

Universidad de Murcia

Grado en Ingeniería Informática

Visión Artificial

Ejercicios - Entrega Parcial

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

Profesor: ALBERTO RUIZ GARCÍA

Índice

Índice de figuras	2
1 Introducción:	3
2 Calibración:	4
2.a Realiza una calibración precisa de tu cámara mediante múltiples imágenes de un <i>chessboard</i> .	4
2.b Haz una calibración aproximada con un objeto de tamaño conocido y compara con el resultado anterior	5
2.b.1 Comparación de la calibración precisa y la calibración aproximada	6
2.c Determina a qué altura hay que poner la cámara para obtener una vista cenital completa de un campo de baloncesto	6
2.d Haz una aplicación para medir el ángulo que definen dos puntos marcados con el ratón en la imagen	8
3 Actividad	9
3.a Utilizando un substractor de fondo de opencv como en los ejemplos <i>backsub0.py</i> y <i>backsub.py</i>	9
3.b Mediante un procedimiento sencillo que construya un modelo de fondo con frames anteriores y compare con el actual.	11
4 Color	14
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	14
5 Filtros	16
5.a Comprueba la propiedad de “cascading” del filtro gaussiano	18
5.b Comprueba la propiedad de “separabilidad” del filtro gaussiano	20
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	21
5.d Añade la posibilidad de seleccionar varios filtros para aplicarlos sucesivamente	22

Índice de figuras

1	Resultados de la calibración precisa.	4
2	Objeto de referencia para la calibración aproximada.	5
3	Resultados de la calibración aproximada.	6
4	Diagrama del ejercicio, pista de baloncesto y cámara con vista cenital	7
5	Ejecución del programa <i>pista.py</i> .	7
6	Ejemplo de ejecución en una imagen. Se seleccionan los dos puntos y se calcula el ángulo entre ellos.	8
7	Detector de actividad con OpenCV. Antes de entrar al ROI.	10
8	Detector de actividad con OpenCV. En el ROI, detecta cambios.	10
9	Detector de actividad con OpenCV. Al salir del ROI.	11
10	Detector de actividad manual. Antes de entrar al ROI.	12
11	Detector de actividad manual. En el ROI, detecta cambios.	13
12	Detector de actividad manual. Al salir del ROI.	13
13	Imagen empleada para las pruebas del código.	14
14	Prueba 1. Seleccionamos las piezas de color morado oscuro.	15
15	Prueba 2. Seleccionamos las piezas de color marrón.	15
16	Ejemplo de ejecución con el filtro <i>gaussian</i> con $\sigma = 3$.	17
17	Ejemplo de ejecución con el filtro <i>maximum (max)</i> .	17
18	Imagen base para comprobar las propiedades del filtro gaussiano.	18
19	Resultados de aplicar el filtro gaussiano en cascading y en 1 solo paso.	19
20	Diferencias entre cascading y aplicación en 1 paso.	19
21	Aplicación de un filtro gaussiano primero en una dirección y luego en otra y aplicación del mismo filtro 2D.	20
22	Diferencias entre los dos métodos anteriores.	21
23	Comparación entre la convolución “manual” y la función de convolución de librería. Imagen de ejemplo.	22
24	Comparación entre la convolución “manual” y la función de convolución de librería. Imagen de gran tamaño.	22
25	Aplicación de varios filtros.	23

1. Introducción:

En este informe se describe en detalle la resolución de los ejercicios de la entrega parcial 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*.

No se han realizado todos los subapartados opcionales para esta entrega parcial por falta de tiempo. La intención es hacer los que faltan para la entrega final, ampliando lo que se muestra en este documento.

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 (*píxeles*)
 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.

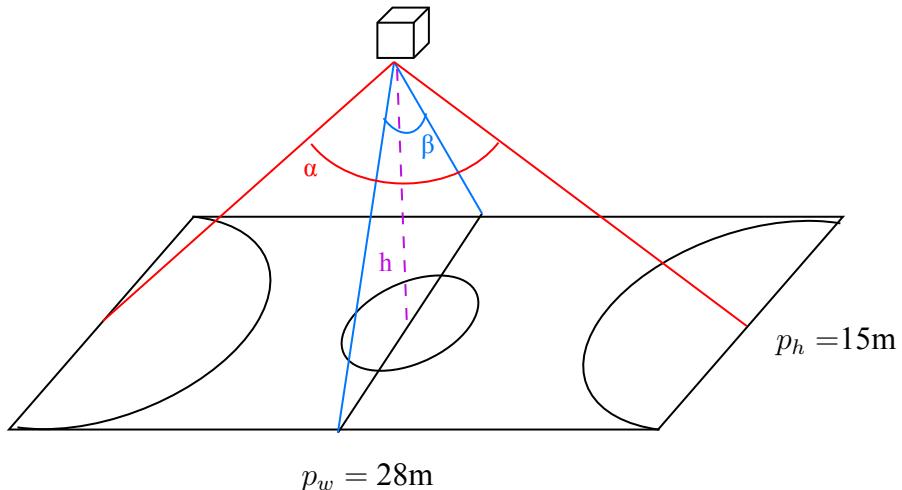


Figura 4: Diagrama del ejercicio, pista de baloncesto y cámara con vista cenital

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:

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

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.
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.

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_subtractor*). Posteriormente, una vez que tenemos definido el ROI, se aplica el extractor de fondo *bg_subtractor.apply()* para eliminar el fondo del ROI, y emplearlo como una máscara para tomar los píxeles distintos del fondo en el ROI.

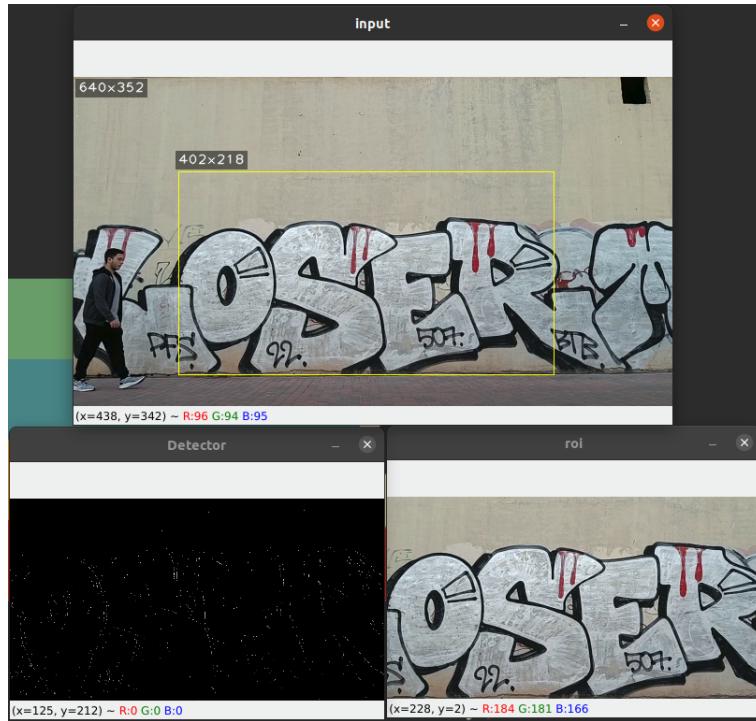


Figura 7: Detector de actividad con OpenCV. Antes de entrar al ROI.

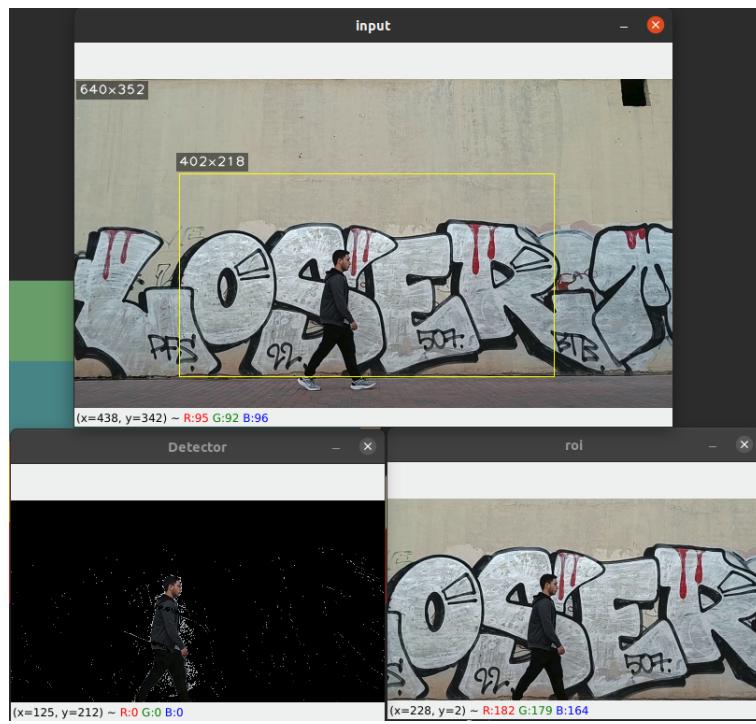


Figura 8: Detector de actividad con OpenCV. En el ROI, detecta cambios.

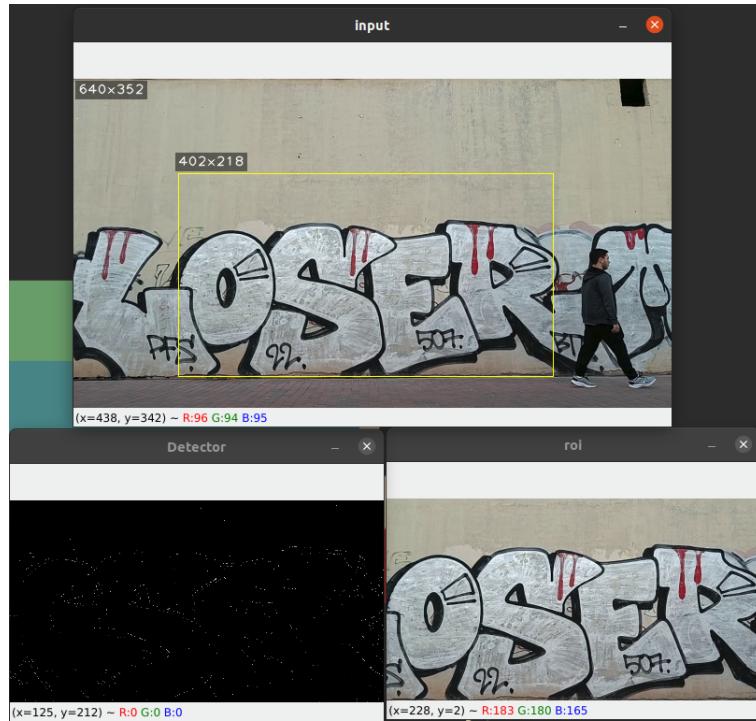


Figura 9: Detector de actividad con OpenCV. Al salir del ROI.

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.VideoWriter_fourcc()* para escoger el codec y *cv.VideoCapture()* 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:

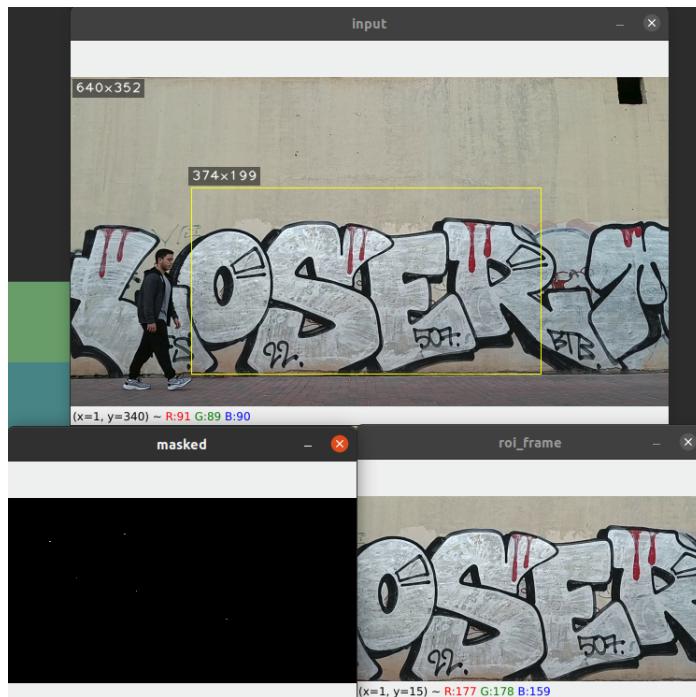


Figura 10: Detector de actividad manual. Antes de entrar al ROI.

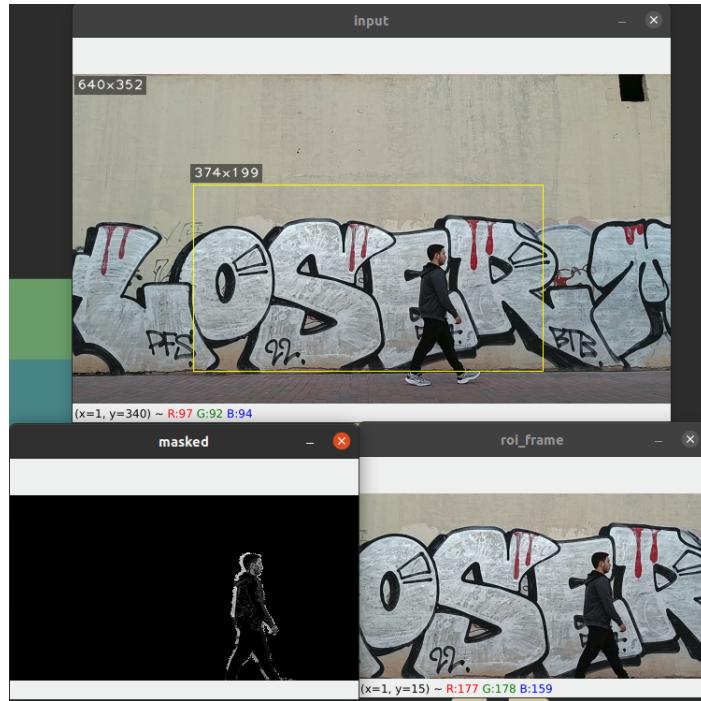


Figura 11: Detector de actividad manual. En el ROI, detecta cambios.

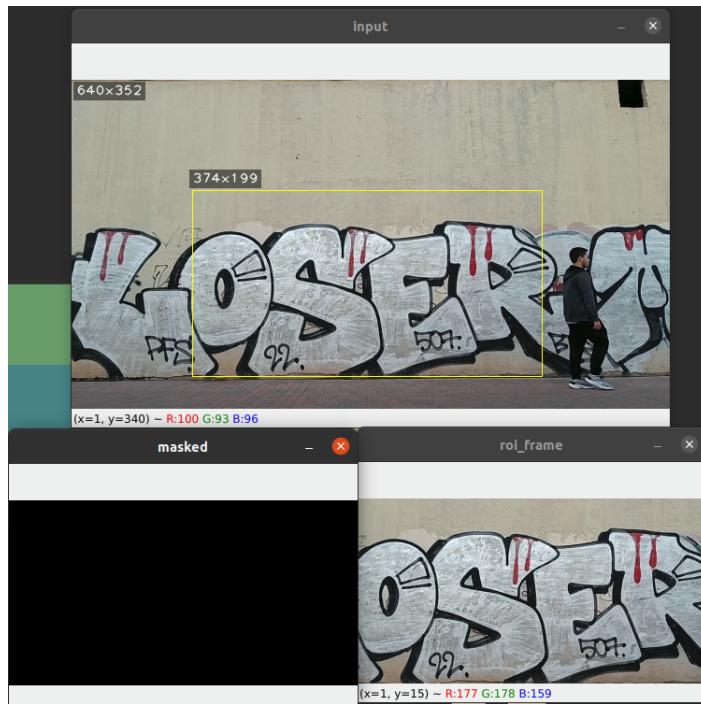


Figura 12: Detector de actividad manual. Al salir del ROI.

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 13: Imagen empleada para las pruebas del código.

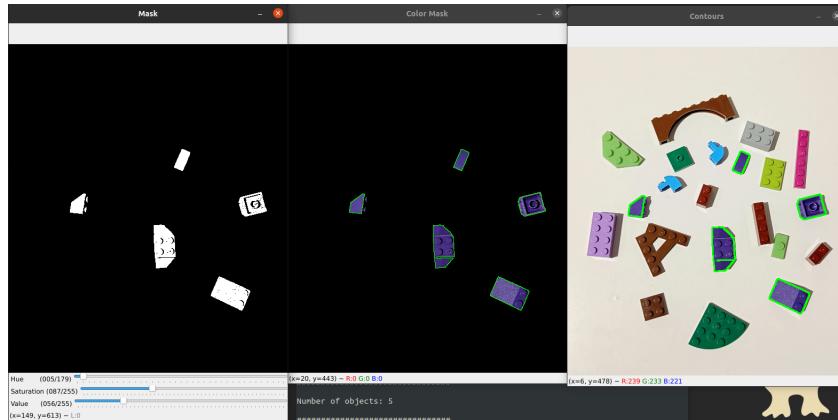


Figura 14: Prueba 1. Seleccionamos las piezas de color morado oscuro.

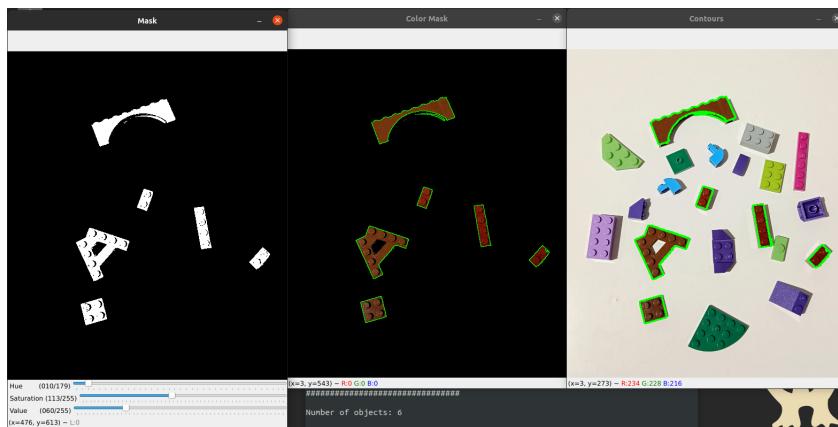


Figura 15: Prueba 2. Seleccionamos las piezas de color marrón.

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.

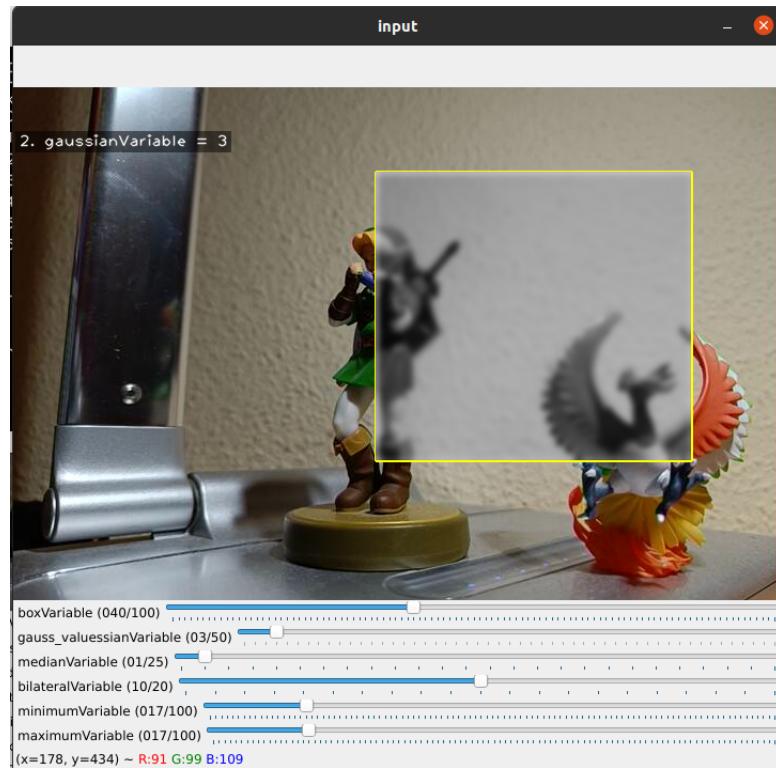


Figura 16: Ejemplo de ejecución con el filtro *gaussian* con $\sigma = 3$.

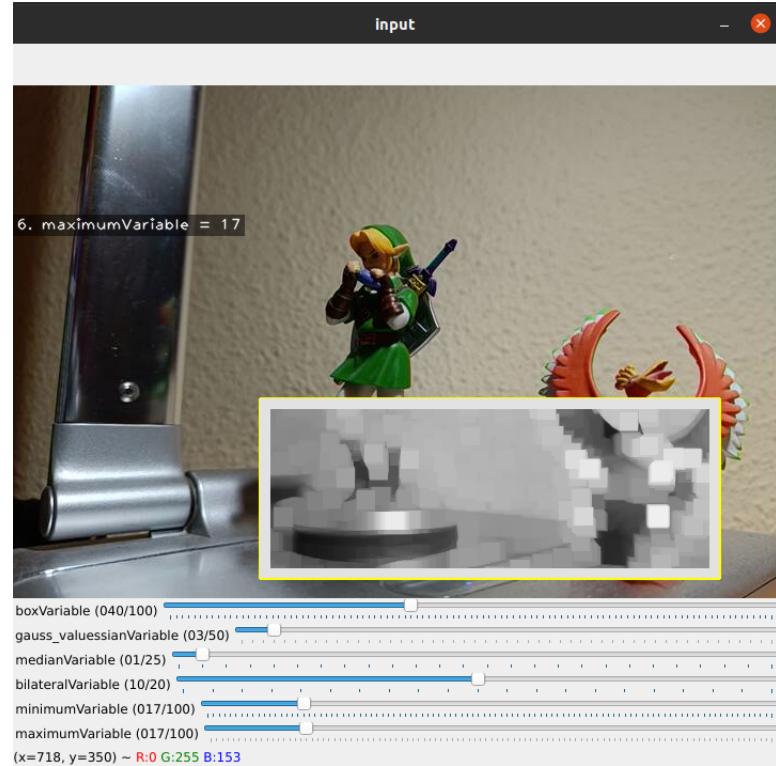


Figura 17: Ejemplo de ejecución con el filtro *maximum (max)*.

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 $\sigma_3 = \sqrt{\sigma_1^2 + \sigma_2^2}$ ([2] 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 18: Imagen base para comprobar las propiedades del filtro gaussiano.

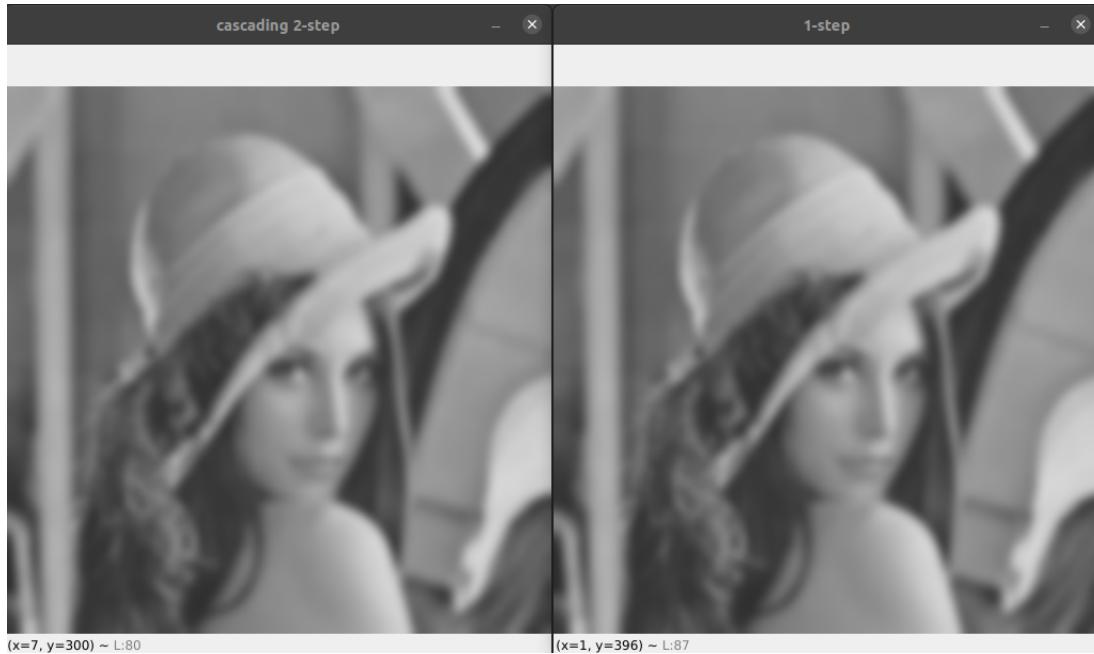


Figura 19: Resultados de aplicar el filtro gaussiano en cascading y en 1 solo paso.

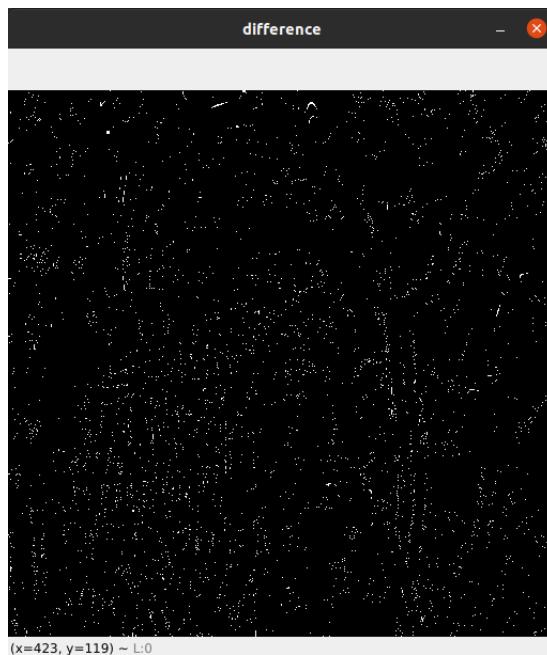


Figura 20: Diferencias entre cascading y aplicación en 1 paso.

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 ([2] 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 ([3]).

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.

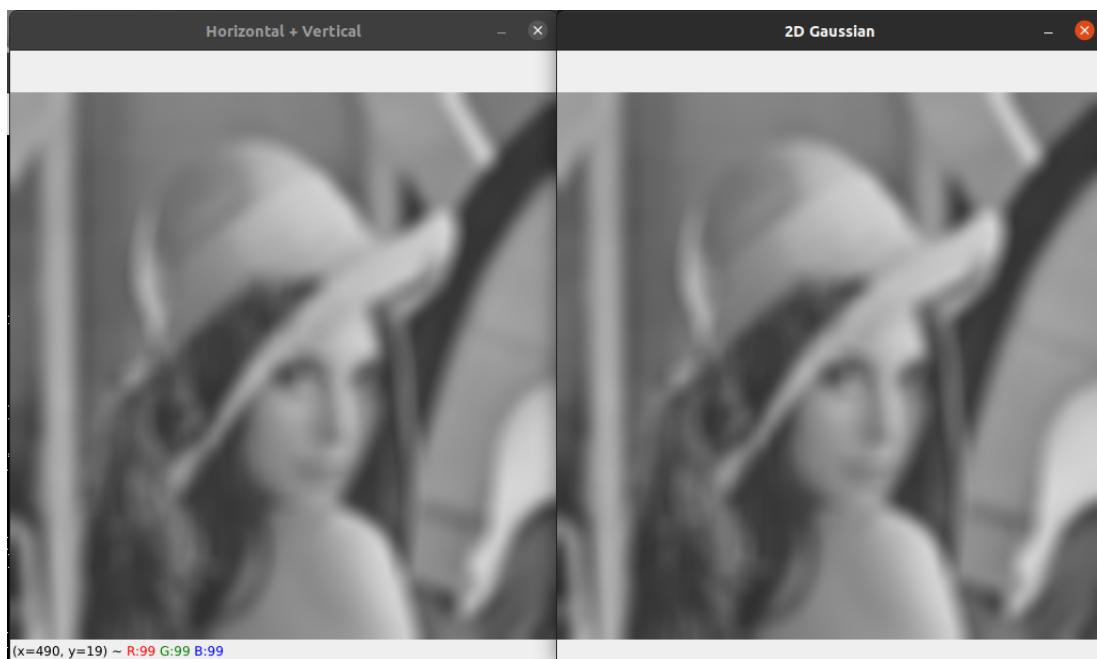


Figura 21: Aplicación de un filtro gaussiano primero en una dirección y luego en otra y aplicación del mismo filtro 2D.



Figura 22: 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 23: 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 24: 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.

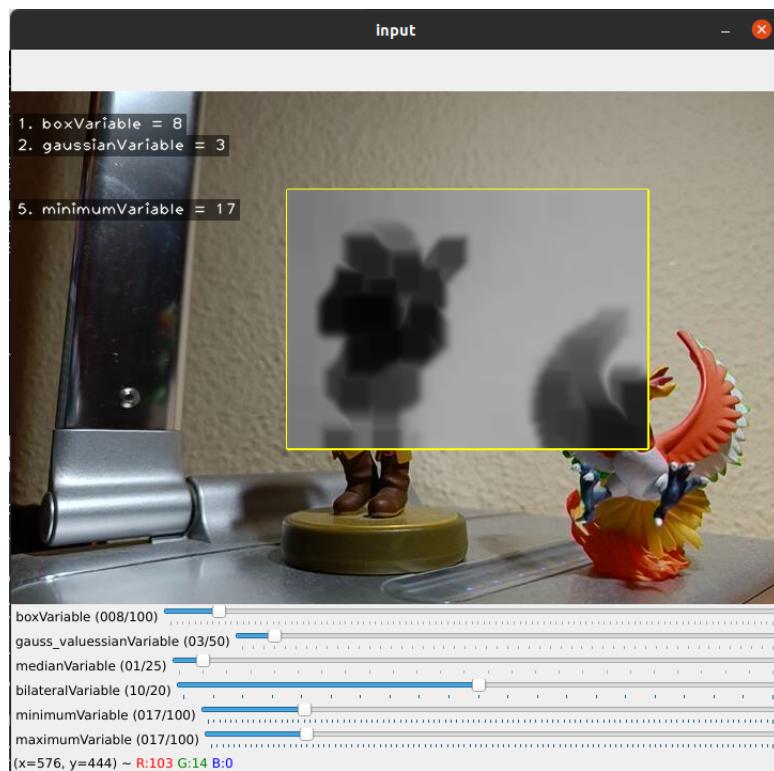


Figura 25: Aplicación de varios filtros.

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] 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.
- [3] *OpenCV Documentation - Image Filtering*. https://docs.opencv.org/4.x/d4/d86/group__imgproc__filter.html#gaabe8c836e97159a9193fb0b11ac52cf1. Accessed: March 8, 2023. 2023.