



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	2
1.1. Apartado A	2
1.2. Apartado B	12
2. EJERCICIO SOBRE PIRÁMIDES Y DETECCIÓN DE REGIONES	15
2.1. Apartado A	15
2.2. Apartado B	18
2.3. Apartado C	22
3. IMÁGENES HÍBRIDAS	28
3.1. Apartado 1	28
3.2. Apartado 2	30
3.3. Apartado 3	33
4. BONUS	37
4.1. Convolución 2D propia	37
4.2. Imágenes híbridas a color	41
4.3. Imagen híbrida propia	41
Referencias	42

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 imagen 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 imagen en cuestión:



Figura 1: Imagen 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 imagen, 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:

```

1 def apply_kernel(img, kx, ky, border):
2     """
3         Funcion que aplica un kernel separable sobre una imagen,
4         realizando una convolucion
5
6     Args:
7         img: Imagen sobre la que aplicar el filtro
8         kx: Kernel en el eje X
9         ky: Kernel en el eje Y
10        border: Tipo de borde
11
12    Return:
13        Devuelve una imagen filtrada
14        """

```

```

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

```

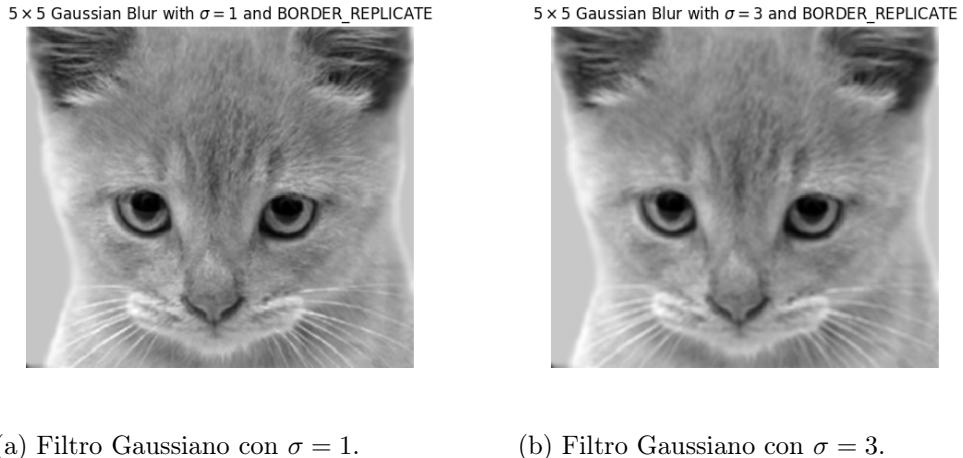
Listing 1: Función que aplica un filtro separable.

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.

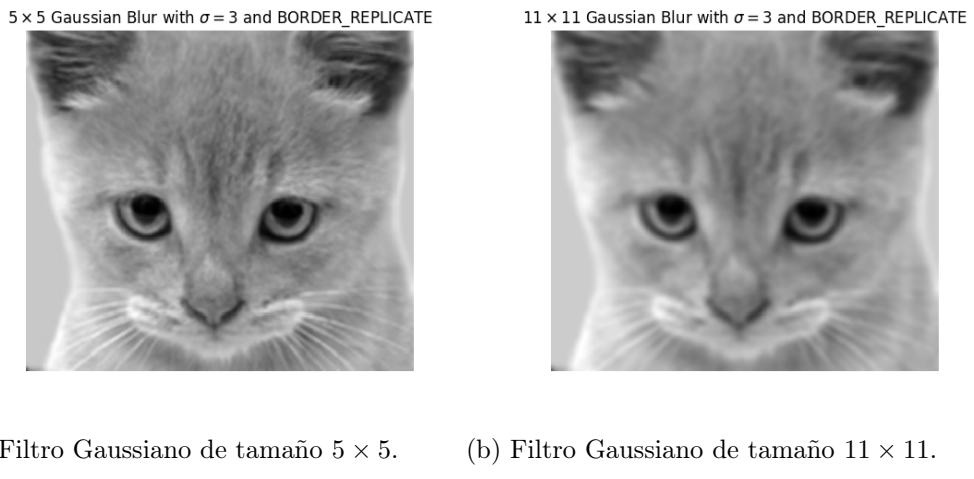
Con esto comentado, ya podemos empezar a hablar de los resultados que se han obtenido. Para ello, vamos a comenzar comentando los resultados que se obtienen para el filtro Gaussiano.

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:

Figura 2: Filtro Gaussiano con diferentes tamaños de σ .

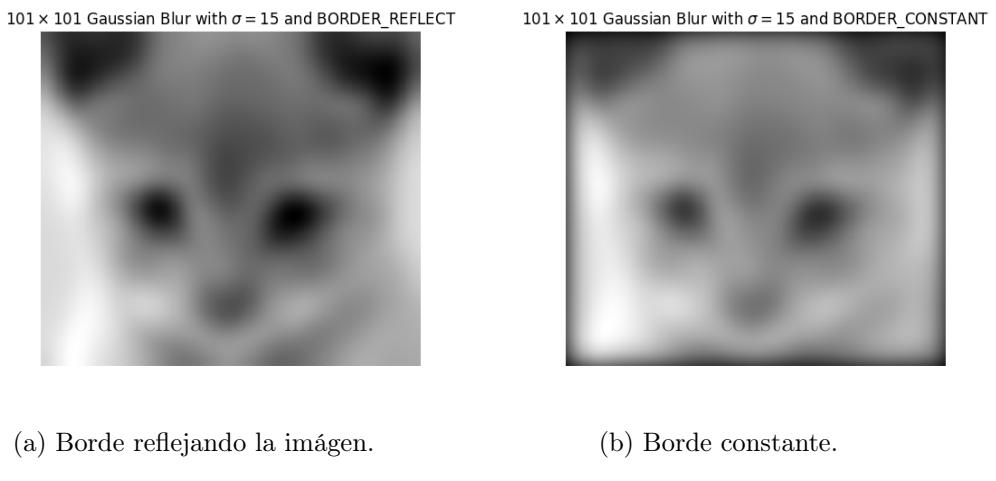
Como se puede ver, comparando cualquiera de las imágenes de la figura 2 con 1, existen ciertas diferencias. Podemos ver claramente como al aplicar el filtro Gaussiano se ha perdido cierto detalle, ya que se están eliminando las frecuencias altas. Además, si comparamos las figuras 2a y 2b, podemos ver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

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 píxeles 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 píxeles 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 píxeles más lejanos. Tampoco hay que coger un tamaño demasiado grande si el σ es pequeño, ya que entonces habrán muchos píxeles 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 imagen 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 imagen. El tamaño de la máscara es de 101×101 , permitiendo que se salga lo suficiente de la imagen para ver la influencia. A continuación se pueden ver los resultados que se han obtenido:



(a) Borde reflejando la imagen.

(b) Borde constante.

101 × 101 Gaussian Blur with $\sigma = 15$ and BORDER_DEFAULT

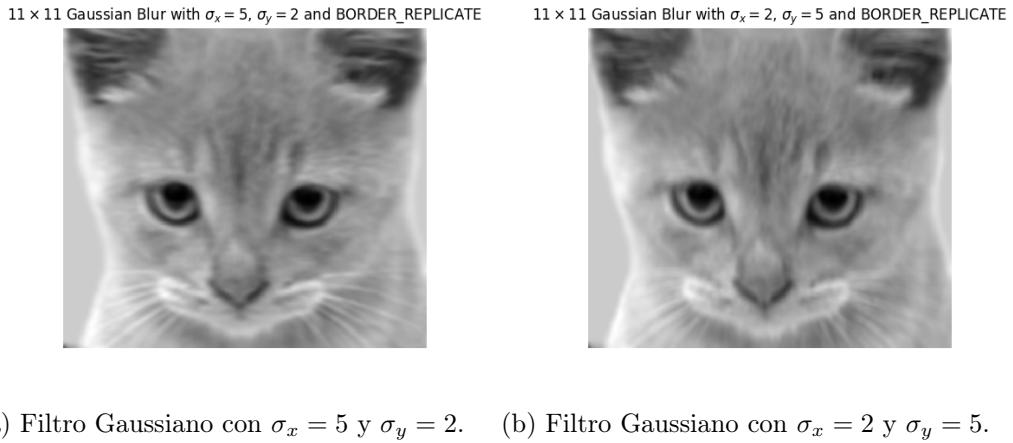
(c) Borde por defecto (reflejado sin contar el primer píxel).

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 imagen queda como si estuviese enmarcada.

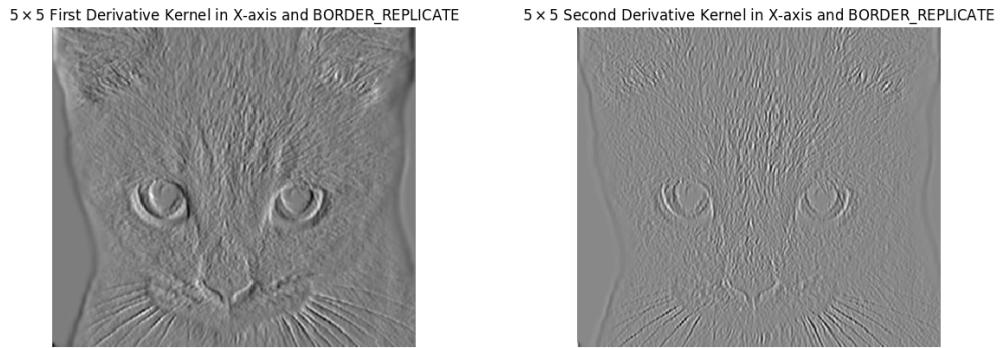
Las diferencias entre las figuras 4a y 4c son mucho más sutiles, pero notables. Se puede ver como por ejemplo en la figura 4c la esquina inferior izquierda es más oscura, mientras que en la figura 4a es más clara. Aunque los dos tipos de borde reflejan la imagen, la forma en la que lo hacen es distinto. *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 resultados 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 imagen se puede ver como los bigotes se han borronado 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 .

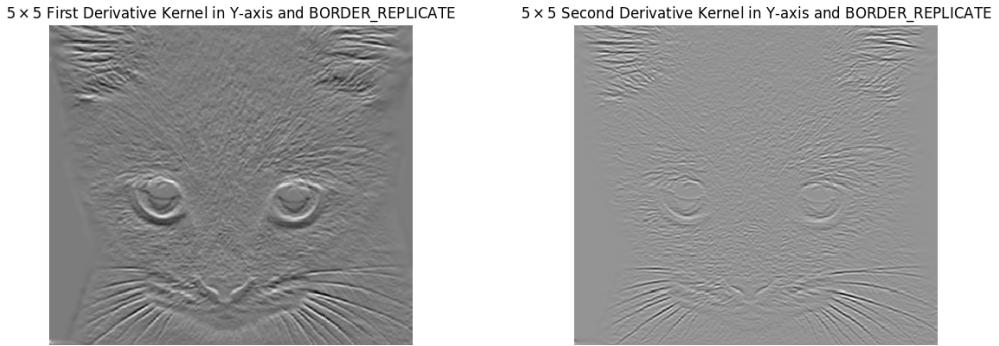
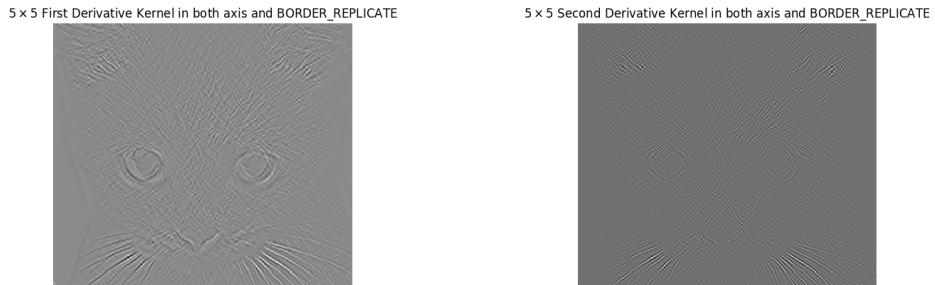
(c) Filtro de primera derivada en el eje Y . (d) Filtro de segunda derivada en el eje Y .

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:

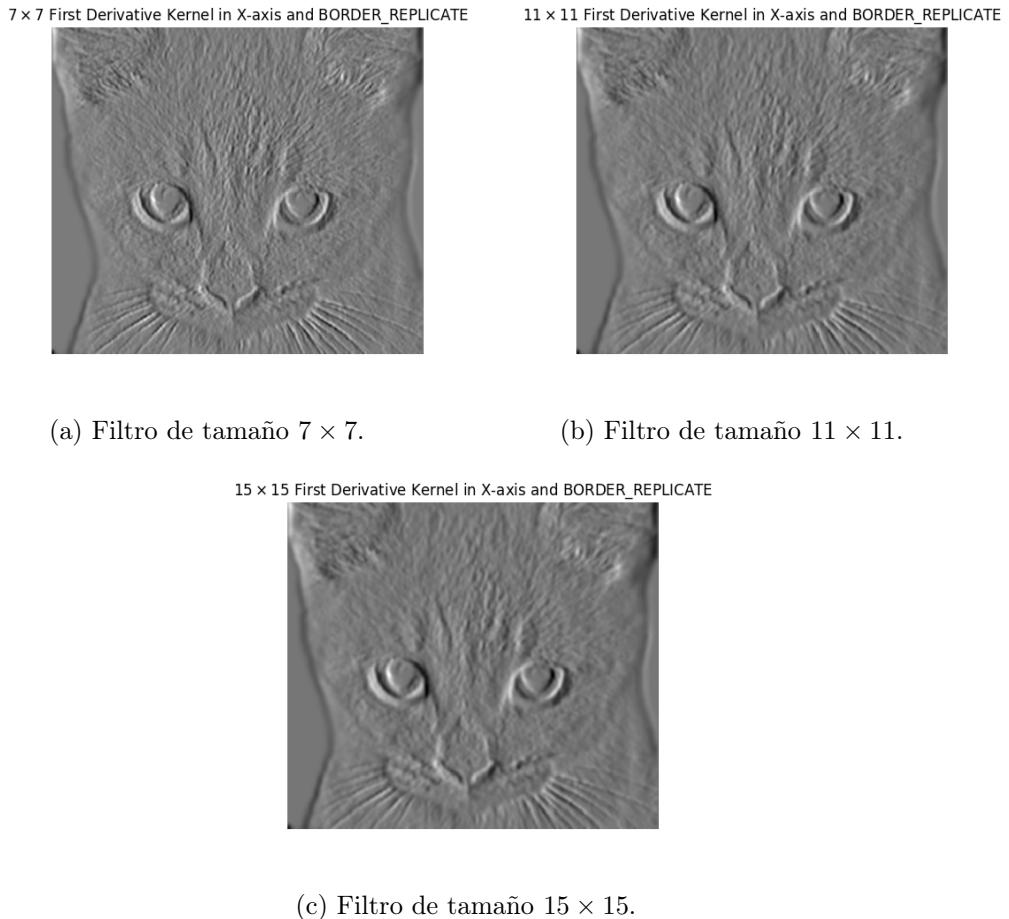


Figura 8: Filtro de primera derivada en el eje *X* variando el tamaño del *kernel*.

Como resultado más obvio, se puede ver que a medida que va aumentando el tamaño del *kernel*, se va emborronando más la imagen. 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 desapareciendo o haciéndose menos notables en esta última. De esta forma, aumentar el tamaño del *kernel* parece que solo nos aporta más suavizado. 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 imagen 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:

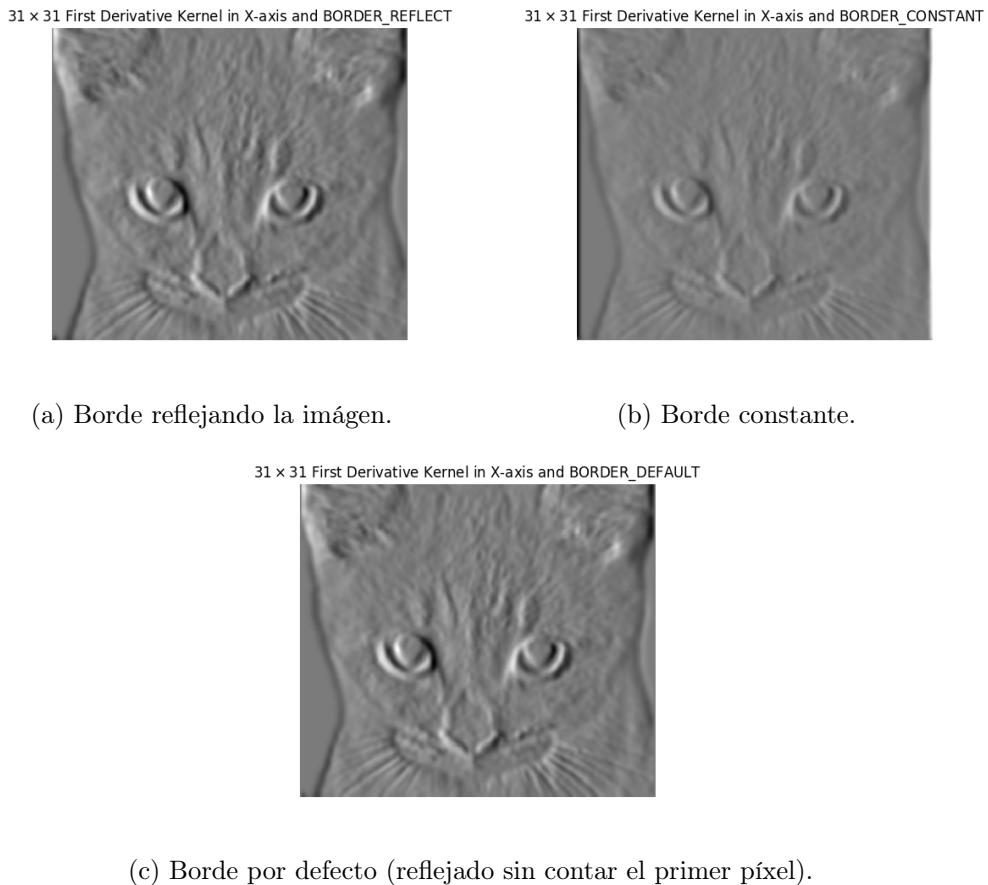


Figura 9: Filtro de primera derivada en el eje *X* variando el tipo de borde.

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 píxeles fuera de la imagen. 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:

```

1 def log_kernel(img, ksize, sigma_x, sigma_y, border):
2     """
3         Funcion que aplica un kernel LoG (Laplacian of Gaussian) sobre
4         una imagen.
5
6     Args:
7         img: Imagen sobre la que aplicar el kernel
8         ksize: Tamaño del kernel Gaussiano y Laplaciano
9         sigma_x: Valor de sigma en el eje X de la Gaussiana
10        sigma_y: Valor de sigma en el eje Y de la Gaussiana
11        border: Tipo de borde
12
13    Return:
14        Devuelve una imagen sobre la que se ha aplicado un filtro
15        LoG
16        """
17
18
19    # Aplicar filtro Gaussiano
20    gauss = gaussian_kernel(img, ksize, ksize, sigma_x, sigma_y,
21                           border)
22
23    # Obtener los filtros de derivada segunda en cada eje, aplicados
24    # sobre
25    # el filtro Gaussiano
26    dx2 = derivative_kernel(gauss, 2, 0, ksize, border)
27    dy2 = derivative_kernel(gauss, 0, 2, ksize, border)
28
29    # Combinar los filtros de derivadas y obtener Laplaciana de
30    # Gaussiana
31    laplace = dx2 + dy2
32
33
34    return laplace

```

Listing 2: Función que aplica una Laplaciana de Gaussiana.

Lo primero que se hace es aplicar un filtro de Gaussiana sobre la imagen de entrada. Al ser éste un filtro de paso bajo, las altas frecuencias (y el ruido) se eliminan, quedando una imagen 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:

$$\text{dst} = \Delta \text{src} = \frac{\partial^2 \text{src}}{\partial x^2} + \frac{\partial^2 \text{src}}{\partial y^2} \quad (1)$$

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 imagen. 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 alisando 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:

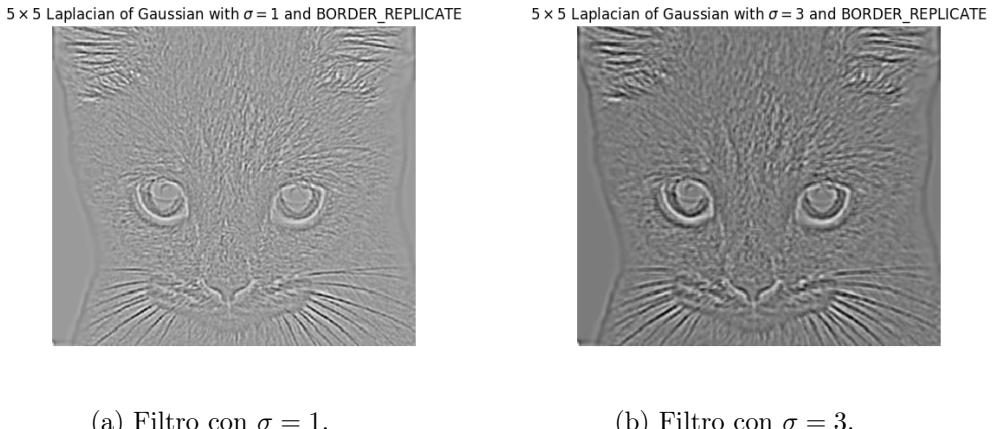


Figura 10: Filtro de Laplaciana de Gaussiana con diferentes valores de σ .

Como se puede ver, al aumentar el tamaño de σ se obtiene un mayor suavizado (debido a la Gaussiana), además de que la imagen 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 emborronamiento.

Como también se ha pedido, se ha probado a modificar el tipo de borde utilizado para ver qué efecto tiene sobre la imagen 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:

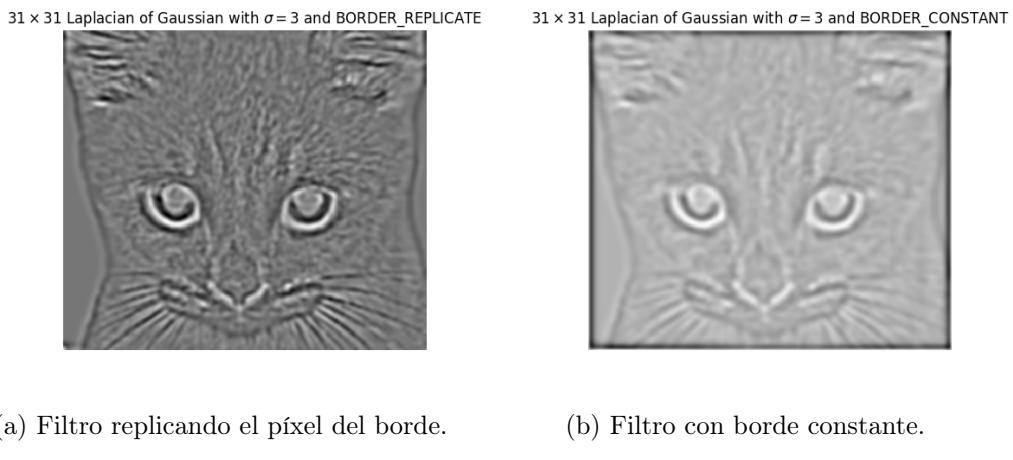


Figura 11: Filtro de Laplaciana de Gaussiana con diferentes valores de σ .

Claramente se pueden apreciar las diferencias. Se puede ver que la imagen que se ve en la figura 11b parece estar enmarcada, debido a que tiene los píxeles 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:

```

1 def gaussian_pyramid(img, ksize, sigma_x, sigma_y, border, N=4):
2     """
3         Funcion que devuelve una piramide Gaussiana de tamaño N
4
5     Args:
6         img: Imagen de la que extraer la piramide
7         ksize: Tamaño del kernel
8         sigma_x: Valor de sigma en el eje X
9         sigma_y: Valor de sigma en el eje Y
10        border: Tipo de borde a utilizar
11        N: Numero de imagenes que componen la piramide (default 4)
12

```

```

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,
23         sigma_y, border)
24
25         # Reducir el tamaño de la imagen a la mitad
26         down_sampled_gauss = gauss[1::2, 1::2]
27
28         # Añadir imagen a la piramide
29         gaussian_pyr.append(down_sampled_gauss)
30
31
31     return gaussian_pyr

```

Listing 3: Función que crea una pirámide 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 imagen original). Por tanto, la imagen resultante estará compuesta de la imagen original y las $N - 1$ reducciones de la imagen original. Para obtener la siguiente imagen 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 imagen, 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 imagen 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 cogiésemos 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:

Gaussian Pyramid using a 5×5 kernel with $\sigma = 3$ and BORDER_REPLICATEFigura 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 imagen 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 imagen, 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 replicado porque en general no modifica mucho la imagen de salida. En este caso, la pirámide es de 4 niveles; es decir, $N = 4$, tal y como se ha pedido.

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:

Gaussian Pyramid using a 5×5 kernel with $\sigma = 3$ and BORDER_CONSTANT

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 imagen 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:

```

1 def laplacian_pyramid(img, ksize, sigma_x, sigma_y, border, N=4):
2     """
3         Funcion que crea una piramide Laplaciana de una imagen
4
5     Args:
6         img: Imagen de la que crar la piramide
7         ksize: Tamaño del kernel
8         sigma_x: Valor de sigma en el eje X
9         sigma_y: Valor de sigma en el eje Y
10        border: Tipo de borde
11        N: Numero de componentes de la piramide. El nivel Gaussiano
12            (ultimo)
13            no esta incluido(default 4)
14
15    Return:

```

```

14     Devuelve una lista que contiene las imagenes que forman la
15     piramide
16     """
17
18     # Obtener piramide Gaussiana de un nivel mas
19     gaussian_pyr = gaussian_pyramid(img, ksize, sigma_x, sigma_y,
20                                     border, N+1)
21
22     # Crear la lista que contendra la piramide Laplaciana
23     # Se inserta el ultimo elemento de la piramide Gaussiana primero
24     laplacian_pyr = [gaussian_pyr[-1]]
25
26     # Recorrer en orden inverso la piramide Gaussiana y generar la
27     # Laplaciana
28     for i in reversed(range(1, N+1)):
29         # Obtener la imagen actual y la anterior
30         current_img = gaussian_pyr[i]
31         previous_img = gaussian_pyr[i - 1]
32
33         # Hacer upsampling de la imagen actual
34         upsampled_img = np.repeat(current_img, 2, axis=0)
35         upsampled_img = np.repeat(upsampled_img, 2, axis=1)
36
37         # Si falta una fila, copiar la ultima
38         if upsampled_img.shape[0] < previous_img.shape[0]:
39             upsampled_img = np.vstack([upsampled_img, upsampled_img
40 [-1]])
41
42         # Si falta una fila, copiar la ultima
43         if upsampled_img.shape[1] < previous_img.shape[1]:
44             upsampled_img = np.hstack([upsampled_img, upsampled_img
45 [:, -1].reshape(-1, 1)])
46
47         # Pasar un Gaussian Blur a la imagen escalada para intentar
48         # suavizar el efecto de las
49         # filas y las columnas repetidas
50         upsampled_gauss = gaussian_kernel(upsampled_img, ksize,
51                                         sigma_x, sigma_y, border)
52
53         # Restar la imagen original de la escalada para obtener
54         # detalles
55         diff_img = previous_img - upsampled_gauss
56
57         # Guardar en la piramide
58         laplacian_pyr.insert(0, diff_img)
59
60
61     return laplacian_pyr

```

Listing 4: Función que crea una pirámide Laplaciana.

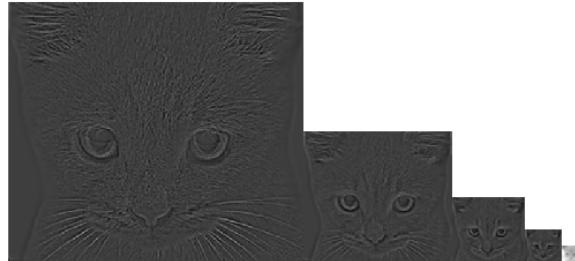
Esta vez, a diferencia de lo que se podía ver en el código 3, el código puede que no sea tan sencillo de entender. Por tanto, vamos a intentar explicar de manera

breve y simple qué es lo que hace.

Para generar una pirámide Laplaciana de N niveles, lo más fácil es partir de una pirámide Gaussiana de $N + 1$ niveles (el último será la imagen más pequeña, la cuál contendrá las bajas frecuencias, a diferencia del resto de imágenes de la pirámide). Comenzando desde esta imagen más pequeña, se realiza un escalado hacia arriba para que tenga las mismas dimensiones de la imagen del nivel anterior. Este escalado se realiza replicando todas las filas y columnas. Sobre este escalado se aplica un filtro Gaussiano para suavizar la imagen, ya que contiene información redundante e interesa modificar los píxeles para que se tenga en cuenta el entorno. Finalmente, se resta la imagen escalada y suavizada al anterior nivel de la pirámide Gaussiana para obtener las altas frecuencias, y se almacena esta información.

El proceso de escalado o *upsampling* es muy importante, y se debe destacar la forma en la que se hace. Este procedimiento se puede ver en las líneas 31-40. Como se dijo anteriormente, se replican las filas y columnas, insertándolas al lado de las originales. Este es el procedimiento más sencillo que se puede realizar, ya que se duplica el tamaño de la imagen con valores “válidos”. Sin embargo, aparece un pequeño problema: puede suceder que alguna de las dimensiones de la imagen del nivel anterior sea impar. Como las imágenes deben tener el mismo tamaño para poder obtener las altas frecuencias, lo que se ha decidido es añadir una fila o una columna más, o ambos, al final de la imagen, replicando por tanto la última fila/columna. De esta forma nos aseguramos de que siempre la imagen escalada y la del nivel anterior de la pirámide Gaussiana tengan el mismo tamaño.

A continuación se pude ver un ejemplo de pirámide Laplaciana. Tal y como se hizo antes, se ha elegido un tamaño de *kernel* de 5×5 con un $\sigma = 3$ en cada eje, ya que así no se emborrona mucho la imagen en cada nivel. De nuevo, el tipo de borde es el de replicado, porque no modifica mucho la imagen. En este caso, se ha generado una pirámide Laplaciana con $N = 4$, lo cuál significa que el resultado tendrá un nivel extra, el cuál será la imagen más pequeña con las frecuencias bajas. El resultado es el siguiente:

Laplacian Pyramid using a 5×5 kernel with $\sigma = 3$ and BORDER_REPLICATEFigura 14: Pirámide Laplaciana utilizando un *kernel* 5×5 en cada nivel con $\sigma = 3$.

Se puede ver que los niveles bajos son los que tienen las frecuencias altas, mientras que el más alto (el último) es el que contiene las frecuencias bajas de la imagen. De esta forma, se puede realizar una reconstrucción de la imagen sencilla y conservando casi todos los detalles, cosa que con la pirámide Gaussiana no ocurría, ya que solo se almacenan las bajas frecuencias.

De nuevo, tal y como hicimos antes, para ver qué efecto tiene cambiar el tipo de borde, vamos a probar a construir la misma pirámide, pero cambiando a un borde de tipo constante. A continuación se puede ver la pirámide resultante:

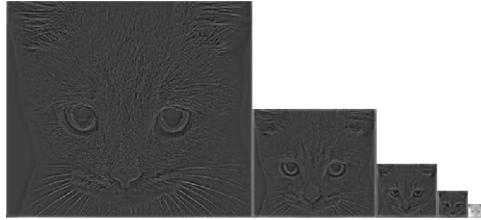
Laplacian Pyramid using a 5×5 kernel with $\sigma = 3$ and BORDER_CONSTANT

Figura 15: Pirámide Laplaciana con tipo de borde constante.

Se puede ver que sí que existen ciertas diferencias si se compara con la figura 14. De nuevo, es como si la imagen estuviese enmarcada, ya que empiezan a aparecer bordes, aunque esta vez son blancos en las imágenes de altas frecuencias, y de forma menos visible pero aún así presente, negros en la imagen de baja frecuencia, tal y como sucedía en la figura 13.

2.3. Apartado C

Para comenzar, vamos a ilustrar cuál es el código que realiza la función de obtener un espacio de escalas Laplaciano:

```

1 def laplacian_scale_space(img, ksize, border, N, sigma=1.0,
2     sigma_inc=1.2):
3     """
4     Funcion que construye el espacio de escalas Laplaciano de una
5     imagen dada
6
7     Args:
8         img: Imagen de la que extraer el espacio de escalas
9         ksize: Tamaño del kernel
10        border: Tipo de borde
11        N: Numero de escalas
12        sigma: Valor inicial del kernel (default 1)
13        sigma_inc: Multiplicador que incrementa el sigma (default
14            1.2)
15    Return:
16        Devuelve una lista con N imagenes, formando el espacio de
17        escalas y los
18        valores de sigma utilizados
19    """
20    # Crear listas que contendran las imagenes y los sigma
21    scale_space = []
22    sigma_list = []
23
24    # Crear las N escalas
25    for _ in range(N):
26        # Aplicar Laplacian of Gaussian
27        level_img = log_kernel(img, ksize, sigma, sigma, border)
28
29        # Normalizar multiplicando por sigma^2
30        level_img **= sigma ** 2
31
32        # Elevar al cuadrado la imagen resultante
33        level_img = level_img ** 2
34
35        # Suprimir no maximos
36        suppressed_level = non_max_supression(level_img)
37
38        # Guardar imagen y sigma
39        scale_space.append(suppressed_level)

```

```

36     sigma_list.append(sigma)
37
38     # Incrementar sigma
39     sigma *= sigma_inc
40
41 return scale_space, sigma_list

```

Listing 5: Función que crea un espacio de escalas Laplaciano

Esta función es bastante sencilla de entender. Se encarga de crear un espacio de escalas Laplaciano de N niveles. Para ello, partiendo siempre de la imagen original, aplica un filtro de Laplaciana de Gaussiana como el que se puede ver en el algoritmo 2. Después, normaliza la imagen multiplicando por σ^2 y, para eliminar todos aquellos valores negativos, se eleva cada píxel de la imagen resultante al cuadrado. Acto seguido se suprimen los no máximos y se guarda la imagen resultante. También se guarda el σ utilizado, aunque esto es para ofrecer información extra luego. Finalmente, se multiplica σ por 1.2, y se repite todo el proceso.

Como se pudo ver, por defecto, el valor de σ inicial es 1, ya que no interesa aplicar mucho alisamiento al principio. Adicionalmente, el incremento por defecto de σ es 1.2, ya que de esta forma se puede ver mejor como van evolucionando las regiones relevantes.

Para la supresión de no máximos se ha utilizado la siguiente función:

```

1 def non_max_supression(img):
2     """
3         Funcion que realiza la supresion de no maximos dada una imagen
4             de entrada
5
6     Args:
7         img: Imagen sobre la que realizar la supresion de no maximos
8     Return:
9         Devuelve una nueva imagen sobre la que se han suprimido los
10            maximos
11
12     # Crear imagen inicial para la supresion de maximos (
13     # inicializada a 0)
14     suppressed_img = np.zeros_like(img)
15
16     # Para cada pixel, aplicar una caja 3x3 para determinar el
17     # maximo local
18     for i,j in np.ndindex(img.shape):
19         # Obtener la region 3x3 (se consideran los bordes para que
20         # la caja
21         # tenga el tamaño adecuado, sin salirse)
22         region = img[max(i-1, 0):i+2, max(j-1, 0):j+2]
23
24         # Obtener el valor actual y el maximo de la region
25         current_val = img[i, j]
26         max_val = np.max(region)

```

```

22     # Si el valor actual es igual al maximo, copiarlo en la
23     # imagen de supresion
24     # de no maximos
25     if max_val == current_val:
26         suppressed_img[i, j] = current_val
27
28 return suppressed_img

```

Listing 6: Función que hace la supresión de no máximos.

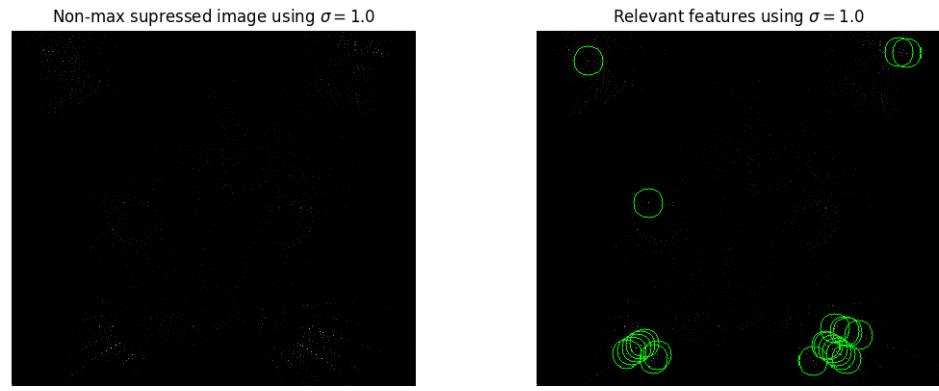
Lo único que se realiza es crear una imagen nueva con todos los valores a 0, y se recorre la imagen sobre la que se ha aplicado el filtro Laplaciana de Gaussiana normalizado y elevado al cuadrado. En cada caso se comprueba si el píxel (i, j) , es decir, el actual, tiene el mismo valor que el máximo de la región. Para la región, se ha escogido un tamaño de 3×3 , porque es el que se escoge normalmente. En caso de que el píxel actual sea igual al máximo, se copia su valor en la posición (i, j) de la imagen resultante. En caso contrario, se deja a 0, ya que no es un máximo.

Puede que la línea que resulte más intrigante del algoritmo 6 es la 17, donde se determina la región. De la forma en la que se hace, nos aseguramos que en ningún momento nos salimos de las dimensiones de la matriz que representa la imagen por la izquierda y por arriba, ya que no importa que nos pasemos de más por los extremos de la derecha y por abajo. De esta forma, controlamos los bordes. En caso de no hacerlo así, obtendríamos un error por problemas de indexación. No hace falta controlar que no nos salgamos de los otros bordes porque Numpy no da problemas al pasarnos de tamaño, siempre y cuando alguno de los píxeles de la imagen formen parte de esa región.

Una vez comentado esto, vamos a ver cómo sería aplicar esta función sobre nuestra imagen del gato. Para ello, vamos a obtener un espacio de escalas con 5 escalas utilizando un *kernel* de tamaño 5 y borde de replicado. Se ha escogido este tamaño de máscara porque nos ha ofrecido unos buenos resultados en los apartados anteriores, y el borde de replicado ofrece unos resultados razonables. El resto de parámetros (σ e incremento) se dejan a los valores por defecto por los motivos comentados anteriormente.

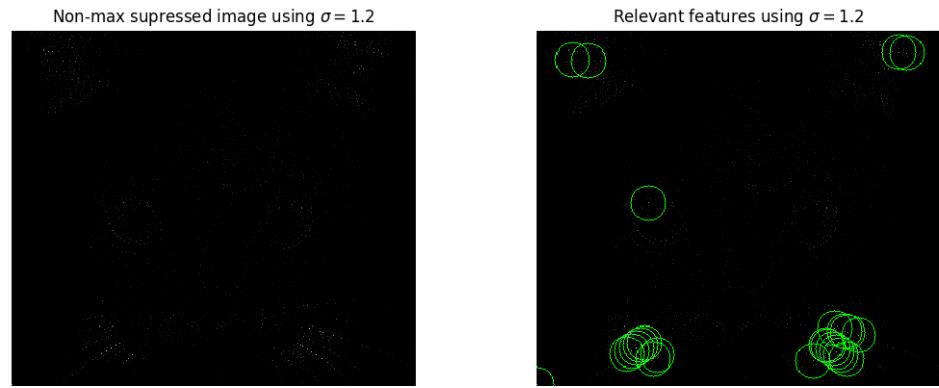
También se pide que, sobre estas escalas se dibujen círculos proporcionales a σ de las regiones relevantes. Ya que no se indica exactamente cómo hacerlo, vamos a suponer que aquellas regiones relevantes son las que tienen una intensidad de gris superior a 128. Como tampoco se indica qué tamaño deben tener los círculos, vamos a suponer que tienen un radio de 15σ según el σ utilizado para la escala.

A continuación se pueden ver la imagen resultado de aplicarle la supresión de no máximos y la imagen con las regiones relevantes:



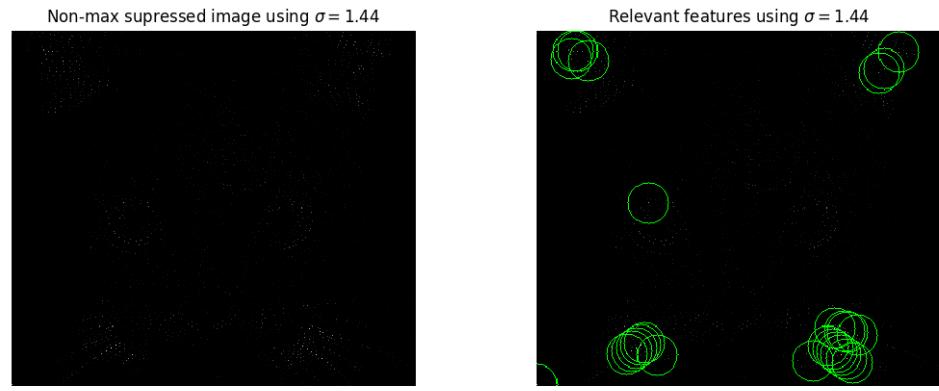
(a) Supresión de no máximos del nivel 1. (b) Regiones relevantes del nivel 1.

Figura 16: Resultados del nivel 1 de escalas Laplaciano.



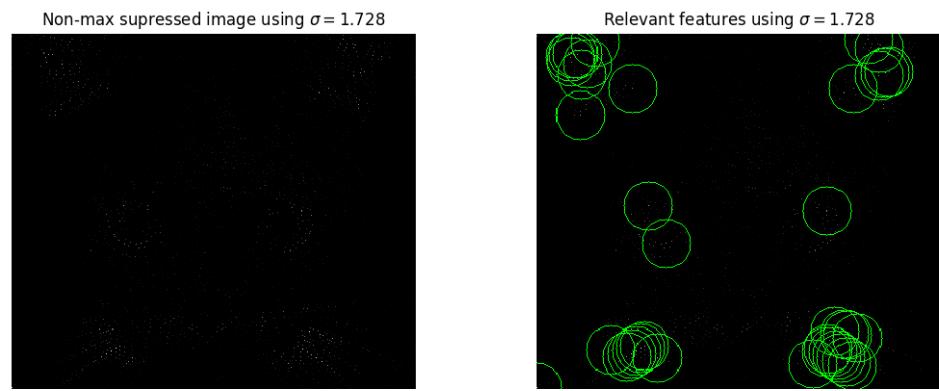
(a) Supresión de no máximos del nivel 2. (b) Regiones relevantes del nivel 2.

Figura 17: Resultados del nivel 2 de escalas Laplaciano.



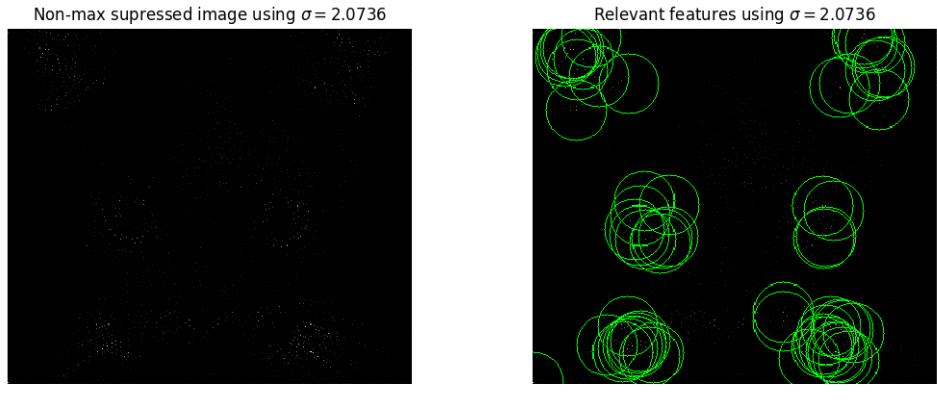
(a) Supresión de no máximos del nivel 3. (b) Regiones relevantes del nivel 3.

Figura 18: Resultados del nivel 3 de escalas Laplaciano.



(a) Supresión de no máximos del nivel 4. (b) Regiones relevantes del nivel 4.

Figura 19: Resultados del nivel 4 de escalas Laplaciano.



(a) Supresión de no máximos del nivel 5.

(b) Regiones relevantes del nivel 5.

Figura 20: Resultados del nivel 5 de escalas Laplaciano.

Como se puede ver, a medida que va aumentando el valor de σ el número de regiones relevantes se va incrementando. Esto puede deberse a que al incrementar el valor de σ se suaviza más, y por tanto, los píxeles comienzan a tener unos valores más similares.

Puede que este incremento de regiones no se vea muy nítido en el documento debido al tamaño de las figuras y a la compresión que se hace de las imágenes, pero si nos fijamos en las regiones de los ojos, las orejas y los bigotes, podemos ver que se van “encendiendo” más píxeles. Esto se pone de manifiesto con la cantidad de círculos nuevos que van apareciendo en estas zonas, ya que a medida que aumenta el σ van apareciendo más y más.

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 niveles con las imágenes resultado. Explicar el efecto que se observa.

3.1. Apartado 1

En este apartado se ha pedido crear las funciones. Por tanto, en el siguiente apartado veremos las composiciones que se han creado. Así que, de momento, vamos a echarle un vistazo a la función que permite crear las imágenes híbridas:

```

1 def hybrid_image_generator(img_low, img_high, ksize, sigma_low,
2     sigma_high, border):
3     """
4         Funcion que permite crear una imagen hibrida combinando dos
5         imagenes
6
7     Args:
8         img_low: Imagen que sera utilizada para las bajas
9             frecuencias

```

```

7     img_high: Imagen que sera utilizada para las altas
8     frecuencias
9     ksize: Tamaño del kernel (es una tupla)
10    sigma_low: Sigma para la imagen de bajas frecuencias
11    sigma_high: Sigma para la imgane de altas frecuencias
12    border: Tipo de borde
13
14    Return:
15        Devuelve una lista que contiene la imagen de bajas
16        frecuencias, la de altas
17        y la hibrida
18    """
19
20    # Crear la imagen de bajas frecuencias aplicando filtro
21    # Gaussiano
22    low_freq_img = gaussian_kernel(img_low, ksize, ksize,
23                                    sigma_low, border)
24
25    # Crear imagen de altas frecuencias aplicando filtro Gaussiano y
26    # restando a la original
27    gauss_high_freq = gaussian_kernel(img_high, ksize, ksize,
28                                       sigma_high, sigma_high, border)
29    high_freq_img = img_high - gauss_high_freq
30
31    # Crear la imagen hibrida combinando las dos
32    hybrid = low_freq_img + high_freq_img
33
34
35    return [low_freq_img, high_freq_img, hybrid]

```

Listing 7: Función que crea imágenes híbridas.

Esta función crea la imagen híbrida con las frecuencias bajas de la primera y las altas de la segunda imagen que se le pasa. Junto a esta, obtiene también las imágenes de bajas y altas frecuencias que se utilizan para componer la híbrida. Para obtener las frecuencias bajas, basta con simplemente pasar un filtro Gaussiano. Para las altas, solo tenemos que obtener las frecuencias bajas y restárselo a la imagen original, obteniendo por tanto las frecuencias altas. Finalmente, solo hace falta combinar las dos imágenes resultantes en una sumándolas. De esta forma, se obtiene una imagen híbrida, compuesta por las altas frecuencias de una de las imágenes y las altas de la otra.

Para visualizarlas, se ha creado la siguiente función:

```

1 def visualize_mult_images(images, titles=None):
2     """
3         Funcion que pinta multiples imagenes en una misma ventana.
4         Adicionalmente
5             se pueden especificar que titulos tendran las imagenes.
6
7     Args:
8         images: Lista con las imagenes
9         titles: Titulos que tendran las imagenes (default None)
10    """

```

```

11 # Obtener el numero de imagenes
12 n_img = len(images)
13
14 # Crear n_cols subplots (un subplot por cada imagen)
15 # El formato sera 1 fila con n_cols columnas
16 _, axarr = plt.subplots(1, n_img)
17
18 # Pintar cada imagen
19 for i in range(n_img):
20     # Convertir la imagen a uint8
21     img = transform_img_uint8(images[i])
22
23     # Pasarla de BGR a RGB
24     img = cv.cvtColor(img, cv.COLOR_BGR2RGB)
25
26     # Obtener siguiente imagen
27     axarr[i].imshow(img)
28
29     # Determinar si hay que poner o no titulo
30     if titles != None:
31         axarr[i].set_title(titles[i])
32
33     axarr[i].axis('off')
34
35 # Mostar imagenes
36 plt.show()

```

Listing 8: Función que muestra múltiples imágenes en la misma ventana.

Para mostrarlas, las imágenes se pasan primero a *uint8* (hace falta recordar que hemos estado trabajando con imágenes en *float64* hasta ahora). Además, ya que se está usando el módulo *matplotlib*, se tiene que cambiar del formato BGR (el que usa OpenCV) a RGB (el que utiliza el módulo mencionado anteriormente). Hace falta recalcar que esto se ha hecho con todas las imágenes hasta ahora, y no es exclusivo de las imágenes híbridas. Lo que sí que es diferente es la forma en la que se dibujan, ya que hay que partir el *plot* que se utiliza en tantos *subplots* como imágenes se quieran pintar a la vez.

3.2. Apartado 2

Una vez vistas las funciones, vamos a ver las composiciones creadas. Hace falta destacar que los valores de σ que se utilizan para la imagen de las altas frecuencias y para la de las bajas dependen mucho del par de imágenes, y se han obtenido mediante experimentación.

Se han realizado las composiciones de todas las parejas de imágenes, las cuáles pueden ser vistas a continuación:



Figura 21: Composición gato-perro.



Figura 22: Composición Einstein-Marilyn.



Figura 23: Composición bicicleta-moto.



Figura 24: Composición ave-avión.

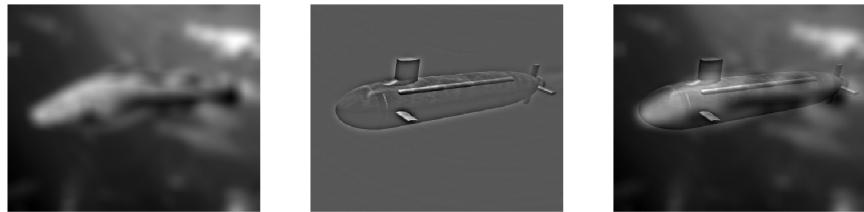


Figura 25: Composición pez-submarino.

En la parte izquierda de cada figura se puede ver la imágen en bajas frecuencias que se ha utilizado. En la parte central aparece la imágen de altas frecuencias. A la derecha se puede ver la imágen híbrida. En general, las imágenes híbridas han quedado bastante bien, ya que se puede ver que la imágen de altas frecuencias es más distingible de cerca que la de bajas. Al revés pasaría lo mismo; ya que al alejarnos de la imágen, solo veríamos las bajas, y no las altas.

3.3. Apartado 3

En este apartado, vamos a simular que nos estamos alejando de la imágen para ver si podemos dejar de ver las altas frecuencias. Para ello, en vez de tener que alejarnos nosotros de la pantalla, podemos ir disminuyendo el tamaño de la imágen híbrida para ver cómo va cambiando lo que vemos a medida que nos vamos alejando. Aquí es donde entra en juego la pirámide Gaussiana, la cuál permite visualizar esta evolución de forma sencilla, ya que incluye versiones más grandes de la imágen y más pequeñas.

Para construir la pirámide, se ha utilizado un $N = 4$ (es decir, que se verán 4 imágenes en la pirámide), con un tamaño de *kernel* de 5 y un $\sigma = 3$ para cada eje, tal y como se ha hecho en los apartados anteriores. Para el tipo de borde, se ha elegido de nuevo el borde de replicado.

A continuación se pueden ver ejemplos de las pirámides Gaussianas:



Figura 26: Pirámide Gaussiana de la composición gato-perro.

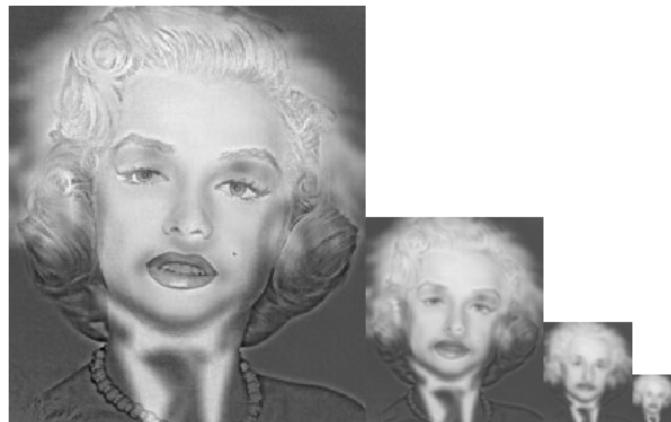


Figura 27: Pirámide Gaussiana de la composición Einstein-Marilyn.

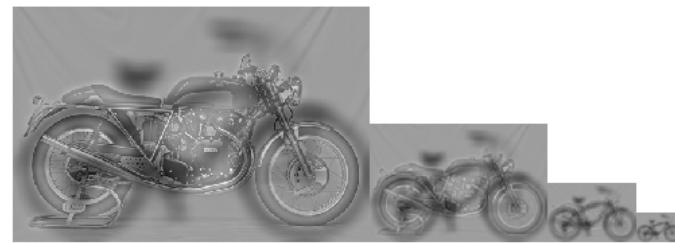


Figura 28: Pirámide Gaussiana de la composición bicicleta-moto.

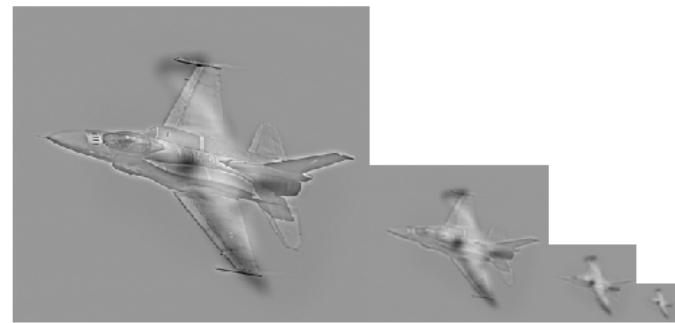


Figura 29: Pirámide Gaussiana de la composición ave-avión.

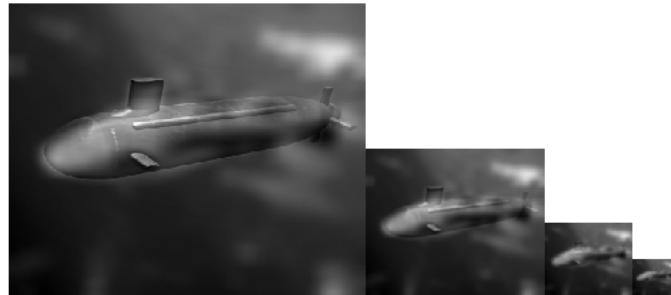


Figura 30: Pirámide Gaussiana de la composición pez-submarino.

Como se puede ver, a medida que se va haciendo más pequeña la imagen, menos se ve la parte de imagen que tiene frecuencias altas, hasta que al final solo se ve la imagen de bajas frecuencias. Esto se debe a que las ondas de frecuencia alta viajan una menor distancia en el espacio debido a que tienen una menor amplitud de onda. Las frecuencias bajas, en cambio, tienen una amplitud de onda mayor, y son capaces de recorrer una mayor distancia. De esta forma, a medida que nos alejamos o reducimos el tamaño de la imagen, menos altas frecuencias nos van a llegar, hasta que al final solo nos lleguen las bajas, tal y como se ha dicho anteriormente.

4. BONUS

4.1. Convolución 2D propia

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

Para esta sección se han hecho dos cosas:

1. Implementar la correlación 1D con código propio.
2. Implementar la convolución 2D utilizando la correlación 1D implementada anteriormente.

Vamos a comenzar explicando la correlación 1D. Se ha implementado el siguiente código:

```

1 def correlation_1D(img, kernel):
2     """
3         Funcion que aplica la correlacion 1D sobre una imagen de entrada
4             utilizando
5                 un kernel dado
6
7     Args:
8         img: Imagen sobre la que aplicar la correlacion
9             kernel: Kernel que se aplicara
10    Return:
11        Devuelve una imagen sobre la que se ha aplicado correlacion
12            1D por filas
13    """
14    # Obtener el numero de elementos que se deben replicar al
15        principio y al final
16    # de la imagen
17    n_replica = kernel.shape[0] // 2
18
19    # Replicar elemento inicial y final de la imagen n_replica veces
20    replica_img = np.hstack([np.tile(img[:, 0].reshape(-1, 1),
21        n_replica), img])
22    replica_img = np.hstack([replica_img, np.tile(img[:, -1].reshape(
23        -1, 1), n_replica)])
24
25    # Crear matriz de salida con el mismo tamaño que img
26    correlation_mat = np.empty_like(img)
27
28    # Aplicar la correlacion sobre cada elemento (i, j) de la imagen
29    for i in range(img.shape[0]):
30        for j in range(n_replica, img.shape[1] + n_replica):
31            # Obtener sub imagen del mismo tamaño que el kernel

```

```

27         sub_img = replica_img[i, j - n_replica:j + n_replica +
28             1]
29
30     # Realizar producto escalar de la sub imagen y el kernel
31     corr_value = np.dot(sub_img, kernel)
32
33     # Actualizar valor de la matriz de correlacion
34     correlation_mat[i, j - n_replica] = corr_value
35
36 return correlation_mat

```

Listing 9: Función que realiza la correlación 1D

Vamos a ir explicando lo que hace el código para que se entienda cuál es la idea general. Lo primero que hay que hacer es extender la imagen que se pasa como argumento, de modo que tenga más píxeles. Esto se hace por los bordes, debido a que también se tiene que calcular la correlación ahí y, en caso de no tener más valores más allá de ellos, se estaría haciendo un cálculo incorrecto. Para ello, se replica el píxel del borde un número de veces. Este número de veces está determinado por la variable $n_replica$, la cuál viene dada por la siguiente expresión:

$$n_replica = \left\lceil \frac{KernelSize}{2} \right\rceil \quad (2)$$

Esto se debe a que los *kernels* tienen un tamaño impar, debido a que consideran $KernelSize / 2 - 1$ vecinos a cada lado del píxel (i, j) de la imagen. Este píxel (el cuál es el central del *kernel*) hace que el tamaño de la máscara sea siempre impar. Por tanto, nos interesa saber cuántas veces tenemos que replicar el borde de la imagen según el tamaño del *kernel* que se le pase.

Se ha elegido hacer un replicado debido a que no estaba nada especificado en el enunciado del ejercicio, además de que es lo más sencillo de hacer. Por tanto, lo que se hace es copiar el primer y el último píxel de cada fila de la imagen $n_replica$ veces, tal y como hace el tipo de borde *BORDER_REPLICATE* de OpenCV. Este replicado se puede ver en las líneas 17 y 18 del código 9, donde se crea una nueva matriz en la que se concatenan primero al principio y luego al final las réplicas del primer y del último píxel de la imagen original, respectivamente. Esta operación está vectorizada, por tanto, se hace para todas las filas a la vez.

Después, se crea una matriz de salida, *correlation_mat*, la cuál tiene las mismas dimensiones que la imagen de entrada. Aquí es dónde se escribirá el resultado de la convolución. A continuación se recorren todas las filas de la de la imagen replicada, lo cuál se corresponde con la línea 24. La imagen replicada tiene el mismo número de filas que la original, así que usaremos el número de filas de la original para delimitar el rango del bucle, por comodidad más que nada. Para cada fila, se comienza a

partir de columna $n_replica$, debido a que queremos dejar tantos vecinos a la izquierda del píxel central inicial, y se recorre la imagen replicada hasta la columna $NumColumnasImagenOriginal + n_replica$, aplicando la correlación. De esta forma, recorremos la imagen original dentro de la replicada, y usamos los píxeles replicados solo para el cálculo de la correlación. Ninguno de estos píxeles replicados estará en el centro de la correlación por las condiciones especificadas anteriormente.

Para el cálculo de la correlación, simplemente obtenemos una parte de la imagen del mismo tamaño que el *kernel* utilizando los índices definidos anteriormente, se realiza el producto escalar entre esta parte de la imagen y el *kernel* y se guarda el resultado en la matriz de salida. Es importante destacar que esta función asume que el *kernel* viene dado como un vector columna, para así poder realizar el producto escalar entre un vector fila (la parte de la imagen) y un vector columna (el *kernel*).

Ahora ya solo nos queda ver como se haría la convolución. A continuación se puede ver el código implementado:

```

1 def convolution(img, kernel_x, kernel_y):
2     """
3         Funcion que aplica la convolucion sobre una imagen dados un
4             kernel para
5                 el eje X y el eje Y
6
7     Args:
8         img: Imagen sobre la que aplicar la convolucion
9         kernel_x: Kernel en el eje X
10        kernel_y: Kernel en el eje Y
11
12    Return:
13        Devuelve la convolucion de la imagen con los kernels de
14            entrada
15        """
16
17    # Hacer que los kernels sean vectores columna
18    kernel_x = kernel_x.reshape(-1, 1)
19    kernel_y = kernel_y.reshape(-1, 1)
20
21    # Realizar un flip sobre los kernels
22    kernel_x_flip = np.flip(kernel_x)
23    kernel_y_flip = np.flip(kernel_y)
24
25    # Realizar la convolucion (primero sobre el eje X y luego sobre
26        el eje Y)
27    convolution = correlation_1D(img, kernel_x_flip)
28    convolution = correlation_1D(convolution.T, kernel_y_flip)
29
30    # Trasponer la convolucion (porque se han cambiado filas por
31        columnas anteriormente)
32    convolution = convolution.T
33
34
35    return convolution

```

Listing 10: Función que aplica la convolución 2D sobre una imagen.

Lo primero que se hace es transformar los *kernels* de entrada para que sean vectores columna (si no lo eran ya). A continuación, para poder aplicar la convolución hace falta realizar un *flip* sobre los *kernels* (líneas 18-19). Como son vectores y no matrices, simplemente basta con darles la vuelta. Si no se hiciese, se aplicaría una correlación en vez de una convolución, lo cuál no produciría el resultado deseado.

Una vez hecho esto, es el momento de aplicar la correlación utilizando los *kernels* sobre los que se ha aplicado un *flip*. Primero se aplica sobre el eje X, lo cuál se puede ver en la línea 22 y no supone ningún problema. Sin embargo, si nos damos cuenta de una cosa, la correlación 1D que se puede ver en el código 9 se aplica por filas. Para aplicarla a las columnas, simplemente basta realizar la traspuesta de la imagen sobre la que se quiere realizar la correlación. De esta forma, se cambiarán las filas por las columnas, y se podrá aplicar sin problema. Lo único que hará falta es trasponer la imagen resultado al final, ya que tiene las filas y las columnas cambiadas. Esto se puede ver en la línea 26.

Para probar nuestra función, hemos aplicado un *kernel* Gaussiano de $\sigma = 3$ en cada eje y tamaño de *kernel* 5 sobre la imagen del gatito que hemos estado usando hasta ahora. También hemos aplicado un filtro de primera derivada en el eje *X* con tamaño de *kernel* 5. A continuación se pueden ver los resultados:

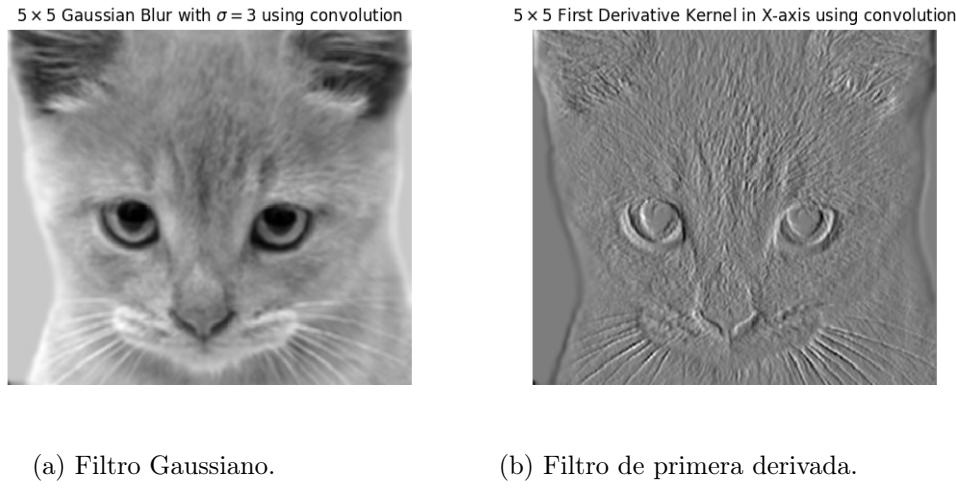


Figura 31: Distintos filtros utilizando la convolución propia.

El código anteriormente mostrado funciona perfectamente ya que, como se puede de comprobar, los resultados son los mismos que los que se pueden ver para las figuras 2b y 6a.

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. Imagen 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.refencia.com>