar el cubo en coordenadas 3D utilizamos la matriz de pose M, y para transformar la ruta del cubo a las coordenadas en el plano del marcador extraemos la matriz de homografía y calculamos su inversa. Con esto queda explicada toda la parte gráfica de interés.

Como observación, a base de experimentar con el programa he descubierto que al definir la ROI para trackear el dedo puede haber fallos debido a que se detecten otros puntos. Principalmente puede deberse a cambios de las condiciones de iluminación o mala calidad de la imagen, que confunden al tracker. En este caso habrá que reiniciar el programa.

9. Opcional: VROT

Observación: Debido a limitación de tiempo, no he podido documentar a fondo este ejercicio. Está hecho íntegramente y puede encontrarse en su directorio correspondiente, en el fichero *vrot.py* gran parte de las explicaciones se encuentran realizadas en los comentarios del código.

Amplía el tracker de Lucas-Kanade

9.a. Determinar en qué dirección se mueve la cámara (UP,DOWN,LEFT,RIGHT, [FORWARD, BACKWARD])

Como idea general, extraemos los puntos de las trayectorias y los utilizamos para calcular el desplazamiento medio en cada eje.

Con esto determinamos si el movimiento es principalmente hacia arriba, abajo, izquierda o derecha (en sentido contrario a lo que obtenemos, puesto que buscamos el movimiento de la *cámara*, no de la *imagen*)

9.b. Estimar de forma aproximada la velocidad angular de rotación de la cámara (grados/segundo)

10. Opcional: SILU

10.a. Amplía el reconocedor de siluetas para que admita múltiples prototipos. Intenta leer placas de matrícula vistas de frente.

En este ejercicio se utiliza el análisis frecuencial de los contornos para reconocer las siluetas de los símbolos que conforman una matrícula de vehículo. Es una ampliación de los ejemplos vistos en las prácticas, ampliando para reconocer múltiples modelos de símbolos.

Como observaciones iniciales, en este ejercicio nos ceñiremos a las matrículas de vehículos con el formato de España. Según se establece en el Boletín Oficial del Estado [[3]], se emplea un sistema de combinación de un número de cuatro cifras y de tres letras, de las cuales se excluyen las cinco vocales, la Ñ y la Q. Los números y letras se disponen como se muestra en la figura (49).



Figura 49: Formato de una placa de matrícula en España.

Para realizar el ejercicio, primero intentaremos reconocer la placa de matrícula y aislarla del resto de la imagen. Para ello detectaremos los contornos blancos sobre fondo negro, los aproximaremos por polígonos de menor lado y nos quedaremos con el mayor de los que se aproximen por 4 lados. Si la matrícula es el elemento principal en la foto, esto nos permite quedarnos con el contorno rectangular de la placa. Para el resto del ejercicio solamente consideraremos esta región.

El siguiente paso es detectar los contornos negros sobre fondo blanco. El primer contorno será la letra de identificación del país, podemos descartarlo. De los siguientes, los 4 primeros son los números y los 3 siguientes las letras, separamos los contornos en 2 conjuntos *numbers* y *alpha*. Como algunas matrículas tienen un formato distinto (4 números en la primera fila y 3 letras en la segunda fila), hemos añadido un análisis de los contorno para ordenar y separar los contornos correctamente incluso en este caso.

Después, se realiza la identificación de símbolos a partir de los contornos obtenidos y los modelos utilizando análisis en el espacio de las frecuencias. Se toman en cuenta las mismas consideraciones que se explican extensamente en el ejercicio (12). Como observaciones particulares de este ejercicio, a partir de las pruebas empíricas realizadas, he decidido establecer dos umbrales distintos para la comparación en el espacio de las frecuencias para el caso de números y el caso de letras. Esto se debe a que, por lo general, las letras se reconocen más fácilmente y no dan lugar a tantos casos confusos. Una razón podría ser que los distintos símbolos son más distintos entre sí, mientras que en los números hay pares de números cuyas frecuencias son muy similares (6 y 9, 0 y 8, 2 y 5). Por eso he establecido un umbral menor para los números.

Además, antes de la detección de contornos de los símbolos de la matrícula, aplico un filtro gaussiano de suavizado para eliminar de la imagen elementos que interfieran con la detección de

los contornos.

A continuación, se muestran algunos ejemplos de ejecución.



Figura 50: Ejemplo de ejecución. Vemos la imagen original, el ROI sobre el que se trabaja y los resultados de la detección de contornos. Detecta la matrícula sin problemas.

Por lo general funciona bien siempre que no haya ni 0 ni 8 entre los números (son los números más problemáticos). También he encontrado problemas para diferenciar la letra “L” de la “J” por su gran parecido en el espacio de las frecuencias.

Los valores de los umbrales para números y letras explicados anteriormente los he ajustado empíricamente de forma que funcione en la mayor parte de los casos. Para algunos casos extremos he incluido un modo “avanzado”, que permite regular estos valores, así como la intensidad del filtro de suavizado, mediante trackbars y solucionar estos casos más difíciles. Para acceder a él basta ejecutar el programa con el último argumento “–advanced”.

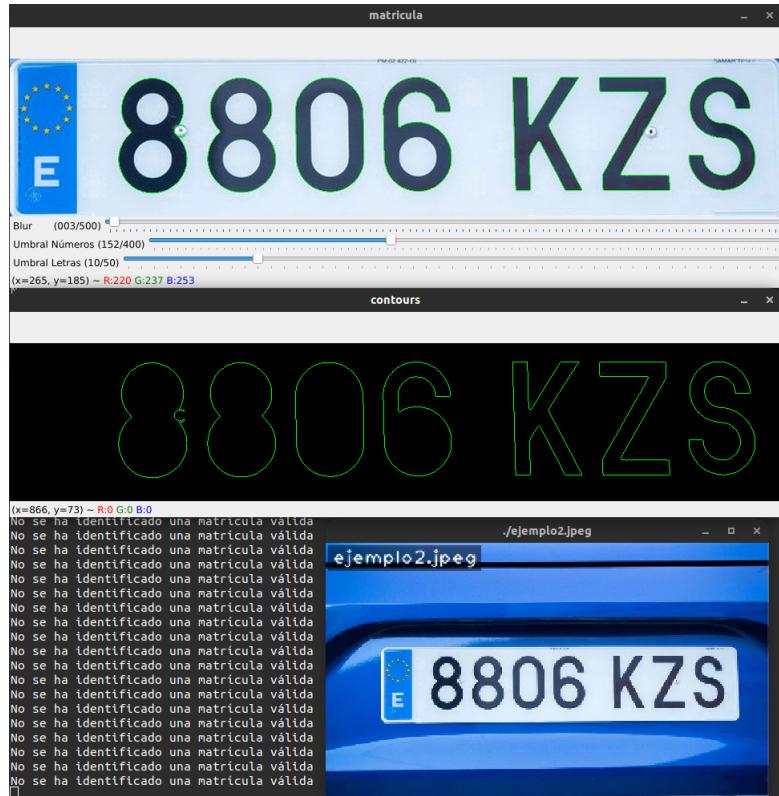


Figura 51: Modo avanzado del programa SILU. En este caso no lo detecta bien, seguramente debido al tornillo que impide reconocer correctamente el contorno del primer 8. Modificando los parámetros podemos intentar afinar la detección.

En el directorio del ejercicio (*/SILU*) he incluído más imágenes de ejemplo. Es interesante probarlas y “jugar” con los parámetros de los umbrales de detección para ver cómo cambia la detección, con qué letras y números encuentra más dificultades, etc.

11. Opcional: SWAP

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

En este ejercicio el primer paso es marcar dos cuadriláteros C_1 y C_2 en la imagen. Puesto que estos cuadriláteros pueden encontrarse en dos planos distintos de la imagen, el siguiente paso es calcular una transformación homográfica, pero en vez de tratarse de una homografía entre una imagen y un “plano en la realidad”, como hemos hecho en otros ejercicio (ver 7), en este caso realizamos la transformación entre los 2 planos de la imagen determinados por los dos cuadriláteros.

De esta manera, conocemos las dos regiones en las que se sitúan los cuadriláteros, y mediante la transformación homográfica y su inversa podemos transformar los puntos de un plano en puntos del otro y viceversa.

Con esto el ejercicio se resuelve fácilmente. Si suponemos que la homografía H se ha calculado para transformar puntos del plano de C_1 al plano de C_2 , basta tomar R_1 la región de la imagen correspondiente a C_1 en la imagen, aplicarle H y obtenemos la región transformada T_1 . De manera análoga obtenemos la región T_2 transformada de C_2 . Terminamos sustituyendo en una copia de la imagen original la región de cada cuadrilátero por la transformada de la región contraria.

Mostramos un ejemplo práctico en el que intercambiamos dos fotografías entre los marcos.

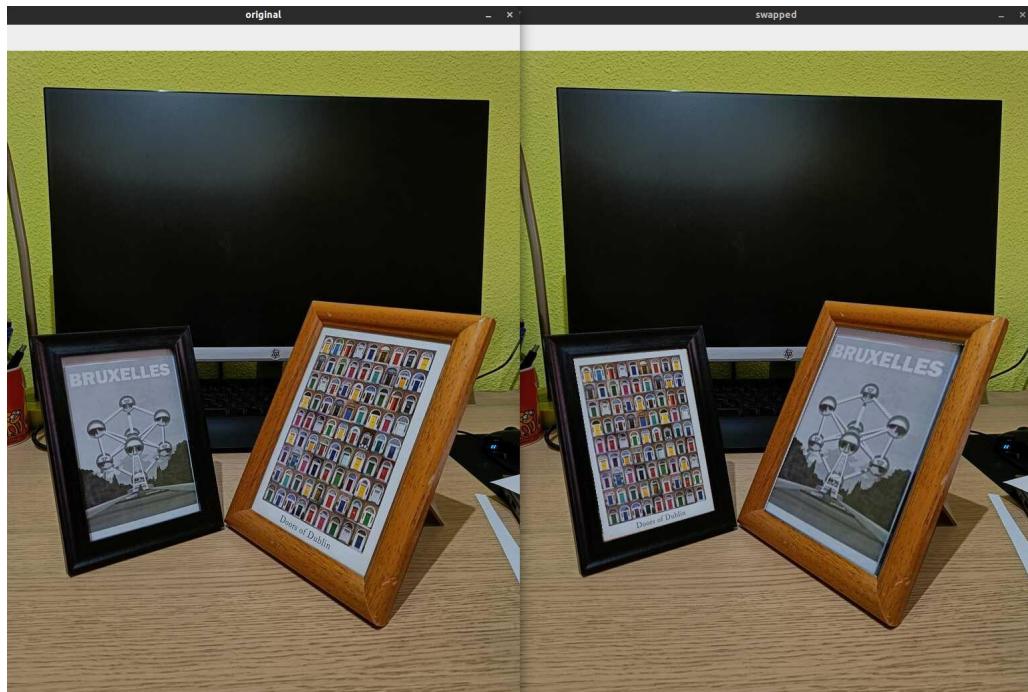


Figura 52: Ejemplo de ejecución de SWAP. Se intercambian las dos fotos entre los marcos. Izquierdo: Antes. Derecho: Después.

12. Opcional: SUDOKU

En el siguiente ejercicio opcional propuesto en clase el objetivo es, a partir de una foto de un juego de sudoku, calcular las soluciones y proyectarlas encima de la imagen del sudoku. Además, el programa debe estar preparado para funcionar con imágenes de sudokus que se vean desde distintas perspectivas, es decir, que funcione incluso cuando la imagen del mismo se ha tomado con un cierto ángulo de inclinación.

Para resolver el ejercicio he empleado los conceptos sobre detección de contornos y análisis frecuencial estudiados en clase.

He seguido los siguientes pasos generales para llegar a la solución:

- a) Detección del contorno exterior del sudoku
- b) Rectificación de perspectiva
- c) Detección de los contornos interiores del sudoku
- d) Detección de los contornos de los números y asignación de los mismos a las casillas interiores correspondientes.
- e) Identificación de los números a partir de los contornos
- f) Resolución del sudoku
- g) Dibujo de las soluciones sobre el sudoku
- h) Deshacer rectificación de perspectiva

Profundizamos en cada uno de ellos.

12.a. Detección del contorno exterior del sudoku

El primer paso es detectar el contorno exterior del sudoku para poder delimitar en qué región de la imagen se encuentra. Para ello empleamos la detección de contornos de opencv con las consideraciones (binarizar la imagen antes de la detección de contornos, considerar solamente contornos razonables, etc.) ya empleadas en ejercicios anteriores. En este caso detectamos los contornos OS-CUROS sobre fondo blanco.

Una vez obtenidos los contornos, empleamos la función *polygons* para aproximar los contornos por polígonos de 4 lados y descartar aquellos contornos oscuros que no se correspondan con cuadriláteros. Tomando el contorno de mayor área tendremos identificado el borde exterior del sudoku.

12.b. Rectificación de perspectiva y detección de los contornos interiores del sudoku

Una vez encontrado el sudoku, se rectifica la imagen para poder analizar su contenido. Para ello calculamos la matriz de homografía (*cv.findHomography()*) a partir de las 4 esquinas del borde del sudoku y usamos *cv.warpPerspective()* para rectificar. Una vez rectificada la imagen, nos quedamos con la región de la imagen que nos interesa (*rectif_crop*, como si fuera un *ROI*)

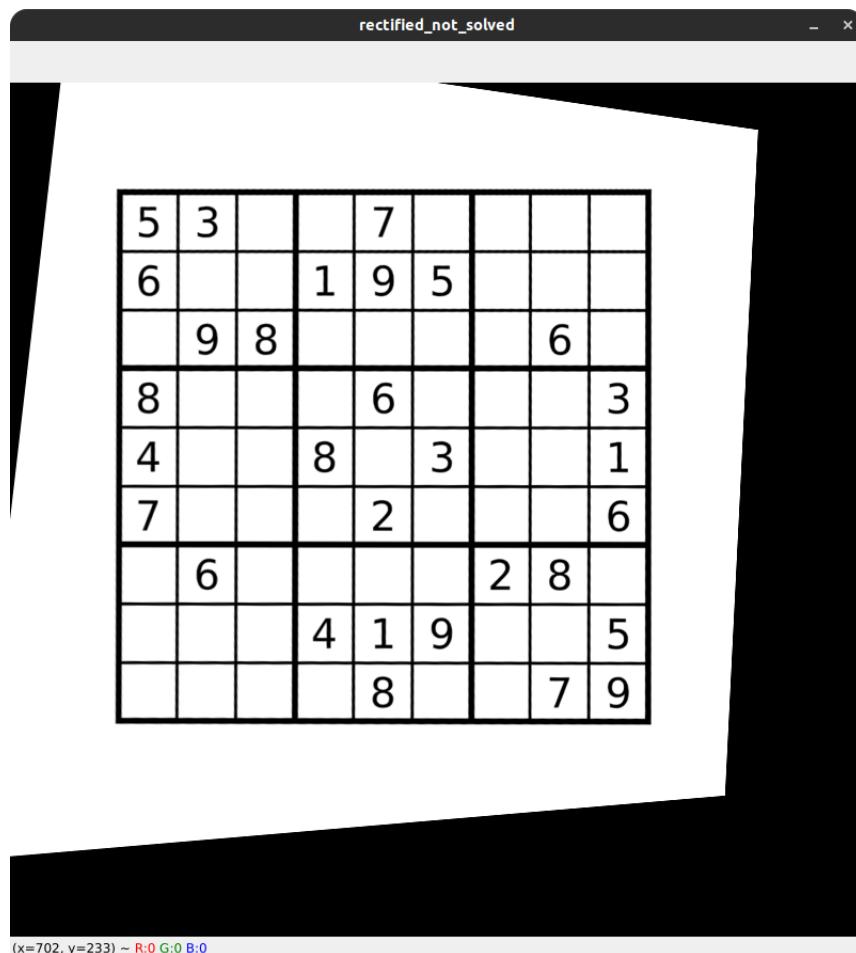


Figura 53: Rectificación del sudoku una vez detectados los bordes.

El siguiente paso es volver a extraer los contornos de la imagen una vez rectificada y recortada. Se podría aplicar la transformación de homografía a los contornos detectados en la imagen original, pero tras hacer unas pruebas me pareció más preciso el resultado si se detectan los contornos en la imagen rectificada. De lo contrario la deformación de perspectiva de la imagen original podría impedir una detección correcta, sobre todo si es muy grande.

Después de obtener los contornos los separamos en INTERIORES (serán las casillas de dentro del sudoku y algún contorno interior de un número, pero nos interesan los primeros) y EXTERIORES (serán los contornos de los números). En este caso la separación es fácil, basta con tomar la orientación positiva o negativa como se vio en los ejemplos de prácticas. Se ordenan las casillas

de izquierda a derecha y arriba a abajo y se recorren los contornos de los números para determinar qué número está dentro de cada casilla. Así se obtiene un array de 9×9 en el que cada contorno de número está asignado en la posición de la casilla en la que está contenido. Con esto tenemos la estructura básica del sudoku. Sabemos qué casillas están llenas y cuáles no.

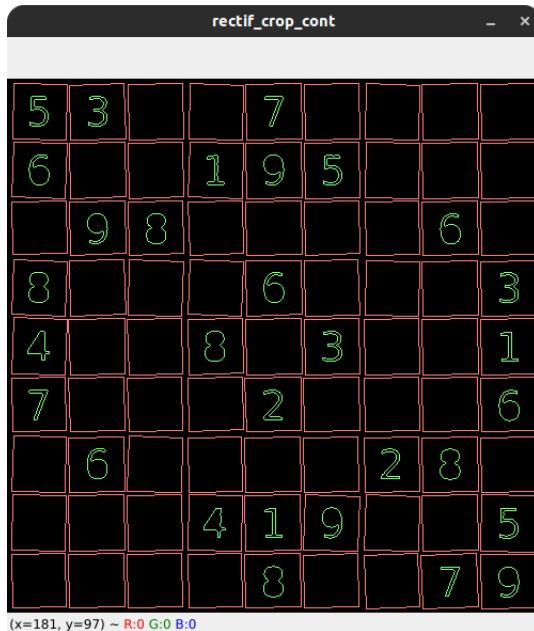


Figura 54: Detección de contornos de las casillas y los números del sudoku.

12.c. Identificación de los números

El siguiente paso sería tomar cada casilla rellena y reconocer de qué número en cuestión se trata. Para ello emplearemos el análisis frecuencial de contornos, de manera similar a cómo se estudió en clase, en especial nos basaremos en el programa *trebol5.py* visto en las prácticas de la asignatura.

La idea es introducir un modelo de cada uno de los números del 1 al 9 [55] y generar el invariante de cada uno. Después se procesa cada uno de los contornos de los números, se calcula su invariante y se compara con los invariantes de los modelos de los números. Esto es, se está realizando la comparación en el *espacio de las frecuencias*. Como observación, he tenido que realizar varios ensayos hasta encontrar la máxima distancia admisible entre el modelo y el contorno. Con la distancia de 0,15 tomada inicialmente, había problemas puesto que no era capaz de distinguir entre el 2 y el 5. Finalmente, determiné que el valor ideal era MAXDIST = 0,08.

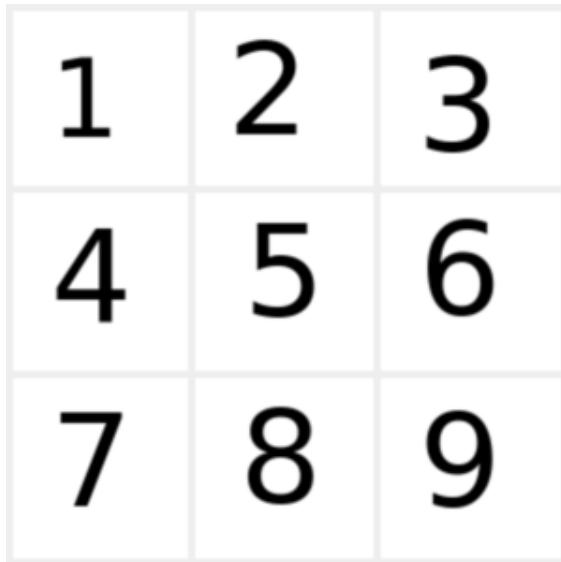


Figura 55: Modelos empleados para el reconocimiento de números mediante análisis frecuencial.

En este punto considero interesante analizar un problema inesperado que encontré. El programa funcionaba perfectamente con todos los números, pero al llegar al contorno del número 6, lo detectaba dos veces, una como el número 6 y otra con el número 9. Lo mismo ocurría con el contorno del número 9, que se detectaba como los dos números. Tras razonar un poco resulta evidente que el problema surgía debido a que al calcular el invariante en el análisis frecuencial, se obtiene el mismo independientemente de la rotación (de ahí el nombre), y al ser el 6 una rotación del 9, ambos números generan el mismo invariante y por tanto son indistinguibles en el espacio de las frecuencias.

Esto, que en otros contextos resulta de lo más útil (reconocer contornos del modelo, aunque se encuentren rotados en cualquier ángulo), supone aquí un problema importante. Ante este problema valoré emplear técnicas más potentes como modelos de machine learning entrenados para el reconocimiento de números. Sin embargo, como quería aprovechar la eficiencia que ofrece el método del análisis de las frecuencias con la *fft* (*Fast Fourier Transform*), opté por implementar una técnica heurística para distinguir los dos números. La heurística se basa en la distribución de píxeles del contorno. Se calcula el centroide (*cv.moments()*) y nos quedamos con la componente vertical del centroide) y se sigue el siguiente razonamiento: en el número 6, el centroide estará desplazado hacia la parte superior del contorno (el “*rabo*” contiene la mayor cantidad de píxeles y está hacia arriba). En el caso del 9 será al contrario, el centroide se encontrará en la mitad inferior del contorno.

```

def is_6_or_9(contour):
    # centroide del contorno
    M = cv.moments(contour)
    cy = int(M['m01']/M['m00'])

    # si es un 6, el centroide estará en la parte superior
    # si es un 9, el centroide estará en la parte inferior
    # calculamos la distribución de píxeles en el contorno
    _, y, _, h = cv.boundingRect(contour)
    center_y = y + h / 2
    if cy > center_y:
        return 6
    else:
        return 9

```

Figura 56: Función para distinguir entre el número 6 y el número 9 a partir de sus contornos.

Esta pequeño razonamiento nos permite solucionar el problema sin realizar cálculos costosos. Como observación, estamos basando el concepto de “arriba” y “abajo” en la orientación definida por el borde exterior del sudoku, considerando que el orden de las esquinas en la imagen se corresponde con el orden de las esquinas en la realidad (lo que es lo mismo, no existe una rotación de más de 90°). Podríamos superar este defecto si situamos algún tipo de marcador junto al sudoku que nos permita fijar de manera inequívoca la orientación, pero esto lo dejo como una propuesta de mejora del ejercicio.

12.d. Resolución del sudoku y muestra de resultados

Una vez identificado cada número, hemos alcanzado una matriz en la que cada casilla rellena tiene el número correspondiente y las casillas vacías tienen un 0. A partir de aquí, resolver el sudoku es una tarea sencilla, se trata de un problema de satisfacción de restricciones (*csp*) sin ninguna particularidad, por lo que no entrará en detalles.

La solución del sudoku será un array 9×9 de números, de forma que iteraremos sobre las casillas vacías del sudoku original y dibujamos en ellas el número correspondiente. Lo dibujamos primero en la imagen rectificada (tenemos las coordenadas del contorno de cada casilla, así que podemos situar fácilmente el número en el centro). El último paso será revertir la transformación de rectificación de la imagen (es decir, aplicar la transformación homográfica *inversa*) a la imagen con las soluciones y obtendremos la imagen original con las soluciones superpuestas, y el ejercicio quedaría resuelto [58].

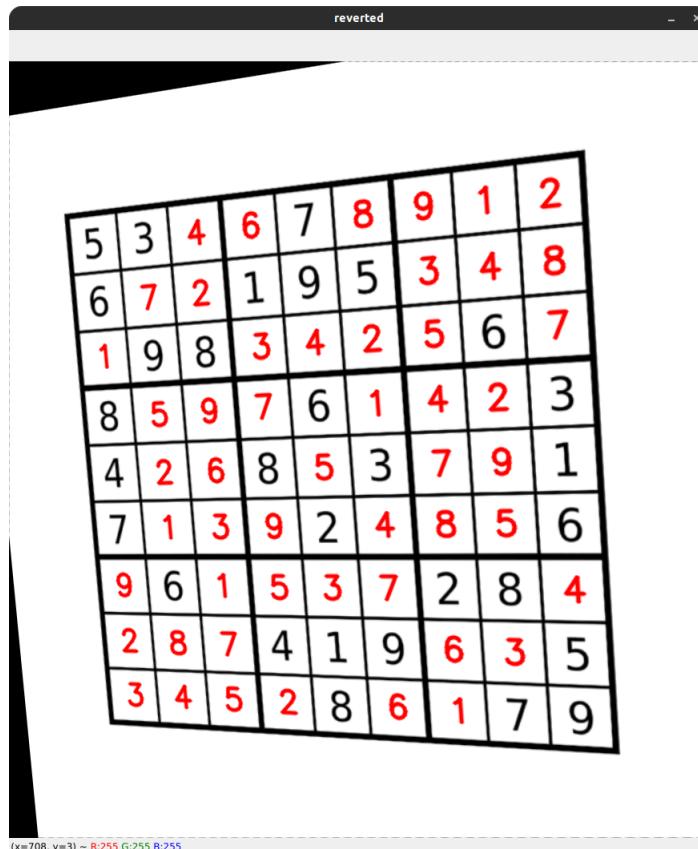


Figura 58: Sudoku resuelto tras deshacer la transformación homográfica de rectificación.

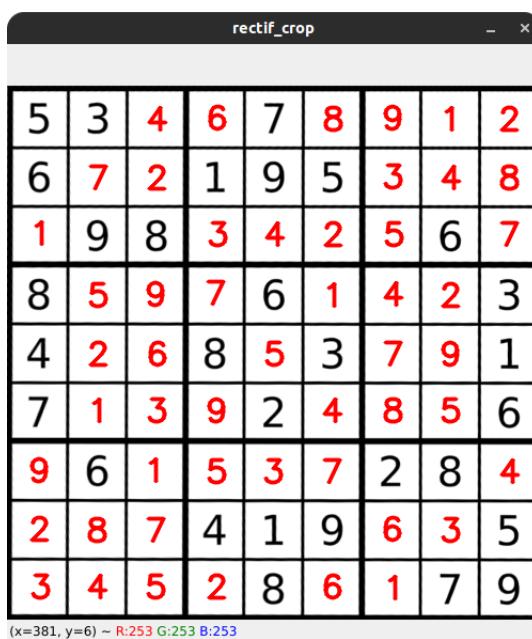


Figura 57: Sudoku resuelto.

Comentarios finales: Al tratarse de un ejercicio largo y relativamente complejo, he ido en-

contrando diversos matices que podrían mejorarse. Por ejemplo, toda la detección de contornos depende en gran medida de los parámetros de sensibilidad escogidos (qué se considera un contorno razonable, sensibilidad para aproximar un contorno por otro de menos puntos, cuánto hay que escalar la imagen, etc.). Tras experimentar con distintas configuraciones, he determinado unos valores para todos estos parámetros que considero adecuados (en combinación con alguna heurística sencilla) y que funcionan bien en la mayoría de los casos, como siempre bajo condiciones controladas y favorables (imágenes de gran tamaño y resolución, que se ven bien, en condiciones de luz óptimas, sin sombras pronunciadas y con un ángulo de perspectiva razonable entre otros). Sin embargo, en casos más extremos puede fallar. Podría implementarse algún tipo de selector para que el usuario introduzca estos valores de manera dinámica, o bien heurísticas más complejas en función del análisis de la imagen. Todo esto se plantea como una posible mejora a partir del estado actual del programa.

13. Conclusión

Realizar todos estos ejercicios me ha permitido interiorizar por completo la materia de la asignatura. Es la primera asignatura que he cursado con esta metodología tan práctica y de aplicación de todos los conceptos estudiados y creo que es todo un acierto.

He disfrutado cada ejercicio, pensando las distintas soluciones, buscando mis propias aproximaciones y creo que he aprendido no solo visión artificial sino a plantear y resolver distintos problemas que bien podría encontrarme en el mundo real.

Solamente me gustaría haber tenido más tiempo para poder perfeccionar los ejercicios y la memoria e intentar todos los ejercicios, pero en general ha sido una experiencia muy positiva e instructiva.

Referencias

- [1] Consejo Superior de Deportes. *BLC1: El campo de juego*. https://www.csd.gob.es/sites/default/files/media/files/2018-10/blc1_el_campo_de_juego.pdf. Oct. de 2018.
- [2] Consejo Superior de Deportes. *Normas NIDE 1-22*. Sitio web. Recuperado de <https://www.csd.gob.es/es/csd/instalaciones/politicas-publicas-de-ordenacion/normativa-tecnica-de-instalaciones-deportivas/normas-nide/nide-1-22>. 2023.
- [3] Gobierno de España. “Real Decreto 293/2000, de 18 de febrero, sobre las condiciones de accesibilidad y no discriminación de las personas con discapacidad en sus relaciones con la Administración General del Estado”. En: *Boletín Oficial del Estado* 72.47 (2000), págs. 7462-7472. url: <https://www.boe.es/buscar/doc.php?id=BOE-A-2000-16805>.
- [4] Euclid. *The Elements*. Vol. 1. Heath’s Mathematics Series. Book 3, Proposition 20. Cambridge: W. Heffer y Sons Ltd., 1956.
- [5] Guido Gerig. *CS7960-AdvImProc-FEV3-4*. <http://www.sci.utah.edu/~gerig/CS7960-S2010/handouts/CS7960-AdvImProc-FEV3-4.pdf>. Accessed: March 8, 2023. 2010.
- [6] *OpenCV Documentation - Image Filtering*. https://docs.opencv.org/4.x/d4/d86/group__imgproc__filter.html#gaabe8c836e97159a9193fb0b11ac52cf1. Accessed: March 8, 2023. 2023.
- [7] Unknown. *Using Homography for Pose Estimation in OpenCV*. <https://medium.com/analytics-vidhya/using-homography-for-pose-estimation-in-opencv-a7215f260fdd>. Fecha de acceso: 30 de mayo de 2023.
- [8] Wikipedia. *Inscribed Angle — Wikipedia, The Free Encyclopedia*. [Online; accessed 8-May-2023]. 2023. url: https://en.wikipedia.org/wiki/Inscribed_angle.