VC – PRÁCTICA 1

Julio Fresneda – juliofresnedag@correo.ugr.es

Apartado 1

1.- USANDO LAS FUNCIONES DE OPENCV:

Escribir funciones que implementen los siguientes puntos:

A) El cálculo de la convolución de una imagen con una máscara Gaussiana 2D (Usar GaussianBlur). Mostrar ejemplos con distintos tamaños de máscara y valores de sigma. Valorar los resultados.

La función GaussianBlur de OpenCV es una función que difumina una imagen convulcionándola con una máscara Gaussiana. Esta máscara es una matriz cuyo tamaño y valores es distinto dependiendo del ksize y sigma que le pasemos a la función.

Lo que conseguimos con convulsionar una imagen con una máscara Gaussiana es que cada píxel de la imagen contenga valores que no difieran mucho de los píxeles vecinos. De esta forma eliminamos el ruido de la imagen, y visualmente parece que la imagen se ha difuminado.

El código para conseguir este resultado es el siguiente.

```
# Cargamos la imagen
img = cv2.imread('imagenes/cat.bmp')

# Aplicamos a la imagen un kernel Gaussiano para obtener distintas imágenes difuminadas, con distintos tamaños de máscara y valores de sigma.
difuminada1 = cv2.GaussianBlur(img,(5,5),1)
difuminada2 = cv2.GaussianBlur(img,(7,7),10)
difuminada3 = cv2.GaussianBlur(img,(3,3),5)

# OpenCV usa BGR, y matplotlib RGB. Por lo tanto, para visualizar bien las imágenes hay que transformarlas.
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
difuminada1 = cv2.cvtColor(difuminada1, cv2.COLOR_BGR2RGB)
difuminada2 = cv2.cvtColor(difuminada2, cv2.COLOR_BGR2RGB)
difuminada3 = cv2.cvtColor(difuminada3, cv2.COLOR_BGR2RGB)
```

El resultado es el siguiente.

En la segunda foto, vemos que no se ha difuminado demasiado, puesto que, aunque el tamaño de máscara es considerable, el valor de sigma es bastante pequeño. En la última foto tampoco se ven muchas diferencias, pero esta vez porque, aunque el valor de sigma es alto, el tamaño de máscara es pequeño. En la tercera foto, al ser altos tanto el valor de sigma como el tamaño de máscara, vemos que se ha difuminado bastante.

B) Usar getDerivKernels para obtener las máscaras 1D que permiten calcular la convolución 2D con máscaras de derivadas. Representar e interpretar dichas máscaras 1D para distintos valores de sigma.

getDerivKernels nos devuelve un par de matrices verticales 1xk, uno corresponde con el eje x y otro con el eje y. Si multiplicamos la traspuesta de la primera matriz por la segunda matriz, obtenemos el kernel de la derivada (respecto al eje x, y o ambos).

Este kernel tiene una cualidad importante: Al convulcionarlo con una imagen, obtenemos sus bordes. Vamos a verlo con ejemplos.

Primero, vamos a verlo con kernels 1D de tamaño 1x3, derivando respecto al eje x.

```
## Ksize = 3

## Máscara derivada respecto a 'x'
d3x_x, d3x_y = cv2.getDerivKernels(1, 0, 3, True)

## Matriz:
d3x_x = d3x_x.transpose()
d3xm = d3x_x * d3x_y
```

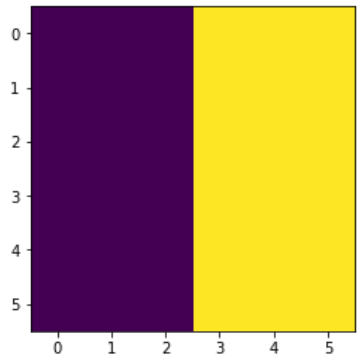
```
In [16]: d3x_x
Out[16]: array([[-1.,  0.,  1.]], dtype=float32)

In [17]: d3x_y
Out[17]:
array([[1.],
       [2.],
       [1.]], dtype=float32)

In [18]: d3xm
Out[18]:
array([[-1.,  0.,  1.],
       [-2.,  0.,  2.],
       [-1.,  0.,  1.]], dtype=float32)
```

Como vemos, esta matriz sigue un patrón claro: La columna de la izquierda contiene valores negativos, la columna central no contiene valores y la columna de la derecha contiene valores positivos. ¿Qué conseguimos con un kernel así? Conseguimos que se resalten los bordes verticales, sin modificar los bordes horizontales.

Vamos a verlo con un ejemplo.

Supongamos que nuestra imagen se compone de la siguiente matriz.

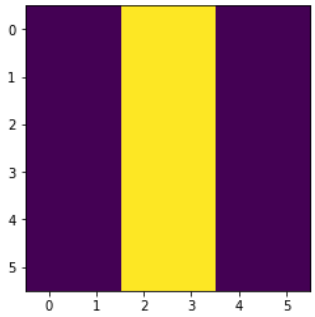
```
[[10, 10, 10, 20, 20, 20],
 [10, 10, 10, 20, 20, 20],
 [10, 10, 10, 20, 20, 20],
 [10, 10, 10, 20, 20, 20],
 [10, 10, 10, 20, 20, 20],
 [10, 10, 10, 20, 20, 20]]
```



Como vemos, hay una división clara entre las dos columnas centrales, lo que podría considerarse un borde.

Si convulsionamos esta matriz con nuestra máscara, obtenemos el siguiente resultado:

```
[[0.0, 0.0, 40.0, 40.0, 0.0, 0.0],
 [0.0, 0.0, 40.0, 40.0, 0.0, 0.0],
 [0.0, 0.0, 40.0, 40.0, 0.0, 0.0],
 [0.0, 0.0, 40.0, 40.0, 0.0, 0.0],
 [0.0, 0.0, 40.0, 40.0, 0.0, 0.0],
 [0.0, 0.0, 40.0, 40.0, 0.0, 0.0]]
```



En esta matriz resultante vemos que todos sus valores son 0, excepto en las dos columnas centrales: Las dos columnas donde antes había un borde. Esto ocurre porque en la convolución, cuando la máscara pasa por zonas sin borde, es decir, zonas de la matriz con valores parecidos, la columna de la izquierda de la convolución obtiene valores negativos (ya que es negativa), mientras que la derecha, obtiene valores positivos. Al hacer la suma total,

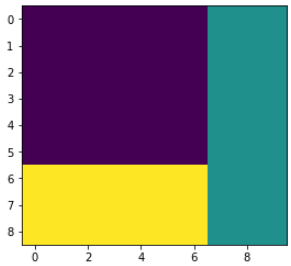
estas dos partes se anulan. En cambio, cuando se llega a una zona con borde (columnas de distinto valor), una columna de la máscara va a aplicarse sobre valores más grandes que la otra columna, haciendo que los valores no se anulen, y en vez de obtener ceros, obtenemos unos valores (positivos o negativos) que pueden representarse como un borde.

Hay que tener en cuenta que, si nuestro kernel se ha obtenido de la derivada respecto a la x, sólo va a detectar bordes verticales, puesto que los horizontales se anularían igualmente, y viceversa.

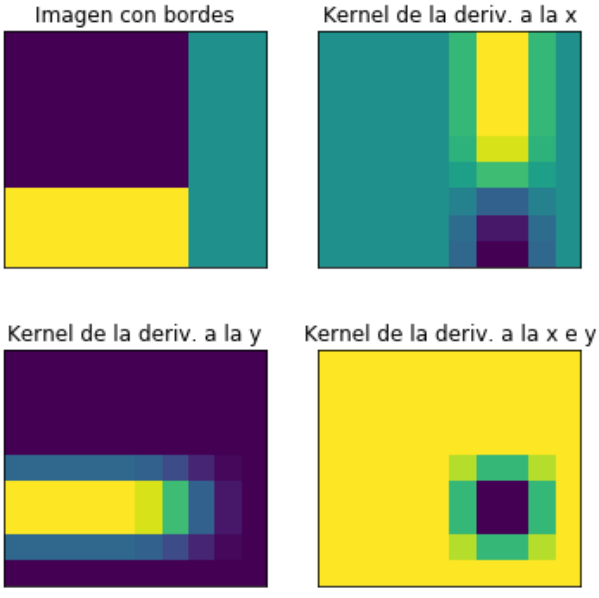
Vamos a ver un último ejemplo más completo, con una matriz con un borde horizontal y otra vertical, y vamos a aplicarles distintos kérnels de tamaño 5x5.

Nuestra matriz de prueba será la siguiente.

```
[[10, 10, 10, 10, 10, 10, 10, 20, 20, 20],
 [10, 10, 10, 10, 10, 10, 10, 20, 20, 20],
 [10, 10, 10, 10, 10, 10, 10, 20, 20, 20],
 [10, 10, 10, 10, 10, 10, 10, 20, 20, 20],
 [10, 10, 10, 10, 10, 10, 10, 20, 20, 20],
 [10, 10, 10, 10, 10, 10, 10, 20, 20, 20],
 [30, 30, 30, 30, 30, 30, 30, 20, 20, 20],
 [30, 30, 30, 30, 30, 30, 30, 20, 20, 20],
 [30, 30, 30, 30, 30, 30, 30, 20, 20, 20]]
```



Como vemos, hay tanto un borde horizontal como otro vertical. Vamos a ver el resultado de aplicarles distintos kérnels 5x5.



Vemos que, al aplicarle el kernel de la derivada respecto a la x, obtenemos el borde vertical. Si le aplicamos el kernel de la derivada respecto a la y, obtenemos el borde horizontal. Y si le aplicamos el kernel de la derivada respecto a la x y a la y, obtenemos el lugar de la imagen donde hay tanto borde vertical como horizontal.

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

El sistema que usa la función laplaciana para detectar bordes es parecido al sistema visto en el ejercicio anterior. También usa getDerivKernels, pero en vez de usar las primeras derivadas, usa las segundas. Formalmente, la función 2D laplaciana es la suma de sus segundas derivadas. En consecuencia a esto, la transición en la matriz obtenida con la laplaciana entre un valor positivo o negativo es un buen indicador de borde.

Nuestra función Laplacian calcula la laplaciana de la imagen y le pasa la máscara Sobel, obteniendo los bordes.

Vamos a ver algunos ejemplos.

Antes de nada, vamos a pasarle GaussianBlur a la imagen para eliminar el ruido antes de buscar los bordes. Vamos a pasárselo con dos valores de sigma: 1 y 3.

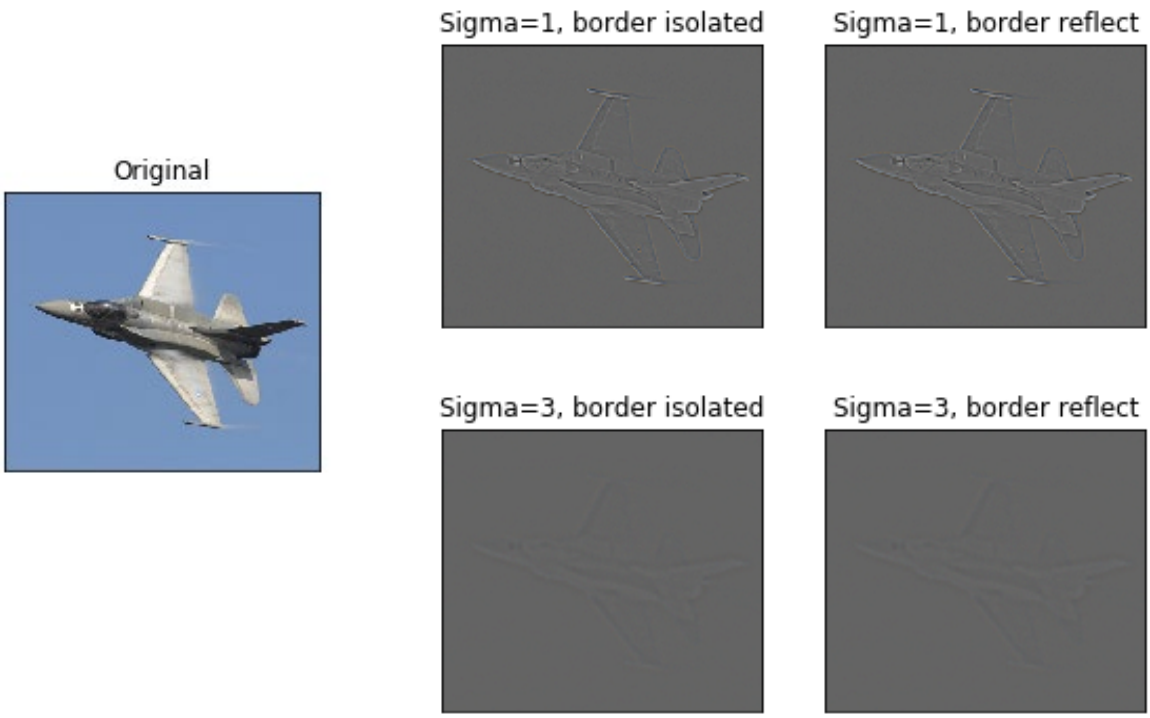
```
# Imagen
img = cv2.imread('imagenes/plane.bmp')
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

## Vamos a pasarle un blur gaussiano para eliminar ruido
imgs1 = cv2.GaussianBlur(img,(0,0),1)
imgs3 = cv2.GaussianBlur(img,(0,0),3)
```

Como tamaño de kernel le pasamos (0,0), así la función lo calcula a partir del valor de sigma.

Para cada valor de sigma, vamos a obtener las imágenes usando dos tipos distinto: BORDER_ISOLATED y BORDER_REFLECT

```
## Laplacianas

# Sigma = 1
laps1i = cv2.Laplacian(imgs1,cv2.CV_8U, borderType = cv2.BORDER_ISOLATED,delta=100)
laps1r = cv2.Laplacian(imgs1,cv2.CV_8U, borderType = cv2.BORDER_REFLECT,delta=100)

# Sigma = 3
laps3i = cv2.Laplacian(imgs3,cv2.CV_8U, borderType = cv2.BORDER_ISOLATED,delta=100)
laps3r = cv2.Laplacian(imgs3,cv2.CV_8U, borderType = cv2.BORDER_REFLECT,delta=100)
```

También se le han pasado un valor de delta de 100. Este valor es añadido al resultado para que podamos apreciar mejor los bordes obtenidos.



Con sigma = 1 los bordes se pueden apreciar relativamente bien, pero con sigma = 3 no solo eliminamos el ruido sino también difuminamos demasiado los bordes, por lo que el avión es difícil de ver.

2.- IMPLEMENTAR apoyándose en las funciones getDerivKernels, getGaussianKernel, pyrUp(), pyrDown(), escribir funciones los siguientes (3 puntos).

Usar solo imágenes de un solo canal (imágenes de gris). Valorar la influencia del tamaño de la máscara y el valor de sigma sobre la salida, en todos los casos.

A. El cálculo de la convolución 2D con una máscara separable de tamaño variable. Usar bordes reflejados. Mostrar resultados.

Las máscaras separables son máscaras de tamaño n x m que se pueden dividir en dos máscaras, la primera es una única fila y la segunda una única columna. Si multiplicas estas dos máscaras, obtienes la máscara original.

Esta propiedad nos puede resultar útil de cara a optimizar el tiempo de cómputo, ya que tratar las dos máscaras por separado cuesta menos que tratar la máscara original. Las máscaras gaussianas siempre son separables, por eso son tan útiles.

Para convolucionar una imagen con una máscara separable de tamaño variable, he implementado mi propia función, convolucionKernelSeparable(imagen, kernelx, kernely, tipo_borde). Esta función la describiré en el apartado 3 del bonus. En tipo de borde, si le pasamos 0, la función usa bordes reflejados. Si le pasamos cualquier otro número, la función usa bordes a 0.

Para obtener los kernels he usado la función getGaussianKernel, el cual devuelve un kernel gaussiano.

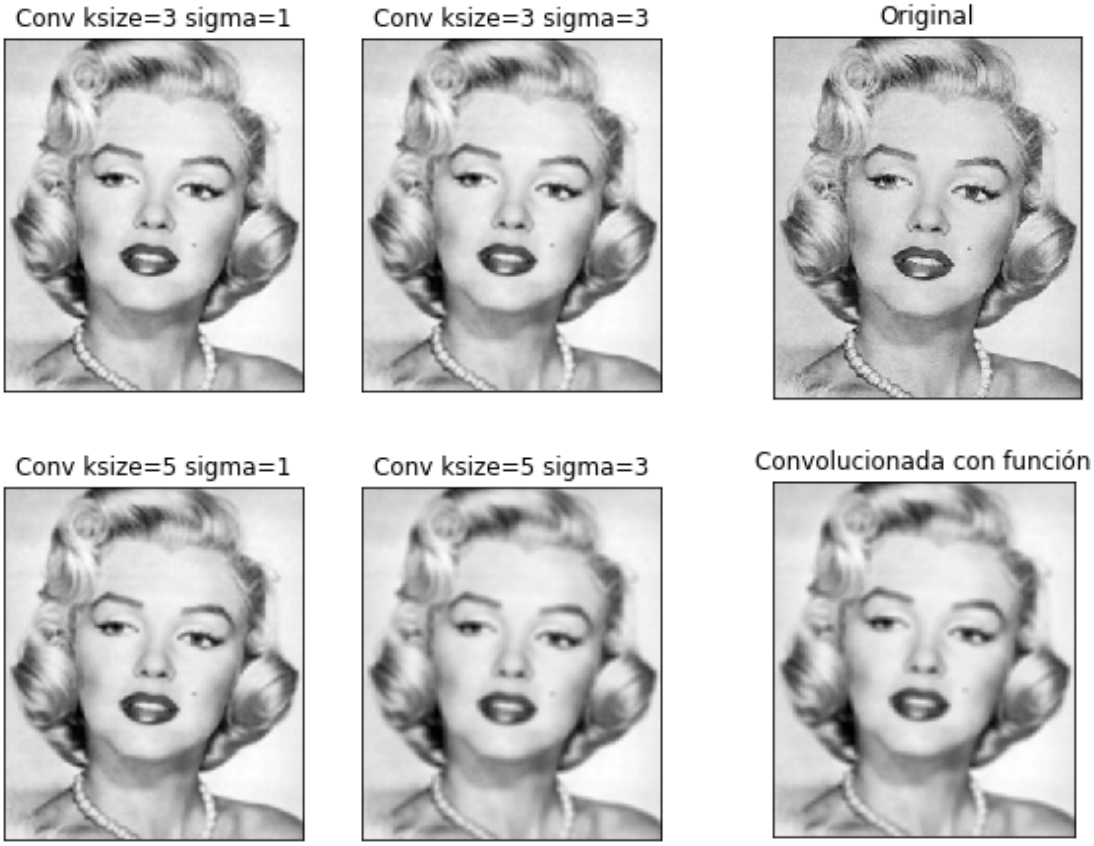
He probado cuatro combinaciones distintas: Sigma = 1 y ksize = 3, sigma = 1 y ksize = 5, sigma = 3 y ksize = 3, y sigma = 3 y ksize = 5.

El código es el siguiente.

```
# Imagen
orig = cv2.imread('imagenes/marilyn.bmp',0)

# Kernel con tamaño de máscara 3 y sigma 1
kernel_array = cv2.getGaussianKernel(3,1)
kernel31 = []
for k in range(0,len(kernel_array)):
    kernel31.append(kernel_array[k][0])

# Kernel con tamaño de máscara 5 y sigma 1
kernel_array = cv2.getGaussianKernel(5,1)
kernel51 = []
for k in range(0,len(kernel_array)):
    kernel51.append(kernel_array[k][0])
```

```
# Kernel con tamaño de máscara 3 y sigma 3
kernel_array = cv2.getGaussianKernel(3,3)
kernel33 = []
for k in range(0,len(kernel_array)):
    kernel33.append(kernel_array[k][0])

 # Kernel con tamaño de máscara 5 y sigma 3
kernel_array = cv2.getGaussianKernel(5,3)
kernel53 = []
for k in range(0,len(kernel_array)):
    kernel53.append(kernel_array[k][0])

## Convolucionamos
imgconv31 = convolucionKernelSeparable(orig,kernel31,kernel31,0)
imgconv33 = convolucionKernelSeparable(orig,kernel33,kernel33,0)
imgconv51 = convolucionKernelSeparable(orig,kernel51,kernel51,0)
imgconv53 = convolucionKernelSeparable(orig,kernel53,kernel53,0)
```

Vamos a ver los resultados.



La última foto la he convolucionado con ksize = 5 y sigma = 3 como la foto de su izquierda, pero con la función GaussianBlur, para comparar resultados y ver que nuestra función convoluciona y realiza el blur correctamente.

B. El cálculo de la convolución 2D con una máscara 2D de 1ª derivada de tamaño variable. Mostrar ejemplos de funcionamiento usando bordes a cero.

En este ejercicio vamos a usar la función getDerivKernels. Esta función nos devuelve los coeficientes en función de 'x' y de 'y' de nuestro kernel separable, en este caso para la primera derivada. Para ver dos ejemplos, vamos a usar un tamaño de apertura de 3 y de 7.

```
## Obtenemos los coeficientes del kernel, para ksize = 3 y ksize = 7.
kernel_array3 = cv2.getDerivKernels(1,1,3)

kernelx3 = []
for k in range(0,len(kernel_array3[0])):
    kernelx3.append(kernel_array3[0][k][0])

kernely3 = []
for k in range(0,len(kernel_array3[1])):
    kernely3.append(kernel_array3[1][k][0])


kernel_array7 = cv2.getDerivKernels(1,1,7)

kernelx7 = []
for k in range(0,len(kernel_array7[0])):
    kernelx7.append(kernel_array7[0][k][0])

kernely7 = []
for k in range(0,len(kernel_array7[1])):
    kernely7.append(kernel_array7[1][k][0])
```

Antes de nada, vamos a cargar la imagen y pasarle GaussianBlur para eliminar posible ruido.

```
## Cargamos la imagen
orig = cv2.imread('imagenes/bird.bmp',0)

## Le pasamos un blur para eliminar ruido
img = cv2.GaussianBlur(orig,(5,5),3)
```

Ahora vamos a obtener la convolución, usando nuestro método convolucionKernelSeparable.

```
## Obtenemos la convolución
imgconvk3 = convolucionKernelSeparable(img,kernelx3,kernely3,1)
imgconvk7 = convolucionKernelSeparable(img,kernelx7,kernely7,1)
```

El resultado es el siguiente.

Vemos que en este caso hemos obtenido mejores resultados con un ksize de 7. Hemos usado bordes a 0.

C. El cálculo de la convolución 2D con una máscara 2D de 2ª derivada de tamaño variable.

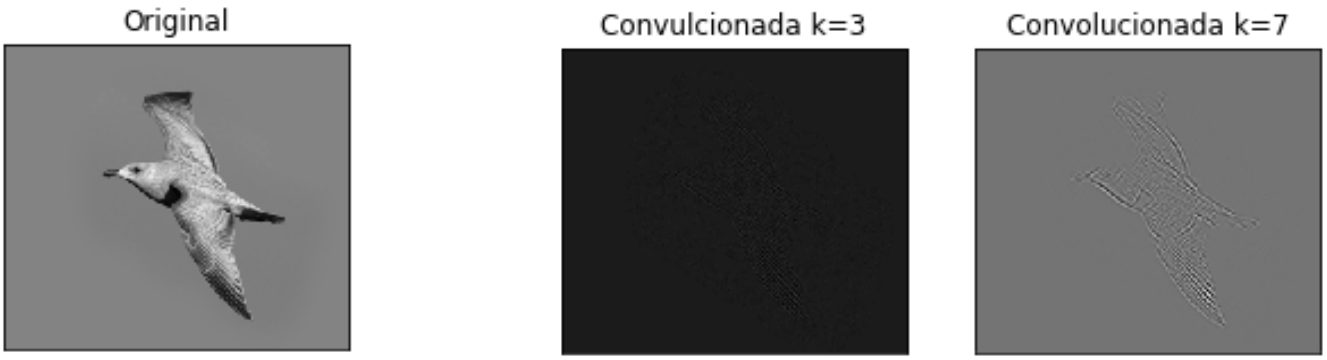
Este apartado es prácticamente igual que el anterior, con la diferencia de que esta vez obtenemos con getDerivKernels los kernels de la segunda derivada.

El código es el mismo que en el ejercicio anterior, excepto en los métodos antes mencionados:

```
## Obtenemos los coeficientes para la derivada
kernel_array3 = cv2.getDerivKernels(2,2,3)
kernel_array7 = cv2.getDerivKernels(2,2,7)
```

Igual que antes, cargamos la imagen, le pasamos GaussianBlur para eliminar ruido, y convolucionamos la imagen con los kernels.

El resultado es el siguiente.



Con tamaño de kernel k=7 podemos apreciar un poco los bordes, pero con k=3 sólo apreciamos una nube de ruido, y si nos fijamos muy bien, la silueta del pájaro. A diferencia de en el resto de ejercicios, esta no parece una buena forma de detectar bordes.

D. Una función que genere una representación en pirámide Gaussiana de 4 niveles de una imagen. Mostrar ejemplos de funcionamiento usando bordes.

Cuando vemos un cuadro de cerca, y nos alejamos, vemos como parece que los detalles se van difuminando. Esto no pasa cuando reescalamos manualmente una imagen en el ordenador. Para simular este efecto y darle realismo a los reescalados, usamos la pirámide gaussiana.

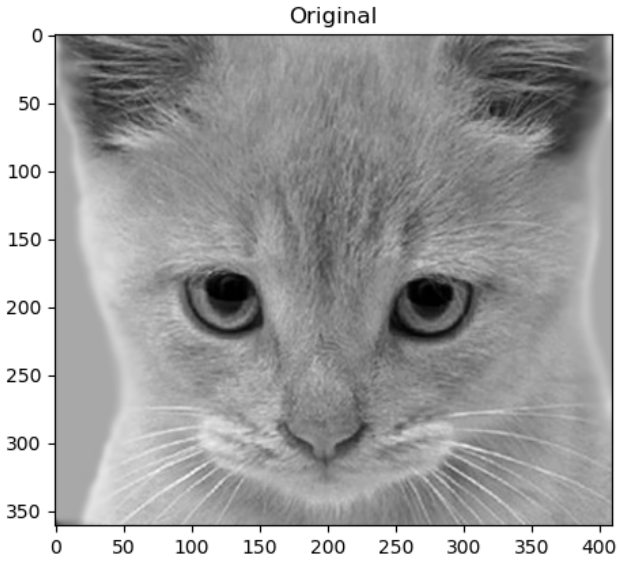
La pirámide gaussiana comienza con el nivel 0 el cual es la imagen original. Para añadirle un nivel a la pirámide, debemos aplicarle un blur gaussiano y dividir tanto su anchura como altura entre dos (hacer la imagen cuatro veces más pequeña). De esta forma obtenemos nuestra pirámide gaussiana.

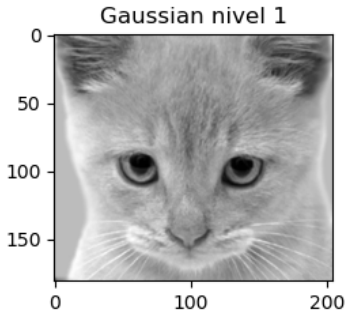
El algoritmo es el siguiente.

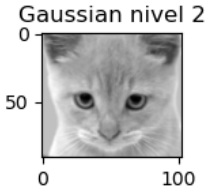
```
## Función
def gaussianPyramid(img,levels,border = 4):


    layer = img

    gaussian_pyramid = [layer]
    for i in range(levels):
        layer = cv2.pyrDown(layer,borderType = border)
        gaussian_pyramid.append(layer)


    return gaussian_pyramid
```

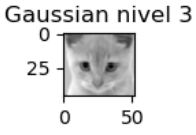
Vamos a ejemplificarlo, tanto como con bordes reflejados como “wrap”:

Bordes reflejados:

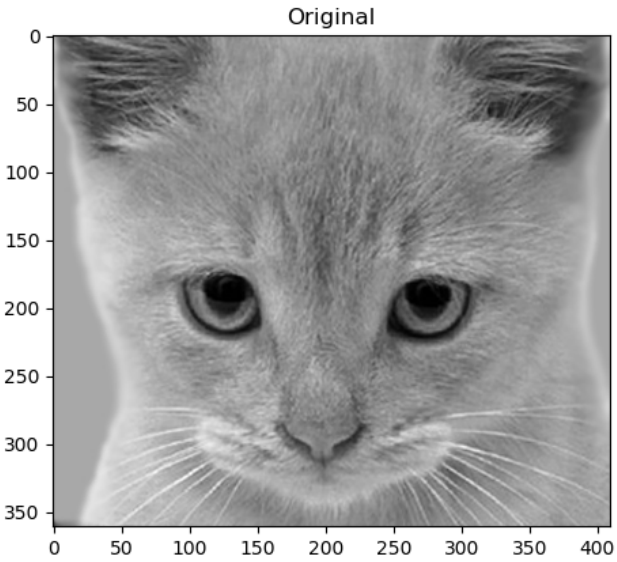


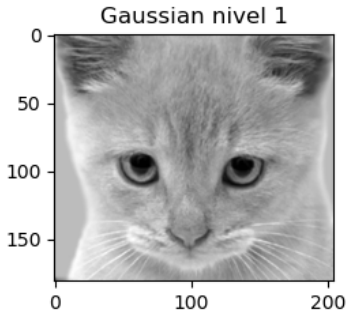


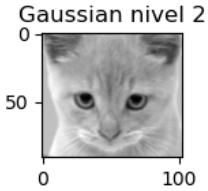


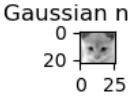


Bordes “wrap”:









Vemos que, aparte de no notarse diferencias en los distintos tipos de bordes, la imagen se va difuminando conforme se hace más pequeña, dando un efecto óptico realista.

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

La pirámide laplaciana se podría decir que está pareja con la gaussiana, puesto que se obtiene a partir de ella.

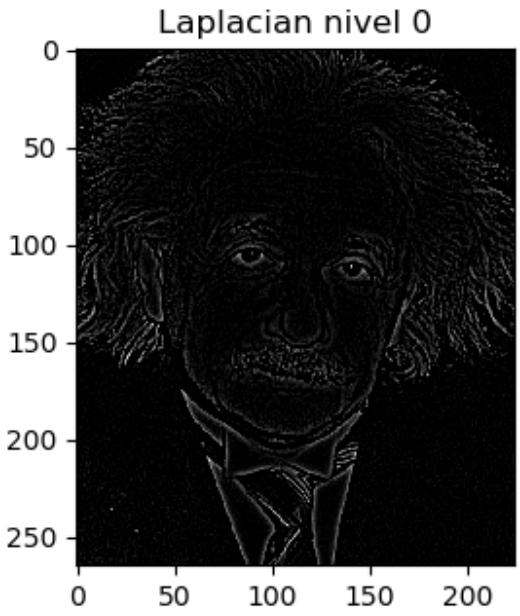
Para obtener un nivel en la pirámide laplaciana, primero obtenemos su nivel de la gaussiana equivalente, después doblamos su anchura y altura y le aplicamos un blur gaussiano, y el resultado se lo restamos al nivel anterior de la pirámide gaussiana. De esta forma obtenemos los bordes que forman el nivel laplaciano. Se obtienen los bordes porque esta imagen reescalada y con blur evidentemente contiene menos información (altas frecuencias) que la imagen de la cual restamos.

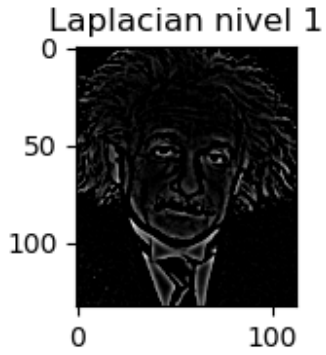
El algoritmo para obtener la pirámide laplaciana es el siguiente.

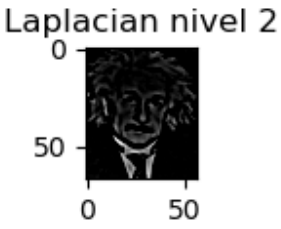
```
## Función
def laplacianPyramid(levels,gp):

    laplacian_pyramid = []
    for i in range(levels,0,-1):
        print(i)
        high_f = gp[i-1]
        size = (high_f.shape[1], high_f.shape[0])

        expanded = cv2.pyrUp(gp[i],dstsize=size)

        laplacian_pyramid.append(cv2.subtract(high_f,expanded))

    return list(reversed(laplacian_pyramid))
```

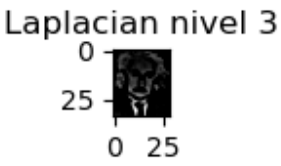
Vamos a ver un ejemplo:

Pirámide Laplaciana

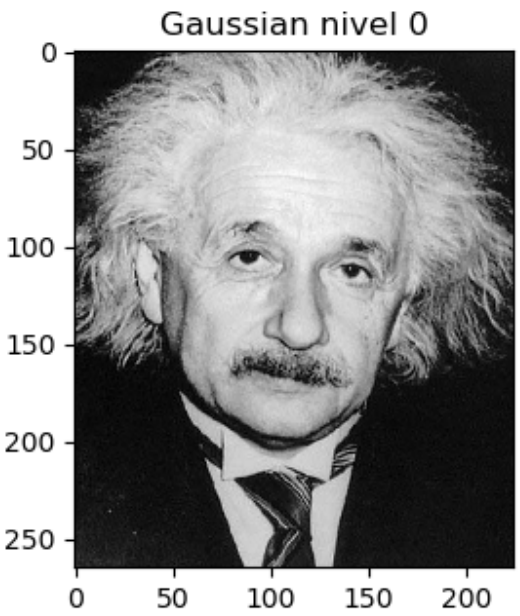


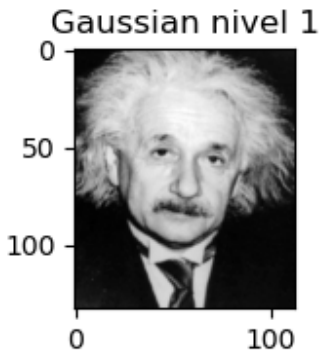


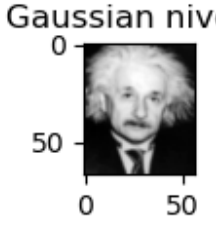


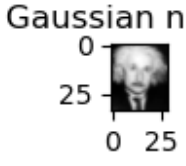


Gaussiana equivalente









3.- Imágenes Híbridas: (SIGGRAPH 2006 paper by Oliva, Torralba, and Schyns). (3 puntos)

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.

Las imágenes híbridas consisten en la suma de las frecuencias bajas de una imagen con las frecuencias altas de otra imagen, de forma que se produce el efecto óptico de ver una imagen a lo lejos, y si nos acercamos, ver otra imagen distinta.

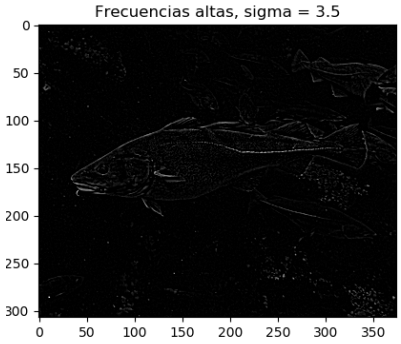
Para obtener las frecuencias bajas de una imagen podemos pasarle GaussianBlur para difuminarla.

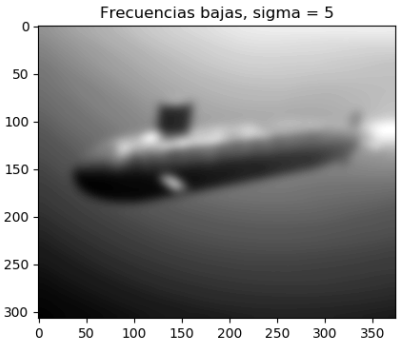
Para obtener las frecuencias altas, podemos restarle a la imagen original la imagen con frecuencias bajas (difuminada), donde nos quedaríamos con las frecuencias altas.

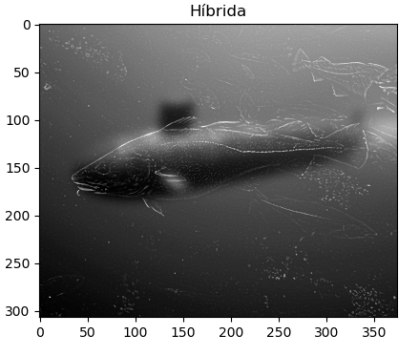
La función que realiza esta tarea es la siguiente.

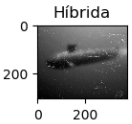
```
## Función que obtiene la imagen híbrida
def hibridas(img_baja, sigma_baja, img_alta, sigma_alta,g=0):

    img_cerca = cv2.GaussianBlur(img_baja,(0,0),sigma_baja)
    gau = cv2.GaussianBlur(img_alta,(5,5),sigma_alta)

    img_lejos = cv2.subtract(img_alta,gau)

    hibrida = cv2.add(img_cerca,img_lejos)

    imShowRealScale(img_cerca,gray=g)
    imShowRealScale(img_lejos,gray=g)
    imShowRealScale(hibrida,gray=g)
    imShowRealScale(hibrida,scale=0.25,gray=g)

    return hibrida
```

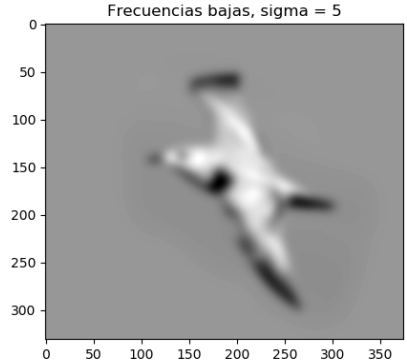
Vamos a ver tres ejemplos distintos de imágenes híbridas. Se han elegido distintos valores de sigma y tamaño de kernel para que el resultado sea óptimo.

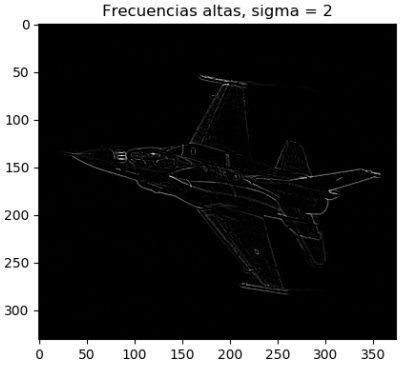


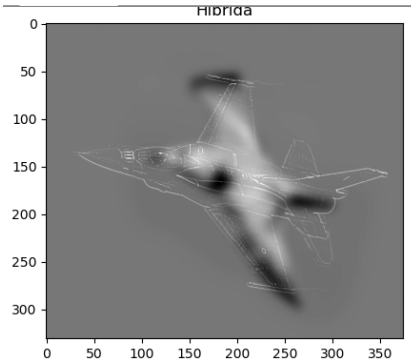


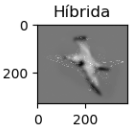


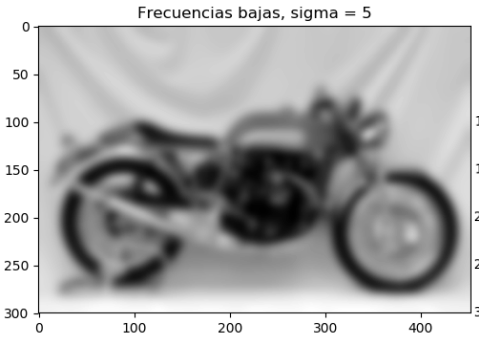


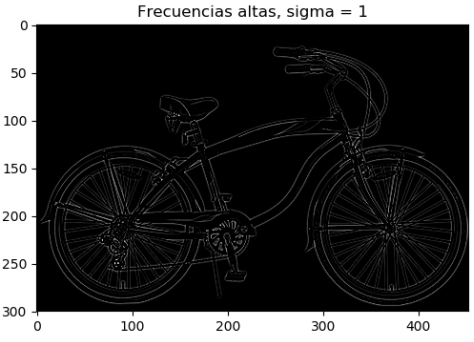


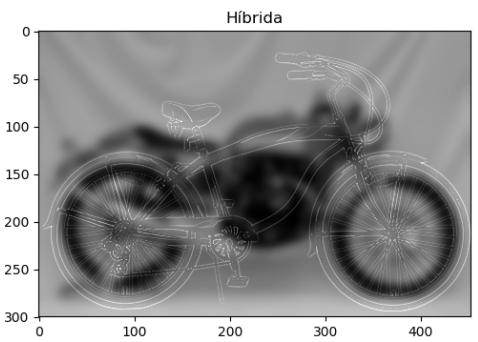


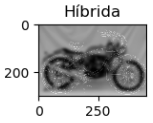












Vemos que en la híbrida grande se aprecia bien la imagen de frecuencias altas, resaltando sobre la imagen de frecuencias bajas, mientras que en la híbrida pequeña la imagen de frecuencias altas apenas se nota.

BONUS

3.- Implementar con código propio la convolución 2D con cualquier máscara 2D de números reales usando máscaras separables. (2 puntos)

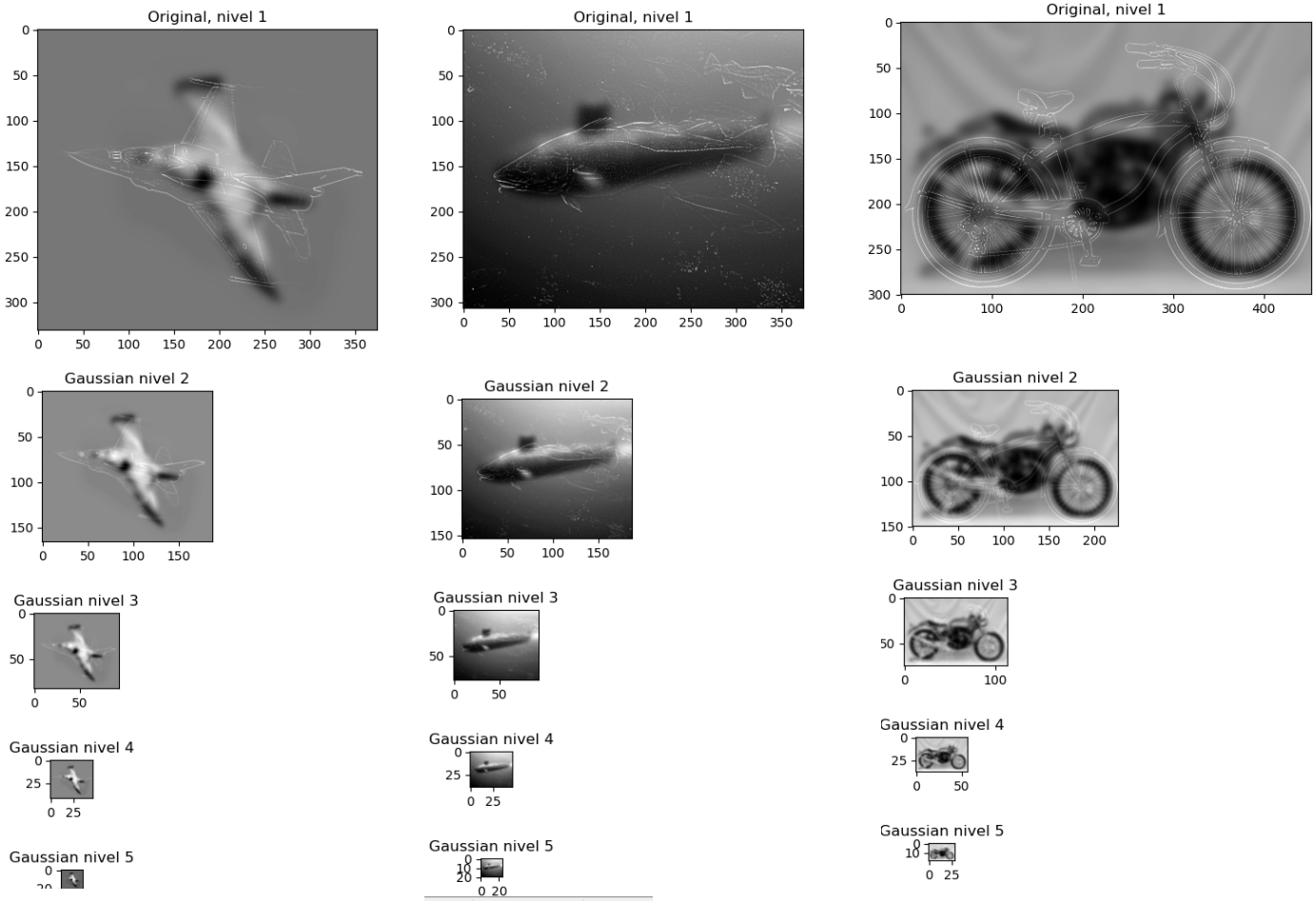
La función, la cual ya hemos usado en ejercicios anteriores, es la siguiente.

```
def convolucionKernelSeparable( orig, kernelx, kernely, tipo_borde ):
```

El código es demasiado largo y está en el archivo .py, creo que no es adecuado incluirlo en el pdf.

3.- Construir una pirámide Gaussiana de al menos 5 niveles con las imágenes híbridas calculadas en el apartado anterior. Mostrar los distintos niveles de la pirámide en un único canvas e interpretar el resultado. Usar implementaciones propias de todas las funciones usadas (0.5 puntos)

Las pirámides son las siguientes. He usado las funciones descritas en apartados anteriores.



4.- Realizar todas las parejas de imágenes híbridas en su formato a color (1 punto)

(solo se tendrá en cuenta si la versión de gris es correcta)

Al ser a color, he modificado un poco los valores de sigma. Las parejas, a color, son las siguientes.