UNIVERSIDAD
DE GRANADA

Visión por Computador
grado en ingeniería informática

Trabajo 1

Filtrado y Detección de Regiones

Autor
Vladislav Nikolov Vasilev

Rama
Computación y Sistemas Inteligentes

Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación

Curso 2019-2020

Índice

1. Ejercicio sobre filtros básicos

USANDO LAS FUNCIONES DE OPENCV: escribir funciones que implementen los siguientes puntos:

A) El cálculo de la convolución de una imagen con una máscara 2D. Usar una Gaussiana 2D (GaussianBlur) y máscaras 1D dadas por getDerivKernels). Mostrar ejemplos con distintos tamaños de máscara, valores de sigma y condiciones de contorno. Valorar los resultados.

B) Usar la función Laplacian para el cálculo de la convolución 2D con una máscara normalizada de Laplaciana-de-Gaussiana de tamaño variable. Mostrar ejemplos de funcionamiento usando dos tipos de bordes y dos valores de sigma: 1 y 3.

1.1. Apartado A

Para realizar todas las pruebas, vamos a utilizar una imágen en blanco y negro. De esta forma, los resultados se podrán ver de forma más clara. Se puede utilizar cualquiera de las imágenes que se han proporcionado, pero vamos a utilizar la foto del gato. A continuación se puede ver la imágen en cuestión:

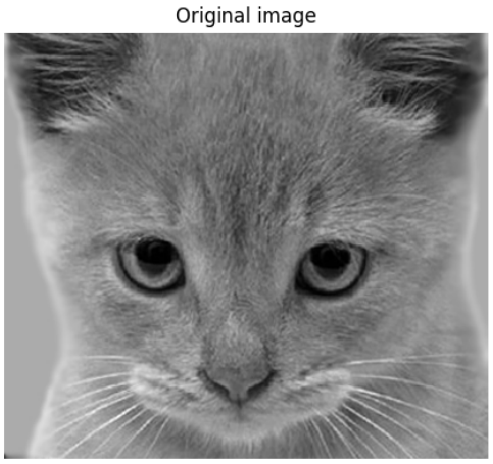


Figura 1: Imágen original del gato en blanco y negro.

Antes de ver cuáles son los resultados y compararlos. Hace falta aclarar algunos puntos:

- Cuando se cargan las imágenes, se convierten a *float64*, para así tener más precisión a la hora de hacer las posteriores operaciones (aplicar filtros, sumar o restar imágenes, etc.), además de que así no nos limitamos a usar solo valores en el rango $[0, 255]$.
- Las operaciones para aplicar filtros de OpenCV aplican una correlación. Por tanto, para aplicar una convolución, se debe realizar un *flip* del *kernel*; esto es, darle la vuelta en el eje X y en el eje Y. Como en todos los casos trabajaremos con *kernels* separables, solo será necesario darles la vuelta en un sentido.
- Respecto al punto anterior, al trabajar con *kernels* separables, con tal de mejorar la eficiencia, se puede aplicar primero el *kernel* del eje X y, sobre el resultado, aplicar el *kernel* en el eje Y. Esto es mucho más rápido que aplicar directamente un *kernel* 2D sobre la imágen, debido a que el número de operaciones es mucho menor.

Como en este ejercicio se piden aplicar *kernels* de Gaussiana y de derivada, se han hecho funciones específicas las cuáles pueden ser consultadas en el código. Estas funciones son *gaussian_kernel()* y *derivative_kernel()*, respectivamente. No se va a especificar exactamente lo que hacen debido a que solo obtienen los *kernels* con los parámetros adecuados (valores de σ y tamaño de cada *kernel* para el caso de la Gaussiana, y tamaño del *kernel* y número de veces que se deriva en cada eje en el caso de la derivada).

Lo que sí que es importante destacar es que estas dos funciones utilizan una común para realizar la convolución. Como se dijo anteriormente, OpenCV realiza como tal la correlación, pero se puede aplicar una convolución realizando unos pocos cambios. La función que se puede ver a continuación recibe un *kernel* para cada eje y aplica el filtro mediante la convolución:

```
def apply_kernel(img, kx, ky, border):
    """
    Funcion que aplica un kernel separable sobre una imagen,
    realizando una convolucion

    Args:
        img: Imagen sobre la que aplicar el filtro
        kx: Kernel en el eje X
        ky: Kernel en el eje Y
        border: Tipo de borde
    Return:
        Devuelve una imagen filtrada
    """
```

```
14      # Hacer el flip a los kernels para aplicarlos como una
        convolucion
15      kx_flip = np.flip(kx)
16      ky_flip = np.flip(ky)
17
18      # Realizar la convolucion
19      conv_x = cv.filter2D(img, cv.CV_64F, kx_flip.T,
20                           borderType=border)
21      conv = cv.filter2D(conv_x, cv.CV_64F, ky_flip,
22                         borderType=border)
23
24      return conv
```

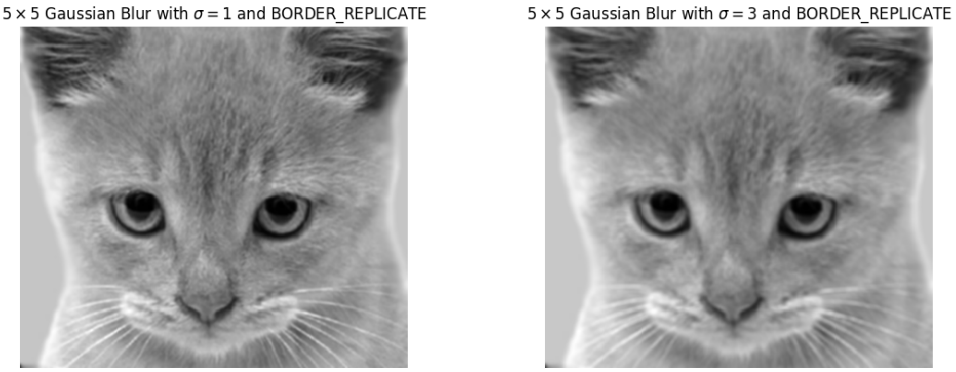
Listing 1: Función que aplica un filtro separable.

Como se puede ver, se tiene que hacer un *flip* de los *kernels* para poder aplicar una convolución. Al aplicar cada *kernel* de forma separada mediante $filter2D$, se consigue una mayor eficiencia (como se ha mencionado anteriormente). Es importante destacar que primero se aplica el *kernel* sobre el eje de las X y, sobre el resultado obtenido, se aplica el *kernel* en el eje Y. También es importante destacar que, cuando se aplica el *kernel* sobre el eje X, se le pasa la traspuesta del *kernel*. Esto se debe a que OpenCV proporciona los *kernels* como vectores columnas, y para pasarlo por las filas, necesitamos que sea un vector fila. Al traponer el vector columna obtenemos, como parece lógico, un vector columna.

Otra cosa muy importante a destacar son los *kernels* que se van a utilizar. Uno de ellos es el *kernel* Gaussiano, el cuál es simétrico tanto en el eje X como en el Y. Al aplicarlo, por tanto, se podría ahorrar la operación del *flip*, ya que lo va a dejar igual. Sin embargo, debido a que se ha implementado la función para que sea genérica, se realiza esta operación. El otro *kernel* que se utiliza es el de las derivadas, que no es más que un *kernel* de Sobel, el cuál combina alisamiento Gaussiano con la derivada. A diferencia del anterior *kernel*, este no es simétrico, y por tanto, la operación de *flip* no lo va a dejar igual; por tanto, no puede ser ahorrada en este caso.

Con esto comentado, ya podemos empezar a hablar de los resultados que se han obtenido. Para ello, vamos a comenzar comentando los resultados que se obtienen para el filtro Gaussiano.

Lo primero que se ha probado es un *kernel* de 5×5, variando los valores de σ para ver cómo cambiaba. A continuación se pueden ver los resultados:

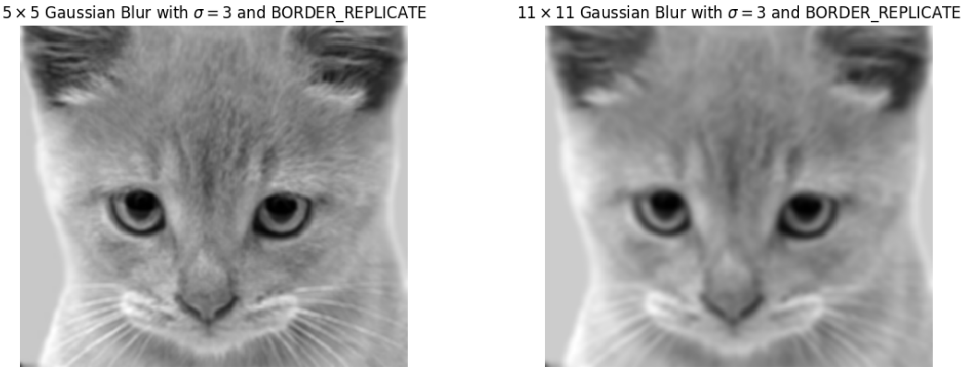


(a) Filtro Gaussiano con $\sigma = 1$. (b) Filtro Gaussiano con $\sigma = 3$.

Figura 2: Filtro Gaussiano con diferentes tamaños de σ.

Como se puede ver, comparando cualquiera de las imágenes de la figura 2 con 1, existen ciertas diferencias. Podemos ver claramente como al aplicar el filtro Gaussiano se ha perdido cierto detalle, ya que se están eliminando las frecuencias altas. Además, si comparamos las figuras 2a y 2b, podemos ver que a medida que aumentamos el tamaño de σ, se van perdiendo más detalles, ya que se emborrona más.

Se ha probado también a variar el tamaño de la máscara, conservando el mismo valor de σ, para ver qué es lo que sucede. A continuación, se puede ver lo que se ha obtenido:

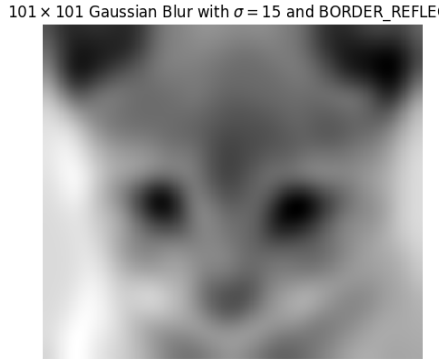


(a) Filtro Gaussiano de tamaño 5×5. (b) Filtro Gaussiano de tamaño 11×11.

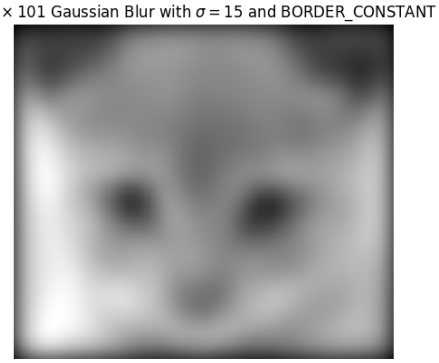
Figura 3: Filtro Gaussiano con diferentes tamaños de *kernel*.

Tal y como se puede ver en la figura 3, al modificar el tamaño del *kernel* se han obtenido diferentes resultados. Esto se debe a que más píxels forman parte de la máscara (como es obvio), y por tanto, se tiene en cuenta una mayor parte de la Gaussiana. Por tanto, existe cierta dependencia entre esto dos valores (el tamaño del *kernel* y el valor de σ). A mayor σ, mayor tendrá que ser el tamaño de la máscara, para así poder modelizar mejor el comportamiento de la función con los píxels vecinos. Si no se adaptase el tamaño del *kernel*, se tendría que solo se coge una parte pequeña de la función Gaussiana más cercana al centro, ignorando el comportamiento de los píxels más lejanos. Tampoco hay que coger un tamaño demasiado grande si el σ es pequeño, ya que entonces habrán muchos píxels que tendrán un valor muy próximo a 0 (aquellos que estén más alejados). En resumen, que hay escoger de forma proporcional el tamaño del *kernel* y el σ.

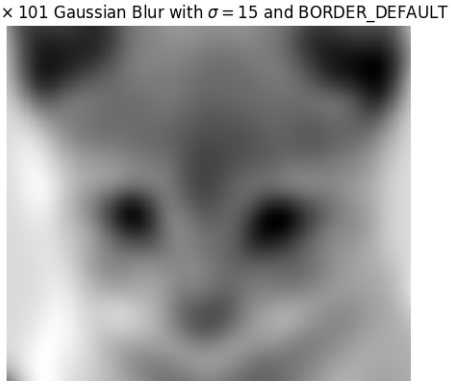
Para ver cómo afecta el tipo de borde a la imágen resultante, se han realizado una serie de pruebas variando el borde utilizado a la hora de aplicar los *kernels*. Para que los efectos fuesen notables, se ha tenido que escoger un tamaño de máscara mucho más grande a los utilizados anteriormente. Esto se debe a que con una máscara pequeña no se puede apreciar muy bien el resultado, debido a que no se coge mucha región fuera de los bordes de la imágen. El tamaño de la máscara es de 101×101, permitiendo que se salga lo suficiente de la imágen para ver la influencia. A continuación se pueden ver los resultados que se han obtenido:



(a) Borde reflejando la imágen.



(b) Borde constante.



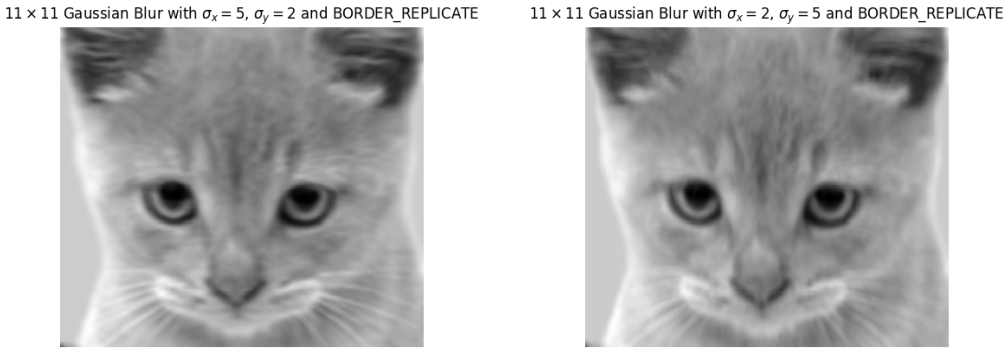
(c) Borde por defecto (reflejado sin contar el primer píxel).

Figura 4: Filtro Gaussiano con diferentes tipos de bordes.

Como se puede ver, existen ciertas diferencias dependiendo de qué tipo de borde se escoja. Si comparamos las figuras 4a y 4c con la figura 4b, se puede ver claramente que los resultados no son los mismos, ya que en este último caso se utiliza un valor constante para las partes más allá de los bordes de las imágenes. Por tanto, la imágen queda como si estuviese enmarcada.

Las diferencias entre las figuras 4a y 4c son mucho más sutiles, pero notables. Se puede ver como por ejemplo en la figura 4c la esquina inferior izquierda es más oscura, mientras que en la figura 4a es más clara. Aunque los dos tipos de borde reflejen la imágen, la forma en la que lo hacen es distinto. $BORDER\_DEFAULT$ no considera el píxel del borde, mientras que $BORDER\_REFLECT$ sí que lo hace. Todo esto se puede ver mejor explicado aquí.

Por último, y antes de pasar al *kernel* de derivada, se ha probado a variar el σ que se ha aplicado en cada eje, manteniendo constante el tamaño del *kernel*. A continuación se pueden ver cuáles han sido los rsultados obtenidos:

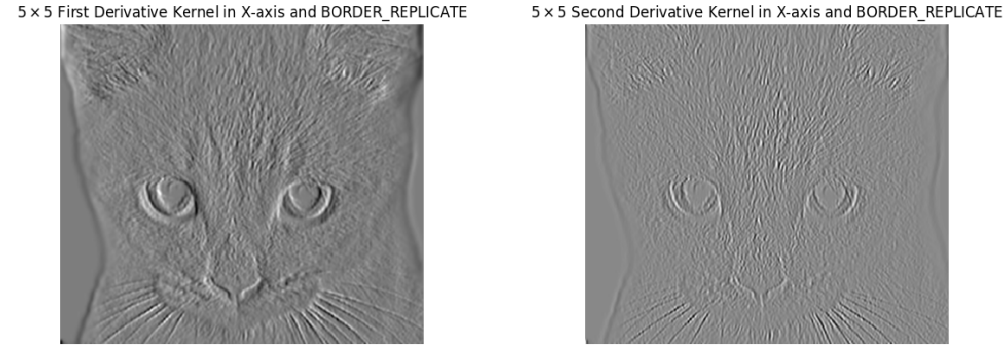


(a) Filtro Gaussiano con $\sigma_x = 5$ y $\sigma_y = 2$. (b) Filtro Gaussiano con $\sigma_x = 2$ y $\sigma_y = 5$.

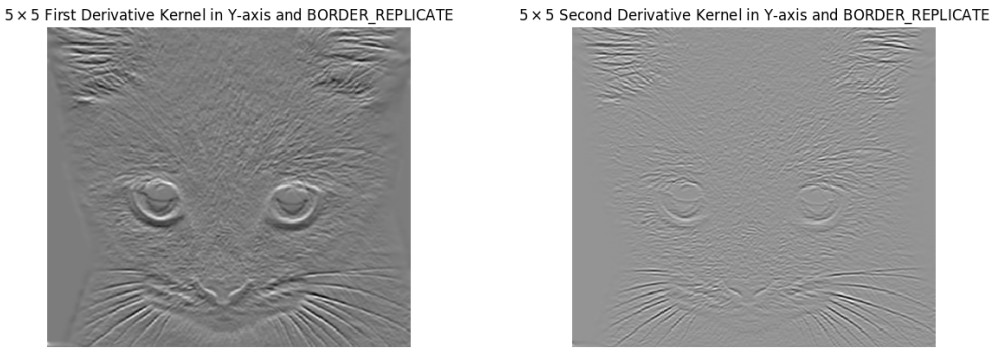
Figura 5: Filtro Gaussiano variando el σ en cada eje.

Como se puede ver claramente, hay diferencias, debido a que el alisamiento solo se ha hecho en uno de los ejes en vez de en los dos. Se puede ver como en la figura 5a el pelo de la frente del gato parece que va hacia la derecha, mientras que en la figura 5b el pelo de la frente parece que va hacia abajo. Además, en esta segunda imágen se puede ver como los bigotes se han emborronado más que en el primer caso, donde son más apreciables.

Con esto ya comentado, pasemos a hablar del *kernel* de las derivadas. Lo primero que podemos hacer es ver qué influencia tiene la derivada dependiendo del eje sobre el que se aplique y según el número de veces que se derive en cada uno de los ejes. Para ello, vamos a probar la primera y la segunda derivada en cada uno de los ejes de forma separada (esto es, que no derivaremos a la vez). A continuación se pueden ver los resultados:



(a) Filtro de primera derivada en el eje X. (b) Filtro de segunda derivada en el eje X.

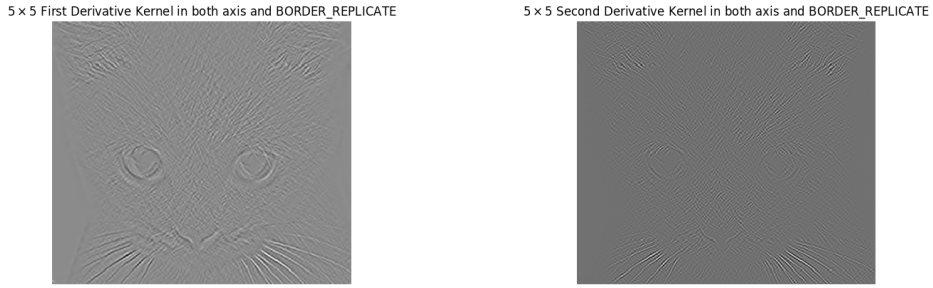


(c) Filtro de primera derivada en el eje Y. (d) Filtro de segunda derivada en el eje Y.

Figura 6: Filtro de derivada variando el número de diferenciaciones en cada eje.

Al pasarlo por el eje X, como se puede ver en las figuras 6a y 6b, se puede ver que se van quedando aquellos bordes verticales, ya que los horizontales se ven muy poco o nada, como es por ejemplo el caso del pelo de las orejas, el cuál casi no se ve, o algunos de los bigotes. En cambio, al pasarlo por el eje Y, tal y como se puede ver en las figuras 6c y 6d, se van quedando aquellos bordes que horizontales. Se puede ver que el pelo en las orejas se puede ver mejor, además de que algunos de los bigote están mejor definidos que en el caso anterior. Lo único que se va perdiendo son los bordes del gato como tal, haciendo que, a medida que se va derivando más en el eje Y, se distinga menos donde empieza el gato y donde está el fondo.

También se ha decidido estudiar qué efecto tiene pasar el *kernel* de las derivadas tanto en el eje X como en el Y a la vez, ya que la función $getDerivKernel$ permite especificar el número de diferenciaciones a realizar en cada eje. Los resultados que se han obtenido son los siguientes:

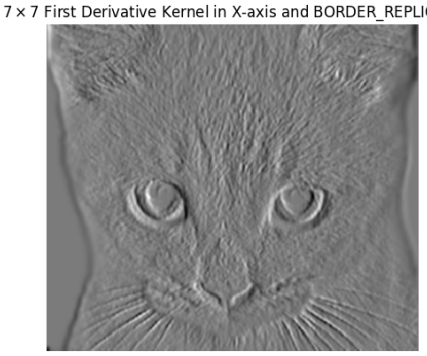


(a) Filtro de primera derivada en ambos ejes. (b) Filtro de segunda derivada en ambos ejes.

Figura 7: Filtro de derivada variando el número de diferenciaciones en los dos ejes.

A diferencia de lo que se ve en la figura 6, en la figura 7 se observa que aquellos bordes que destacan más son los que están en diagonal (es decir, una combinación de los dos ejes). Los horizontales y los verticales casi no se notan, como por ejemplo es el caso de los contornos del gato, ya que aquí es incluso más difícil o casi imposible distinguir donde empieza el gato y donde empieza el fondo.

Ahora, tal y como hicimos antes, pasemos a ver qué pasa si mantenemos el número de diferenciaciones constante y aumentamos el tamaño del *kernel*. Para ello, vamos a establecer que en todos los casos se obtiene la primera derivada en el eje X. Los resultados de estas variaciones del tamaño del *kernel* se pueden ver a continuación:



(a) Filtro de tamaño 7×7.

(b) Filtro de tamaño 11×11.

(c) Filtro de tamaño 15×15.

Figura 8: Filtro de primera derivada en el eje X variando el tamaño del *kernel*.

Como resultado más obvio, se puede ver que a medida que va aumentando el tamaño del *kernel*, se va emborronando más la imágen. Esto se debe a que uno de

los *kernels* de Sobel hace un suavizado, de forma que se va eliminando el ruido. A mayor tamaño del *kernel*, más se va a emborronar. Por tanto, aquellos bordes muy finos se van a ir perdiendo. Por ejemplo, si comparamos la figura 8a con la figura 8c, podemos ver que los bordes o contornos que se pueden ver en la primera figura en la frente del gato han ido desapariciendo o haciéndose menos notables en esta última. De esta forma, aumentar el tamaño del *kernel* parece que solo nos aporta más suaviazado. Un tamaño de *kernel* relativamente pequeño es suficiente para extraer los contornos.

Por último, vamos a ver como variando el tipo de borde cambia la imágen que obtenemos. Para ello, tal y como hicimos antes, vamos a fijar un tamaño de *kernel* relativamente grande para poder apreciar el efecto del borde. En este caso, estamos limitados a un tamaño máximo de 31, ya que OpenCV no permite obtener más. Por tanto, vamos a usar este tamaño de *kernel* a la hora de variar los bordes, ya que se ha considerado el más adecuado para la labor. A continuación se pueden ver los resultados:

(a) Borde reflejando la imágen.

(b) Borde constante.

(c) Borde por defecto (reflejado sin contar el primer píxel).

Figura 9: Filtro de primera derivada en el eje X variando el tipo de borde.

En este caso, sucede algo parecido a lo que se podía ver en la figura 4. Podemos ver que la figura más diferente de todas es la figura 9b. Sin embargo, no parece que haya mucha diferencia destacable o notable entre las figuras 9a y 9c, con lo cuál podríamos decir que permiten obtener un resultado más o menos parecido, sin demasiadas diferencias. Parece que en este caso el tipo de borde que más afecta es el constante, aquél en el que se supone un valor constante para todos los píxels fuera de la imágen. Por tanto, si utilizamos cualquier otro tipo de borde no notaríamos mucha diferencia, lo cuál no sucedía en el caso del filtro Gaussiano, ya que sí que había algunas diferencias, aunque fuesen muy pocas.

1.2. Apartado B

Antes de analizar los resultados, vamos a comentar brevemente cómo es la función de la Laplaciana de Gaussiana. El código es el que se puede ver a continuación:

```
def log_kernel(img, ksize, sigma_x, sigma_y, border):
    """
    Funcion que aplica un kernel LoG (Laplacian of Gaussian) sobre una imagen.

    Args:
        img: Imagen sobre la que aplicar el kernel
        ksize: Tamaño del kernel Gaussiano y Laplaciano
        sigma_x: Valor de sigma en el eje X de la Gaussiana
        sigma_y: Valor de sigma en el eje Y de la Gaussiana
        border: Tipo de borde
    Return:
        Devuelve una imagen sobre la que se ha aplicado un filtro LoG
    """

    # Aplicar filtro Gaussiano
    gauss = gaussian_kernel(img, ksize, ksize, sigma_x, sigma_y, border)

    # Obtener los filtros de derivada segunda en cada eje, aplicados sobre
    # el filtro Gaussiano
    dx2 = derivative_kernel(gauss, 2, 0, ksize, border)
    dy2 = derivative_kernel(gauss, 0, 2, ksize, border)

    # Combinar los filtros de derivadas y obtener Laplaciana de Gaussiana
    laplace = dx2 + dy2

    return laplace
```

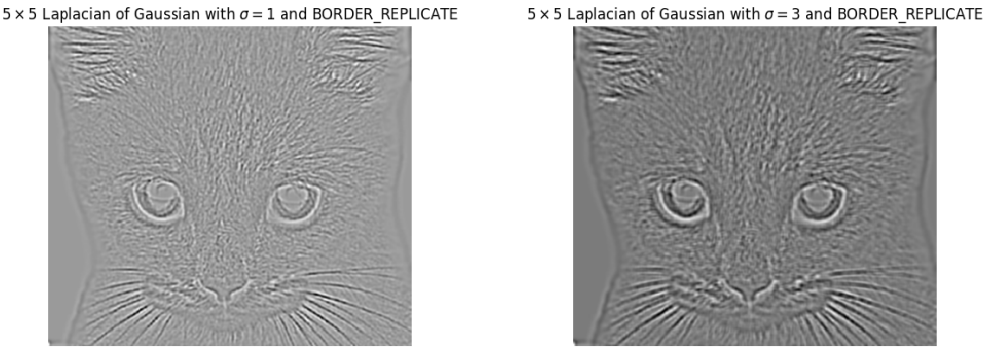
Listing 2: Función que aplica una Laplaciana de Gaussiana.

Lo primero que se hace es aplicar un filtro de Gaussiana sobre la imágen de entrada. Al ser éste un filtro de paso bajo, las altas frecuencias (y el ruido) se eliminan, quedando una imágen más suavizada. Después, sobre el resultado de aplicar el filtro, se obtienen los filtros de segunda derivada para cada eje. Lo único que hay que hacer al final es combinar la segunda derivada en el eje X y la segunda derivada en el eje Y. Estos dos últimos pasos (diferenciación y suma) se han seguido según la fórmula de la Laplaciana, la cuál es:

$$\texttt{dst} = \Delta\texttt{src} = \frac{\partial^2 \texttt{src}}{\partial x^2} + \frac{\partial^2 \texttt{src}}{\partial y^2} \tag{1}$$

Hay que tener en cuenta que el filtro de derivada tiene una parte que es también un filtro de alisamiento. Por tanto, al aplicarlo junto con la Gaussiana, se alisa más la imágen. Esto nos hace replantearnos el uso del filtro Gaussiano del principio. Sin embargo, también hay que considerar que el alisamiento de la derivada por sí solo no es muy grande. Por consiguiente, vamos a dejar el filtro Gaussiano del principio, aunque se acabe alisndo algo más de lo que se debe.

Una vez comentado esto, vamos a ver cuáles son los resultados que se han obtenido. Lo primero que se ha probado es a variar el tamaño del σ utilizado, utilizando en ambos casos un *kernel* de 5×5. Los resultados son los que se pueden ver a continuación:



(a) Filtro con $\sigma = 1$.

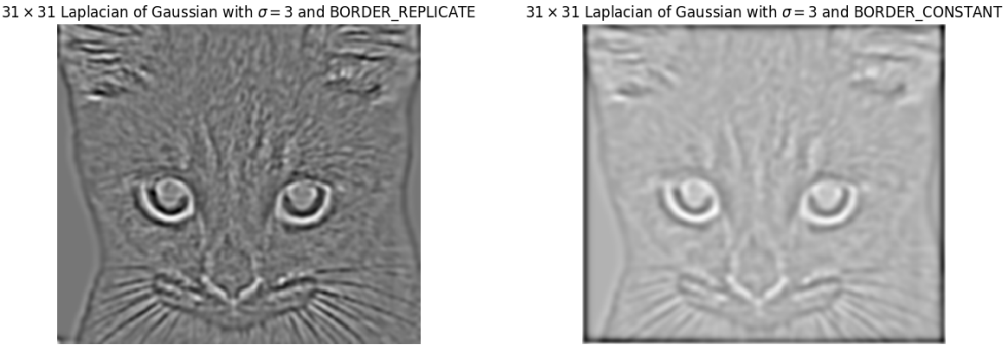
(b) Filtro con $\sigma = 3$.

Figura 10: Filtro de Laplaciana de Gaussiana con diferentes valores de σ.

Como se puede ver, al aumentar el tamaño de σ se obtiene un mayor suavizado (debido a la Gaussiana), además de que la imágen pierde intensidad debido a esto mismo. Además, se puede ver como los bordes son más gruesos. En la figura 10a los bordes en las orejas y los bigotes son mucho más finos que los que se

pueden ver en la figura 10b. Esto se debe a lo comentado anteriormente: un mayor emoborronamiento.

Como también se ha pedido, se ha probado a modificar el tipo de borde utilizado para ver qué efecto tiene sobre la imágen resultante. Para ello, tal y como se ha hecho anteriormente se ha utilizado un *kernel* de mayor tamaño, concretamente de 31. De esta forma, el efecto sería más apreciable. Se ha utilizado el mismo borde que se puede ver en la figura 10, y otro más. Además, se ha fijado un $\sigma = 3$ para los dos casos. Los resultados se pueden ver a continuación:



(a) Filtro replicando el píxel del borde. (b) Filtro con borde constante.

Figura 11: Filtro de Laplaciana de Gaussiana con diferentes valores de σ.

Claramente se pueden apreciar las diferencias. Se puede ver que la imágen que se ve en la figura 11b parece estar enmarcada, debido a que tiene los píxels de color negro en los bordes. Además, tiene una mayor intensidad que la que se puede ver en la figura 11a, aunque el precio a pagar ha sido que se ha perdido demasiado detalle por el camino.

2. Ejercicio sobre pirámides y detección de regiones

IMPLEMENTAR funciones para las siguiente tareas:

A) Una función que genere una representación en pirámide Gaussiana de 4 niveles de una imagen. Mostrar ejemplos de funcionamiento usando bordes y justificar la elección de los parámetros.

B) Una función que genere una representación en pirámide Laplaciana de 4 niveles de una imagen. Mostrar ejemplos de funcionamiento usando bordes.

C) Construir un espacio de escalas Laplaciano para implementar la búsqueda de regiones usando el siguiente algoritmo:

- **a.** Fijar sigma
- **b.** Repetir para N escalas
  - I. Filtrar la imagen con la Laplaciana-Gaussiana normalizada en escala
  - II. Guardar el cuadrado de la respuesta para el actual nivel del espacio de escalas
  - III. Incrementar el valor de sigma por un coeficiente k.(1.2-1.4)
- **c.** Realizar supresión de no-máximos en cada escala
- **d.** Mostrar las regiones encontradas en sus correspondientes escalas. Dibujar círculos con radio proporcional a la escala.

2.1. Apartado A

Para crear la pirámide Gaussiana, se ha utilizado la siguiente función:

```
def gaussian_pyramid(img, ksize, sigma_x, sigma_y, border, N=4):
    """
    Funcion que devuelve una piramide Gaussiana de tamaño N

    Args:
        img: Imagen de la que extraer la piramide
        ksize: Tamaño del kernel
        sigma_x: Valor de sigma en el eje X
        sigma_y: Valor de sigma en el eje Y
        border: Tipo de borde a utilizar
        N: Numero de imagenes que componen la piramide (default 4)

```

```
13      """
14      # Inicializar la piramide con la primera imagen
15      gaussian_pyr = [img]
16
17      for i in range(1, N):
18          # Obtener el elemento anterior de la piramide Gaussiana
19          prev_img = np.copy(gaussian_pyr[i - 1])
20
21          # Aplicar Gaussian Blur
22          gauss = gaussian_kernel(prev_img, ksize, ksize, sigma_x,
    sigma_y, border)
23
24          # Reducir el tamaño de la imagen a la mitad
25          down_sampled_gauss = gauss[1::2, 1::2]
26
27          # Añadir imagen a la piramide
28          gaussian_pyr.append(down_sampled_gauss)
29
30
31      return gaussian_pyr
```

Listing 3: Función que crea una priámide Gaussiana.

Entender el funcionamiento del código es bastante directo, pero por si acaso, vamos a describir brevemente lo que hace. Esta función crea la pirámide Gaussiana de N niveles (contando como primer nivel la imágen original). Por tanto, la imágen resultante estará compuesta de la imágen original y las $N - 1$ reducciones de la imágen original. Para obtener la siguiente imágen de la pirámide, simplemente basta con coger el nivel anterior, aplicarle un filtro Gaussiano y reducir su tamaño a la mitad, y posteriormente guardar el resultado. Todo esto se corresponde con la parte del bucle, la cuál se realiza $N - 1$ veces.

Lo más importante a destacar es el escalado de la imágen, el cuál puede ser visto en línea 25. Lo que se hace es coger todas las filas y columnas pares, ignorando las impares. De esta forma, si se escala una imágen que tenga alguna de sus dimensiones (o bien el número de filas o bien el número de columnas) impar, en el siguiente nivel estos valores serán pares. De la otra forma, si cogíesemos las filas y columnas impares, esto no se daría, ya que si alguna de las dimensiones fuese impar, en el siguiente nivel también lo sería. Es preferible trabajar con dimensiones pares, ya que así se pierde menos información al tomar una muestra de su espacio.

Una vez habiendo comentado esto, vamos a ver algún ejemplo de pirámide Gaussiana:

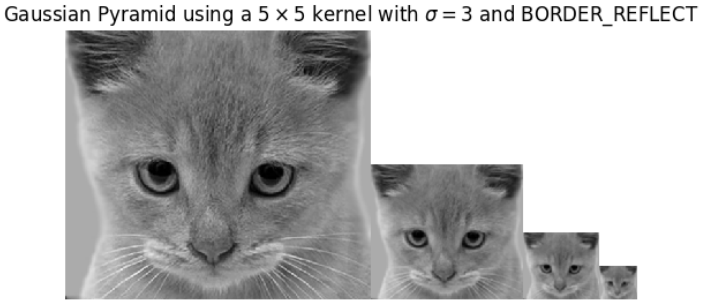


Figura 12: Pirámide Gaussiana utilizando un *kernel* 5×5 en cada nivel con $\sigma = 3$.

Tal y como se puede ver en la figura 12 se crea una pirámide en la que al reducir el tamaño de la imágen no se pierde demasiado detalle, ya que la figura del gato sigue siendo distinguible en cada nivel.

Se ha elegido un tamaño de *kernel* de 5×5 con un $\sigma = 3$ en cada eje porque es una máscara no muy grande y que aplica un suavizado suficiente para cada nivel, es decir, que ni aplica demasiado ni demasiado poco, tal y como se comprobó en la figura 2. Otro motivo importante es que, cuantos más niveles tenga la pirámide, más pequeña se irá haciendo la imágen, y por tanto, no interesa tener una máscara enorme con un sigma muy grande, ya que se va a emborronar demasiado. Se ha elegido un borde de tipo reflejo porque en general no modifica mucho la imágen de salida.

Para ver si cambiando el tipo de borde obtenemos algo diferente, se ha probado a utilizar un borde constante. El resto de parámetros se han dejado igual. A continuación se pueden ver los resultados:

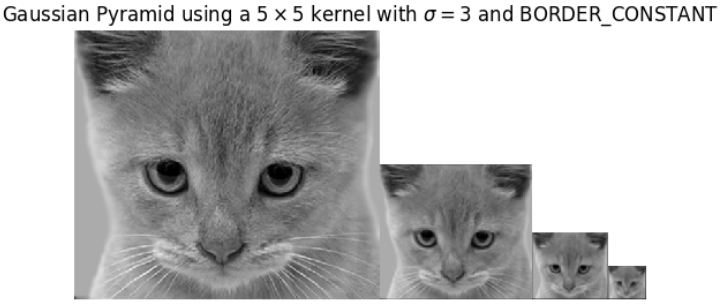


Figura 13: Pirámide Gaussiana con borde constante.

Se puede ver que existen unas pequeñas diferencias si lo comparamos con la figura 12. En los niveles más altos, no tiene mucha influencia el haber cambiado el tipo de borde, pero a medida que la imágen se va haciendo más pequeña, se pueden ver como comienzan a aparecer los recuadros negros, tal y como hemos visto anteriormente en la figura 4b. A parte de esto, no hay muchas más diferencias apreciables, lo cuál se puede deber a que el tamaño del *kernel* no es tan grande a como lo era antes.

2.2. Apartado B

Para crear la pirámide Laplaciana, se ha utilizado la siguiente función:

```
def laplacian_pyramid(img, ksize, sigma_x, sigma_y, border, N=4):
    """
    Funcion que crea una piramide Laplaciana de una imagen

    Args:
        img: Imagen de la que crar la piramide
        ksize: Tamaño del kernel
        sigma_x: Valor de sigma en el eje X
        sigma_y: Valor de sigma en el eje Y
        border: Tipo de borde
        N: Numero de componentes de la piramide. El nivel Gaussiano (ultimo)
            no esta incluido(default 4)
    Return:
```

```
14          Devuelve una lista que contiene las imagenes que forman la
    piramide
15      """
16
17      # Obtener piramide Gaussiana de un nivel mas
18      gaussian_pyr = gaussian_pyramid(img, ksize, sigma_x, sigma_y,
    border, N+1)
19
20      # Crear la lista que contendra la piramide Laplaciana
21      # Se inserta el ultimo elemento de la piramide Gaussiana primero
22      laplacian_pyr = [gaussian_pyr[-1]]
23
24      # Recorrer en orden inverso la piramide Gaussiana y generar la
    Laplaciana
25      for i in reversed(range(1, N+1)):
26          # Obtener la imagen actual y la anterior
27          current_img = gaussian_pyr[i]
28          previous_img = gaussian_pyr[i - 1]
29
30          # Hacer upsampling de la imagen actual
31          upsampled_img = np.repeat(current_img, 2, axis=0)
32          upsampled_img = np.repeat(upsampled_img, 2, axis=1)
33
34          # Si falta una fila, copiar la ultima
35          if upsampled_img.shape[0] < previous_img.shape[0]:
36              upsampled_img = np.vstack([upsampled_img, upsampled_img
    [-1]])
37
38          # Si falta una fila, copiar la ultima
39          if upsampled_img.shape[1] < previous_img.shape[1]:
40              upsampled_img = np.hstack([upsampled_img, upsampled_img
    [:, -1].reshape(-1, 1)])
41
42          # Pasar un Gaussian Blur a la imagen escalada para intentar
    suavizar el efecto de las
43          # filas y las columnas repetidas
44          upsampled_gauss = gaussian_kernel(upsampled_img, ksize,
    ksize, sigma_x, sigma_y, border)
45
46          # Restar la imagen orignal de la escalada para obtener
    detalles
47          diff_img = previous_img - upsampled_gauss
48
49          # Guardar en la piramide
50          laplacian_pyr.insert(0, diff_img)
51
52
53      return laplacian_pyr
```

Listing 4: Función que crea una pirámide Laplaciana.

Esta vez, a diferencia de lo que se podía ver en el código 3

2.3. Apartado C

3. Imágenes híbridas

Mezclando adecuadamente una parte de las frecuencias altas de una imagen con una parte de las frecuencias bajas de otra imagen, obtenemos una imagen híbrida que admite distintas interpretaciones a distintas distancias (ver hybrid images project page).

Para seleccionar la parte de frecuencias altas y bajas que nos quedamos de cada una de las imágenes usaremos el parámetro sigma del núcleo/-máscara de alisamiento gaussiano que usaremos. A mayor valor de sigma mayor eliminación de altas frecuencias en la imagen convolucionada. Para una buena implementación elegir dicho valor de forma separada para cada una de las dos imágenes (ver las recomendaciones dadas en el paper de Oliva et al.). Recordar que las máscaras 1D siempre deben tener de longitud un número impar.

Implementar una función que genere las imágenes de baja y alta frecuencia a partir de las parejas de imágenes (solo en la versión de imágenes de gris). El valor de sigma más adecuado para cada pareja habrá que encontrarlo por experimentación.

1. Escribir una función que muestre las tres imágenes (alta, baja e híbrida) en una misma ventana. (Recordar que las imágenes después de una convolución contienen número flotantes que pueden ser positivos y negativos)
2. Realizar la composición con al menos 3 de las parejas de imágenes
3. Construir pirámides gaussianas de al menos 4 níveles con las imágenes resultado. Explicar el efecto que se observa.

3.1. Apartado 1

3.2. Apartado 2

3.3. Apartado 3

4. Bonus

4.1. Convolución 2D propia

Implementar con código propio la convolución 2D con cualquier máscara 2D de números reales usando máscaras separables.

4.2. Imágenes híbridas a color

Realizar todas las parejas de imágenes híbridas en su formato a color (solo se tendrá en cuenta si la versión de gris es correcta).

4.3. Imágen híbrida propia

Realizar una imagen híbrida con al menos una pareja de imágenes de su elección que hayan sido extraídas de imágenes más grandes. Justifique la elección y todos los pasos que realiza.

Referencias

[1] Texto referencia
`https://url.referencia.com`