Universidad de Granada

Ingeniería Informática

Computación y Sistemas Inteligentes

Practica 1

Autor: José Antonio Ruiz Millán
Asignatura: Visión por Computador
20 de octubre de 2018

1. **USANDO LAS FUNCIONES DE OPENCV : escribir funciones que implementen los siguientes puntos: (2 puntos)**

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

   En este apartado he utilizado una funcion que permite imprimir varios elementos en una misma ventana. La funcion se define:

```
def multIM(img,nfil,ncol,tamx,tamy,color=True):
    fig=plt.figure( figsize =(tamx, tamy))
    for i,im in enumerate(img):
        fig.add_subplot(nfil, ncol, i+1)
        imgt = (np.clip(im,0,1)*255.).astype(np.uint8)
        if color:
            nimg = cv2.cvtColor(imgt, cv2.COLOR_BGR2RGB)
        else:
            nimg = cv2.cvtColor(imgt,cv2.COLOR_GRAY2RGB)
        plt.imshow(nimg)
    plt.show()
```

   Esta funcion crea una figura conjunta y va recorriendo cada una de las subfiguras y poniendo cada imagen en una posicion para finalmente mostrarla completa.

   Una vez mostrado esto, pongo la resolucion del ejercicio:

```
img = leeimagen('imagenes/bicycle.bmp',1)

concat = [img]

for i in range(3,12,2):
    concat.append(cv2.GaussianBlur(img,(i,i),0,0))

multIM(concat,2,3,28,13)
```

   Como se puede apreciar, leemos la imagen en mi caso en color, y vamos concatenando todas las imagenes con distintos valores de sigma (ksize) para finalmente mostrarlas por pantalla. La imagen que queda finalmente es la siguiente:

Podemos apreciar en la imagen como a medida que vamos aumentando el ksize y por lo tanto el sigma (se calcula a partir del ksize), obtenes un difuminado mas fuerte, esto se debe a que al aumentar el kernel, la matriz es de mayor tamaño y al aplicarle la convolución, el valor central se calcula con respecto a más pixeles de su alrededor y esto hace que en su totalidad, la imagen se muestre mas difuminada.

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

En mi caso, he utilizado kernels de tamaño 3,5 y 7. Utilizando 3 tipos de máscaras, la máscara con la primera derivada respecto a x únicamente, otra con sólo la derivada respecto a y, y finalmente la primera derivada respecto a x e y. Para ello, he utilizado el siguiente código:

```
            for i in range(3,8,2):
2               kx, ky = cv2.getDerivKernels(1,0,i)

4               print('Kernel de tamano '\+str(i)\+ ' con dx=1 y dy=0')
                print(np.dot(kx,np.transpose(ky)))
6
        for i in range(3,8,2):
8           kx, ky = cv2.getDerivKernels(1,1,i)

10          print('Kernel de tamano '+str(i)+ ' con dx=1 y dy=1')
            print(np.dot(kx,np.transpose(ky)))
12
        for i in range(3,8,2):
14          kx, ky = cv2.getDerivKernels(0,1,i)

16          print('Kernel de tamano '+str(i)+ ' con dx=0 y dy=1')
            print(np.dot(kx,np.transpose(ky)))
18
```

Y como resultado de esta ejecución obtenemos los siguientes resultados:

Kernel de tamaño 3 con dx=1 y dy=0

```
[[-1. -2. -1.]
 [ 0.  0.  0.]
 [ 1.  2.  1.]]
```

Esta máscara como podemos comprobar lo que nos ofrece es una muy buena localizacion de los contrastes que se encuentran de forma vertical, por ejemplo, una linea que se encuentre en la imagen de forma vertical, se va a conseguir obtenerla con facilidad, mientras que por ejemplo las lineas horizontales perderemos o no le daremos tanta significatividad.

Kernel de tamaño 5 con dx=1 y dy=0

```
[[ -1.  -4.  -6.  -4.  -1.]
 [ -2.  -8. -12.  -8.  -2.]
```

[0. 0. 0. 0. 0.]

[2. 8. 12. 8. 2.]

[1. 4. 6. 4. 1.]]

Como el anterior, este tipo de kernel tiene mas fuerza en lineas verticales, y al aumentar el kernel lo que conseguimos es que la transformación de la imagen sea mas suavizada y por lo tanto mas difuminada. Se puede comprobar esto ya que ahora, el pixel central depende de más valores que en anterior, por lo que ahora es mas sensible a los pixeles que le rodea y esto hace que el pixel esté mas suavizado.

Kernel de tamaño 7 con dx=1 y dy=0

[[-1. -6. -15. -20. -15. -6. -1.]

[-4. -24. -60. -80. -60. -24. -4.]

[-5. -30. -75. -100. -75. -30. -5.]

[0. 0. 0. 0. 0. 0. 0.]

[5. 30. 75. 100. 75. 30. 5.]

[4. 24. 60. 80. 60. 24. 4.]

[1. 6. 15. 20. 15. 6. 1.]]

Como seguimos con el mismo tipo de kernel pero únicamente hemos aumentado el kernel, tenemos el mismo caso que el anterior, y como podemos ver sigue el mismo camino. El aumento del valor ded los pixeles contiguos al centro es básicamente para darle más importancia a estos ya que se encuentran mas cerca del pixel que realmente queremos evaluar/transformar. El resto de caracteristicas son las mismas que los anteriores.

Kernel de tamaño 3 con dx=1 y dy=1

[[1. 0. -1.]

[0. 0. 0.]

[-1. 0. 1.]]

Ahora hemos cambiado de kernel y tenemos el kernel conjnuto de la primera derivada respecto a x y respecto a y. Lo que podemos ver ahora es que este kernel se centra en buscar lineas que se encuentran en diagonal, dejando un poco mas alejadas las verticales y horizontales. Por ejemplo si aplicamos esta convolución en una imagen en la que tenemos una rueda, este filtro conseguirá encontrar perfectamente los cambios de contraste en las partes donde la rueda esta curvada respecto al eje x o al y, miestras que la zona que pisa el suelo, la zona de arriba y los dos laterales estarán menos señados por este filtro.

Kernel de tamaño 5 con dx=1 y dy=1

[[1. 2. -0. -2. -1.]

[2. 4. -0. -4. -2.]

[-0. -0. 0. 0. 0.]

[-2. -4. 0. 4. 2.]

[-1. -2. 0. 2. 1.]]

En este caso, seguimos con el kernel 1,1; por lo que aparte de las propiedades comentadas en la matriz anterior, tenemos la ampliación de la misma, que conlleva a reevalorar los valores de la matriz, que se conservan como la anterior pero dandonde más importancia a los pixeles que se encuentran proximos al centro y dandole menos importancia a los lejanos, teniendo en cuenta como he comentado anteriormente que se centra en lineas diagonales.

Kernel de tamaño 7 con dx=1 y dy=1

[[1. 4. 5. -0. -5. -4. -1.]

[4. 16. 20. -0. -20. -16. -4.]

[5. 20. 25. -0. -25. -20. -5.]

[-0. -0. -0. 0. 0. 0. 0.]

[-5. -20. -25. 0. 25. 20. 5.]

[-4. -16. -20. 0. 20. 16. 4.]

[-1. -4. -5. 0. 5. 4. 1.]]

Continuamos con el mismo kernel que el anterior pero aumentando el valor de *ksize*, por lo que lo que conseguimos ahora es aumentar aún mas los pixeles que utilizaremos para evaluar el pixel central, lo que esto conlleva a una mayor difuminacion del mismo. Seguimos conservando el valor mayor a los valores cercanos al pixel y mostrandole la mayor antencion a las lienas que se encuentren en diagonal.

Kernel de tamaño 3 con dx=0 y dy=1

[[-1. 0. 1.]

[-2. 0. 2.]

[-1. 0. 1.]]

Pasamos ahora al caso contrario que teníamos al principio del ejercicio, ahora tenemos un kernel que hemos obtenido con la primera derivada respecto a y en vez de respecto a x como en el primer ejemplo. Por lo que teniendo lo contrario que al principio, este kernel se encarga de buscar los contrastes en líneas horizontales, no siendo muy significativo para lineas verticales.

Kernel de tamaño 5 con dx=0 y dy=1

[[-1. -2. 0. 2. 1.]

[-4. -8. 0. 8. 4.]

[-6. -12. 0. 12. 6.]

[-4. -8. 0. 8. 4.]

[-1. -2. 0. 2. 1.]]

Al igual que el anterior, seguimos con el mismo tipo de kernel, favoreciendo a las líneas horizontales, pero aumentando el tamaño del kernel, lo que hace que el pixel central esté con un valor más repartido entre los pixeles contiguos a él. Esto aumenta el alisado y el difuminado.

Kernel de tamaño 7 con dx=0 y dy=1

[[-1. -4. -5. 0. 5. 4. 1.]

[-6. -24. -30. 0. 30. 24. 6.]

[-15. -60. -75. 0. 75. 60. 15.]

[-20. -80. -100. 0. 100. 80. 20.]

[-15. -60. -75. 0. 75. 60. 15.]

[-6. -24. -30. 0. 30. 24. 6.]

[-1. -4. -5. 0. 5. 4. 1.]]

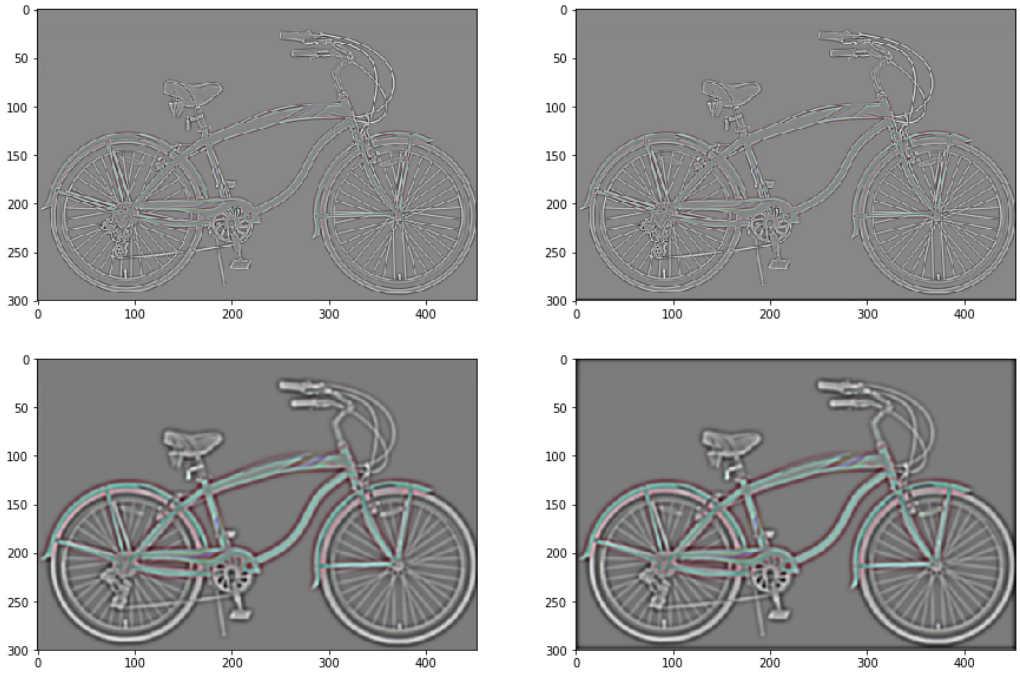
Al igual que el anterior tenemos lo mismo, pero con un aumento del kernel que este cambio afecta igual que el aumento anterior pero teniendo aún más efecto

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

Para este ejercicio he utilizado la imagen de una bicicleta, y como nos pedian dos tipos de bordes he decidido utilizar el borde reflejado y el borde constante para cada uno de los valores de sigma que nos piden. El codigo sería el siguiente:

```
1               dst = []

3               aux     = cv2.Laplacian(src=img,ddepth=-1,ksize=3,borderType=cv2.
    BORDER_REFLECT_101)
                dst.append(aux)
5               aux     = cv2.Laplacian(src=img,ddepth=-1,ksize=3,borderType=cv2.
    BORDER_CONSTANT)
                dst.append(aux)
7
                aux     = cv2.Laplacian(src=img,ddepth=-1,ksize=17,borderType=cv2.
    BORDER_REFLECT_101)
9               dst.append(aux)
                aux     = cv2.Laplacian(src=img,ddepth=-1,ksize=17,borderType=cv2.
    BORDER_CONSTANT)
11              dst.append(aux)

13              dst = map(transformar,dst)

15              multIM(dst,2,2,15,10)
```

En la que obtenemos como resultado:



Podemos observar como al aumentar el valor de σ (calculado respecto a ksize), aumentamos el contraste y la informacion alrededor de los constrastes formados en la imagen, lo que nos permite en este caso una mejor visualizacion de los bordes. Como podemos ver en la imagen, al cambiar el borde de la imagen de reflejado a constante, nos mete unos pixeles de borde negros, que al aumentar el sigma y por lo tanto el tamaño del kernel, el borde se acopla más en la imagen y aparece con un tamaño más grande ya que el kernel utiliza más pixeles para la transformación.

Como podemos observar en el codigo he tenido que crear una funcion llamada *transformar* que lo que hace es normalizar los valores que estan fuera del rango 0-1 que no serían visibles al rango 0-1. Este metodo hace lo siguiente:

```
                minimo = np.min(img)
                maximo = np.max(img)

                if minimo < 0 or maximo > 1:
                    img = (img-minimo)/(maximo-minimo)

                return img
```

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

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

   Para este ejercicio he utilizado el siguiente codigo:

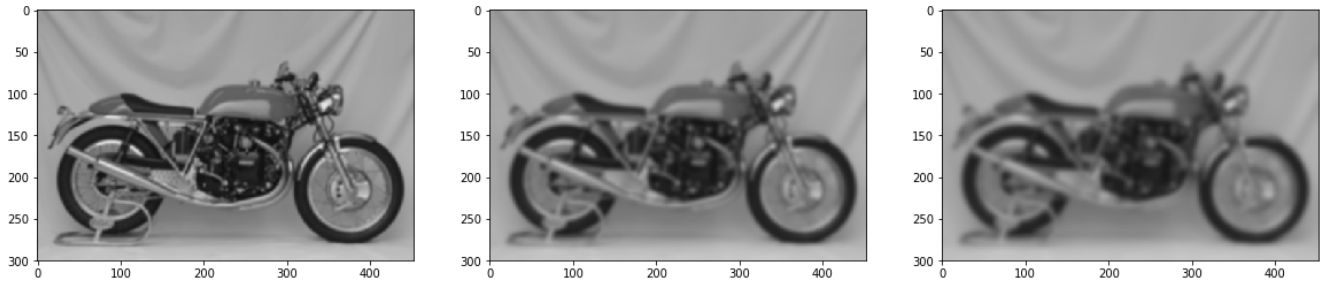
```
                img = leeimagen('imagenes/motorcycle.bmp',0)

                conv_gauss1 = convolucion_gauss(img,7)
                conv_gauss2 = convolucion_gauss(img,15)
```

```
conv_gauss3 = convolucion_gauss(img,21)

multIM([conv_gauss1,conv_gauss2,conv_gauss3],1,3,20,10,False)
```

Donde he creado un metodo llamado *convolucion_gauss* que nos permite dada un imagen y un tamaño que puede variar de kernel, nos crea y devuelve la convolucion aplicada sobre la imagen. Este metodo hace lo siguiente:

```
def convolucion_gauss(img,tam):
    val = cv2.getGaussianKernel(ksize=tam,sigma=-1)

    res = cv2.sepFilter2D(src=img,ddepth=-1,kernelX=val,kernelY=val,
borderType=cv2.BORDER_REFLECT_101)

    return res
```

Del cual obtenemos como resultado:



Como podemos observar, se ha creado en la imagen un difuminado que aumenta cambiando el valor del ksize que le pasamos al metodo creado.

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

Para este ejercicio el código que he utilizado a sido el siguiente:

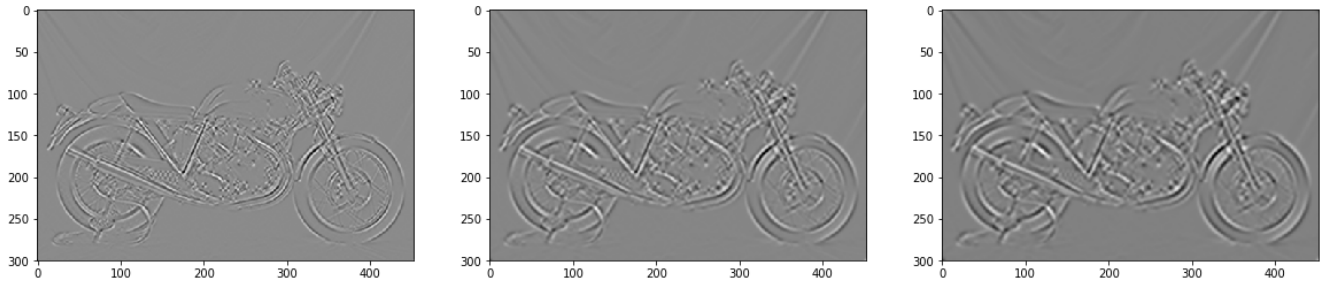
```
conv_1deriv1 = transformar(convolucion_1deriv(img,7))
conv_1deriv2 = transformar(convolucion_1deriv(img,15))
conv_1deriv3 = transformar(convolucion_1deriv(img,21))

multIM([conv_1deriv1,conv_1deriv2,conv_1deriv3],1,3,20,10,False)
```

Para ello, he creado una funcion llamada *convolucion_1deriv* que me devuelve la convolucion de derivada 1 en x e y aplicando bordes constantes. Podemos verla en el siguiente codigo:

```
def convolucion_1deriv(img,tam)
kx,ky = cv2.getDerivKernels(1,1,tam)

res = cv2.sepFilter2D(src=img,ddepth=-1,kernelX=kx,kernelY=ky,borderType=
cv2.BORDER_CONSTANT)

return res
```

```
7
```

Del cual obtenemos como resultado:



Vemos como esta convolucion, conforme aumentamos el valor del sigma, hace mas incapié en los contrastes de los bordes y conseguimos hacerlos mas notables, mas gruesos.

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

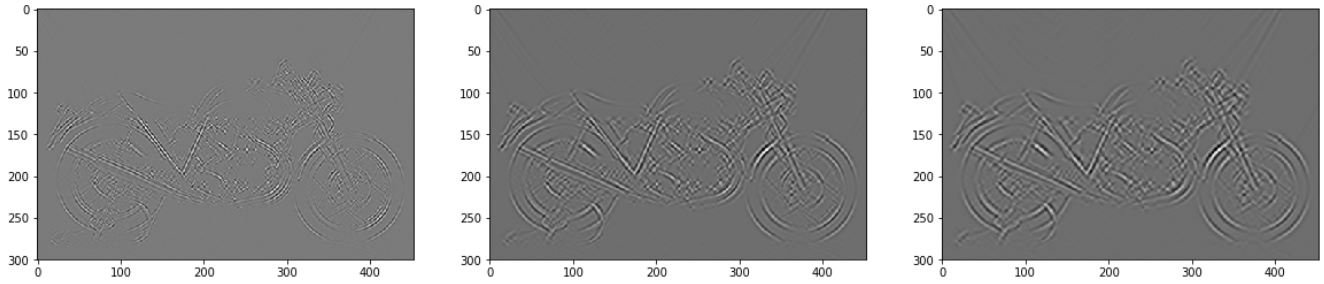
Para este ejercicio el codigo que he utilizado es similar al anterior:

```
                conv_2deriv1 = transformar(convolucion_2deriv(img,7))
2               conv_2deriv2 = transformar(convolucion_2deriv(img,15))
                conv_2deriv3 = transformar(convolucion_2deriv(img,21))
4
                multIM([conv_2deriv1,conv_2deriv2,conv_2deriv3],1,3,20,10,False)
6
```

Donde ahora en este caso tengo la nueva funcion llamada *convolucion_2deriv* que es igual que la funcion vista anteriormente pero aplicando la segunda derivada, no obstante, la funcion es la siguiente:

```
1               def convolucion_2deriv(img,tam)
                    kx,ky = cv2.getDerivKernels(2,2,tam)
3
                    res  = cv2.sepFilter2D(src=img,ddepth=-1,kernelX=kx,kernelY=ky)
5
                    return res
7
```

Lo que nos devuelve como resultado:



COmo vemos en las imágenes, al hacer la segunda derivada, enfocamos el esfuerzo en los puntos donde obtenemos el máximo contraste y al aumentar al sigma conseguimos aumentar el tamaño de los bordes donde se encuentran estos puntos.

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

Para este ejercicio he utilizado la imagen de la motocicleta en blanco y negro, he creado una piramide de 4 niveles utilizando bordes reflejados. El codigo sería el siguiente:

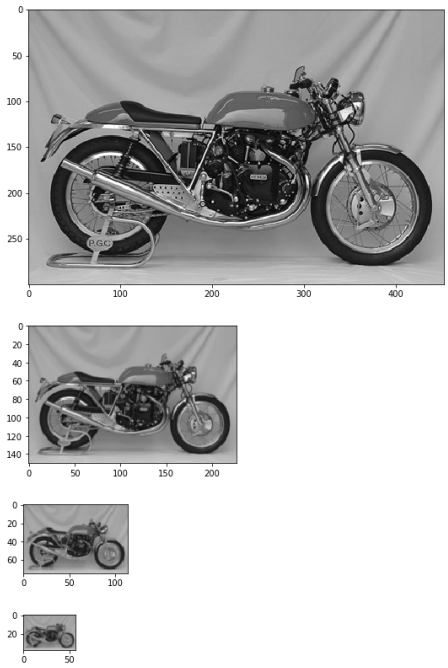
```
dst = piramide_gauss(img)

display_piramide(dst,False)

```

Donde *piramide_gauss* y *display_piramide* se define:

```
def piramide_gauss(img):
    res = [img]

    aux = cv2.pyrDown(img,borderType=cv2.BORDER_REFLECT_101)
    res.append(aux)
    aux = cv2.pyrDown(aux,borderType=cv2.BORDER_REFLECT_101)
    res.append(aux)
    aux = cv2.pyrDown(aux,borderType=cv2.BORDER_REFLECT_101)
    res.append(aux)

    return res
```

```
def display_piramide(img,color=True):
    for im in img:
        imgt = (np.clip(im,0,1)*255.).astype(np.uint8)
        if color:
            nimg = cv2.cvtColor(imgt, cv2.COLOR_BGR2RGB)
        else:
            nimg = cv2.cvtColor(imgt,cv2.COLOR_GRAY2RGB)
        dpi = 50
        height, width, depth = nimg.shape
        figsize = width / float(dpi), height / float(dpi)
        plt.figure( figsize =figsize)
        plt.imshow(nimg)
    plt.show()
```

Con esto obtenemos la siguiente imagen donde podemos ver perfectamente la pirámide.



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

AL igual que en el caso anterior, utilizo 4 niveles para la pirámide y bordes reflejados, pero en este caso el proceso será distinto ya que voy a mostrar tanto las imagenes cuando creamos la pirámide hacia abajo, como las imágenes que guardamos para poder realizar la pirámide hacia arriba, y finalmente la pirámide hacia arriba. Con lo que tenemos el siguiente codigo:

```
dest = []
dif = []

dst, dif = piramide_laplaciana_down(img)

display_piramide(dst,False)

trans = map(transformar,dif)

display_piramide(trans,False)

up = piramide_laplaciana_up(dst[len(dst)-1],dif)

display_piramide(up,False)
```

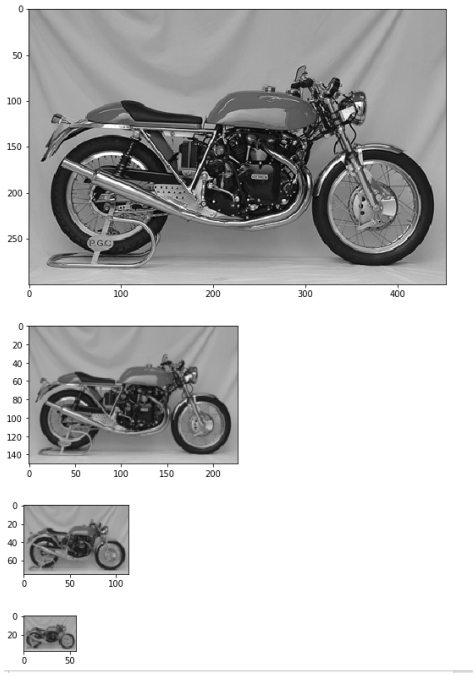
Donde los métodos *piramide_laplaciana_down* y *piramide_laplaciana_up* se definen:

```
def piramide_laplaciana_down(img):
    res = [img]
    dif = []
    aux = img

    for i in range(3):
        mini = cv2.pyrDown(aux,borderType=cv2.BORDER_REFLECT_101)
```

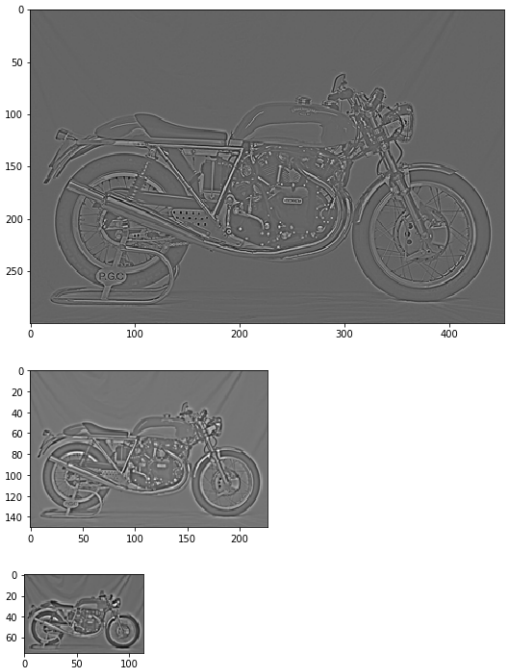
```
                col, fil  = aux.shape
                maxi = cv2.pyrUp(mini,None,(fil,col),borderType=cv2.
    BORDER_REFLECT_101)
                dif.append(aux-maxi)
                res.append(mini)
                aux = mini

            return res, dif
```

```
        def piramide_laplaciana_up(img,dif):
            res = [img]
            aux = img

            for im in reversed(dif):
                col, fil  = im.shape
                maxi = cv2.pyrUp(aux,None,(fil,col),borderType=cv2.
    BORDER_REFLECT_101)
                real  = maxi+im
                res.append(real)
                aux = real

            return res
```

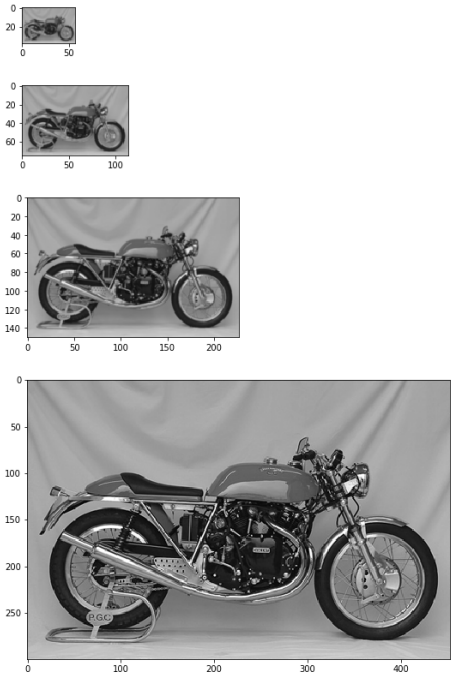
Por lo que tenemos como salida, en primer lugar la piramide hacia abajo.



Luego tendríamos las distintas imagenes que almacenamos para poder realizar el up, que son la diferencia entre la imagen real y la imagen escalada con *pyrUp*.



Y por ultimo, el resultado final para comprobar que realizamos el up correctamente.



Para terminar, en resumen hemos podido comprobar que el valor de sigma o el tamaño del kernel afecta a la hora del calculo, en cómo ese pixel se va a representar, por ejemplo cuando tenemos un sigma muy alto, las imagenes se difuminan mas en el caso de un filtro gaussiano y tambien nos permite aumentar el tamaño de los bordes en la laplaciana o en la derivada ya que exapandimos o difuminamos los pixeles.

3. **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.

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

Para este ejercicio he creado una funcion que dadas dos imagenes, nos crea la imagen hibrida entre ellas y nos la devuelve. Pero antes de nada, mostrare el codigo main, que en mi caso lo realizo con 3 parejas de imagenes donde primero para cada una de ellas creo su imagen híbrida, luego muestro las 3 imagenes juntas como nos piden en el siguiente ejerciio y por último las muestro en una pirámide de gauss para comprobar el resultado.

```
img1 = leeimagen('imagenes/cat.bmp',0)
img2 = leeimagen('imagenes/dog.bmp',0)

animales = crear_hibrida(img1,img2,41,31)

multIM([img1,img2,animales],1,3,15,10,False)

img1 = leeimagen('imagenes/einstein.bmp',0)
img2 = leeimagen('imagenes/marilyn.bmp',0)

personas = crear_hibrida(img1,img2,21,17)

multIM([img1,img2,personas],1,3,15,10,False)

img1 = leeimagen('imagenes/bird.bmp',0)
img2 = leeimagen('imagenes/plane.bmp',0)

aire  = crear_hibrida(img1,img2,24,17)

multIM([img1,img2,aire],1,3,15,10, False)

pir  = piramide_gauss(personas)

display_piramide(pir, False)
```

Para el correcto funcionamiento de este codigo se necesita la implementacion de las distintas funciones que vemos en el código que iré mostrando en cada uno de los apartados del ejercicio. En este caso, toca enseñar la funcion *crear_hibrida*, que dadas dos imagenes con los dos valores del ksize, devuelve su imagen hibrida.

```
def crear_hibrida(img1,img2,ksize1,ksize2):
    gauss = cv2.getGaussianKernel(ksize1,-1)
    gauss = cv2.sepFilter2D(img1,-1,gauss,gauss)

    g2 = cv2.getGaussianKernel(ksize2,-1)
    g2 = cv2.sepFilter2D(img2,-1,g2,g2)
    laplacian = img2-g2

    res = gauss+laplacian

    return res

```

a) **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)**

Una vez tenemos lo anterior, ya podemos definir este método, que en mi caso es *multIM* que permite pasandole las imagenes, el numero de filas y columnas y un tamaño para la salida, escribir las imagenes en una misma imagen. Como trabajamos con las imagenes con valores entre 0-1 y flotantes, en este método que sirve para visualizar, realiza la transformacion.

```
def multIM(img,nfil,ncol,tamx,tamy,color=True):
    fig=plt.figure( figsize =(tamx, tamy))
    for i,im in enumerate(img):
        fig.add_subplot(nfil, ncol, i+1)
        imgt = (np.clip(im,0,1)*255.).astype(np.uint8)
        if color:
            nimg = cv2.cvtColor(imgt, cv2.COLOR_BGR2RGB)
        else:
            nimg = cv2.cvtColor(imgt,cv2.COLOR_GRAY2RGB)
        plt.imshow(nimg)
    plt.show()
```

b) **Realizar la composición con al menos 3 de las parejas de imágenes**

Por último en este apartado mostraré los distintos resultados.

Imagen híbrida perro-gato:

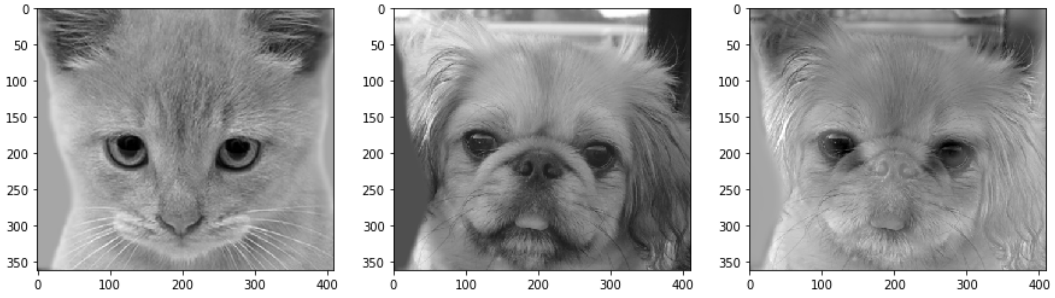


Imagen híbrida Einstein-Marilyn:

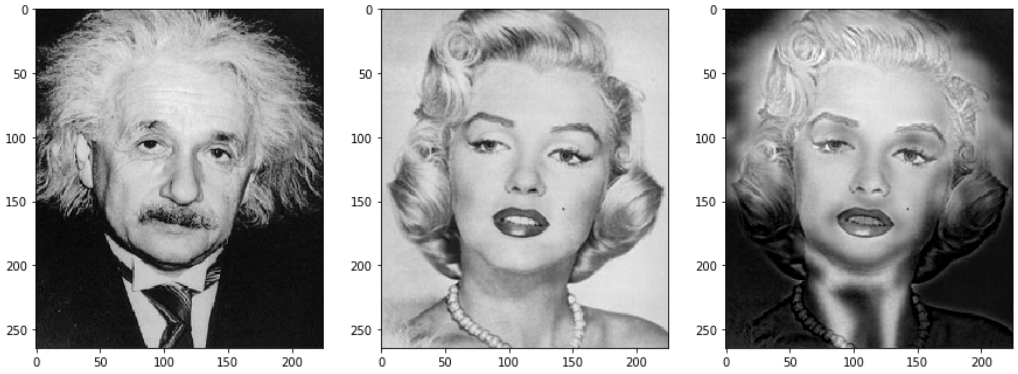
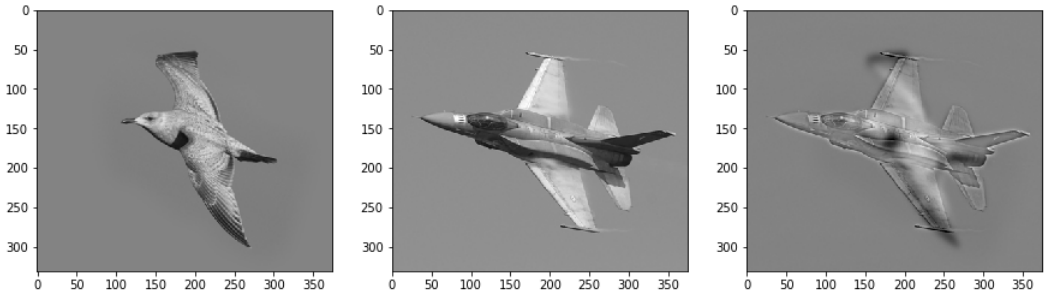


Imagen híbrida avion-pajaro:



Y por último para comprobar el efecto, realicé una piramide gaussiana con Einstein-Marilyn donde se puede apreciar el cambio.