Práctica 1 Visión por Computador:

Filtrado y Muestreo

Antonio Manuel Milán Jiménez
20 de octubre de 2017

Ejercicio 1

Apartado A: Imprimir imágenes

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
from math import sqrt,pi,exp,pow
from PIL import Image
from scipy import signal
from copy import deepcopy

def imprimir(imagenes,fila,col,titulos,grises=False):

    n_img=1
    for img in imagenes:
        plt.subplot(fila,col,n_img)
        plt.subplots_adjust(hspace=0.8)
        if(grises==True):
            plt.imshow(img,cmap='gray')
        else:
            plt.imshow(cv2.cvtColor(img,cv2.COLOR_BGR2RGB))
        plt.title(titulos[n_img-1])
        n_img=n_img+1
    plt.show()
```

Con esta función podremos imprimir una lista de imágenes eligiendo cuántas deben aparecer por fila y columna. Para ello proporcionaremos la lista de imágenes ("imagenes") y las filas y columnas ("fila","col"). Entonces, mediante las funciones "subplot()" y "imshow()", mostraremos dichas imagenes en el formato de filas y columnas que hayamos indicado.

Contamos con una lista de "titulos" para que cada una de las imágenes vaya acompañada con el título que le corresponda. Para mostrarlo, utilizamos la función "plt.title()".

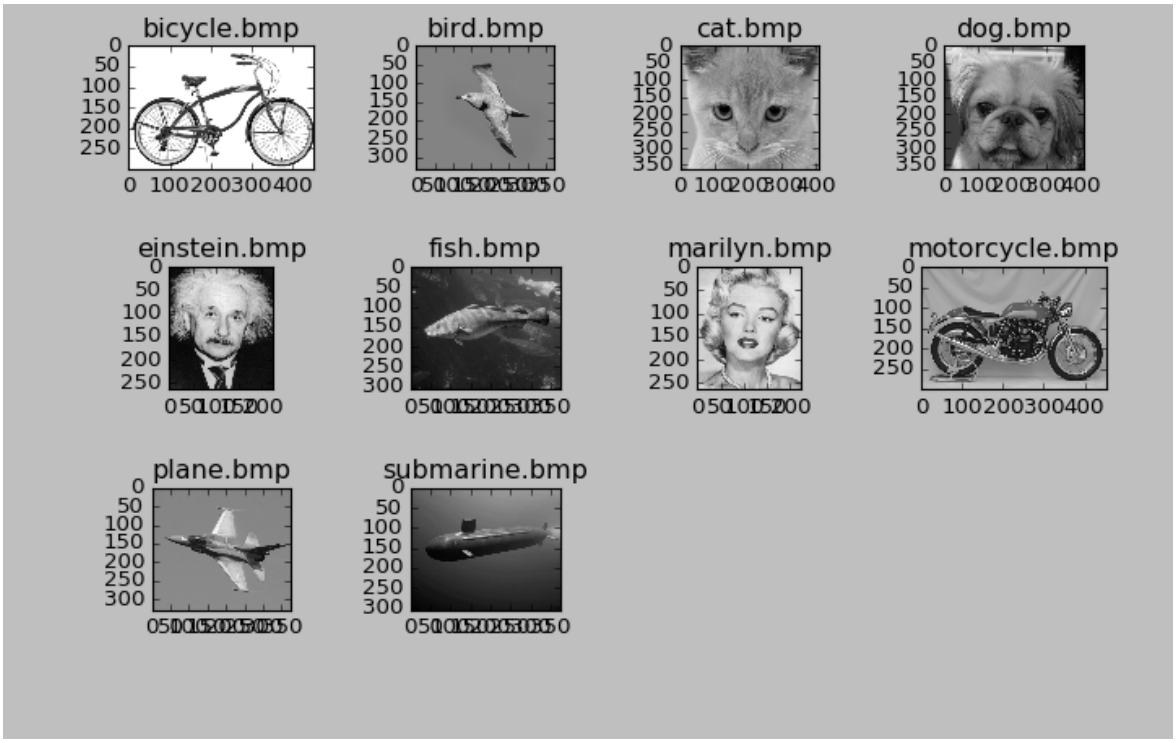
Adicionalmente, la imagen que se muestra puede ser a color o en escala de grises. Esto nos puede interesar en función del formato en el que hayamos leído la imagen. Para conseguir esto, le indicamos mediante un booleano "grises" el color en el que imprimirla, lo cuál hará que utilicemos "imshow()" con un mapa de colores o en tono de grises. Utilizamos la función "cv2.cvtColor()" en "imshow()" para que se impriman los colores correctos de la imagen.

```
mis_imagenes =
["bicycle.bmp","bird.bmp","cat.bmp","dog.bmp","einstein.bmp","fish.bmp","m
arilyn.bmp","motorcycle.bmp","plane.bmp","submarine.bmp"]
lista_imagenes = []

bordes=[cv2.BORDER_WRAP,cv2.BORDER_WRAP,cv2.BORDER_CONSTANT,cv2
.BORDER_CONSTANT]
nombre_bordes=['WRAP','WRAP','Constante','Constante']

for image in mis_imagenes:
    lista_imagenes.append(cv2.imread(image,0))
```

Así es como mostramos las imagenes. En un bucle, leemos mediante "cv2.imread()" cada uno de las imágenes proporcionadas por una lista con sus nombres. Además, creamos ya una lista con tipos de bordes que utilizaremos en futuras muestras de imágenes. El "0" en la función "imread()" indica que se lean las imágenes es escala de grises.

Ya sólo queda llamar a nuestra función "imprimir()". El "True" del final indica que se imprima en escala de grises. Es algo que se hará en todos los demás ejercicios.

```
imprimir(lista_imagenes,4,4,mis_imagenes,True)
```



Apartado B: Convolución con máscara Gaussiana

En este apartado crearemos una función que haga una convolución con una máscara Gaussiana para conseguir así un alisamiento en la imagen. Esta es la función que se encargará de ello:

```
def Gauss_alisamiento(img,σX,σY,borde):

    if(σX<1):
        tamX=7
    else:
        tamX=int(σX*6+1)

    if(σY<1):
        tamY=7
    else:
        tamY=int(σY*6+1)

    img_mod=np.zeros((img.shape[0],img.shape[1],3),dtype=np.float32)

    img_mod=cv2.GaussianBlur(np.array(img, dtype=np.float32),
(tamX,tamY),σX=σX,σY=σY)
    img_mod=cv2.copyMakeBorder(img_mod,10,10,10,10,borde)

    return img_mod
```

Inicialmente, a partir de los σs proporcionados, establecemos el tamaño que va a tener nuestra máscara Gaussiana. Este tamaño se calcula mediante la fórmula: σ*6+1

Escogemos esta expresión ya que el tamaño será aproximadamente de un 95% del área de la máscara de Gauss. Adicionalmente, si el σ proporcionado es inferior a 1, automáticamente se establece el tamaño a 7.

A continuación, aplicamos la convolución mediante la función "cv2.GaussianBlur()", que se encargará directamente de hacer el alisamiento a nuestra imagen con una máscara Gaussiana de un tamaño y σ especificado. A ésta proporcionaremos entonces la imagen, el tamaño de la máscara y los σs.

Por último, llamamos a la función "cv2.copyMakeBorder()" que añade un borde a nuestra imagen. Le especificaremos el tipo de borde y el tamaño de éste a cada uno de los lados de la imagen.

Vemos que tanto la imagen que recoge el resultado de "GaussianBlur()" cómo la imagen en sí que le pasamos están en formato "float32". Esto se hace para evitar que funciones externas automáticamente cambien los valores de nuestra imagen a valores enteros, positivos y entre 0 y 255. En todas las funciones que hemos implementado en la práctica se trabaja de la misma forma, indicándole a las diferentes imágenes con las que trabajamos que manejen valores flotantes. Justo antes de pintarlas será cuando se pasen a enteros entre 0 y 255 mediante la función "Reescalar()" y se cambie el formato a "uint8" para eliminar los decimales de los diferentes valores de la imagen.

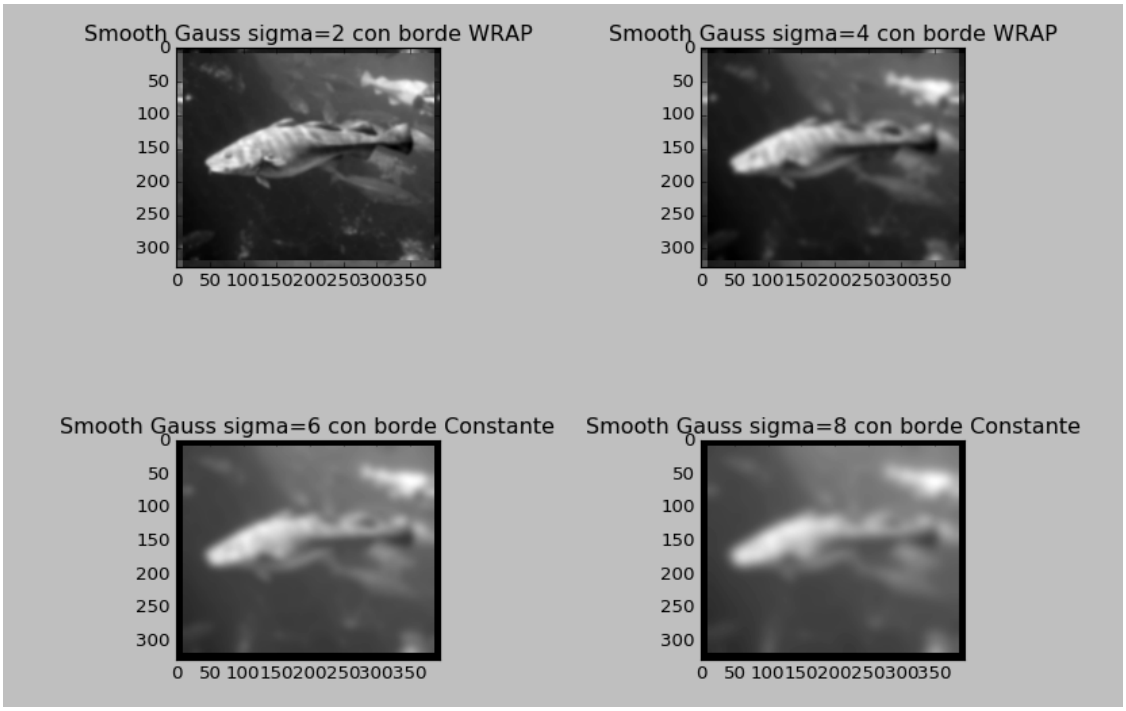
Así es como llamamos a nuestra función:

```
lista_imagenesB = []
   titulos=[]

   for i in range(1,5):
      mi_img=Gauss_alisamiento(lista_imagenes[5],i*2,i*2,bordes[i-1])
      lista_imagenesB.append(np.array(mi_img, dtype=np.uint8))
      titulos.append("Smooth Gauss σ=" + str(i*2) + " con borde " +
nombre_bordes[i-1])
```

En un bucle llamamos 4 veces a nuestra función para mostrar la imagen con diferentes σs de alisamiento y diferentes bordes. Los σs serán 2,4,6 y 8; iguales en X e Y. Aprovechamos el mismo bucle para crear los diferentes títulos que acompañarán a las imágenes. Vemos que una vez que obtenemos la imagen resultado la transformamos a "uint8" para poder imprimirla como hemos comentado anteriormente.

Finalmente mostramos el resultado:

```
imprimir(lista_imagenesB,2,2,titulos,True)
```



Vemos claramente como, al aumentar σ, el alisamiento es mayor y perdemos una buena cantidad de detalles. Mientras que con σ=2 podemos ver la mayor parte de los detalles, con σ=8 lo único que podemos diferenciar es el pez del resto de la imagen.

Esto sucede debido a que al aumentar σ, también en consecuencia el tamaño de la máscara, le estamos dando una mayor ponderación a los píxeles de alrededor del píxel en cuestión, además de tener en cuenta un mayor número de "vecinos" al haber aumentado la máscara de tamaño. Pasamos de centrarnos en un entorno muy local del píxel, a tener en cuenta muchos más píxeles, que al hacer la media significará que los píxeles se asemejen más entre ellos y perdamos ese nivel de detalle marcado por la diferencia entre los píxeles.

Apartado c: Convolución con núcleo separable

Si ahora aprovechamos que la máscara 2D podemos separarla en dos kernels de 1D, conseguimos mejorar el orden de complejidad de $O(n^2m^2)$ a $O(n^2m)$.

Esta es nuestra función que se encargará de ello:

```
def Gauss_kernel_RGB(img,σ,ind_kernel,kernel,borde):

    img_mod = np.array(img, dtype=np.float32)

    if(ind_kernel==0):
        if(σ<1):
            tam=7
        else:
            tam=int(σ*6+1)

        kernel = cv2.getGaussianKernel(tam,σ)


    for i in range(len(img)):
        img_mod[i]=cv2.filter2D(img[i],cv2.CV_32F,np.flipud(kernel))


    for i in range(len(img[0])):
        img_mod[:,i]=cv2.filter2D(img[:,i],cv2.CV_32F,np.flipud(kernel))

    if(borde!=0):
        img_mod=cv2.copyMakeBorder(img_mod,10,10,10,10,borde)

    return img_mod
```

Inicialmente comprueba si ya le estamos proporcionando un kernel a la función. De no ser así, utiliza la función "cv2.getGaussianKernel()" con la que, a partir del σ y tamaño del kernel que le proporcionamos, crea el núcleo separable que hemos comentado anteriormente. El tamaño del kernel se calcula de igual forma que en el anterior apartado.

Una vez que tenemos nuestro kernel, tenemos que aplicarlo separadamente por las filas y columnas de la imagen. De esto se encarga la función "cv2.filter2D()", que realiza la convolución con un kernel y un array. El parámetro CV_32F indica que trabaje con números flotantes.

Entonces, iremos aplicando "filter2D()" a cada una de las filas y columnas de nuestra imagen mediante dos bucles "for" separados que las irán recorriendo y actualizando. En la documentación de openCV se indica que en "filter2D()", debemos "voltear" el kernel con flip() para que realmente se haga una convolución. Aunque en este caso no sería necesario ya que el kernel es simétrico, en otros casos, como el kernel de las derivadas, sí será necesario ya que no será simétrico.

Finalmente añadimos un borde con la función "copyMakeBorder()".

Adicionalmente hemos creado otra función para que trabaje con imágenes de un sólo canal. Se diferencia de esta función en que al resultado de "filter2D()" le realizamos "ravel()" para que devuelva un array y no una matriz de una columna. Esto se hace para que sean compatibles los formatos:

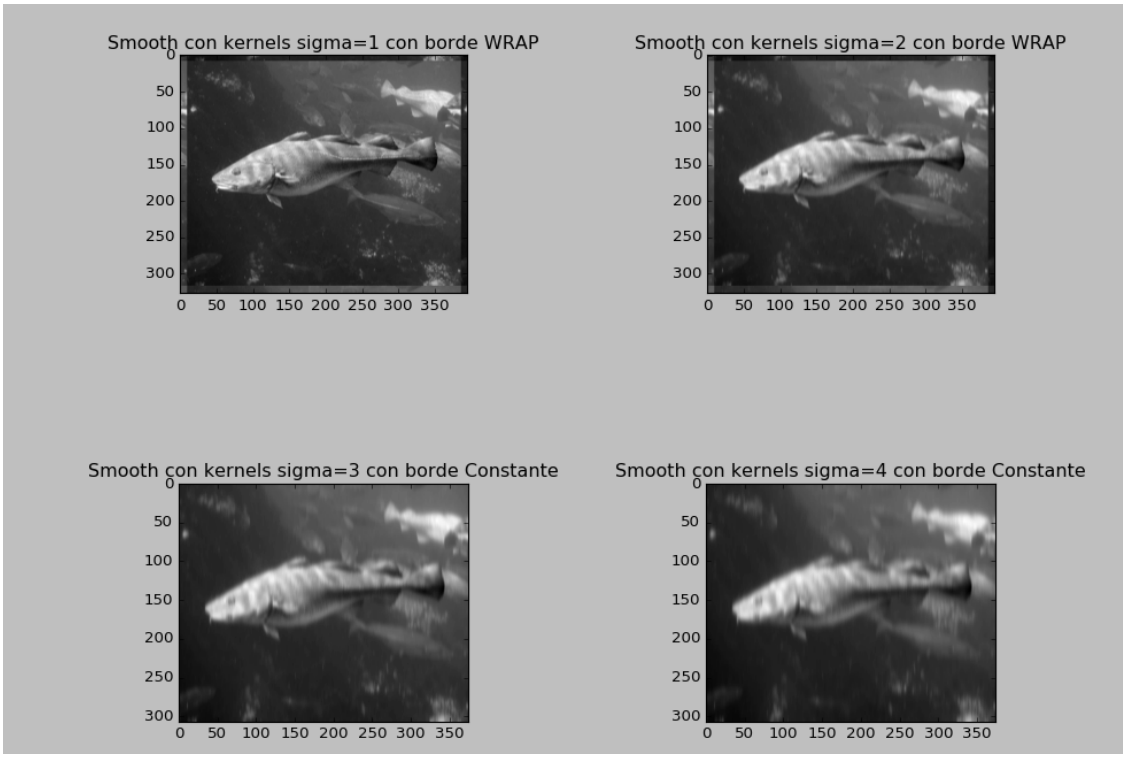
```
def Gauss_kernel_Grey(img,σ,ind_kernel,kernel,borde):

    img_mod = np.array(img, dtype=np.float32)

    if(ind_kernel==0):
        if(σ<1):
            tam=7
        else:
            tam=int(σ*6+1)

        kernel = cv2.getGaussianKernel(tam,σ)

    for i in range(len(img)):
        img_mod[i]=cv2.filter2D(img[i],cv2.CV_32F,np.flipud(kernel)).ravel()


    for i in range(len(img[0])):
        img_mod[:,i]=cv2.filter2D(img[:,i],cv2.CV_32F,np.flipud(kernel)).ravel()


    if(borde!=0):
        img_mod=cv2.copyMakeBorder(img_mod,10,10,10,10,borde)


    return img_mod
```

Entonces llamamos a la función con diferentes σs como hicimos en el anterior apartado:

```
lista_imagenesB = []
titulos = []

for i in range(1,5):
    mi_img=Gauss_kernel_Grey(lista_imagenes[5].copy(),i,0,0,bordes[i-1])
    lista_imagenesB.append(np.array(mi_img, dtype=np.uint8))
    titulos.append("Smooth con kernels σ=" + str(i) + " con borde " +
nombre_bordes[i-1])
```

Y éste es el resultado:

```
imprimir(lista_imagenesB,2,2,titulos,True)
```



Vemos que obtenemos el resultado que ya obtuvimos en el anterior apartado, cuando lo realizábamos sin núcleos separables, con la diferencia de que ahora estaríamos obteniendo el mismo resultado con un orden de complejidad menor.

Nuevamente, al aumentar σ provocamos que se vayan perdiendo los detalles de la imagen.

Apartado D: Convolución con núcleo de 1ª derivada

A continuación, vamos a trabajar con núcleos de 1ª derivada. Al tratar con estos núcleos podremos detectar los puntos en los que haya un rápido cambio de intensidad, es decir, los bordes o detalles de la imagen. Estos bordes coinciden con los extremos de la derivada.

Para lograr esto obtendremos los kernels derivados y a continuación realizaremos una convolución respecto a uno de los ejes con uno de estos kernels. En el otro eje aplicaremos un kernel que no está derivado para que la convolución sea sólo respecto a uno de los ejes.

Para realizar ésto hemos creado dos funciones:

```
def Gauss_kernel_deriv(img,ind_deriv,kernel,kernel_no_derivado):

    img_mod = np.array(img, dtype=np.float32)

    if(ind_deriv ==0):
        for i in range(len(img)):
            img_mod[i]=cv2.filter2D(img[i],cv2.CV_32F,np.flipud(kernel)).ravel()
        for i in range(len(img[0])):

img_mod[:,i]=cv2.filter2D(img_mod[:,i],cv2.CV_32F,np.flipud(kernel_no_deriv
ado)).ravel()


    if(ind_deriv ==1):
        for i in range(len(img_mod[0])):

img_mod[:,i]=cv2.filter2D(img[:,i],cv2.CV_32F,np.flipud(kernel)).ravel()
        for i in range(len(img)):

img_mod[i]=cv2.filter2D(img_mod[i],cv2.CV_32F,np.flipud(kernel_no_derivad
o)).ravel()


    if(np.amin(img_mod)  < 0):            #Hay negativos
        img_mod = Reescalar(img_mod,False,True)

    return img_mod




def kernel_derivada(img,ordenX,ordenY,var,borde,σ):

    img_mod = np.array(img, dtype=np.float32)

    if(var==0):
        kernels=cv2.getDerivKernels(ordenX,0,σ*6+1,normalize=True)
    else:
        kernels=cv2.getDerivKernels(0,ordenY,σ*6+1,normalize=True)

    img_mod=Gauss_kernel_deriv(np.array(img,
dtype=np.float32),var,kernels[var],kernels[(var+1)%2])

    if(σ>1):

        n_kernel=cv2.getGaussianKernel(int(sqrt(σ-1)*6+1),int(sqrt(σ-1)))
        img_mod=Gauss_kernel_Grey(img_mod,int(sqrt(σ-1)),1,n_kernel,0)

    if(borde!=0):
        img_mod=cv2.copyMakeBorder(img_mod,10,10,10,10,borde)
```

```
    return img_mod
```

Inicialmente, en la función "kernel_derivada()", llamamos a la función "cv2.getDerivKernels()" para obtener los kernels derivados respecto "x" y respecto "y". Para ello indicamos el orden de "x" e "y" y el tamaño que tendrá el kernel.

A continuación, llamamos a la función "Gauss_kernel_deriv()" que se encargará de hacer la convolución con éste kernel. Esta función sigue una estructura muy similar a la del anterior apartado aunque en esta ocasión, el kernel derivado respecto "x" sólo lo pasamos por filas y el derivado respecto "y" lo pasamos sólo por columnas. La variable "ind_deriv" indicará si queremos entonces respecto "x" o respecto "y". Como hemos comentado anteriormente, se aplicará en el otro eje un kernel no derivado.

Además, dado que "getDerivKernels()" sólo trabaja bien con σ=1, si queremos trabajar con un σ superior, tendremos que utilizar la propiedad de que la convolución de $G(\sigma^2)$ y $G(\sigma b^2)$ es igual a $G(\sigma^2 + \sigma^2)$. Por ello, la imagen que en un primer momento obtenemos del "Gauss_kernel_deriv()" (la de σ=1), la convolucionamos con un kernel Gaussiano de σ igual a sqrt(σO-1), siendo σO el σ que deseamos aplicar.

Una vez conseguido ésto, tenemos que tratar la imagen para que se pueda imprimir bien, es decir, con valores entre 0 y 255. Para ello utilizamos otra nueva función, "Reescalar()":

```
def Reescalar(img,negativos=False):

    if(negativos):
        minimo=np.amin(img)
        img=img+minimo*-1

    maximo=np.amax(img)
    if(maximo>255):
        coef=(float)(255/maximo)
        img=img*coef


    return img
```

Consta de dos pasos básicamente. Primero, encontramos el mínimo de todos los valores, es decir, el negativo más bajo de todos. Entonces, sumamos el valor absoluto de este mínimo a todos los valores de la imagen, consiguiendo así que todos ellos se queden en positivo. Permitimos que esta operación sea opcional por si en futuras ocasiones reescalamos imagenes que no tienen número negativos.

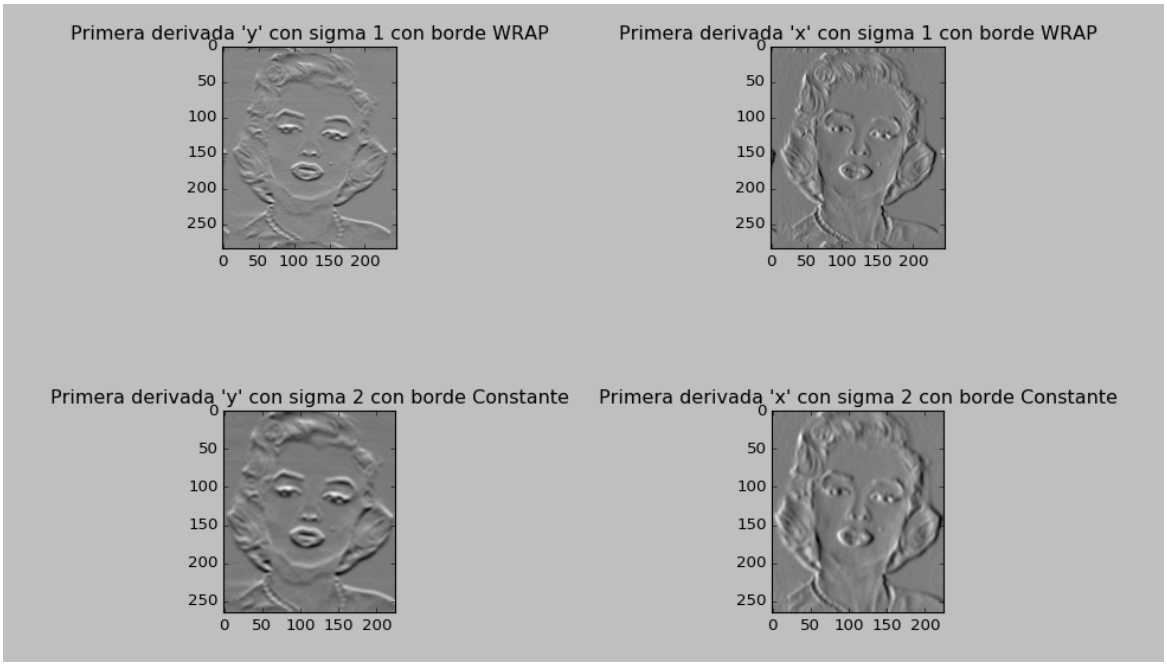
Seguimos necesitando que los valores se queden entre 0 y 255, así que realizaremos un reescalado multiplicando cada uno de los valores de la imagen por un coeficiente calculado cómo 255/maximo_imagen. Esto hará que todos los valores se queden entre 0 y 255.

Ya nos queda llamar a la función en un bucle para obtener las imágenes derivadas respecto "x" y respecto "y", junto con dos tipos de bordes. Indicamos en cada llamada que queremos hacer la primera derivada con un 1 en los argumentos "ordenX" y "ordenY".

```
lista_imagenesB = []
titulos=[]
var=["x","y"]
σs=[1,1,2,2]

for i in range(1,5):
    mi_img=kernel_derivada(lista_imagenes[6],1,1,i%2,bordes[i-1],σs[i-1])
    lista_imagenesB.append(np.array(mi_img, dtype=np.uint8))
    titulos.append("Primera derivada '" + var[i%2] +"' con σ " + str(σs[i-1]) + " con borde " + nombre_bordes[i-1])
```

Y aquí tenemos el resultado:

```
imprimir(lista_imagenesB,2,2,titulos,True)
```



Observamos una importante diferencia entre las derivadas respecto "x" y las de "y", la "dirección" de los bordes. Cuando hacemos la derivada respecto "y" resaltan más los bordes que tienen una dirección horizontal (por ejemplo, las cejas), sin embargo,

en la que es respecto "x" destacan más los bordes con una dirección vertical (por ejemplo, el tabique nasal).

Esto sucede debido a que en la derivada respecto "y", pasabamos el kernel derivado por columnas, detectando los cambios de intensidad entre los píxeles de la misma vertical. En cambio, respecto "x" se detectan los cambios de intensidad entre los píxeles que están en la misma horizontal. Al utilizar un σ mayor obtenemos unos bordes bastantes más gruesos.

Apartado E: Convolución con núcleo de 2ª derivada

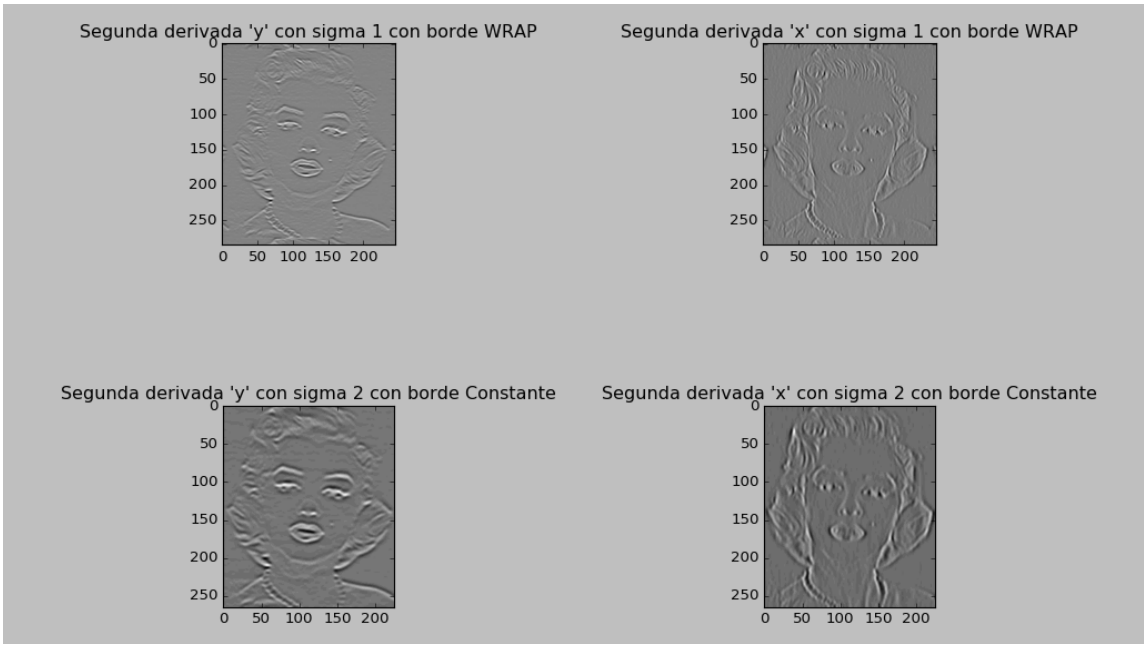
Si ahora trabajamos con la segunda derivada, detectaremos los puntos anteriores y posteriores de donde hay un cambio rápido de intensidad.

Utilizamos las mismas funciones que en el anterior apartado aunque ahora indicaremos un 2 en la función "getDerivKernels()" para trabajar el kernel de segunda derivada respecto "x" y respecto "y":

```
lista_imagenesB = []
titulos=[]

for i in range(1,5):
    mi_img=kernel_derivada(lista_imagenes[6],2,2,i%2,bordes[i-1],σs[i-1])
    lista_imagenesB.append(np.array(mi_img, dtype=np.uint8))
    titulos.append("Segunda derivada '" + var[i%2] +"' con σ " + str(σs[i-1]) +
" con borde " + nombre_bordes[i-1])
```

Y aquí tenemos las imágenes obtenidas:

```
imprimir(lista_imagenesB,2,2,titulos,True)
```



Nuevamente vemos que en la derivada respecto "y" destacan los bordes con una dirección horizontal y en la de "x" los bordes con una dirección vertical. Sin embargo, ahora los bordes o "edges" se encuentran mejor definidos ya que se encuentran entre dos cambios rápidos de intensidad.

Apartado F: Convolución Laplaciana-de-Gaussiana

Para calcular la convolución de Laplaciana de Gaussiana tendremos que calcular las segundas derivadas respecto "x" y respecto "y" por separado. Una vez que las tengamos, realizaremos la convolución con núcleos de 2ª derivada con ellas, sumaremos las imágenes y normalizaremos el resultado. Aquí tenemos la función:

```
def Laplaciana_Gaussiana(img,borde,σ):

    img_X = np.array(img, dtype=np.float32)
    img_Y = np.array(img, dtype=np.float32)

    img_X=kernel_derivada(np.array(img, dtype=np.float32),2,0,0,0,σ)
    img_Y=kernel_derivada(np.array(img, dtype=np.float32),0,2,1,0,σ)


    img_mod=img_X +img_Y
    img_mod=img_mod*pow(σ,2) #Normalizamos

    if(np.amin(img_mod)<0):
        img_mod=Reescalar(img_mod,False,True)
    else:
        img_mod=Reescalar(img_mod)
```

```
    if(borde!=0):
        img_mod=cv2.copyMakeBorder(img_mod,10,10,10,10,borde)


    return img_mod
```

Vemos que para realizar las convoluciones utilizamos la función que hemos presentado en el anterior apartado. Realizamos una normalización multiplicando por σ^2 para evitar que la derivada del filtro decrezca conforme σ crece, en otras palabras, para evitar que con σs mayores todos los valores se acerquen al 0.

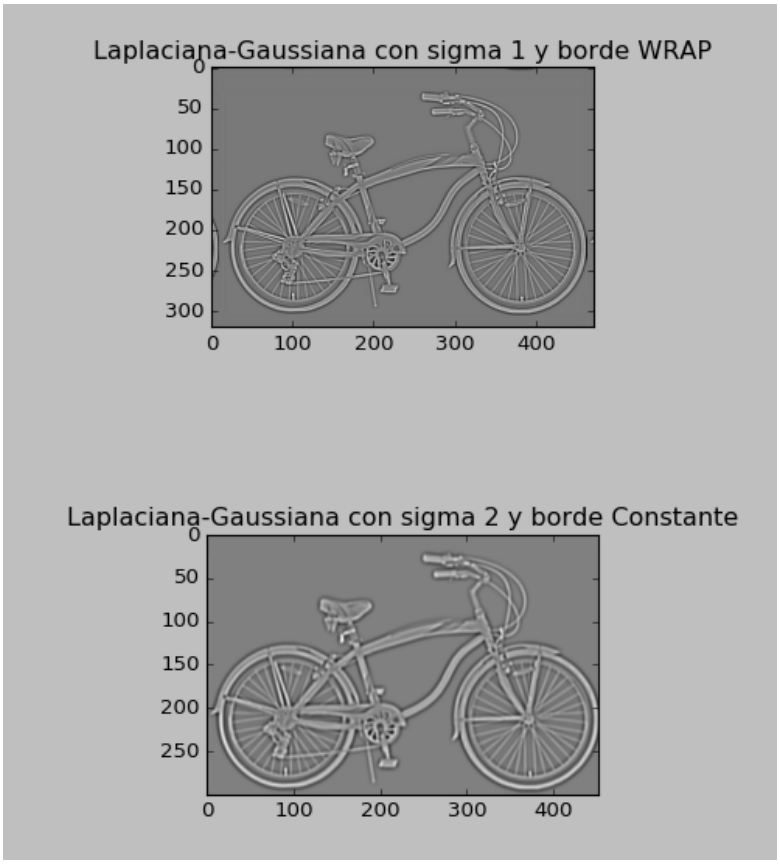
Por último, antes de terminar, reescalamos por si tenemos números negativos y/o mayores que 255 y así poder mostrar bien la imagen.

Así llamamos a la función, con σ 1 y 2:

```
lista_imagenesB = []
titulos=[]

for i in range(1,3):
    mi_img=Laplaciana_Gaussiana(lista_imagenes[0],bordes[i],i)
    lista_imagenesB.append(np.array(mi_img, dtype=np.uint8))
    titulos.append("Laplaciana-Gaussiana con σ " + str(i) + " y borde " +
nombre_bordes[i])
```

Y éstas son las imágenes obtenidas:

```
imprimir(lista_imagenesB,2,1,titulos,True)
```



La única diferencia notable al cambiar σ es que la imagen se hace algo más clara y con unos bordes algo más gruesos.

Apartado G: Pirámide Gaussiana

La idea de la pirámide Gaussiana es que en cada iteración reduzcamos el tamaño de la imagen a la mitad. Es necesario además pasar un filtro Gaussiano pues, de lo contrario, al reducir la imagen perdería mucha calidad. Esta es la función:

```
def piramide_Gaussiana(img,borde,n):


    lista_img_piramides = []
    img_mod = np.array(img, dtype=np.float32)
    img_aux=np.array(img_mod, dtype=np.float32)

    for i in range(1,n+1):
        img_aux=cv2.copyMakeBorder(img_mod,int(16/pow(2,i-
1)),int(16/pow(2,i-1)),int(16/pow(2,i-1)),int(16/pow(2,i-1)),borde)
        lista_img_piramides.append(img_aux)
        img_mod=cv2.pyrDown(img_mod)

    return lista_img_piramides
```

La función "cv2.pyrDown()" se encarga de aplicar el filtro Gaussiano y de reducir a la mitad la imagen. Además, es necesario también que en cada iteración reduzcamos el tamaño del borde.

Para construir la imagen de la piramide, creamos una imagen mucho más grande que la original donde aparecerán todas las imagenes reducidas que obtengamos. Esta función se encarga de crear así la pirámide, proporcionándole la lista de imágenes que hemos ido reduciendo:

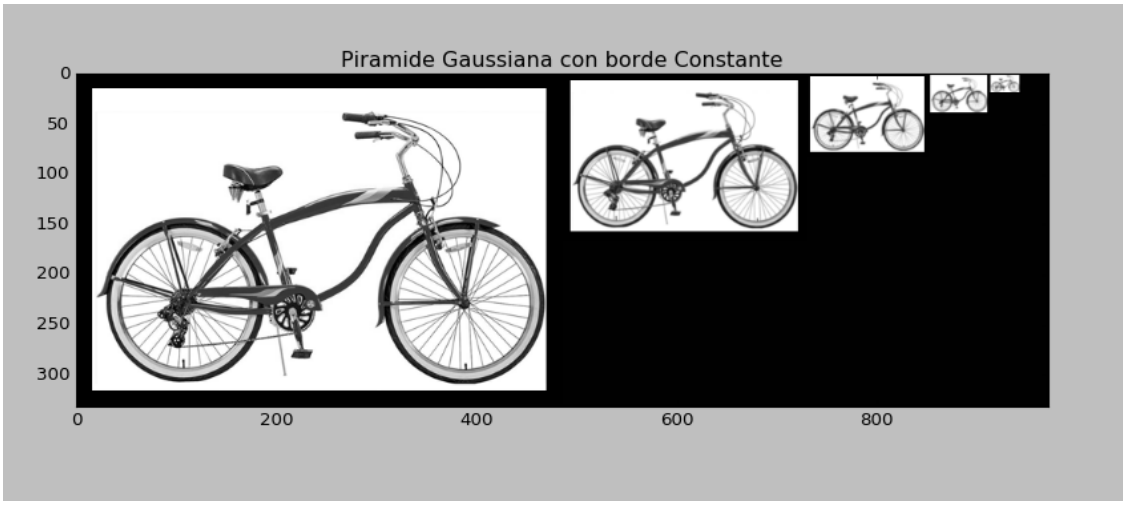
```
def construir_piramide(lista_imagenes,colores=False):

    tamY=(lista_imagenes[0].shape)[0]
    tamX=int((lista_imagenes[0].shape)[1]*2+1)

    if(colores==True):
        piramide=np.ones((tamY,tamX,3),dtype=np.float32)
    else:
        piramide=np.ones((tamY,tamX),dtype=np.float32)


    limiteX=0

    for i in range(len(lista_imagenes)):
        x= (lista_imagenes[i].shape)[1]
        y= (lista_imagenes[i].shape)[0]

        piramide[0:y,limiteX:x+limiteX]=lista_imagenes[i]

        limiteX=x+limiteX

    return piramide
```

Así llamamos a la función, indicándole que queremos 5 niveles en la piramide, la original y otras 4 reducciones:

```
mi_img=cv2.imread("bicycle.bmp",0)
lista_img_piramides= piramide_Gaussiana(mi_img,bordes[2],5)

mi_lista=[]
titulos=[]
mi_lista.append(construir_piramide(lista_img_piramides))
titulos.append("Piramide Gaussiana con borde " + nombre_bordes[2])
```

Y ésta es nuestra pirámide:

```
imprimir(mi_lista,1,1,titulos,True)
```



Vemos que, aunque reduzcamos el tamaño, gracias a que realizamos una convolución, no perdemos calidad conforme más pequeña se hace la imagen.

Apartado H: Pirámide Laplaciana

La pirámide Laplaciana se construye con las imágenes "diferencia" de la imagen original y la imagen original reducida y ampliada seguidamente. Luego la imagen original pasa a ser la imagen original reducida a la mitad. Mostramos la función:

```
def piramide_Laplaciana(img,borde,n):


    lista_img_piramides = []
    img_mod = np.array(img, dtype=np.float32)
    img_r = np.array(img, dtype=np.float32)
    img_a = np.array(img, dtype=np.float32)
    img_lap = np.array(img, dtype=np.float32)
    img_aux=np.array(img_mod, dtype=np.float32)

    for i in range(1,n+1):

        if(img_mod.shape[0]%2==1):
            img_mod=img_mod[:-1,:]

        if(img_mod.shape[1]%2==1):
            img_mod=img_mod[:,:-1]

        img_r=cv2.pyrDown(img_mod)
        img_a=cv2.pyrUp(img_r)
        img_lap=img_mod-img_a
        img_lap=Reescalar(img_lap,False,True)
        img_aux=cv2.copyMakeBorder(img_lap,int(16/pow(2,i-1)),int(16/pow(2,i-
1)),int(16/pow(2,i-1)),int(16/pow(2,i-1)),borde)
```

```
        lista_img_piramides.append(img_aux)
        img_mod=img_r

    return lista_img_piramides
```

La imagen la reducimos con cv2.pyrDown() e inmediatamente la ampliamos con cv2.pyrUp().

Es necesario al inicio de la iteración quitar la última fila y/o columna si es impar el número de filas o columnas. Se hace porque podríamos tener problemas de tamaños de las imagenes que reducimos y ampliamos a la hora de hacer la resta. También reescalamos la imagen resultante por si aparece algún número negativo.

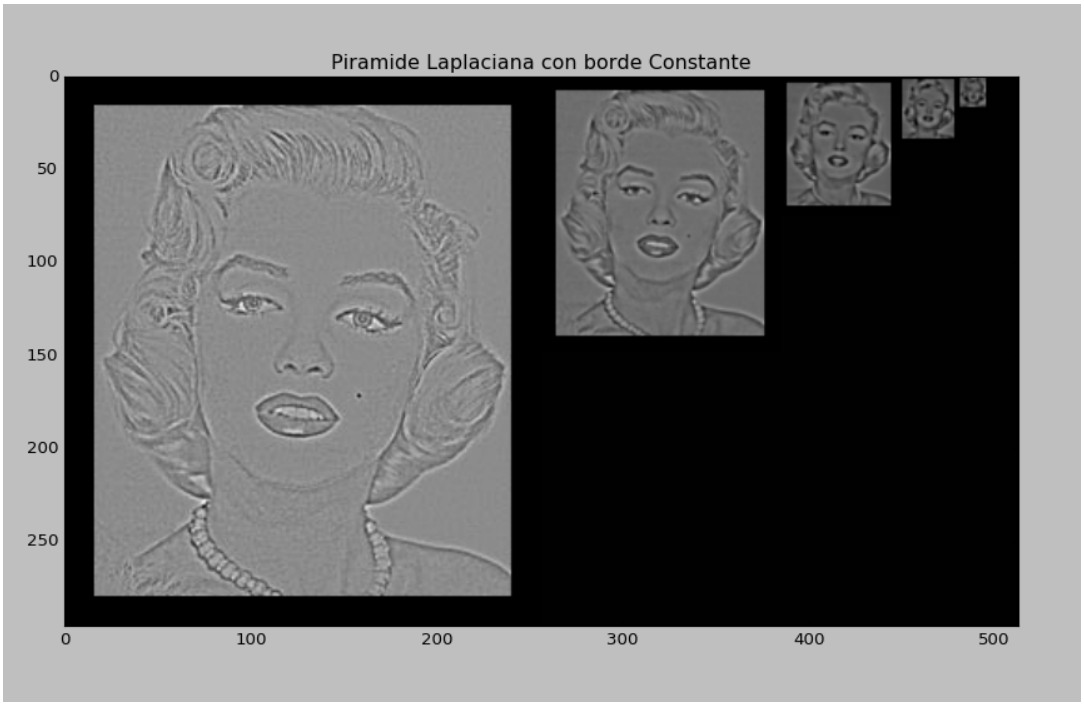
Utilizamos la función del anterior apartado para construir en sí la pirámide.

Esta es la llamada:

```
mi_img=cv2.imread("marilyn.bmp",0)
lista_img_piramides= piramide_Laplaciana(mi_img,bordes[2],5)

mi_lista=[]
titulos=[]
mi_lista.append(construir_piramide(lista_img_piramides))
titulos.append("Piramide Laplaciana con borde " + nombre_bordes[2])
```

Y ésta la salida:

```
imprimir(mi_lista,1,1,titulos,True)
```



Conseguimos así ver la imagen diferencia que existe entre la imagen original y el resultado de hacer pyrDown() y pyrUp() seguidos.

Ejercicio 2: Imágenes Híbridas

En este ejercicio trataremos las frecuencias altas y bajas de las imágenes consiguiendo que, al juntar las frecuencias altas de una imagen con las bajas de otra imagen, se obtenga una imagen en la que, dependiendo de la distancia a la que estemos, veremos una u otra imagen.

Para conseguir las altas o bajas frecuencias, jugaremos con el σ del alisamiento, que, cuanto mayor sea, nos quedaremos más con las frecuencias bajas de la imagen.

El filtro de paso-bajo se aplica a la imagen de la que queremos eliminar las frecuencias altas, un σ alto. El filtro de paso-alto nos dejará entonces con las altas frecuencias de la imagen, un σ menor.

Entonces la imagen de frecuencias bajas se obtiene aplicando un filtro Gaussiano con un σ alto y nuestra imagen.

En cambio, la imagen de frecuencias altas se calcula mediante la fórmula:

I*(1-filtro_paso_alto)* que equivale *a I-I**filtro_paso_alto,

es decir, la imagen original menos la imagen aplicando un filtro Gaussiano con un σ pequeño.

Finalmente se sumarán ambas imágenes resultantes y se hará un reescalado para obtener nuestra imagen híbrida. Todo esto se realiza en nuestra función:

```
def Get_Imagenes_Hibridas(img_alta,img_baja,σ_alto,σ_bajo):

    mi_lista=[]
    img_paso_alto = np.array(img_alta, dtype=np.float32)
    img_paso_bajo = np.array(img_baja, dtype=np.float32)
    img_hibrida = np.array(img_baja, dtype=np.float32)

    img_paso_alto=Gauss_kernel_Grey(img_alta.copy(),σ_alto,0,0,0)
    img_paso_bajo=Gauss_kernel_Grey(img_baja.copy(),σ_bajo,0,0,0)

    img_paso_alto=img_alta - img_paso_alto

    img_hibrida = img_paso_alto + img_paso_bajo

    img_hibrida=Reescalar(img_hibrida,False,True)

    mi_lista.append(img_paso_alto)
    mi_lista.append(img_paso_bajo)
```

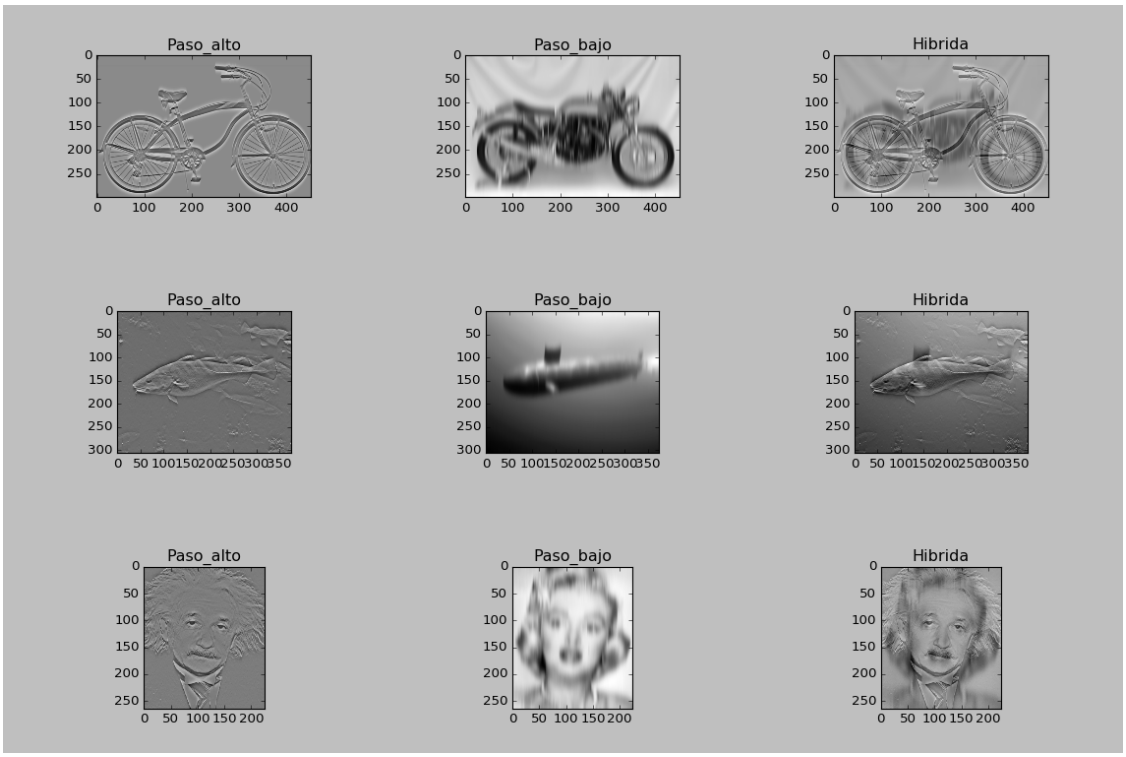
```
    mi_lista.append(img_hibrida)

    return mi_lista
```

Experimentalmente se ha probado con diferentes σs para conseguir el mejor efecto y finalmente se ha optado por σ_bajo=5 y σ_alto=7. Llamamos 3 veces a la función para mostrar 3 parejas diferentes:

```
img_bicycle=cv2.imread("bicycle.bmp",0)
img_motorcycle=cv2.imread("motorcycle.bmp",0)
img_fish=cv2.imread("fish.bmp",0)
img_submarine=cv2.imread("submarine.bmp",0)
img_einstein=cv2.imread("einstein.bmp",0)
img_marilyn=cv2.imread("marilyn.bmp",0)

mi_lista1=Get_Imagenes_Hibridas(img_bicycle,img_motorcycle,5,7)
mi_lista2=Get_Imagenes_Hibridas(img_fish,img_submarine,5,7)
mi_lista3=Get_Imagenes_Hibridas(img_einstein,img_marilyn,5,7)

mi_lista=mi_lista1+mi_lista2+mi_lista3
titulos=["Paso_alto","Paso_bajo","Hibrida","Paso_alto","Paso_bajo","Hibrida","P
aso_alto","Paso_bajo","Hibrida"]
```

Y éste es el resultado:

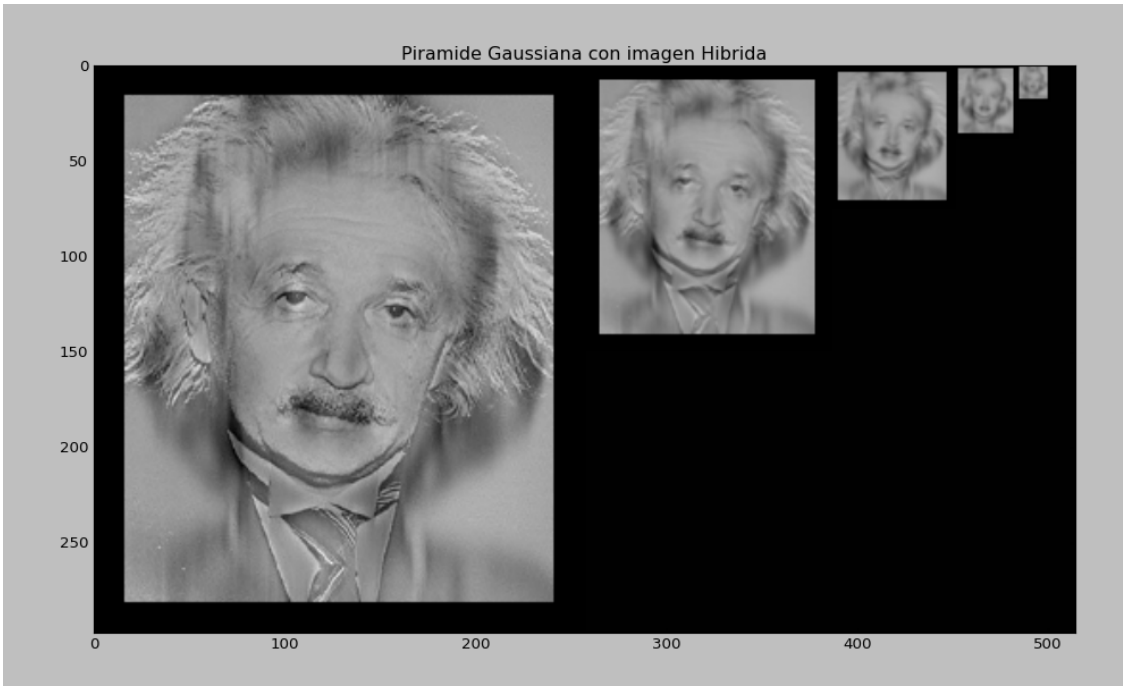
```
imprimir(mi_lista,3,3,titulos,True)
```



Observando las 3 parejas podemos observar que en algunas parejas se logra un mejor efecto que en otras. Esto se debe según los σs que estemos utilizando. En este caso, con la imagen de la bicicleta y la moto parece ser que se logra un mejor efecto.

Dado que el efecto se consigue cuando se aleja o acerca la imagen, incluimos una de las parejas en una pirámide Gaussiana de las anteriormente construidas para lograr este efecto:

```
mi_img=mi_lista3[2]
lista_img_piramides= piramide_Gaussiana(mi_img,0,5)


mi_lista=[]
mi_lista.append(construir_piramide(lista_img_piramides))
titulos=["Piramide Gaussiana con imagen Hibrida"]

imprimir(mi_lista,1,1,titulos,True)
```



Gracias a la pirámide Gaussiana podemos apreciar mucho mejor el efecto que se logra con las imágenes híbridas. En la primera vemos mucho mejor a Einstein que a Marilyn y pasa lo contrario conforme se aleja la imagen.

Bonus

Apartado 1: Cálculo del vector máscara Gaussiano

Para calcularlo, dado que sería imposible un vector infinito que representase cada uno de los puntos de la Gaussiana, discretizamos un vector que estará en el intervalo [-3σ,3σ], teniendo un total de $6\sigma+1$ *valores.* Éste tamaño del vector es en base a representar ese 95% de la Gaussiana que comentábamos en anteriores apartados.

El siguiente paso es sustituir cada uno de estos valores del vector en la función que se nos proporciona en el enunciado.

Esta máscara Gaussiana cumpliría que es simétrica y sus valores están entre 0 y 1. Sin embargo, faltaría que la suma del vector fuese 1 por lo que dividimos cada uno de los valores entre la sumatoria del vector para así conseguir al final que la suma total sea 1.

Todo esto es lo que se realiza en nuestra función:

```
def f(x,σ):
    t=exp(-0.5*(float)(pow(x,2)/pow(σ,2)))
    return t

def calcular_kernel(σ):
    tam=σ*3+1
    mi_kernel=np.ones(σ*6+1)

    for i in range(tam):
        mi_kernel[i]= σ*3-i

    for i in range(tam,tam+σ*3):
        mi_kernel[i]= i-3

    for i in range(len(mi_kernel)):
        mi_kernel[i]=f(mi_kernel[i],σ)

    mi_kernel=mi_kernel/sum(mi_kernel)

    return mi_kernel
```

Apartado 2: Convolución 1D

En este apartado tenemos que implementar la función que realizaba "filter2D()" de openCV, es decir, que dado un vector 1D y una máscara 1D, se realiza una convolución con ellos. La máscara la calculamos con el apartado anterior.

Para conseguirlo, en un bucle vamos recorriendo el vector posición a posición, tomando subvectores del mismo tamaño que la máscara, multiplicando uno a uno los valores del subvector y la máscara, y realizando la media de estos valores

resultantes. Esta media será entonces la que sustituyamos en nuestro vector 1D en lo posición que actualmente estemos recorriendo.

Esta posición será la que coincida con el máximo de la máscara y la que irá iterando a la par que movemos la máscara.

Para evitar sobrepasar el vector 1D crearemos a sus extremos unos marcos de 0s del tamaño de la mitad de la máscara, empezando y terminando el paso de la máscara por estos marcos. Esta es la función que añade los marcos a una imagen (de cara al siguiente apartado):

```
def anadir_marco(img,tam_borde):


img_mod=np.zeros((img.shape[0]+tam_borde*2,img.shape[1]+tam_borde*2)
)

    for i in range(tam_borde,img.shape[0]+tam_borde):
        for j in range(tam_borde,img.shape[1]+tam_borde):
            img_mod[i,j]=img[i-tam_borde,j-tam_borde]

    return img_mod
```

Y ésta la función que hace la convolución en sí:

```
def calcular_convolucion(img_vector,kernel,tam_borde,tam_orig):


    vector_mod=np.zeros(len(img_vector))

    for i in range(tam_borde,tam_borde+tam_orig):
        vector_mod[i]= sum(img_vector[i-tam_borde:i-
tam_borde+len(kernel)]*kernel)/len(kernel)


    return vector_mod
```

Apartado 3: Convolución 2D

En este apartado crearemos una convolución 2D, convolución de una imagen, apoyandonos en las 2 funciones de los 2 apartados anteriores.

Para poder tratar con imágenes a color y facilitar los cálculos, dividiremos la imagen de entrada en 3 bandas, RGB, realizaremos los mismos cálculos sobre ellas, y posteriormente las juntaremos para tener nuestra imagen a color. Esta es la función en cuestión:

```
def mi_convolucion(img,σ,colores=True):

    kernel=calcular_kernel(σ)
```

```
    img_conv=np.array(img, dtype=np.float32)

    tam_origX=img.shape[1]
    tam_origY=img.shape[0]

    if(colores):
        img_conv1=img_conv[:,:,0]
        img_conv2=img_conv[:,:,1]
        img_conv3=img_conv[:,:,2]
    else:
        img_conv1=img_conv

    img_conv1=anadir_marco(img_conv1,int(len(kernel)/2))
    if(colores):
        img_conv2=anadir_marco(img_conv2,int(len(kernel)/2))
        img_conv3=anadir_marco(img_conv3,int(len(kernel)/2))



    for i in range(img_conv1.shape[0]):

img_conv1[i]=calcular_convolucion(img_conv1[i],kernel,int(len(kernel)/2),tam
_origX)
        if(colores):

img_conv2[i]=calcular_convolucion(img_conv2[i],kernel,int(len(kernel)/2),tam
_origX)

img_conv3[i]=calcular_convolucion(img_conv3[i],kernel,int(len(kernel)/2),tam
_origX)


    for i in range(img_conv1.shape[1]):

img_conv1[:,i]=calcular_convolucion(img_conv1[:,i],kernel,int(len(kernel)/2),t
am_origY)
        if(colores):

img_conv2[:,i]=calcular_convolucion(img_conv2[:,i],kernel,int(len(kernel)/2),t
am_origY)

img_conv3[:,i]=calcular_convolucion(img_conv3[:,i],kernel,int(len(kernel)/2),t
am_origY)

    if(colores):
        img_conv=cv2.merge((img_conv1,img_conv2,img_conv3))
    else:
        img_conv=img_conv1
```

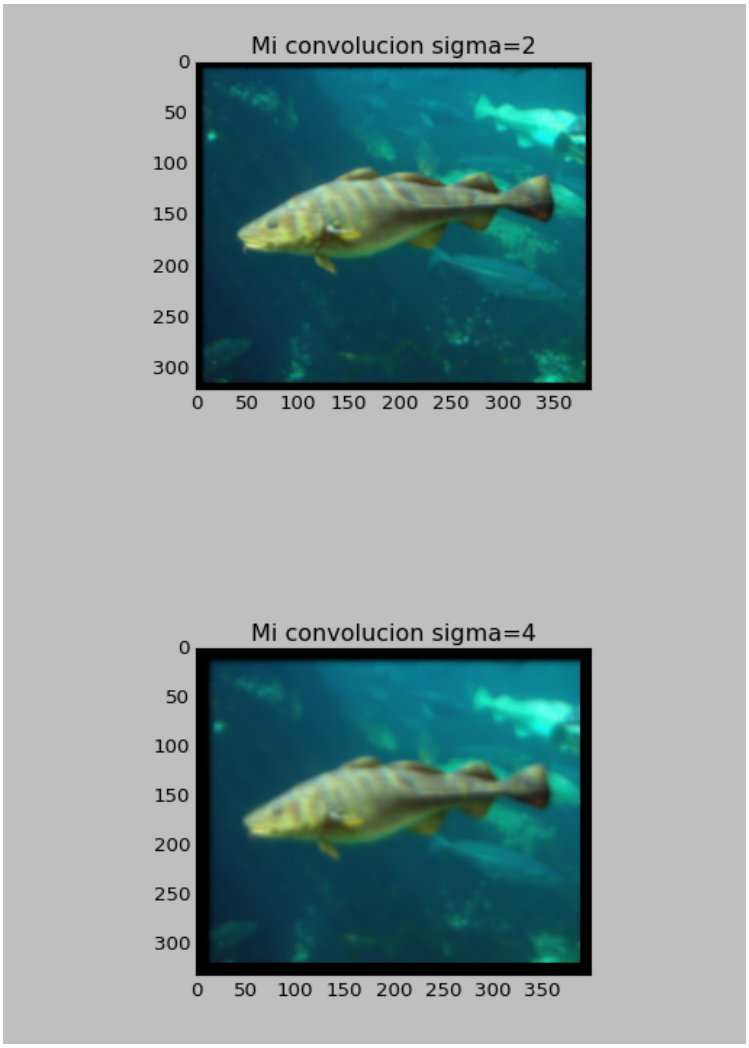
```
    img_conv=Reescalar(img_conv,True)


    return img_conv
```

Vemos que sigue una estructura muy similar a la que teniamos en el apartado C del ejercicio 1 aunque ahora todo es con implementaciones propias.

Añadimos marcos alrededor de las 3 imágenes para poder aplicar correctamente la máscara o kernel del apartado 1. Vemos que en los 2 bucles tratamos a la vez con las imágenes de las 3 bandas y en que en ellos llamamos a nuestra convolución 1D del 2º apartado.

Pasamos el kernel por filas y columnas como hacíamos entonces y al final realizamos un "merge()" para juntar las 3 bandas en una sola imagen. Finalmente realizamos un reescalado y devolvemos nuestra imagen. Éste es el resultado que obtenemos:

```
lista_imagenesB = []
titulos = []

img=cv2.imread("fish.bmp")

for i in range(1,3):
    mi_img=mi_convolucion(img.copy(),i*2)
    lista_imagenesB.append(np.array(mi_img, dtype=np.uint8))
    titulos.append("Mi convolucion σ=" + str(i*2))

imprimir(lista_imagenesB,2,1,titulos)
```



Vemos que al igual que pasaba en la convoluciones que veiamos en el ejercicio 1, al aumentar σ perdemos detalles en las imágenes, no son tan nítidas.

Apartado 4: Pirámide Gaussiana

Esta es la función que se encarga de crear las imágenes híbridas:

```
def Mis_Imagenes_Hibridas(img_alta,img_baja,σ_alto,σ_bajo):

    mi_lista=[]


img_paso_alto=np.zeros((img_alta.shape[0]+σ_alto*6,img_alta.shape[1]+σ_alto*6),dtype=np.float32)

img_paso_bajo=np.zeros((img_baja.shape[0]+σ_bajo*6,img_baja.shape[1]+σ_bajo*6),dtype=np.float32)
```

```
    img_paso_alto=mi_convolucion(img_alta.copy(),σ_alto,False)
    img_paso_bajo=mi_convolucion(img_baja.copy(),σ_bajo,False)


    img_paso_alto=quitar_marcos(img_paso_alto,σ_alto*3)
    img_paso_bajo=quitar_marcos(img_paso_bajo,σ_bajo*3)


    img_paso_alto=img_alta - img_paso_alto

    img_hibrida = img_paso_alto + img_paso_bajo

    img_hibrida=Reescalar(img_hibrida,False,True)

    mi_lista.append(img_paso_alto)
    mi_lista.append(img_paso_bajo)
    mi_lista.append(img_hibrida)

    return mi_lista


def quitar_marcos(img,tam_marco):

    img=img[tam_marco:img.shape[0]-tam_marco,tam_marco:img.shape[1]-
tam_marco]

    return img
```

Vemos que sigue exactamente la misma estructura que veíamos en el ejercicio 2 salvo que ahora utilizamos nuestra propia convolución que creamos en el anterior apartado y utilizamos la función "quitar_marcos()" para quitar los marcos que coloca la convolución para tratar más facilmente las imágenes. Utilizamos la misma función "Reescalar()" que ya vimos que era una implementación propia.

Respecto a la pirámide, la función sigue la misma estructura que la del ejercicio 1:

```
def mi_piramide_Gaussiana(img,σ,n):

    lista_img_piramides = []

    for i in range(1,n+1):
        img=mi_convolucion(img,σ,False)
        img=quitar_marcos(img,σ*3)
        lista_img_piramides.append(img)
        img=reducir_img(img,False)

    return lista_img_piramides
```

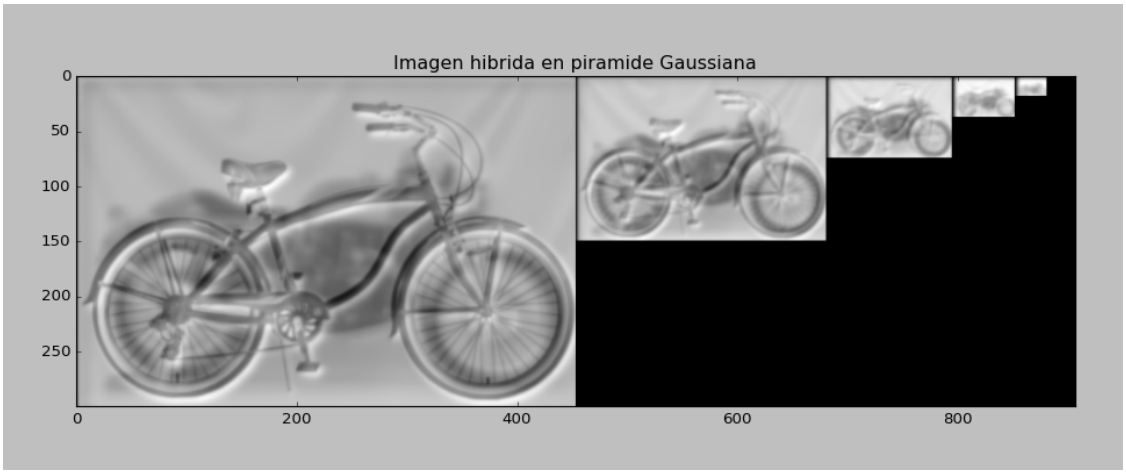
Salvo que ahora realizamos nuestro propio alisamiento y la reducción de la imagen la hacemos nosotros, basicamente quitando las filas y columnas impares de la imagen:

```
def reducir_img(img,colores=True):

    k=0
    t=0

    list_i=range(len(img))
    list_j=range(len(img[0]))
    list_i=[x for x in list_i if x%2 == 0]
    list_j=[x for x in list_j if x%2 == 0]

    if(colores):
        img_r=np.zeros((len(list_i),len(list_j),3))
    else:
        img_r=np.zeros((len(list_i),len(list_j)))

    for i in list_i:
        t=0
        for j in list_j:
            img_r[k,t]=img[i,j]
            t=t+1
        k=k+1

    return img_r
```

Además, utilizamos la misma función que ya creamos para construir la piramide ya que era una implementación propia. Éste es el resultado que al final obtenemos:

```
img_bicycle=cv2.imread("bicycle.bmp",0)
img_motorcycle=cv2.imread("motorcycle.bmp",0)
img_fish=cv2.imread("fish.bmp",0)
img_submarine=cv2.imread("submarine.bmp",0)
img_einstein=cv2.imread("einstein.bmp",0)
img_marilyn=cv2.imread("marilyn.bmp",0)


mi_lista=Mis_Imagenes_Hibridas(img_bicycle.copy(),img_motorcycle.copy(),5,
7)



mi_img=mi_lista[2]
lista_img_piramides=mi_piramide_Gaussiana(mi_img,2,5)

mi_piramide = construir_piramide(lista_img_piramides)

titulo=["Imagen hibrida en piramide Gaussiana"]
mi_lista=[]
```

```
mi_lista.append(mi_piramide)

imprimir(mi_lista,1,1,titulo,True)
```



Vemos que de cerca vemos mucho mejor que la bicicleta que la moto, pero al alejarse la imagen, la bicicleta no se ve tan clara y empieza a aparece la moto.