Visión por Computador (2017-2018)
Grado en Ingeniería Informática
Universidad de Granada

Trabajo 1: Filtrado y Muestreo

Gema Correa Fernández

20 de Octubre de 2017

1.- Usando las operaciones de OpenCV: filter2D, GaussianBlur, Scharr, Sobel, getDerivKernels, getGaussianKernel, sepFilter2D, Laplacian, pyrUp(), pyrDown(), (matploptlib) subplot(), escribir funciones que implementen los siguientes puntos:

A) Una función que sea capaz de representar varias imágenes con sus títulos en una misma ventana. Usar esta función en todos los demás apartados

En este apartado nos hemos creado dos métodos. La primera función lee una imagen desde un archivo y devuelve su matriz. Para ello hacemos uso de la función imread propocionada por OpenCV. En la entrada de nuestra función, pasaremos el nombre del fichero donde está la imagen y la opción de leer la imagen en color o en escala de grises. Por tanto, además de leer imágenes en color y en escala de grises, podremos leer una imagen a color como escala de grises. Por defecto, estableceremos la lectura en color.

```
def leer_imagen(filename, flag_color=True):

  # Si flag_color es True, carga una imagen en color de 3 canales
  if (flag_color):
     image = cv2.imread(filename, flags=1);
  # Si flag_color es False, carga una imagen en escala de grises
  else:
     image = cv2.imread(filename, flags=0);

  return image
```

Con la segunda función representaremos varias imágenes con sus respectivos títulos en una misma ventana. Además, deberemos distinguir si la imagen es en gris o en color, ya que a la hora de la visualización se hará de manera distinta. Dentro de la función, comprobaremos que el número de imágenes se corresponde con el número de títulos. Para mostrar varias imágenes en una misma ventana, lo que haremos será fijar el número de columnas y a partir de ahí, calcular el número de filas necesarias. Una vez definido todo esto, pasaremos a mostrar las imágenes en una misma ventana. Para ello, deberemos obtener el tipo de imagen que estamos leyendo (en color o en gris).

```
def representar_imagenes(lista_imagen_leida, lista_titulos):

  # Comprobamos que el num de imagenes corresponde con el num de tiulos pasados
  if len(lista_imagen_leida) != len(lista_titulos): return -1

  # Calculamos el num de imagenes
  n_imagenes = len(lista_imagen_leida)

  # Establecemos por defecto el num de columnas
  n_columnas = 3
  # Calculamos el num de filas, a partir
  n_filas = (n_imagenes // n_columnas) + (n_imagenes % n_columnas)

  # Establecemos por defecto un tamanio a las imagenes
  plt.figure(figsize=(8,8))

  # Recorremos la lista de imagenes
  for i in range(0, n_imagenes):

      plt.subplot(n_filas, n_columnas, i+1) # plt.subplot empieza en 1

      if (len(np.shape(lista_imagen_leida[i]))) == 2: # Si la imagen es en gris
            plt.imshow(lista_imagen_leida[i], cmap = 'gray')
      else: # Si la imagen es en color
            plt.imshow(cv2.cvtColor(lista_imagen_leida[i], cv2.COLOR_BGR2RGB))

      plt.title(lista_titulos[i]) # Aniadimos el titulo a cada imagen

plt.xticks([]), plt.yticks([]) # Para ocultar los valores de tick en los ejes X
    e Y

  plt.show()
```

Deberemos tener cuidado con la codificación del color de visualización de OpenCV, ya que OpenCV lee en BGR, y si nosotros queremos ver la imagen con los colores de la manera en que los percibimos, deberemos cambiar a RGB con cv2.cvtColor(imagen, cv2.COLOR_BGR2RGB).

B) Una función de convolución con máscara gaussiana de tamaño variable y sigma variable. Mostrar ejemplos de funcionamiento usando dos tipos de bordes y dos valores distintos de sigma.

Antes de nada debemos esclarecer si calculamos el tamaño de la máscara a partir del σ o viceversa. En nuestro caso, introduciremos el σ y con ese valor, calcularemos el tamaño de la máscara. *A lo largo de la práctica realizaremos el mismo procedimiento.* Por tanto, la longitud de la máscara vendrá en función de σ. ¿Pero cómo calculamos el tamaño de la máscara? ¿Cómo relacionamos el tamaño de la máscara con σ? Pues bien, como la función Gaussiana está centrada en el origen, consideraremos los valores más significativos de ambos lados, tanto los de la derecha como los de la izquierda. Como conocemos que los valores más significativos se encuentran a una distancia 3 del centro (origen), cogeremos los valores dentro del intervalo $[-3\sigma, 3\sigma]$. Por consiguiente, el tamaño de nuestra máscara será de $3\sigma \cdot 2 + 1$, donde:

- 3σ: hace referencia a nuestro intervalo $[-3\sigma, 3\sigma]$
- $\cdot 2$: hace referencia a que debemos contabilizar tanto los valores del la derecha como de la izquierda
- $+1$: es el valor del centro (origen) y que debemos de contar

Por ejemplo, si le pasamos un $\sigma = 1$, el tamaño de nuestra máscara será $7 = 3 \cdot 1 \cdot 2 + 1$. Además, siempre tendremos máscaras de tamaño impar.

Para implementar este apartado, hacemos uso de la función GaussianBlur de OpenCV, la cual alisa (desenfoca) una imagen usando un filtro Gaussiano. Le pasamos como entrada el archivo de la imagen a la cual se le va a aplicar el filtro (alisado), el valor del sigma y el tipo de borde que queremos aplicar. Dentro de la función, calcularemos el tamaño de la máscara a partir del σ, y alisaremos.

OpenCV nos ofrece distintos tipos de bordes, entre ellos destacamos:

- **BORDER_REPLICATE**: aaaaaa — abcdefgh — hhhhhhh
- **BORDER_REFLECT**: fedcba — abcdefgh — hgfedcb
- **BORDER_REFLECT_101**: gfedcb — abcdefgh — gfedcba
- **BORDER_WRAP**: cdefgh — abcdefgh — abcdefg
- **BORDER_CONSTANT**: iiiiii — abcdefgh — iiiiiii

```python
def alisar_imagen(imagen, sigma, borde):

  tamanio = 3*sigma * 2 + 1  # Calculamos el tamanio de la mascara

  # Aplicamos el filtro a la imagen
  blur = cv2.GaussianBlur(imagen, ksize=(tamanio,tamanio), sigmaX = sigma,
    sigmaY=sigma, borderType=borde)

  # Cambiamos la imagen a una profundidad(color) de 8 o 16 bits
  # sin signo int (8U, 16U) o 32 bit flotante (32F)
  return blur.astype(np.uint8)
```

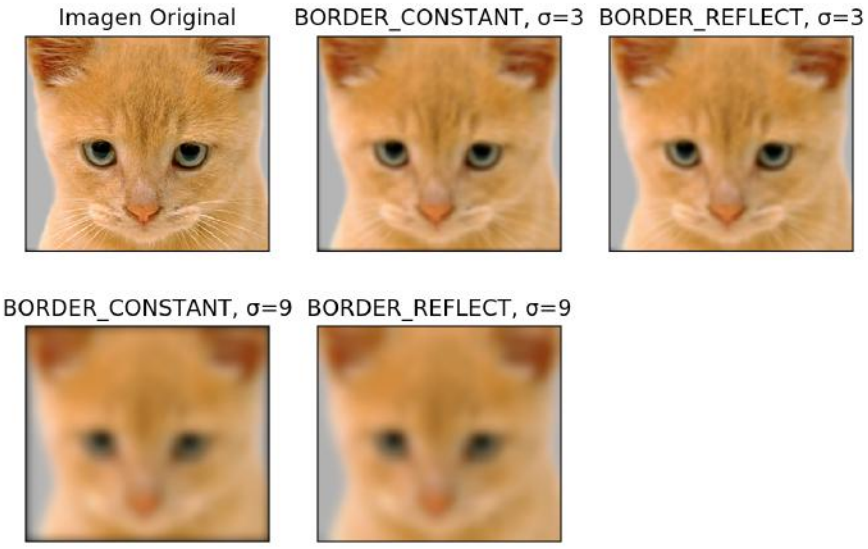

Figura 1: Aplicado del alisado a una imagen con distintos σ y bordes

Como se puede ver en la imagen, ha habido modificaciones. Analicemos cada uno de esos detalles, primero empecemos por hablar del valor del σ. Como se aprecia en la imagen, las fotos con un valor de σ más bajo realizan poco alisamiento (difuminación), sin embargo, con un σ más grande, se obtiene un mayor alisado. Esto es así, ya que cuando el sigma es más

grande estamos aumentando el tamaño de la máscara. Al aumentar el tamaño de la máscara, hacemos que afecten más píxeles a uno solo. Por último, hemos usado estos dos tipos de bordes distintos:

- *BORDER_CONSTANT* (C): borde constante, se aplica un borde negro a la imagen.
- *BORDER_REFLECT* (R): borde reflejo, se reflejan en el borde los últimos píxeles de la imagen.

Por ejemplo, en la imagen **BORDER_CONSTANT,** $\sigma = 9$ se observa que se ha aplicado un borde negro a la imagen. Como en esa imagen la difuminación es mayor, se ve bastante bien la aplicación de ese borde. Ahora comparemos **BORDER_CONSTANT,** $\sigma = 3$ con **BORDER_REFLECT,** $\sigma = 9$, podemos ver que los bordes son distintos, ya que se puede apreciar que en *BORDER_REFLECT,* $\sigma = 9$ los bordes son reflejados.

C) Una función de convolución con núcleo separable de tamaño variable. Mostrar ejemplos de funcionamiento usando dos tipos de bordes y dos valores distintos de sigma.

Antes de nada, deberemos distinguir dos casos para hacer la convolución, ya que se realizará un tratamiento distinto de convolución dependiendo de los canales de la imagen. Si la imagen de entrada tiene solo un canal es una imagen en escala de grises y si tiene tres canales es una imagen en formato color. Básicamente, la idea principal de este apartado consiste en aplicar la convolución a una imagen (matriz) primero por filas, obteniendo así una nueva matriz. Esa nueva matriz la transponemos y volvemos a realizar convolución por filas (siendo en ese caso las columnas de la matriz que hemos transpuesto). Una vez terminado, volvemos a trasponer la matriz para así obtener nuestra imagen final. Veamos un pequeño esquema donde se explica el proceso.

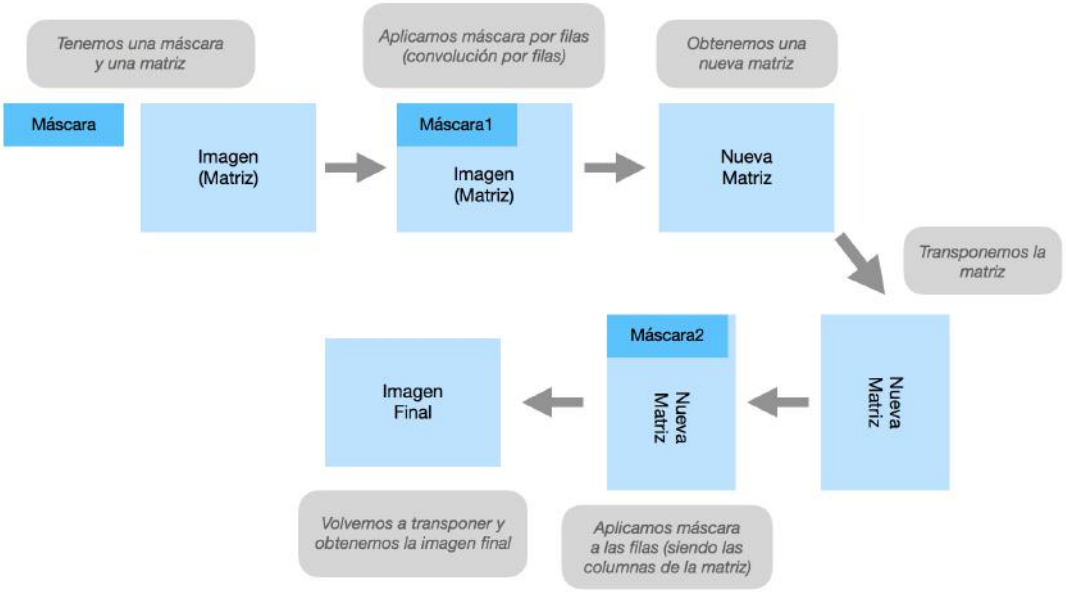

Figura 2: Esquema de aplicar convolución primero por filas y luego por columnas

***Nota**: Podremos pasar la misma máscara para filas o columnas o distintas máscaras.*

Lo importante de este ejercicio es saber qué máscara aplicar por filas y qué máscara aplicar por columnas. Como estamos con una máscara Gaussiana (máscara isotrópica), vamos a aplicar la

misma máscara tanto para filas como para columnas.

Veamos lo que hace nuestro código. Para realizar convolución haremos uso de la función filter2D de OpenCV, la cual convoluciona una imagen con un kernel. Primero, comprobaremos el tamaño de la imagen con *np.shape*(*imagen*) para diferenciar entre imagen en color o en escala de grises. Una vez, hecho eso, realizaremos el mismo proceso de convolución de la *Figura2* en ambas partes. Simplemente que si la imagen resulta ser en color, deberemos hacer convolución por cada uno de sus canales y si la imagen es en escala de grises solo lo haremos una vez.

```
def convolucion_separable(imagen, borde, mask_fila, mask_col):

  nueva_imagen = np.copy(imagen) # Hago copia de la imagen de entrada

  if len(np.shape(imagen)) == 2: # Si imagen en escala de grises

     # Iteramos por filas
     for i in range(np.shape(imagen)[0]):

          # Aplicamos convolucion
          fila = cv2.filter2D(imagen[i,:], -1, mask_fila, borderType=borde)
          nueva_imagen[i,:] = fila[:,0]
          # debemos poner fila[:,0] ya que filter2D crea una estructura de
          # canales, pero la imagen no tiene canales

     # Transponemos para cambiar filas por columnas
     nueva_imagen = nueva_imagen.transpose(1,0)

     # Iteramos ahora las columnas
     # como hemos transpuesto las tratamos como filas
     for i in range(np.shape(imagen)[1]):

          # Aplicamos convolucion
          columna = cv2.filter2D(nueva_imagen[i,:], -1, mask_col,
                                 borderType=borde)
          nueva_imagen[i,:] = columna[:,0]

     # Volvemos a trasponer para obtener la imagen definitiva
     nueva_imagen = nueva_imagen.transpose(1,0)

  else: # Imagen en color

     # Iteramos filas
     for i in range(np.shape(imagen)[0]):

          # Aplicamos convolucion por cada cada canal
          # filter2D separa por canales --> imagen[i,:,:]
          fila = cv2.filter2D(imagen[i,:,:], -1, mask_fila, borderType=borde)
          nueva_imagen[i,:,:] = fila

     # Transponemos para cambiar filas por columnas
     nueva_imagen = nueva_imagen.transpose(1,0,2)

     # Iteramos ahora por columnas
     # como hemos transpuesto las tratamos como filas
     for i in range(np.shape(imagen)[1]):
          # Aplicamos convolucion por cada cada canal
          columna = cv2.filter2D(nueva_imagen[i,:,:], -1, mask_col,
                                 borderType=borde)
          nueva_imagen[i,:,:] = columna

     # Volvemos a trasponer para obtener la imagen original definitiva
     nueva_imagen = nueva_imagen.transpose(1,0,2)

  return nueva_imagen.astype(np.uint8)
```

Ahora vamos a probar su funcionamiento, para ello calculo la máscara utilizando la función getGaussianKernel de OpenCV, la cual devuelve la máscara a usar para filtrar la imagen. Como hemos comentado anteriormente usaremos la misma máscara tanto para filas como para columnas.

Establecemos un valor σ de 2, donde el tamaño de la máscara será de 19; y otro con un valor σ de 7, donde el tamaño de la máscara será de 43. Se puede ver que se obtiene la misma conclusión que en el apartado anterior, cuanto más grande es σ mayor difuminado se produce. Vamos a usar dos tipos de bordes **BORDER_CONSTANT** y **BORDER_REPLICATE**, el cual copia el último pixel de cada fila/columna.

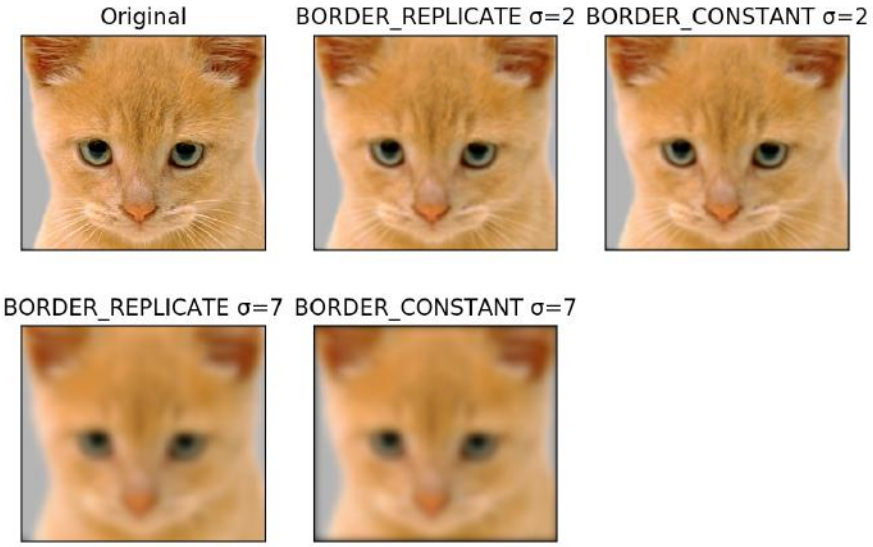

Figura 3: Ejemplos de imagen convolucionada por filas y por columnas con distintos σ y bordes

Podemos ver que da unos resultados razonables a simple vista, pero hagamos una comparación con la función sepFilter2D de OpenCV, la cual aplica un filtro lineal separable a una imagen. Para comparar, usaremos la misma máscara $\sigma = 2$ y borde $BORDER\_REPLICATE$.

```
# Calculamos las mascaras
mask_fila1 = cv2.getGaussianKernel(ksize=13, sigma=2)
mask_col1 = mask_fila1
# my_sepFilter2D
img2 = convolucion_separable(img1, cv2.BORDER_REPLICATE, mask_fila1, mask_col1)
# opencv_sepFilter2D
img6 = cv2.sepFilter2D(img1, -1, kernelX = mask_fila1, kernelY = mask_col1,
    borderType = cv2.BORDER_REPLICATE)
```

Figura 4: Comparación entre convolución hecha por mí y la de OpenCV

A simple vista parece que realiza el mismo difuminado, pero para asegurarnos comparemos las matrices de cada una de las convoluciones. Debido a que poner la matriz de cada imagen es mucha ocupación, pondré el principio y el final de ambas.

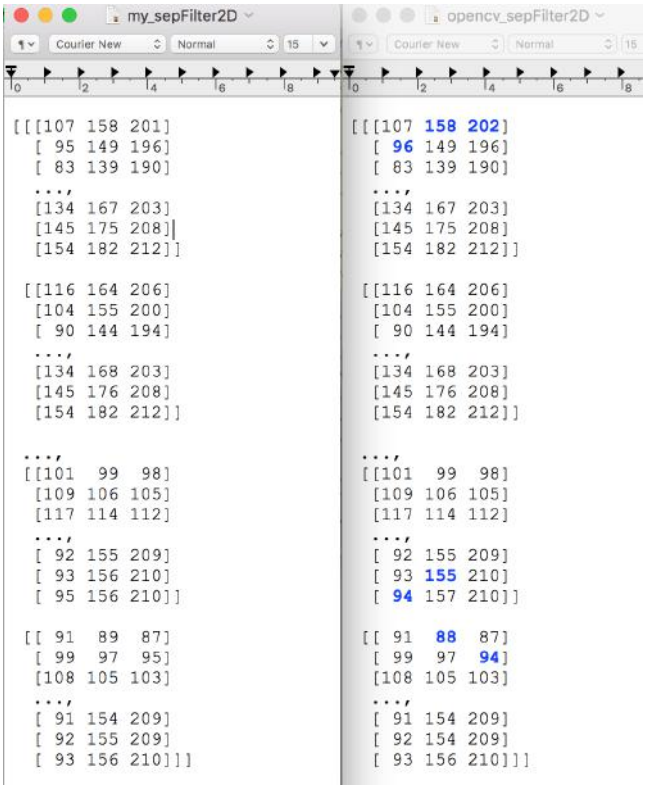

Figura 5: Comparación de matrices entre mi convolución y la de OpenCV

La matriz de la izquierda es la resultante de mi función y la de la derecha es la de OpenCV. Como se puede apreciar hay pocos valores distintos y los que son diferentes, difieren en 1 arriba o abajo, por lo que se puede despreciar.

Pero para estar más seguros de nuestra función, vamos a realizar un pequeño experimento. Vamos a aplicar dos máscara distintas a una imagen, es decir, vamos a producir un distinto alisado por filas ($\sigma = 3$) y por columnas ($\sigma = 7$). Para ello nos hemos creamos una imagen con un fondo blanco, donde haya una cruz negra.

```
# Mascara para filas
mask_fila = cv2.getGaussianKernel(ksize=19, sigma=3)
```

```
# Mascara para columnas
mask_col = cv2.getGaussianKernel(ksize=43, sigma=7)
# Aplicamos convolucion
img5 = convolucion_separable(img4,cv2.BORDER_REFLECT, mask_fila, mask_col)
```

Antes de visualizar nada, podemos llegar a la conclusión de que hay un mayor alisado por columnas que por filas, puesto que el valor de σ es más grande. Por lo tanto, se obtendrá un mayor difuminado de arriba abajo, afectando a los bordes que estén en horizontal . Veámoslo:

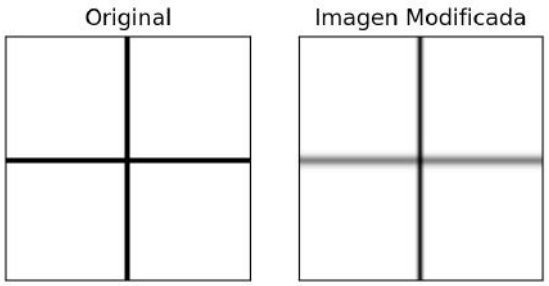

Figura 6: Aplicación de máscaras distintas por columnas y por filas

Como se observa, se produce un mayor alisamiento por columnas.

D) Una función de convolución con núcleo de 1ª derivada de tamaño variable. Mostrar ejemplos de funcionamiento usando dos tipos de bordes y dos valores distintos de sigma.

En este apartado haremos uso de la la convolución implementada en el apartado C, simplemente que ahora el kernel será otro. En concreto, nos pide que usemos la convolución con núcleo de 1ª derivada. Para calcular el núcelo, hacemos uso de la función getDerivKernels de OpenCV, la cual devuelve coeficientes de filtro para calcular derivadas de imágenes. Simplemente, tendremos que pasarle a esta función, el orden de la derivada y el tamaño de la máscara.

Como estamos usando la máscara Guassiana, podríamos haber optado por utilizar la función de separabilidad del filtro gaussiano que viene en las transparencias de teoría. La cual puede ser expresada como producto de dos funciones, una función para la x y otra función para la y. Siendo en este caso las dos funciones idénticas.

$$G_\sigma(x,y) = \frac{1}{2\pi\sigma^2}\exp^{-\frac{x^2+y^2}{2\sigma^2}} = \frac{1}{\sqrt{2\pi}\sigma}exp^{-\frac{x^2}{2\sigma^2}} \cdot \frac{1}{\sqrt{2\pi}\sigma}exp^{-\frac{y^2}{2\sigma^2}}$$

En nuestra función primero calculamos la derivada de primer orden del núcleo y obtener la derivada respecto de X $\frac{\partial(x,y)}{\partial x}$ y la derivada respecto de Y $\frac{\partial(x,y)}{\partial y}$. Asignamos la derivada respecto de X a las filas y la derivada respecto de Y a las columnas. Una vez hecho, hago convolución con la máscara asignada a las filas y hago convolución con la máscara asignada a las columnas y represento ambas imágenes. Veamos la función que hemos implementado:

```
def primera_derivada(imagen, sigma, borde):

  # Calculamos el tamanio
  tamanio = 3*sigma*2+1
```

```
# Hacemos la primera derivada del nucleo y nos devolvera
# una para X y otra para Y
mascara = cv2.getDerivKernels(dx=1, dy=1, ksize=tamanio)
# Le asignamos la primera derivada a las filas (X)
mascara_fila = mascara[0]
# Le asignamos la primera derivada a las filas (Y)
mascara_col = mascara[1]

# Usamos la convolucion del apartado C
# Para conseguir los bordes horizontales
# --> por eso necesitas pasar por columnas
bordes_horizontales = convolucion_separable(imagen, cv2.BORDER_REFLECT, None,
  mascara_fila)
# Para conseguir los bordes verticales
# --> por eso necesitas pasar por filas (es cuando cruza)
bordes_verticales = convolucion_separable(imagen, cv2.BORDER_REFLECT,
  mascara_col, None)

# Normalizamos las matrices
# Representamos las imagenes
```

Debemos tener cuidado con los valores que puede contener la matriz una vez hecha la derivada, por eso necesitamos normalizarla.

Vamos a probar su resultado. Aunque antes de aplicar la derivada tenemos que alisar. Ya que si mi imagen tiene ruido, es decir, donde un píxel tiene una frecuencia más alta que con los de su alrededor, al alisar, nos desprendemos en gran parte del ruido. Por lo tanto, nos quedamos con los bordes más representativos de la figura. En mi caso, aplicaré un alisado a la imagen de entrada.

Aliso la imagen antes y le asigno un $\sigma = 1$ y borde reflejo.

```
img1 = alisar_imagen(img, sigma = 1, borde = cv2.BORDER_REFLECT)
img2 = primera_derivada(img1, sigma = 1, borde = cv2.BORDER_REFLECT)
```

Figura 7: Visualización de la primera derivada con $\sigma = 1$

Aliso la imagen antes y le asigno un $\sigma = 2$ y un borde constante. Como vamos a usar un valor de σ más alto, eliminaremos más frecuencias altas, por lo que los bordes estarán más difusos. Sin embargo con esta representación estamos detectando cuales son las fronteras más definidas en el imagen.

```
img1 = alisar_imagen(img, sigma = 2, borde = cv2.BORDER_CONSTANT)
img2 = primera_derivada(img1, sigma = 1, borde = cv2.BORDER_CONSTANT)
```

Figura 8: Visualización de la primera derivada con $\sigma = 2$

Como se puede ver en las imágenes, en la imagen de **bordes horizontales** se ve la figura más por filas, sin embargo para la de **bordes verticales** se ve que como la imagen acentúa más las columnas. Ahora, hablemos de la diferencia entre los valores de σ, en la Figura 7, le hemos dado un valor de 1 y como sabemos no alisa mucho la imagen, por lo que los bordes son más distinguibles. Sin embargo, en la Figura 8, como el alisado es mayor, los bordes son cogidos de menor manera. Por ejemplo, con un σ mayor, perdemos los bordes que definen los pelitos del bigote del gato. O fronteras representativas, que con un σ valor no pasaba.

E) Una función de convolución con núcleo de 2a derivada de tamaño variable. Mostrar ejemplos de funcionamiento usando dos tipos de bordes y dos valores distintos de sigma.

Para realizar este ejercicio, hacemos exactamente el mismo procedimiento que el apartado anterior, simplemente que ahora el orden de las derivadas es 2, por lo tanto en la función de *detDerivKernel*, debemos cambiar el orden.

```
def segunda_derivada(imagen, sigma, borde):

  # Calculamos el tamanio
  tamanio = 3*sigma*2 + 1

  # CAMBIO CON RESPECTO LA PRIMERA DERIVADA
  # Hacemos la segunda derivada del nucleo y nos devolvera
  # una para X y otra para Y
  mascara = cv2.getDerivKernels(dx=2, dy=2, ksize=tamanio)

  # Le asignamos la primera derivada a las filas (X)
  mascara_fila = mascara[0]
  # Le asignamos la primera derivada a las filas (Y)
  mascara_col = mascara[1]

  # Usamos la convolucion del apartado C
  # Para conseguir los bordes horizontales
  # --> por eso necesitas pasar por columnas
  bordes_horizontales = convolucion_separable(imagen, cv2.BORDER_REFLECT, None,
    mascara_fila)
  # Para conseguir los bordes verticales
  # --> por eso necesitas pasar por filas (es cuando cruza)
  bordes_verticales = convolucion_separable(imagen, cv2.BORDER_REFLECT,
    mascara_col, None)

  # Normalizamos imagenes de derivadas
  # Representamos las imagenes
```

Hago los mismos ejemplos que antes. Primero aliso la imagen y le asigno un $\sigma = 1$ y borde reflejo.

```
img1 = alisar_imagen(img, sigma = 1, borde = cv2.BORDER_REFLECT)
img2 = segunda_derivada(img1, sigma = 1, borde = cv2.BORDER_REFLECT)
```

Figura 9: Visualización de la segunda derivada con $\sigma = 1$

Aliso la imagen antes y le asigno un $\sigma = 2$ y borde constante.

```
img1 = alisar_imagen(img, sigma = 2, borde = cv2.BORDER_CONSTANT)
img2 = segunda_derivada(img1, sigma = 1, borde = cv2.BORDER_CONSTANT)
```

Figura 10: Visualización de la segunda derivada con $\sigma = 2$

Como se ve en la Figura 9, obtenemos las imágenes con un menor contorno, del que hemos obtenido con la primera derivada. Porque como bien sabemos, donde hay máximos en la primera derivada, ahora son 0 en la segunda derivada. Como hemos usado un valor de σ más alto, eliminaremos más frecuencias altas, por lo que los bordes estarán más difusos.

F) Una función de convolución con núcleo Laplaciana-de-Gaussiana de tamaño variable. Mostrar ejemplos de funcionamiento usando dos tipos de bordes y dos valores distintos de sigma.

Debemos aplicar el mismo procedimiento que en los anteriores apartados, simplemente hay que saber que la Laplaciana-de-Gaussiana es la suma de las segundas derivadas.

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

Por tanto, hacemos lo mismo que antes, solo que ahora obtendremos una imagen con los bordes horizontales y los bordes verticales. Para comprobar que nuestra laplaciana está bien, haremos una comparación con la laplaciana de OpenCV. En nuestra función, calcularemos la segunda derivada del núcleo, sumaremos ambas y haremos convolución.

```python
def laplaciana(imagen, sigma, borde):

    tamanio = 3*sigma*2 + 1 # Calculamos el tamanio mascara

    # Hacemos la segunda derivada del nucleo y nos devolvera una
    # para X y otra para Y
    mascara = cv2.getDerivKernels(dx=2, dy=2, ksize=tamanio)
    # Sumamos ambas derivadas
    laplaciana = mascara[0] + mascara[1]

    # Normalizamos la imagen
    imagen = cv2.normalize(imagen, imagen, 0, 255, cv2.NORM_MINMAX)

    # Hacemos convolucion
    laplace = cv2.filter2D(imagen, -1, kernel = laplaciana, borderType=borde)

    return laplace.astype(np.uint8)
```

Figura 11: Comparación de la Laplaciana con σ 1 y 2

Se puede ver en la imagen, que ahora mostramos en una misma foto tanto los bordes horizontales, como verticales. Ahora comparemos con la función de OpenCV:

```python
# opencv_laplacian
img = alisar_imagen(imagen, 1, borde=cv2.BORDER_REFLECT)
img = cv2.Laplacian(img, -1, ksize=1, borderType=cv2.BORDER_REFLECT)
img = img.astype(np.uint8)
# my_laplacian
img1 = alisar_imagen(img1, 1, borde=cv2.BORDER_REFLECT)
img2 = laplaciana(img1, sigma = 1, borde = cv2.BORDER_REFLECT)
```

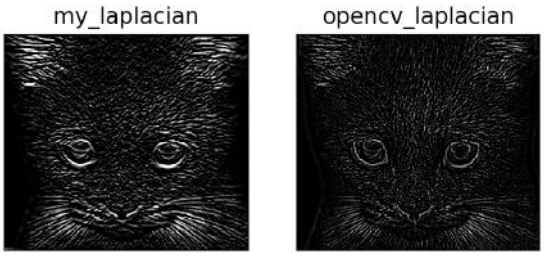

Figura 12: Comparación de la Laplaciana mía y la de OpenCV

A simple se ve que ambas imágenes son similares, pero vamos a comparar las matrices, para

estar más seguros de la conclusión. Una vez que visualizamos las matrices, vemos que ambas toman valores parecidos, y que defieren en algunos valores, ya que opencv es más preciso que mi función.

G) Una función que genere una representación en pirámide Gaussiana de 4 niveles de una imagen. Mostrar ejemplos de funcionamiento usando dos tipos de bordes y dos valores distintos de sigma.

Nota: le paso un nivel 5, porque yo cuento que el primer nivel es la imagen original, por lo tanto escalo las otras 4 imágenes.

Antes de nada, debemos saber que las pirámides gaussiana son un tipo de *collage*, en el cual se representa la misma imagen con distintas escalas. Partimos de una imagen ventana, donde se irán poniendo las demás imágenes. Nuestra pirámide tendrá la imagen original a la izquierda y colocará el resto de imágenes a la derecha, una detrás de otra. Entendemos que cuando dice una pirámide de 5, se refiere a la imagen original más las tres de la derecha. Por lo que las imágenes de la derecha tendrán una escala de 1/2, 1/4, 1/8 y 1/16. Para obtener todo, debemos hacer un redimensiado y un alisamiento, ambas tareas serán llevadas a cabo por la función pyrDown de OpenCV, la cual difumina una imagen y disminuye la resolución de la misma. Un dato a tener en cuenta, es que debemos saber que en este apartado no podremos usar distintos valores de σ ya que la función de OpenCV no nos lo permite.

Nuestra función recibe por entrada la imagen sobre la que se realizará la pirámide, los niveles para la pirámide y el tipo de borde. Y mostrará en la salida la pirámide. Además, deberemos distinguir cuando la imagen es en color y cuando es en escala de grises. Dentro de cada posibilidad, me creo una nueva imagen, que será la la matriz ventana, la cual tendrá de alto el tamaño de las filas de la imagen original y de ancho tendrá la suma de los tamaños de las columnas de la imagen original y de la siguiente imagen escalada. Meteremos la primera imagen y la segundo a mano, y a partir de ahí, como tenemos las imágenes escaladas en un array, solo habrá que calcular la altura que separa unas de otras para meterlas en el lugar correspondiente.

```
def piramide_gaussiana(imagen, levels, borde):

  gaussianPyramid = [imagen.copy()]  # Genero una copia de la imagem de entrada

  for i in range(1, levels): # Recorro los niveles

     # Obtengo cada imagen con su tamanio y su alisado
     img = cv2.pyrDown(gaussianPyramid[i - 1], borderType=borde)
     gaussianPyramid.append(img) # Y la meto en una lista

  if (len(np.shape(imagen))) == 3: # Compruebo si la imagen es en color

     # Creo una imagen, con el tamanio de todas las imagenes a meter
     nueva_imagen = np.empty((np.shape(imagen)[0],math.ceil(np.shape(imagen)
    [1]*1.5), np.shape(imagen)[2]))

     # Meto la primera imagen a mano (estara a la izquierda)
     nueva_imagen[0:np.shape(gaussianPyramid[0])[0], 0:np.shape(gaussianPyramid
    [0])[1], 0:3] = gaussianPyramid[0]
     # Meto la segunda imagen a mano (estara a la derecha)
     nueva_imagen[0:np.shape(gaussianPyramid[1])[0], np.shape(gaussianPyramid
    [0])[1]:math.ceil(np.shape(gaussianPyramid[1])[1]+np.shape(gaussianPyramid
    [0])[1]),0:3] = gaussianPyramid[1]

     # Recorro los niveles que me quedan
     for i in range(2, levels):

          # Calculo la altura de cada imagen, para aniadirla a continuacion
          # de la segunda imagen. Por eso voy sumando las alturas, para
```

```
            # que asi vaya una detras de otra
            altura = 0
            for j in range (1, i):
                altura = altura + np.shape(gaussianPyramid[j])[0]

            # Ya tengo mi imagen con la piramide en color
            nueva_imagen[altura:(np.shape(gaussianPyramid[i])[0]+altura), np.shape
    (gaussianPyramid[0])[1]:math.ceil(np.shape(gaussianPyramid[i])[1]+np.shape(
    gaussianPyramid[0])[1]), 0:3] = gaussianPyramid[i]

    # Compruebo si la imagen es en blanco y negro
    # Hago el mismo procedimiento que para las imagenes en color,
    # simplemente que ahora no tengo que tener en cuenta los canales
    [...]

    # Devuelvo la piramide
    return nueva_imagen.astype(np.uint8)
```

Figura 13: Pirámide Gaussiana en color

Figura 14: Pirámide Gaussiana en escala de grises

A continuación, muestro los mismos ejemplos con distintos bordes.

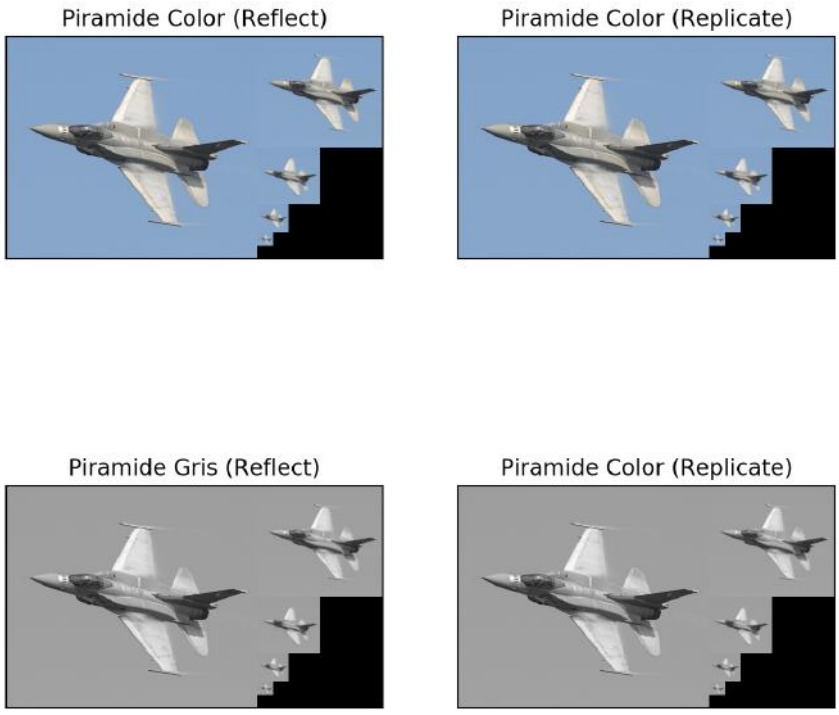

Figura 15: Pirámide Gaussiana con distintos bordes

H) Una función que genere una representación en pirámide Laplaciana de 4 niveles de una imagen. Mostrar ejemplos de funcionamiento usando dos tipos de bordes y dos valores distintos de sigma. Valorar la influencia del tamaño de la máscara y el valor de sigma sobre la salida, en todos los casos.

En este apartado, hacemos el mismo proceso que con la pirámide gaussiana, la funciones prácticamente igual. Simplemente que ahora el filtro que se pasa es distinto, ahora a la imagen, le paso la Laplacian. Pero el proceso de crear la pirámide y de redimensionar cada imagen es el mismo. Sin embargo, el borde se lo aplicaré a mi función laplaciana.

```
def piramide_laplaciana(imagen, levels):

  # Aplicar mi funcion laplaciana a la imagen original
  # Aliso la imagen
  imagenL = laplaciana(imagen, 1, cv2.BORDER_REFLECT)

  # Genero una copia de la imagem de entrada
  laplacianPyramid = [imagenL]

  # Recorro los niveles
  for i in range(1, levels):

      # Alisado y reescala
      img = cv2.pyrDown(laplacianPyramid[i - 1])
      # Creo una lista con las imagenes laplacianas
      laplacianPyramid.append(img)

  # Compruebo si la imagen es en color
  if (len(np.shape(imagen))) == 3:

      # Creo una matriz imagen, con el tamanio de todas las imagenes a meter
      nueva_imagen = np.empty((np.shape(imagen)[0], math.ceil(np.shape(imagen)
    [1]*1.5), np.shape(imagen)[2]))
```

```
    # Meto la primera imagen a mano (estara a la izquierda)
    nueva_imagen[0:np.shape(laplacianPyramid[0])[0], 0:np.shape(
    laplacianPyramid[0])[1], 0:3] = laplacianPyramid[0]
    # Meto la segunda imagen a mano (estara a la derecha)
    nueva_imagen[0:np.shape(laplacianPyramid[1])[0], np.shape(laplacianPyramid
    [0])[1]:math.ceil(np.shape(laplacianPyramid[1])[1]+np.shape(laplacianPyramid
    [0])[1]), 0:3] = laplacianPyramid[1]

    # Recorro los niveles que me quedan
    for i in range(2, levels):

        # Calculo la altura de cada imagen, para aniadirla a continuacion
        # de la segunda imagen. Por eso voy sumando las alturas, para
        # que asi vaya una detras de otra
        altura = 0
        for j in range (1, i):
            altura = altura + np.shape(laplacianPyramid[j])[0]

        # Ya tengo mi imagen con la piramide en color
        nueva_imagen[altura:(np.shape(laplacianPyramid[i])[0]+altura), np.
    shape(laplacianPyramid[0])[1]:math.ceil(np.shape(laplacianPyramid[i])[1]+np.
    shape(laplacianPyramid[0])[1]), 0:3] = laplacianPyramid[i]

# Compruebo si la imagen es en blanco y negro
else:
    # Hago el mismo procedimiento que para las imagenes en color,
    # simplemente que ahora no tengo que tener en cuenta los canales
    [...]
return nueva_imagen.astype(np.uint8)
```

En este ejemplo he usado $\sigma = 1$ y 2 y borde reflejo y replicado.

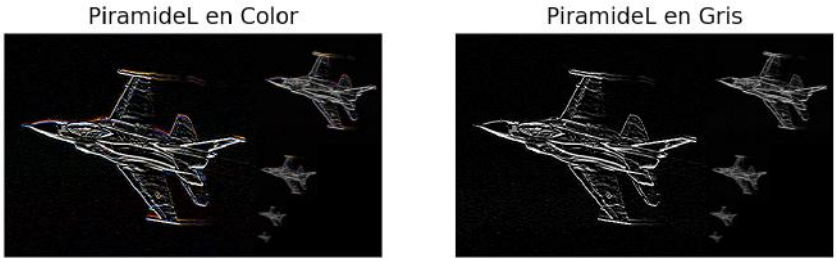

Figura 16: Pirámide Laplaciana con $\sigma = 1$ y BORDER_REFLECT

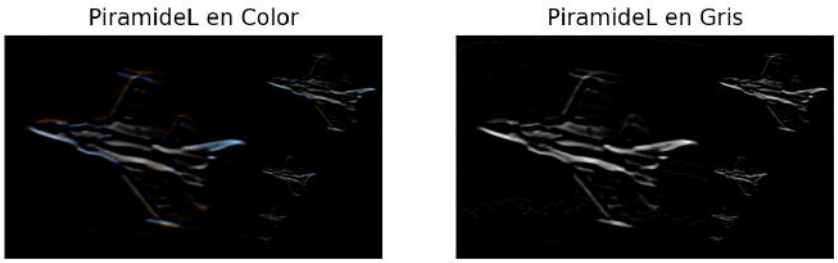

Figura 17: Pirámide Laplaciana con $\sigma = 2$ y BORDER_REPLICATE

A mayor valor de σ, mayor pérdida de fronteras, ya que vamos eliminando las frecuencias altas.

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

En este ejercicio, usaremos imágenes en gris, las de color, están implementadas en el bonus.

1. 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

En este apartado, he implementado dos funciones, una para generar las bajas frecuencias y otra para generar las altas. Empecemos por la función de generar bajas frecuencias. ¿Qué hará esta función? Lo que queremos hacer es alisar la imagen ya que estamos eliminando información (alisando). A nuestra función, le pasamos como entrada el sigma, el cual indica el nivel de alisado de la imagen, de tal manera que cuanto mayor sea su valor más alisada será la imagen

```
def genera_baja_frecuencia(imagen, sigma):

  # Calculo el tamanio de la mascara
  tamanio = 6*sigma+1
  # Filtro la imagen con un filtro Gaussiano para alisarla
  mascara = cv2.getGaussianKernel(ksize = tamanio, sigma = sigma)
  filtrada = convolucion_separable(imagen, cv2.BORDER_REFLECT, mascara, mascara)

  return filtrada
```

La otra función a implementar es la de altas frecuencias, en ella se quiere obtener la diferencia entre la imagen original y sus bajas frecuencias.

```
def genera_alta_frecuencia(imagen, sigma):
  # Nos quedamos con los detalles de la imagen al restarle a esta su alisada
  nueva_imagen = imagen - genera_baja_frecuencia(imagen, sigma)

  return nueva_imagen
```

2. 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)

Como puede que las imágenes contengan números flotantes que pueden ser positivos y negativos para ello debemos normalizar. Los parámetros de nuestra función son *imagenA*, *imagenB*, *sigmaA*, *sigmaB*, siendo la variable con A, las que afectaran a las de alta frecuencia y las de B a las de baja frecuencia. Para que esta función funcione, las dos imágenes para hacer el híbrido tienen que ser del mismo tamaño. Nuestra función, *podrá devolver dos tipo de imágenes, o la híbrida o las tres imágenes puestas en una misma ventana*
.

```
def genera_hibrida_gris(imagenA, imagenB, sigmaA, sigmaB):

  # Genero las imagenes de alta y baja frecuencia
  img_alta = genera_alta_frecuencia(imagenA, sigmaA);
  img_baja = genera_baja_frecuencia(imagenB, sigmaB);

  # Las sumo
  hibrida = img_baja + img_alta

  # Normalizo la hibrida por si hay valores negativos o superiores a 255
  hibrida = cv2.normalize(hibrida,hibrida,0,255, cv2.NORM_MINMAX)

  imagen_compuesta = np.empty((np.shape(img_alta)[0], np.shape(img_alta)[1]*3))
  imagen_compuesta[0:np.shape(img_alta)[0],0:np.shape(img_alta)[1]] = img_alta
  imagen_compuesta[0:np.shape(img_baja)[0], np.shape(img_alta)[1]:np.shape(
    img_alta)[1]+np.shape(img_baja)[1]] = img_baja
  imagen_compuesta[0:np.shape(hibrida)[0], np.shape(img_alta)[1]+np.shape(
    img_baja)[1]:np.shape(img_alta)[1]+np.shape(img_baja)[1]+np.shape(hibrida)
    [1],] = hibrida

  return (hibrida.astype(np.uint8),imagen_compuesta.astype(np.uint8))
```

Para devolver las tres imágenes en una misma ventana, defino las filas como el alto de una imagen, puesto que serán del mismo tamaño todos y defino las columnas como la suma de las columnas de las tres imágenes, ya que la imagen ventana, tendrá de ancho la suma de las tres imágenes juntas, una pegada a la otra.

Primero calculamos las frecuencias altas sobre la imagen, por lo que aplicamos la función *genera_alta_frecuencia* para hacer la convolución, usaremos los bordes reflejados. Luego calculamos las frecuencias bajas sobre la otra imagen.

Debemos saber que el valor de σ irá variando en función de las imágenes, por tanto debemos hacer a prueba y error cuál se adapta mejor.

Figura 18: Híbrida Pájaro-Avión

- Imagen para alta frecuencia y valor de σ: Pájaro y $\sigma = 10$
- Imagen para baja frecuencia y valor del σ: Avión y $\sigma = 1$

Figura 19: Híbrida Submarino-Pez

- Imagen para alta frecuencia y valor de σ: Submarino y $\sigma = 10$
- Imagen para baja frecuencia y valor del σ: Pez y $\sigma = 2$

Figura 20: Híbrida Gato-Peroo

- Imagen para alta frecuencia y valor de σ: Perro y $\sigma = 10$
- Imagen para baja frecuencia y valor del σ: Gato y $\sigma = 2$

3. Realizar la composición con al menos 3 de las parejas de imágenes

Las parejas de imágenes posibles son: Fish - Submarine, Motorcycle - Bicycle, Dog-Cat.

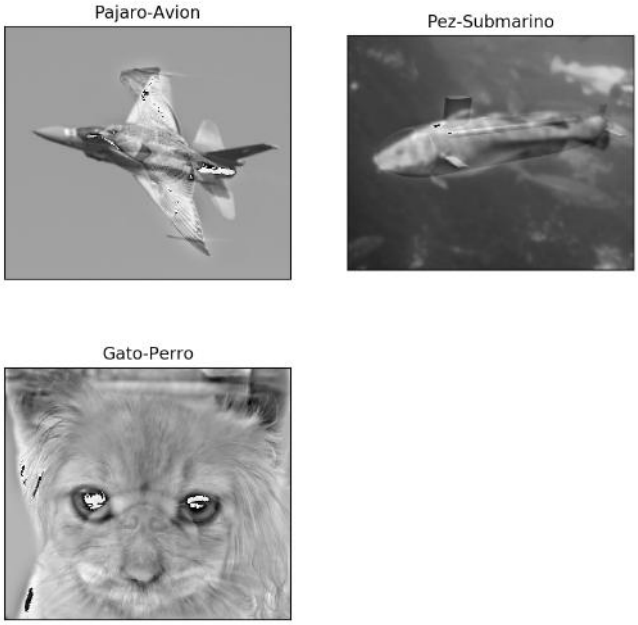

Figura 21: Híbrida para 3 parejas

Podemos ver que se obtiene una imagen híbrida bastante buena, pero como sabemos si nuestra híbrida está bien. Pues para ello, podemos crear un pirámide gaussiana con una híbrida y ver como se comporta desde que está la imagen grande a la imagen pequeñita. Si estamos muy cerca de la image debemos visualizar mejor la de altas frecuencias, pero si estamos a larga distancia debemos visualizar mejor las de baja frecuencia. Como pasa en la siguiente figura.

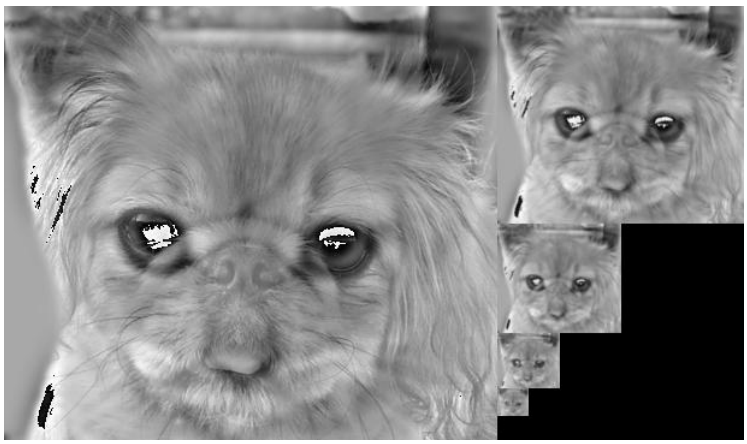

Figura 22: Pirámide Gaussiana con una imagen híbrida

Bonus

1.- Cálculo del vector máscara Gaussiano: Sea una función $f(x) = exp(-0{,}5\frac{x^2}{\sigma^2})$donde ($\sigma$) representa un parámetro en unidades píxel. Implementar una función que tomando sigma como parámetro de entrada devuelva una máscara de convolución 1D representativa de dicha función. Justificar los pasos dados (Ayuda: comenzar calculando la longitud de la máscara a partir de sigma y recordar que el valor de la suma de los valores del núcleo es 1)

La función *calcular_mascara_gaussiana* tiene el mismo objetivo que la función *getGaussianKernel*. Mi función calcula el tamaño de la máscara en función del σ, necesario para saber el intervalo donde se muestrea $[-3\sigma, 3\sigma]$. Una vez calculado el tamaño del vector, asignamos los pesos a cada componente del array. Es decir, que si yo tengo un vector de tamaño 7, mi array será de la siguiente manera $[-3, -2, -1, 0, 1, 2, 3]$, ya que debemos asignar al centro del vector el 0 y a partir de ahí, ir contando hacia arriba o hacia abajo dependiendo de la longitud del vector. Siempre que multipliquemos por σ, tenemos que obtener números enteros, así que si obtenemos reales, debemos redondear hacia arriba. Una vez hecho eso, pasamos a calcular el valor de cada componente del array, haciendo uso de la función proporcionada en el enunciado. Por último debemos normalizar, para que la suma total de nuestra máscara de 1.

```
# Funcion gaussiana para la mascara
# Implementamos la funcion que viene definida en el apartado
def funcion(x, sigma):
  k = np.exp(-0.5 * ((x**2)/(sigma**2)))
  return k

# Funcion que realiza el mismo proceso que getGaussianKernel
def calcular_mascara_gaussiana(sigma):

  # Debemos tomar 3*sigma por cada lado
  # siendo el tamanio total de la mascara: 6*sigma + 1
```

```
    tam = 3*sigma
    kernel = np.arange(-math.ceil(tam), math.ceil(tam+1))

    # Aplicamos la gaussiana a la mascara
    kernel = np.array([funcion(i, sigma) for i in kernel])

    # Y la devolvemos normalizada (algo normalizado su su suma da 1)
    normalizada = kernel/(sum(kernel))
    # print(np.sum(normalizada))

    return normalizada
```

Probemos un ejemplo donde σ valga 1:

```
print(calcular_mascara_gaussiana(1))
[ 0.00443305  0.05400558  0.24203623  0.39905028  0.24203623  0.05400558
  0.00443305]

print(cv2.getGaussianKernel(7,1))
[[ 0.00443305]
 [ 0.05400558]
 [ 0.24203623]
 [ 0.39905028]
 [ 0.24203623]
 [ 0.05400558]
 [ 0.00443305]]
```

Como podemos comprobar, ambos valores coinciden en ambos valores.

2.- Implementar una función que calcule la convolución de un vector señal 1D con un vector-máscara de longitud inferior al de la señal usando condiciones de contorno reflejada. La salida será un vector de igual longitud que el vector señal de entrada. (Ayuda: En el caso de que el vector de entrada sea de color, habrán de extraerse cada uno de los tres vectores correspondientes a cada una de las bandas, calcular la convolución sobre cada uno de ellos y volver montar el vector de salida. Usar las funciones split() y merge())

En este ejercicio lo que hacemos es coger un vector 1D y aplicarle una máscara, haciendo el borde reflejado *BORDER_REFLECT: fedcba — abcdefgh — hgfedcb*. Para realizar el borde reflejado hemos usado la forma *same* que viene en las transparencias de teoría. Por lo tanto tenemos que distinguir 3 posibilidades:

1. Posibilidad 1: que la máscara sobresalga a la izquierda, para ello habrá que ver cuál es el último píxel de la imagen para así reflejar los píxeles.

2. Posibilidad 2: que la máscara esté en el centro, por lo que no habrá qué hacer nada, ya que ningún valor de la mascara sobresale)

3. Posibilidad 3: que la máscara sobresalga a la derecha, para ello habrá que ver cuál es el último píxel de la imagen para así reflejar los píxeles.

A nuestra función le pasaremos una mascara, un vector imagen y el tipo de borde que queramos aplicar. Como en este caso solo he implementado el de contorno reflejada, no aplicará otro que no sea ese. Además tendremos que diferenciar cuando la imagen está en formato color y cuando en escala de grises.

```
def convolucion_1D(mascara_gaussiana, vector_imagen, borde):

  nueva_imagen = np.empty(np.shape(vector_imagen))

  # Obtengo el numero de canales
  canales = len(np.shape(vector_imagen))

  # Cantidad de elementos que hay a la izquierda o a la derecha del elemento
  # central de la mascara
  len_mascara_gaussiana = len(mascara_gaussiana)
  cantidad = (len_mascara_gaussiana - 1) // 2;

  len_vector_imagen = len(vector_imagen) # Longitud del vector

  # Por cada canal paso la mascara si es color es 3
  # y si es blanco y negro es 1
  if canales == 1: # Si la imagen es en gris

     # Borde reflejo
     if (borde == 0):
          # Ahora tenemos que distinguir la posibilidad de que de
          # si la mascara se pone a la derecha, al centro o a la izquierda

          # Posibilidad a la izquierda --> indice va de 0 a cantidad
          # Muevo la mascara sobre el array de la imagen
          # (la parte izquierda de la mascara sobresale)
          for j in range(0, cantidad): # tengo una parte de la mascara fuera
               valor = 0

               # Tengo mi mascara e itero sobre los elementos de la mascara
               # para multiplicarlos con los del array
               for k in range(0, len_mascara_gaussiana):

                    # i es la dimension, es el canal -> abs((0-1+0))
                    valor = valor + vector_imagen[abs((j-cantidad+k)+1)] *
    mascara_gaussiana[k]

               # hemos cogido la mascara a solo un valor
               nueva_imagen[j] = valor

          # La mascara esta en el centro (ningun valor de la mascara sobresale)
          for j in range(cantidad, len_vector_imagen): # tengo la mascara dentro
               valor = 0

               # Tengo mi mascara e itero sobre los elementos de la mascara
               # para multiplicarlos con los del array
               for k in range(0, len_mascara_gaussiana):

                    if (j-cantidad + k < len_vector_imagen):
                         # i es la dimension, es el canal abs((0-1+0))
                         valor = valor + vector_imagen[j-cantidad+k] *
    mascara_gaussiana[k]
                    else:
                         # La mascara esta a la derecha
                         # (la parte derecha de la mascara sobresale)
                         valor = valor + vector_imagen[(len_vector_imagen-((j-
    cantidad+k)-len_vector_imagen)-1)] * mascara_gaussiana[k]

               # hemos cogido la mascara a solo un valor
               nueva_imagen[j] = valor

  # Si la imagen es en color
  else:

     for i in range(canales+1): # Recorro los canales

          if (borde == 0):  # Borde reflejo
               # Ahora distinguimos las tres posibilidades
```

```
                # Posibilidad a la izquierda --> indice va de 0 a cantidad
                # Muevo la mascara sobre el array de la imagen
                # (la parte izquierda de la mascara sobresale)
                for j in range(0, cantidad):
                    valor = 0

                    # Tengo mi mascara e itero sobre los elementos de la
                    # mascara para multiplicarlos con los del array
                    for k in range(0, len_mascara_gaussiana):

                        # i es la dimension, es el canal --> abs((0-1+0))
                        valor = valor + vector_imagen[abs((j-cantidad+k)+1),i]
    * mascara_gaussiana[k]

                        # Hemos cogido la mascara a solo un valor
                        nueva_imagen[j,i] = valor

                # La mascara esta dentro
                # (ningun valor de la mascara sobresale)
                for j in range(cantidad, len_vector_imagen):
                    valor = 0

                    # Tengo mi mascara e itero sobre los elementos de la
                    # mascara para multiplicarlos con los del array
                    for k in range(0, len_mascara_gaussiana):

                        if (j-cantidad + k < len_vector_imagen):
                            # i es la dimension es el canal abs((0-1+0))
                            valor = valor + vector_imagen[j-cantidad+k,i]
     * mascara_gaussiana[k]
                        else:
                            # La mascara esta a la derecha
                            # (la parte derecha de la mascara sobresale)
                            valor = valor + vector_imagen[(
    len_vector_imagen-((j-cantidad+k)-len_vector_imagen)-1),i] *
    mascara_gaussiana[k]

                    # Hemos cogido la mascara a solo un valor
                    nueva_imagen[j,i] = valor

    return nueva_imagen.astype(np.uint8)
```

Podemos hacer un pequeño ejemplo, para comprobar si nuestra función funciona:

```
mascara = calcular_mascara_gaussiana(1)
img1 = leer_imagen('imagenes/cat.bmp', flag_color=True)
img2 = convolucion_1D(mascara,img1[0,:],0)
print("imagen salida:", img2)
print("imagen entrada:", img1)
print("mascara:", img2)
```

Si se comprueba luego esta función en el apartado siguiente, se ve que la salida será un vector de igual longitud que el vector de entrada.

3.- Implementar una función que tomando como entrada una imagen y el valor de sigma calcule la convolución de dicha imagen con una máscara Gaussiana 2D. Usar las funciones implementadas en los dos bonus anteriores.

En este ejercicio, haremos uso de la anterior función, para realizar la convolución de un vector señal 1D con un vector máscara. Además, tendremos dos máscaras, donde una de ellas se le pasará a las filas y la otra a las columnas. En este ejercicio estamos plasmando la misma idea que hemos implemetado en el *Ejercicio* $1C$ o en la Figura 2. En este caso, solo tenemos un tipo de borde (reflejado). Haremos una distinción entre si la imagen leída tiene uno o tres canales.

```
# Hay que convolucionar la imagen con esa convolucion1D
def convolucionar_imagen(mascara_x, mascara_y, imagen2D, borde):

  # Estoy copiando la estructura de la imagen (copiando la matriz sin los
    valores)
  nueva_imagen = np.empty(np.shape(imagen2D))

  # Si imagen en escala de grises B/N
  if len(np.shape(imagen2D)) == 2:

     # Por filas convolucionando
     # Recorro la imagen por filas para aplicarle el filtro Gaussiano 1D
     for i in range(np.shape(imagen2D)[0]):
     # Hago convolucion
          fila = convolucion_1D(mascara_x, imagen2D[i,:], borde)
          nueva_imagen[i,:] = fila

     # Transponemos la imagen
     # al transponer lo que cambiamos es filas por columnas
     nueva_imagen = nueva_imagen.transpose()

     # Ahora hacemos convolucion a las columnas
     # pero como hemos transpuesto seran las filas
     for i in range (np.shape(nueva_imagen)[0]):
          # Hago convolucion
          fila = convolucion_1D(mascara_y, nueva_imagen[i,:], borde)
          nueva_imagen[i,:] = fila

     # Vuelvo a la imagen original
     nueva_imagen = nueva_imagen.transpose()

  else:

     # Por filas convolucionando
     # Recorro la imagen por filas para aplicarle el filtro Gaussiano 1D
     for i in range(np.shape(imagen2D)[0]):
          # Hago convolucion
          fila = convolucion_1D(mascara_x, imagen2D[i,:,:], borde)
          nueva_imagen[i,:,:] = fila

     # Transponemos la imagen
     # al transponer lo que cambiamos es filas por columnas
     nueva_imagen = nueva_imagen.transpose(1,0,2)

     # Ahora hacemos convolucion a las columnas
     # pero como hemos transpuesto seran las filas
     for i in range (np.shape(nueva_imagen)[0]):
          # Hago convolucion
          fila = convolucion_1D(mascara_y, nueva_imagen[i,:,:], borde)
          nueva_imagen[i,:,:] = fila

       # Vuelvo a la imagen original
       nueva_imagen = nueva_imagen.transpose(1,0,2)
```

```
54    return nueva_imagen.astype(np.uint8)
```

Hagamos un pequeño ejemplo con $\sigma = 3$, para ver si en la salida se obtiene un resultado coherente:

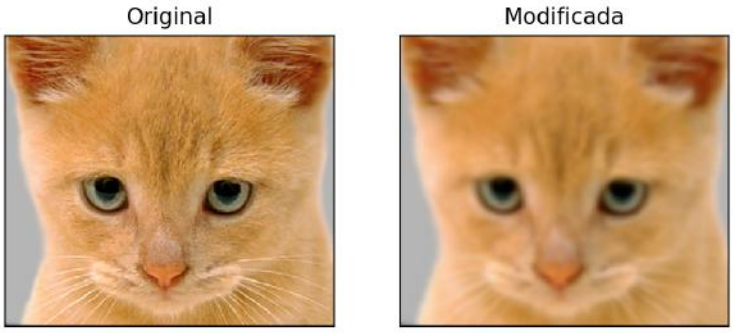

Figura 23: Convolución con una máscara Gaussiana 2D

Podemos ver que aplica un alisado a la imagen, pero para estar seguros de nuestro resultado vamos a comparar las matrices (la mía y la que obtengo con OpenCV).

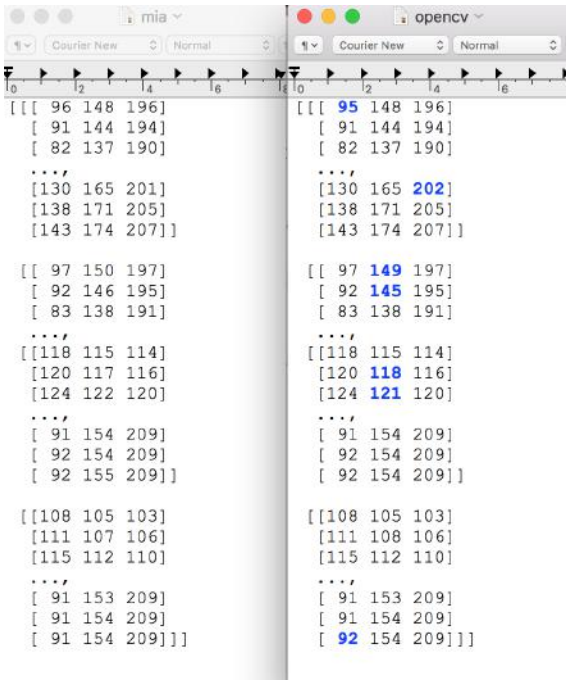

Figura 24: Pirámide Gaussiana con una imagen híbrida

Debido al tamaño de las matrices, he puesto solo un fragmento de las mismas. Pero si se ven enteras, se puede comprobar que existen pocos valores diferentes, y los que difieren tienen un valor arriba o un valor abajo.

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.

Vamos hacer el mismo apartado que el ejercicio 1G, simplemente que ahora tenemos que implementar nosotros *pyrDown*(). Primero aplicaremos un filtro gaussiano a la imagen y luego para redimensionar nos quedaremos con las filas o columnas que sean pares, es decir, cogemos una y descartamos la siguiente y así sucesivamente.

```
def my_pyrDown(imagen):

  # Aplicamos un alisado (convolucionar_imagen)
  # Y nos quedamos con las  columnas y filas pares para reescalar

  # Hacemos convolucion
  sigma = 1
  m_x = calcular_mascara_gaussiana(sigma)
  m_y = calcular_mascara_gaussiana(sigma)
  alisado = convolucionar_imagen(m_x, m_y, imagen, 0)

  # Nos creamos una nueva imagen para guardar
  nueva_imagen = alisado[range(0,alisado.shape[0],2)]
  # y nos quedamos con las con las filas y columnas pares
  nueva_imagen = nueva_imagen[:,range(0,alisado.shape[1],2)]

  return nueva_imagen.astype(np.uint8)
```

Este alisado lo realizaremos con la función *convolucionar_imagen* y con $\sigma = 1$. Nos quedamos solo con lo valores pares de las filas y columnas de la matriz de entrada. Para la función de la *piramide_gaussiana* es la misma, solo que ahora hacemos uso de nuestra función implementada $piramide_gaussiana_bonus()$. No escribiré la función nueva, puesto que es la misma del ejercicio 1G, solo que ahora hace uso de todas las implementaciones llevadas a cabo en los bonus, sin usar OpenCV.

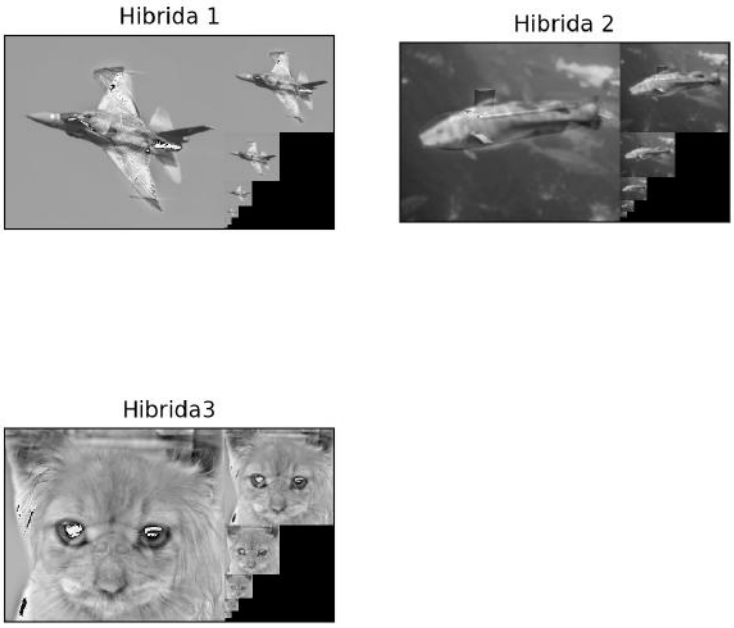

Figura 25: Convolución con una máscara Gaussiana 2D

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

Para realizar este apartado nos hemos basado en el ejercicio 2 de la práctica, básicamente ahora debemos hacer el tratamiento con imágenes en color, es decir, con 3 canales. En este ejercicio, hacemos uso de todas las funciones que hemos implementado a lo largo de los bonus, no usamos opencv.

```
# Genera una imagen con baja frecuencia
def genera_baja_frecuencia_bonus(imagen, sigma):

  # Filtro la imagen con un filtro Gaussiano para alisarla
  mascara = calcular_mascara_gaussiana(sigma)
  # filtrada = cv2.GaussianBlur(imagen, (tamanio,tamanio), sigma)
  filtrada = convolucionar_imagen(mascara, mascara, imagen, 0)

  return filtrada

# Funcion que genera una imagen en alta frecuencia
def genera_alta_frecuencia_bonus(imagen, sigma):
  # Nos quedamos con los detalles de la imagen al restarle a esta su alisada
  nueva_imagen = imagen - genera_baja_frecuencia_bonus(imagen, sigma)

  return nueva_imagen

# Funcion que genera una imagen hibrida
def genera_hibrida_bonus(imagenA, imagenB, sigmaA, sigmaB):

  # Generar alta y baja frecuencia
  img_alta = genera_alta_frecuencia_bonus(imagenA, sigmaA);
  img_baja = genera_baja_frecuencia_bonus(imagenB, sigmaB);
  hibrida = img_baja + img_alta

  # Vamos a crear una nueva imagen componiendo 3 imagenes (alta, baja e hibrida)
  # Se supone que ambas imagenes (imagenA e imagenB) miden lo mismo
```

```
    if len(np.shape(imagenA)) == 3:

        imagen_compuesta = np.empty((np.shape(img_alta)[0],np.shape(img_alta)[1]*3,
    np.shape(img_alta)[2]))

        imagen_compuesta[0:np.shape(img_alta)[0],0:np.shape(img_alta)[1],0:3] =
    img_alta

        imagen_compuesta[0:np.shape(img_baja)[0],
        np.shape(img_alta)[1]:np.shape(img_alta)[1]+np.shape(img_baja)[1], 0:3] =
    img_baja

        imagen_compuesta[0:np.shape(hibrida)[0], np.shape(img_alta)[1]+np.shape(
    img_baja)[1]:np.shape(img_alta)[1]+np.shape(img_baja)[1]+np.shape(hibrida)
    [1], 0:3] = hibrida
    else:
        imagen_compuesta = np.empty((np.shape(img_alta)[0], np.shape(img_alta)
    [1]*3))

        imagen_compuesta[0:np.shape(img_alta)[0], 0:np.shape(img_alta)[1]] =
    img_alta

        imagen_compuesta[0:np.shape(img_baja)[0], np.shape(img_alta)[1]:np.shape(
    img_alta)[1]+np.shape(img_baja)[1]] = img_baja

        imagen_compuesta[0:np.shape(hibrida)[0], np.shape(img_alta)[1]+np.shape(
    img_baja)[1]:np.shape(img_alta)[1]+np.shape(img_baja)[1]+np.shape(hibrida)
    [1]] = hibrida

    return hibrida.astype(np.uint8)
```

Figura 26: Híbridas en su formato a color

Podemos ver como nos muestra las imágenes híbridas en su color.