

Doble Grado en Ingeniería Informática y Matemáticas

VISIÓN POR COMPUTADOR
(E. Computación y Sistemas Inteligentes)

TRABAJO-1: Filtrado y Detección de regiones



**UNIVERSIDAD
DE GRANADA**

Carlos Santiago Sánchez Muñoz

Grupo de prácticas 2 - Jueves

Email: carlossamu7@correo.ugr.es

21 de octubre de 2019

Índice

Ejercicio 1	2
Apartado A	2
Apartado B	8
Ejercicio 2	10
Apartado A	10
Apartado B	11
Apartado C	12
Ejercicio 3	16
Apartado 1	16
Apartado 2	16
Apartado 3	18
Bonus 1	20
Bonus 2	22
Bonus 3	24

Ejercicio 1

USANDO LAS FUNCIONES DE OPENCV: escribir funciones que implementen los siguientes puntos:

Apartado A

El cálculo de la convolución de una imagen con una máscara 2D. Usar una Gaussiana 2D (GaussianBlur) y máscaras 1D dadas por getDerivKernels. Mostrar ejemplos con distintos tamaños de máscara, valores de sigma y condiciones de contorno. Valorar los resultados.

El cálculo de la convolución lo he resuelto programando una función que llamado `convolution` a la cual le pasamos la imagen a convolucionar y las máscaras, una para el eje 0 (filas) y otra para el eje 1 (columnas). Para ello necesitamos transponer solo la máscara en la dirección X y llamar a `cv2.flip` dos veces una con cada máscara. Este paso es fundamental ya que si no hacemos esa llamada estaríamos haciendo la correlación y no la convolución. Finalmente, con las máscaras listas llamaremos dos veces a `cv2.filter2D` para que haga la convolución.

```
def convolution(image, kernel_x, kernel_y, border_type = cv2.BORDER_DEFAULT):
    kernel_x = np.transpose(kernel_x)
    kernel_x = cv2.flip(kernel_x, 0)
    kernel_y = cv2.flip(kernel_y, 1)
    im_conv = cv2.filter2D(image, -1, kernel_x, borderType = border_type)
    im_conv = cv2.filter2D(im_conv, -1, kernel_y, borderType = border_type)
    return im_conv
```

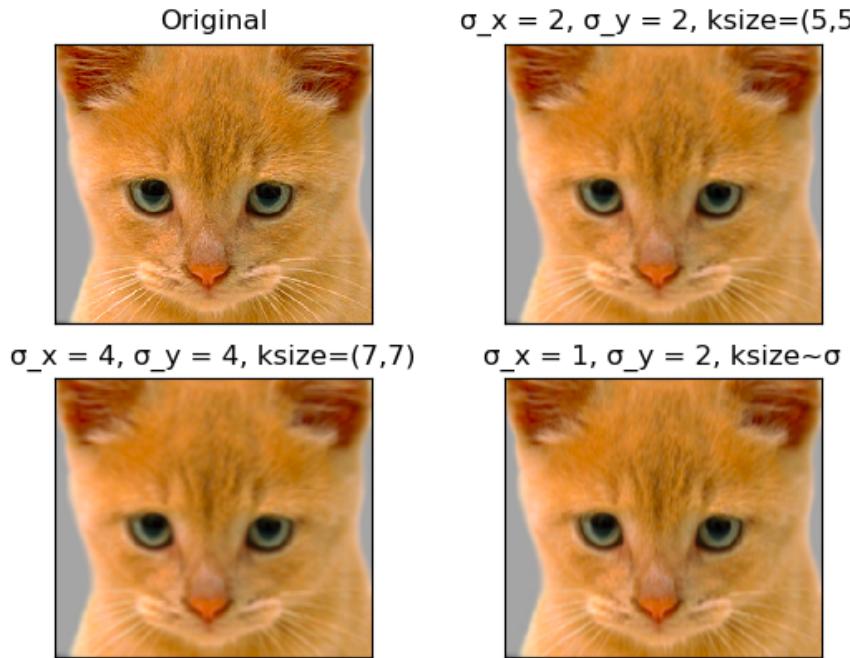
La función `gaussian_blur` ahora es sencilla de programar, simplemente obtenemos las máscaras con el σ y `ksize` deseados a través de `getGaussianKernel` y llamamos a `convolution`. Si `ksize` es 0 lo obtenemos de la ecuación $ksize = 6\sigma + 1$, del mismo modo faremos con σ_y (dirección Y), si fuese 0 o no nos dieran su valor lo faremos igual a σ_x

```
def gaussian_blur(image, sigma_x, sigma_y = 0, k_size_x = 0, k_size_y = 0,
                  border_type = cv2.BORDER_DEFAULT):
    if sigma_y == 0:
        sigma_y = sigma_x
    if k_size_x == 0:
        k_size_x = int(6*sigma_x + 1)
    if k_size_y == 0:
        k_size_y = int(6*sigma_y + 1)

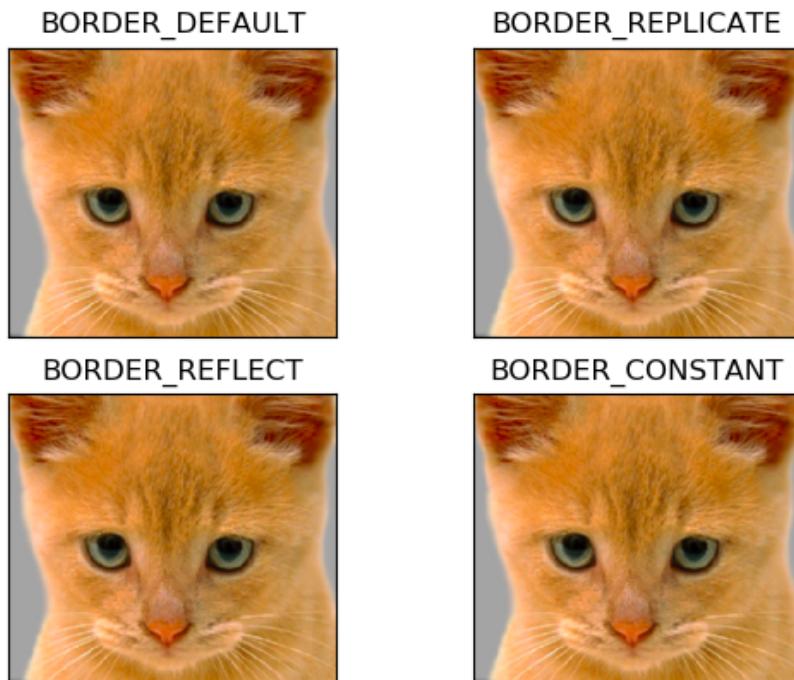
    kernel_x = cv2.getGaussianKernel(k_size_x, sigma_x)
    kernel_y = cv2.getGaussianKernel(k_size_y, sigma_y)
    return convolution(image, kernel_x, kernel_y, border_type)
```

A continuación se muestra la imagen `cat.bmp` tras hacerle un alisado gaussiano con varios valores de σ . En la segunda y tercera imagen `ksize` no está calculado a través de σ , le fuerzo a usar ese en concreto. Cuanto más grande sea el valor de σ más borroso se va a ver porque va a tener en cuenta más píxeles al hacer la media

en la convolución luego el alisado es mayor. En la última imagen sí calculo `ksize` a través de σ en cada una de las direcciones lo cual me da valores distintos.



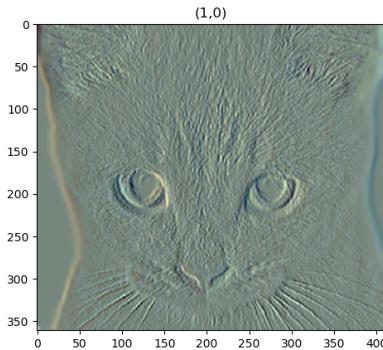
Por último se muestra un ejemplo usando diferentes bordes. He procurado no usar a lo largo de la práctica `cv2.BORDER_CONSTANT` porque hacía efectos extraños.



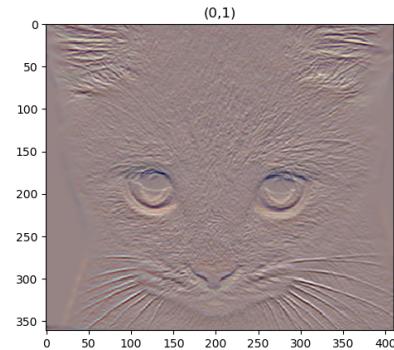
Para calcular la convolución con máscaras 1D dadas por la función `cv2.getDerivKernels` he implementado `derive_convolution`, que por supuesto usa el método `convolution` ya implementado. Veamos:

```
def derive_convolution(image, dx, dy, k_size, border_type = cv2.BORDER_DEFAULT):
    kx, ky = cv2.getDerivKernels(dx,dy,k_size)
    im_conv = convolution(image, kx, ky, border_type)
    return im_conv
```

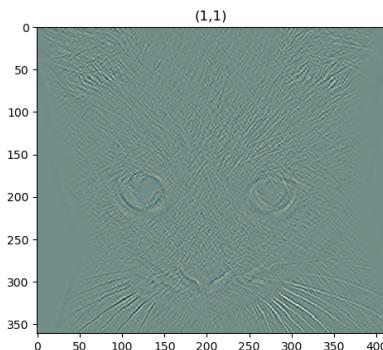
Mostramos los resultados para un `ksize` de tamaño 3 y luego lo haremos para tamaño 5.



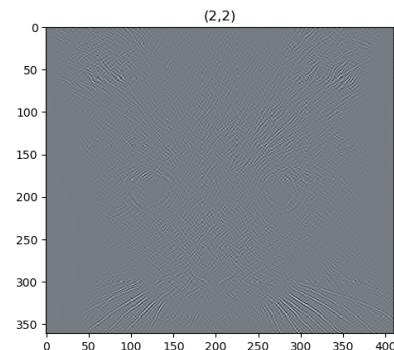
Tamaño=3, orden de derivada (1,0)



Tamaño=3, orden de derivada (0,1)



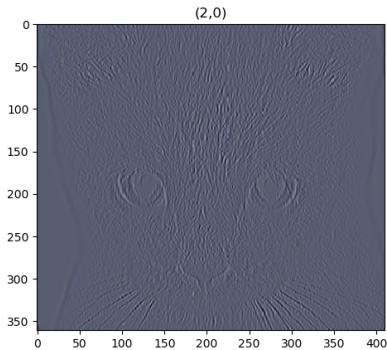
Tamaño=3, orden de derivada (1,1)



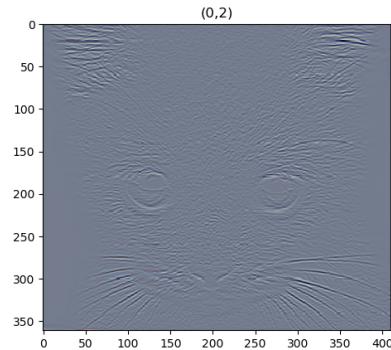
Tamaño=3, orden de derivada (2,2)

En función del orden de la derivada observamos los cambios horizontales, verticales y en el orden (1,1) la zonas de cambio que se ve como diagonales.

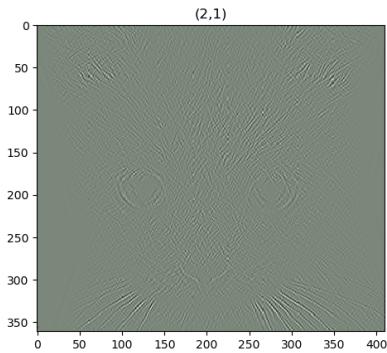
Mostramos las 4 que faltan para tamaño 3:



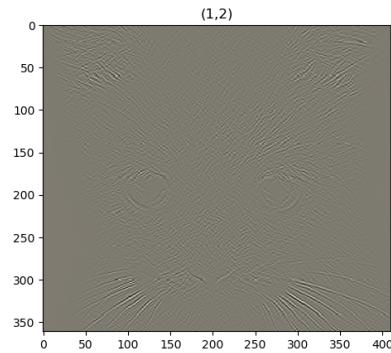
Tamaño=3, orden de derivada (2,0)



Tamaño=3, orden de derivada (0,2)

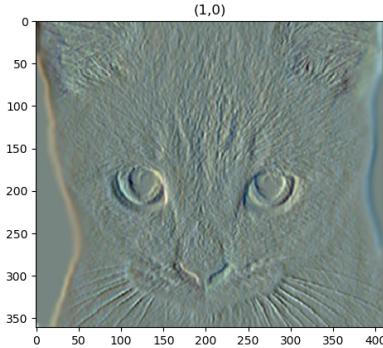


Tamaño=3, orden de derivada (2,1)

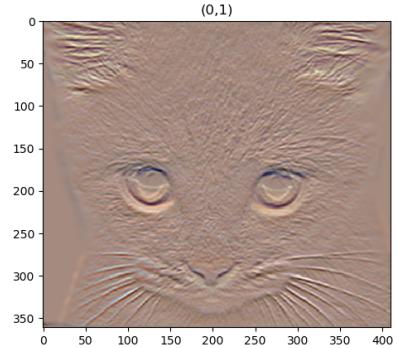


Tamaño=3, orden de derivada (1,2)

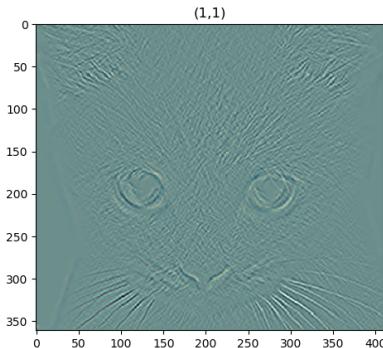
A continuación mostramos como queda la convolución con máscaras de tamaño 5 y todos los órdenes de derivadas hasta el (2, 2). Al tener un `ksize` más grande observamos que se resaltan más los bordes.



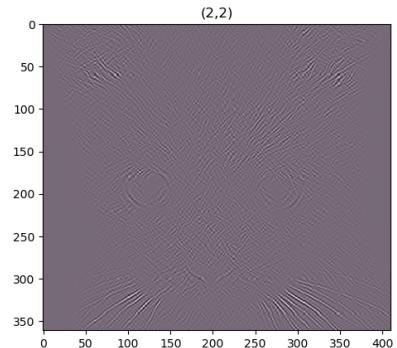
Tamaño=5, orden de derivada (1,0)



Tamaño=5, orden de derivada (0,1)

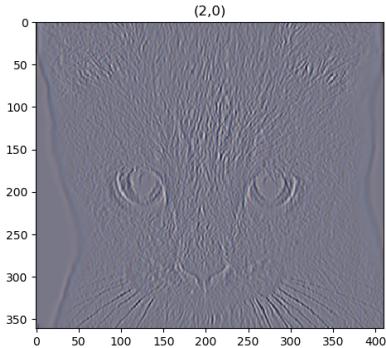


Tamaño=5, orden de derivada (1,1)

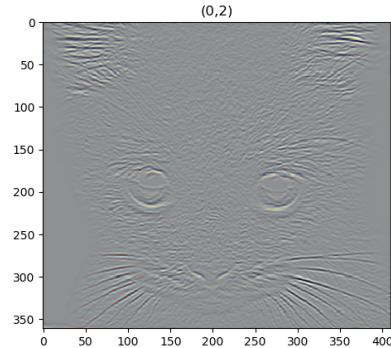


Tamaño=5, orden de derivada (2,2)

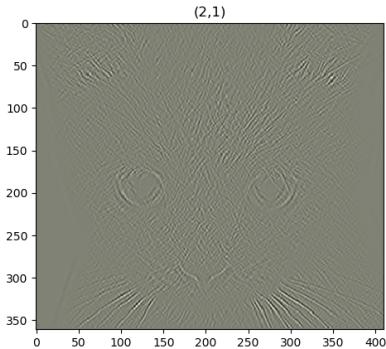
Mostramos las 4 que faltan para tamaño 5:



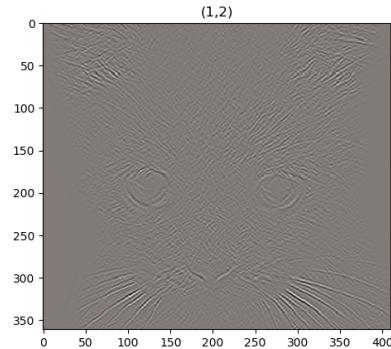
Tamaño=5, orden de derivada (2,0)



Tamaño=5, orden de derivada (0,2)



Tamaño=5, orden de derivada (2,1)



Tamaño=5, orden de derivada (1,2)

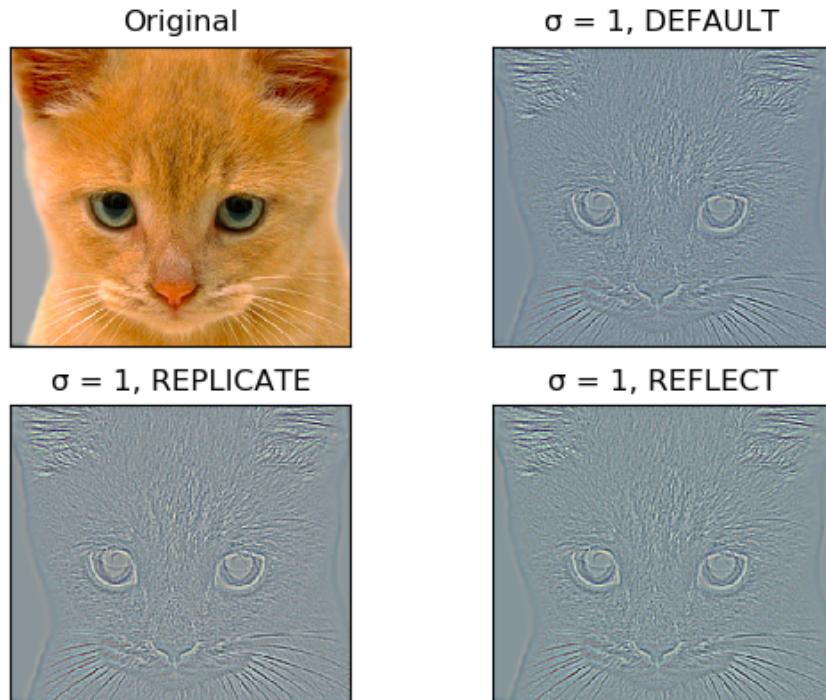
Apartado B

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

He implementado mi propia función `laplacian_gaussian` que hace un calculo similar a `cv2.Laplacian`. Hay que derivar dos veces, una con $(dx, dy) = (2, 0)$ y otra con $(dx, dy) = (0, 2)$. y guardar las máscaras. Aquí se presentan dos opciones válidas:

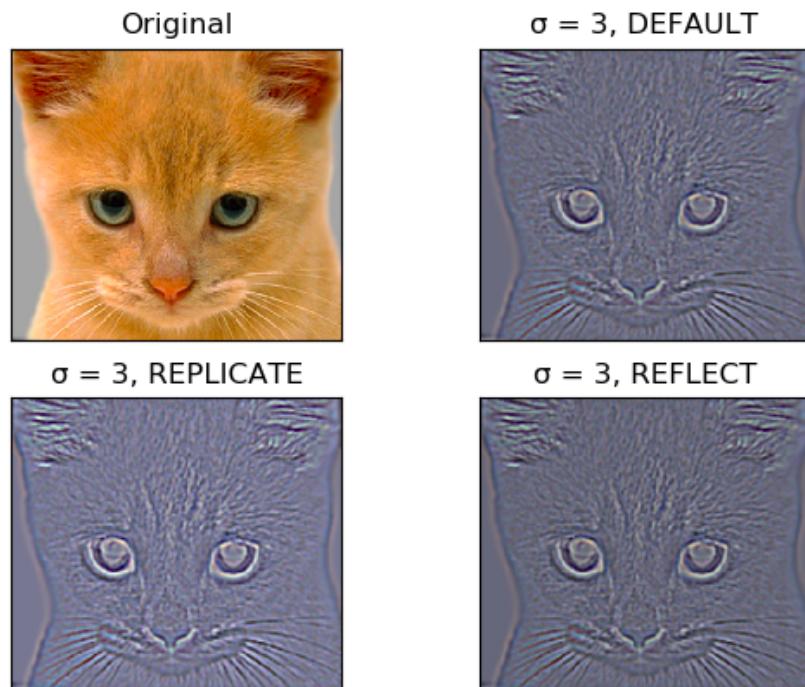
- I) Usar la máscara en la dirección X del primer `getDerivKernels` y la máscara en la dirección Y del segundo y además hacer el alisado gaussiano.
- II) Usar las 4 máscaras obtenidas y no hacer alisado gaussiano pues dos de las máscaras (una para cada dirección) se encargarían de eso. Esta es mi opción

```
def laplacian_gaussian(image, k_size, border_type = cv2.BORDER_DEFAULT):  
    k_x1, k_y1 = cv2.getDerivKernels(2, 0, k_size, normalize = True)  
    k_x2, k_y2 = cv2.getDerivKernels(0, 2, k_size, normalize = True)  
    im_convolution_x = convolution(image, k_x1, k_y1, border_type)  
    im_convolution_y = convolution(image, k_x2, k_y2, border_type)  
    return im_convolution_x + im_convolution_y
```



El operador laplaciano detecta algunos bordes ya que en los puntos donde haya cambio en la intensidad de los píxeles en una dirección el operador es positivo a un lado y negativo al otro y si no hay cambio escribiríamos 0. Conforme aumentamos el tamaño de `ksize` aumenta el grosor del borde.

La siguiente imagen es un buen ejemplo de lo anterior porque $\sigma = 3$ y `ksize = 19`. En la llamada a la función `laplacian_gaussian` multiplicamos la salida por σ^2 para normalizar en escala. En el caso anterior, $\sigma^2 = 1$ luego no teníamos que hacer nada.



Ejercicio 2

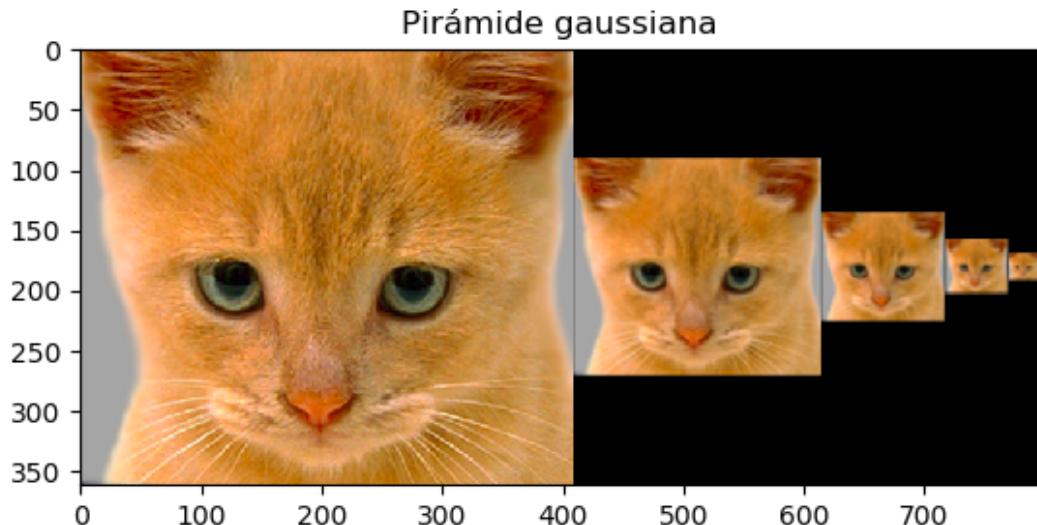
Apartado A

Una función que genere una representación en pirámide Gaussiana de 4 niveles de una imagen. Mostrar ejemplos de funcionamiento usando bordes y justificar la elección de los parámetros.

Hacer una pirámide gaussiana se basa en hacer iteraciones de `blurry + subsampling`. El primero es el alisamiento gaussiano ya programado y el segundo es un método en el que borraremos filas y columnas pares o impares para dejar la imagen en la mitad de su tamaño.

```
def gaussian_pyramid(image, levels = 4, border_type = cv2.BORDER_DEFAULT):
    pyramid = [image]
    blur = np.copy(image)
    for n in range(levels):
        blur = gaussian_blur(blur, 1, 1, 7, 7, border_type = border_type)
        blur = subsampling(blur)
        pyramid.append(blur)
    return pyramid
```

La función `cv2.pyrDown` es equivalente a lo anterior. A continuación se muestra la pirámide gaussiana calculada con 4 niveles más la imagen original.



Apartado B

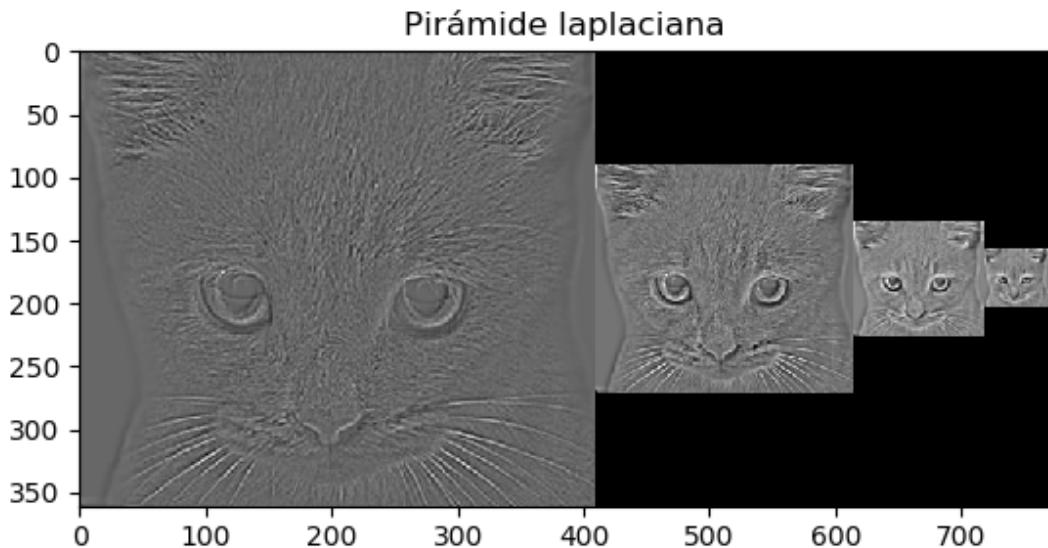
Una función que genere una representación en pirámide Laplaciana de 4 niveles de una imagen. Mostrar ejemplos de funcionamiento usando bordes.

Para hacer una pirámide laplaciana con n niveles es necesario hacer una gaussiana con $n + 1$ niveles. Restaremos a cada nivel de la pirámide gaussiana el siguiente, pero como éste tiene la mitad de tamaño, dicha resta sería imposible. Para ello, se programa una función *upsampling*. Se nos vuelven a presentar dos opciones:

- I) Rellenar las filas y columnas pares o impares con 0 y luego hacer alisado gaussiano. El resultado multiplicarlo por 4 para recuperar la intensidad o cada máscara por 2.
- II) Replicar las filas y columnas pares o impares con el valor que ya había y luego hacer alisado gaussiano. Esta es mi opción.

```
def laplacian_pyramid(image, levels = 4, border_type = cv2.BORDER_DEFAULT):  
    gau_pyr = gaussian_pyramid(image, levels+1, border_type)  
    lap_pyr = []  
    for n in range(levels):  
        gau_n_1 = upsampling(gau_pyr[n+1], gau_pyr[n].shape[0], gau_pyr[n].shape[1])  
        gau_n_1 = gaussian_blur(gau_n_1, 1, 1, 7, 7, border_type = border_type)  
        lap_pyr.append(normaliza(gau_pyr[n] - gau_n_1))  
    return lap_pyr
```

Es fundamental que los σ y tamaños de kernel usados coincidan con los de *gaussian_pyramid*. En este caso $\sigma = 1$ en ambas direcciones y *ksize* = 7. A continuación se muestra la pirámide laplaciana calculada con 4 niveles más la imagen original.



Apartado C

Construir un espacio de escalas Laplaciano para implementar la búsqueda de regiones usando el siguiente algoritmo:

- a. Fijar sigma
- b. Repetir para N escalas
 - I. Filtrar la imagen con la Laplaciana-Gaussiana normalizada en escala.
 - II. Guardar el cuadrado de la respuesta para el actual nivel del espacio de escalas.
 - III. Incrementar el valor de sigma por un coeficiente k.(1.2-1.4)
- c. Realizar supresión de no-máximos en cada escala.
- d. Mostrar las regiones encontradas en sus correspondientes escalas. Dibujar círculos con radio proporcional a la escala.

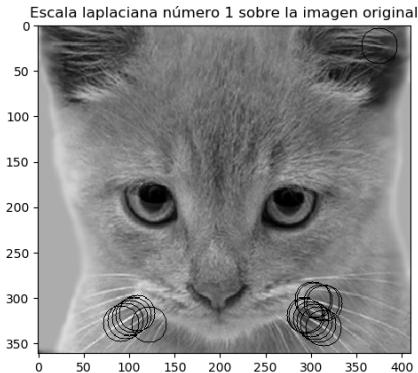
Para la realización de este ejercicio del cálculo del espacio de escalas laplaciano hay que tener en cuenta varias cosas:

- I) Vamos a tener un vector en el que guardamos las escalas. La escala 0 es la imagen original.
- II) En cada iteración actualizo σ y por tanto también `ksize`.
- III) Calculamos la laplaciana de gaussiana con ese tamaño de máscara y la multiplicamos por σ^2 .
- IV) Elevamos al cuadrado cada elemento de esa matriz. Para ello he programado una función `eleva_cuadrado`.
- V) He programado también `non_maximum_supression`, función que hace la supresión de no máximos teniendo en cuenta sólo esa escala. Es decir, para cada píxel tengo en cuenta los 8 píxeles más cercanos, calculo el máximo y veo si es más grande que el de el píxel evaluado, en cuyo caso escribiría 0 porque no es máximo y si es más pequeño escribo el valor del píxel. Para ello se puede usar una matriz con dos filas y columnas más de ceros.
- VI) Normalizamos la matriz resultante de esa escala.
- VII) Señalo las regiones obtenidas a través de la función `select_regions` (haciendo uso de `cv2.circle`) a la cual hay que pasarle la imagen original y la matriz resultante de este proceso. Para ello fijo un valor umbral a partir del cual señalaré la región. A mayor valor menos regiones y círculos voy a pintar.
- VIII) Finalmente pinto el resultado.

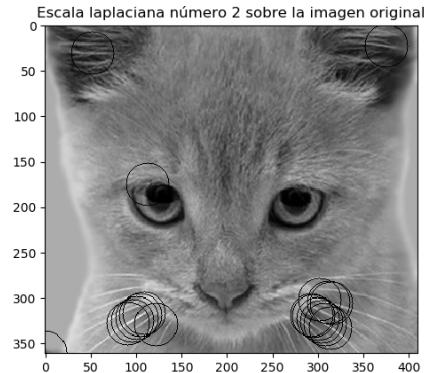
He implementado todo lo anterior para B/N y para color. Echemos un ojo al código que administra todo:

```
def ejercicio_2C(image, sigma, k, umbral = 120, levels = 4, flag_color = 1):
    if len(image.shape) == 2:
        scale = np.zeros((levels+1, image.shape[0], image.shape[1]))
        im = np.zeros((levels+1, image.shape[0], image.shape[1]))
    elif len(image.shape) == 3:
        scale = np.zeros((levels+1, image.shape[0], image.shape[1], image.shape[2]))
        im = np.zeros((levels+1, image.shape[0], image.shape[1], image.shape[2]))
    scale[0] = np.copy(image)

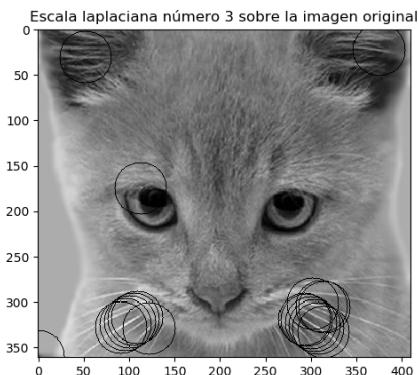
    for i in range(1, levels+1):
        k_size = int(6*sigma)+1
        if k_size % 2 == 0:
            k_size = k_size + 1
        scale[i] = sigma * sigma * laplacian_gaussian(image, k_size)
        scale[i] = eleva_cuadrado(scale[i])
        scale[i] = non_maximum_supresion(scale[i])
        sigma = k * sigma
        scale[i] = normaliza(scale[i], "Escala laplaciana número {} ".format(i))
        im[i] = np.copy(image)
        im[i] = select_regions(im[i], scale[i], umbral, int(11*sigma))
        pintaI(im[i], flag_color, "Espacio de escalas laplaciano")
```



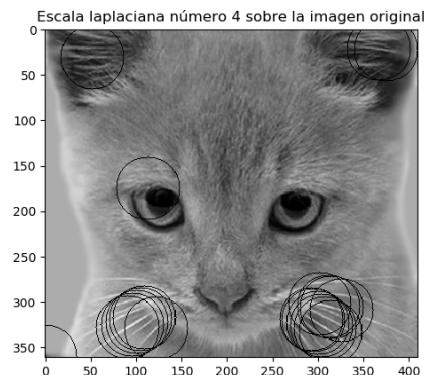
Regiones de la escala 1 (Gris)



Regiones de la escala 2 (Gris)

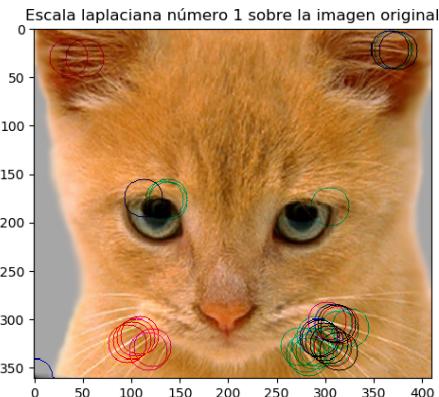


Regiones de la escala 3 (Gris)

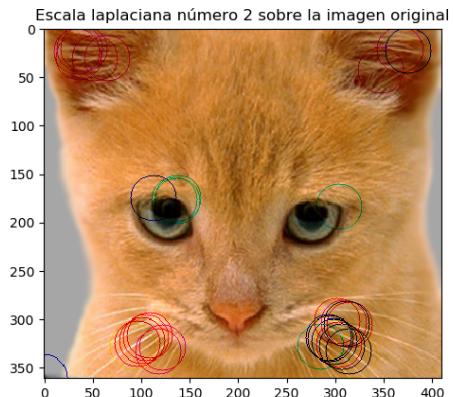


Regiones de la escala 4 (Gris)

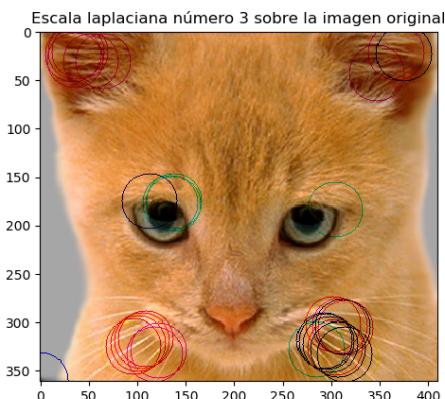
A continuación muestro los resultados que me proporciona a **color**. Me señala las regiones encontradas para cada escala y el círculo viene en el color de esa banda, es decir, Rojo, Verde o Azul. He bajado el umbral a 100 para que me detecte más zonas. Vemos que la mayoría de los círculos se agrupan en los ojos, orejas y bigotes.



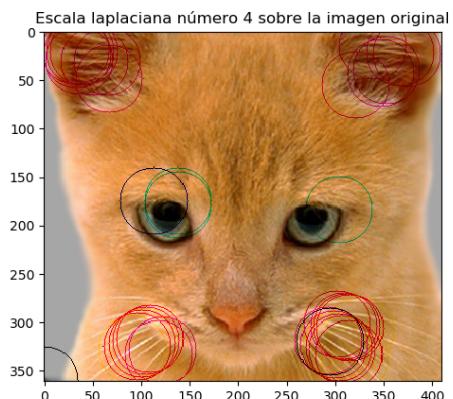
Regiones de la escala 1 (Color)



Regiones de la escala 2 (Color)

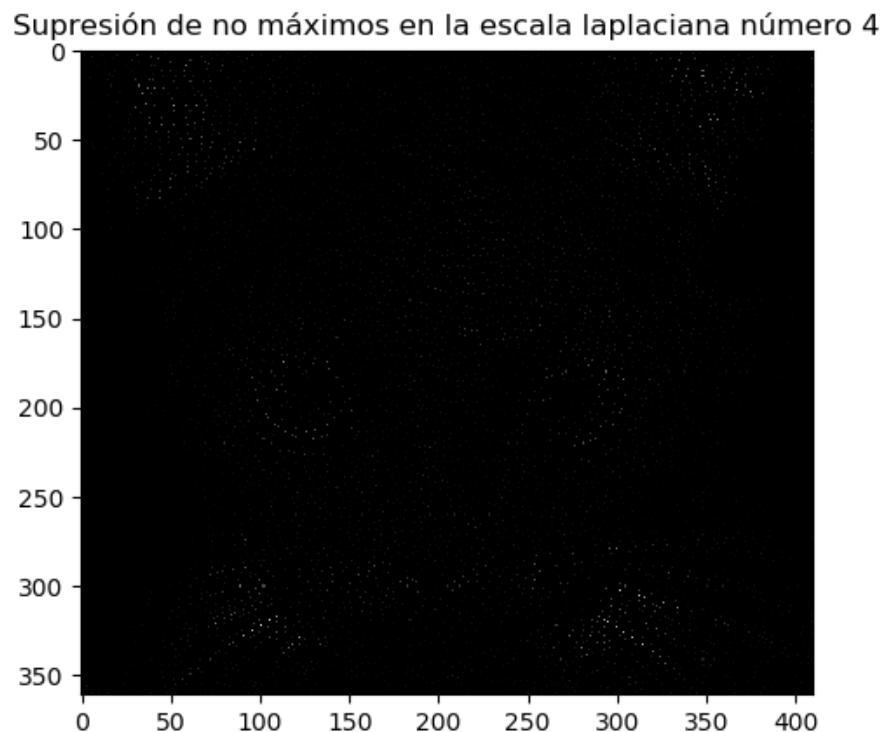


Regiones de la escala 3 (Color)



Regiones de la escala 4 (Color)

Para entender mejor lo que sucede podríamos pintar alguna de las matrices obtenidas después de la supresión de no máximos. Para finalizar este ejercicio vemos el resultado donde destacan los rasgos de las orejas, ojos y bigotes.



Ejercicio 3

Imágenes Híbridas. 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.

Apartado 1

Escribir una función que muestre las tres imágenes (alta, baja e híbrida) en una misma ventana. (Recordar que las imágenes después de una convolución contienen número flotantes que pueden ser positivos y negativos).

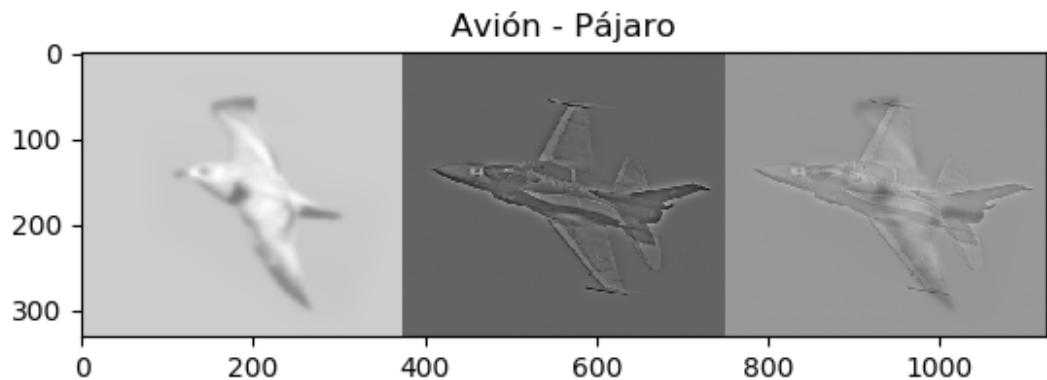
He programado una función `hybridize_images` que saca las frecuencias bajas de la primera imagen haciendo una alisado gaussiano, las altas de la segunda imagen restándole a dicha imagen un alisado gaussiano de ella e hibrida usando `cv2.addWeighted` (multiplica las dos matrices por 0,5 y las suma). Finalmente devuelve un vector con las tres imágenes. El código es sencillo:

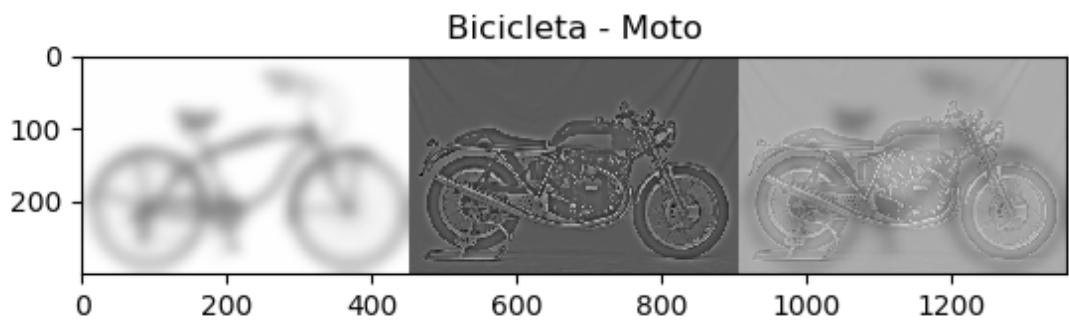
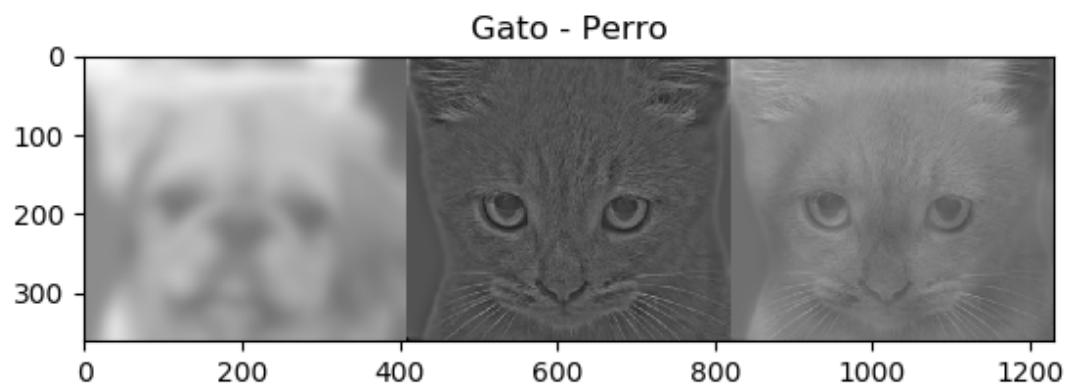
```
def hybridize_images(im1, im2, sigma1, sigma2):
    # Sacando las frecuencias a im1 usando alisado gaussiano
    freq_bajas = gaussian_blur(im1, sigma1, sigma1)
    # Sacando las frecuencias altas a im2 restando alisado gaussiano
    freq_altas = cv2.subtract(im2, gaussian_blur(im2, sigma2, sigma2))
    return [freq_bajas, freq_altas, cv2.addWeighted(freq_bajas, 0.5, freq_altas, 0.5, 0)]
```

Apartado 2

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

He elegido las parejas *Avión - Pájaro*, *Gato - Perro* y *Bicicleta - Moto*:



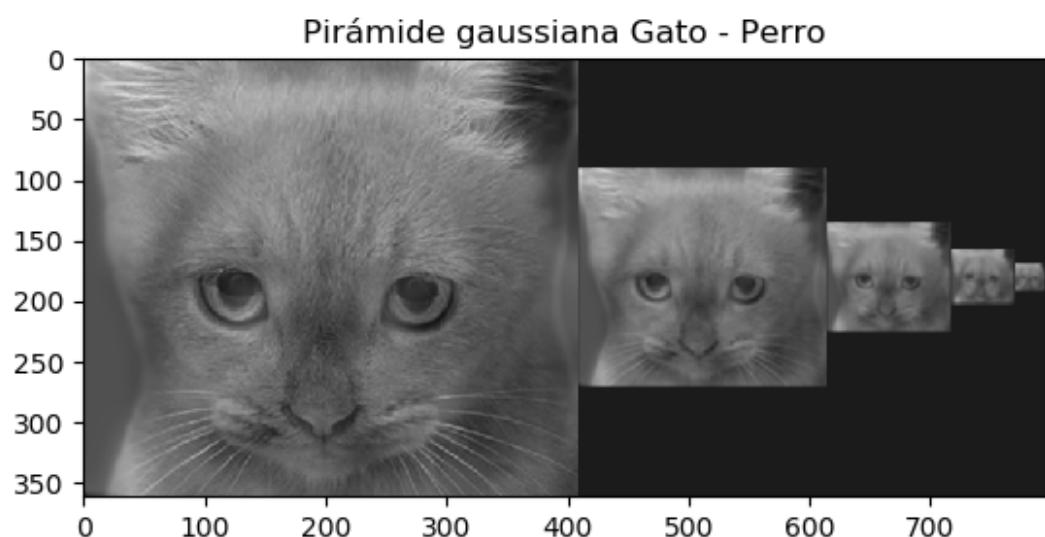
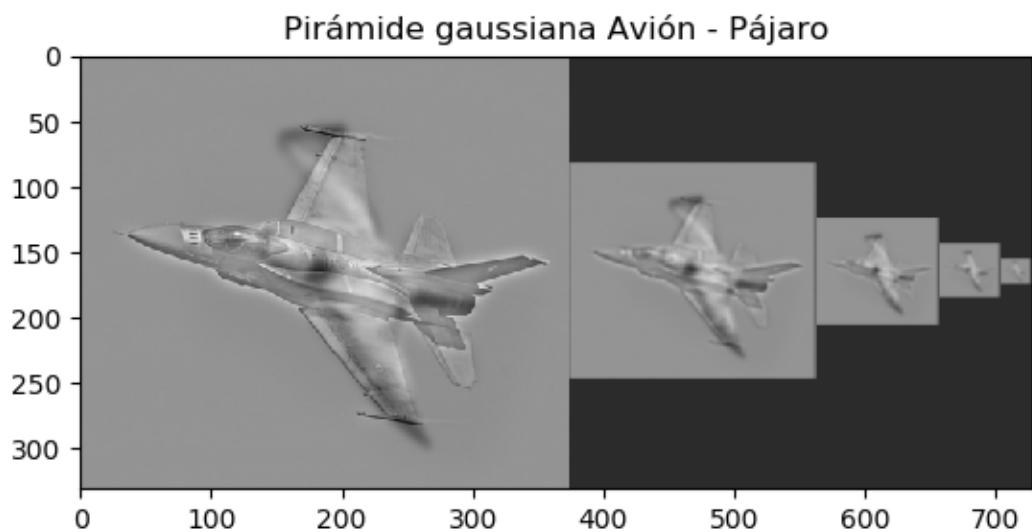


Es claro que si miramos de cerca vemos el *Avión*, *Gato* y *Moto* respectivamente y si miramos de lejos vemos *Pájaro*, *Perro* y *Bicicleta*. En el apartado siguiente podremos apreciar esto mejor. Para cada imagen he elegido valor de σ , es decir, al llamar a `hybridize_images` pasamos como parámetro las dos imágenes y dos valores de σ .

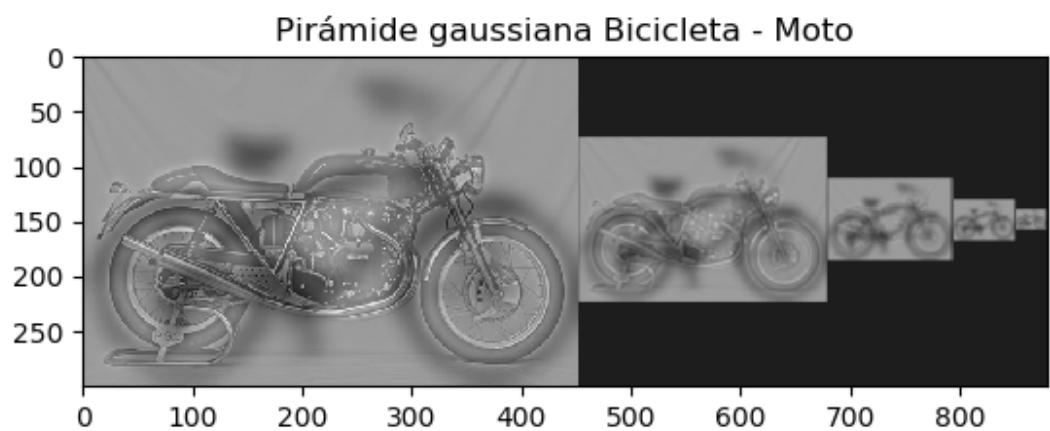
Apartado 3

Construir pirámides gaussianas de al menos 4 niveles con las imágenes resultado. Explicar el efecto que se observa.

Usando lo implementado en el apartado 2B para pirámides gaussianas y la imagen híbrida del apartado anterior construimos las pirámides gaussianas. Muestro directamente los resultados:



Si procedemos a valorar los resultados vemos que conforme nos movemos a la derecha en dichas imágenes y avanzamos por tanto en la pirámide gaussiana pasamos de ver una imagen a otra. Es decir, en la más grande vemos el *Avión*, *Gato* y *Moto* y



en la última el *Pájaro*, *Perro* y *Bibicleta*. Más aún creo que de la tercera en adelante ya vemos las segundas.

Bonus 1

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

He programado una función `convolution2D` sabiendo que la máscara es separable. No he programado ninguna función que dada una máscara 2D separable me devuelva las máscaras 1D que la componen (serían vector columna por vector fila para obtener la máscara 2D).

```
def convolution2D(image, kernel_x, kernel_y):
    salida = np.copy(image)
    kernel_x = cv2.flip(kernel_x, -1)
    kernel_y = cv2.flip(kernel_y, -1)

    salida = correlation1D(salida, kernel_x)
    salida = np.transpose(salida)
    salida = correlation1D(salida, kernel_y)
    salida = np.transpose(salida)
    salida = normaliza(salida, "Convolucion2D")

return salida
```

El otro método relevante es `correlation1D` que pasa un máscara por filas y hace la correlación. En los bordes he puesto ceros pero eso se puede cambiar y hacer `REPLICATE` o `REFLECT`.

Para facilitarme un poco la vida el tamaño de la máscara es impar luego lo puedo escribir como $2k+1$, $k \in \mathbb{N}$ y meto la matriz que me dan en una que tiene k columnas más a cada lado, en total $2k$ columnas extra. Pongo el borde que quiero y me pongo a correlar teniendo en cuenta que he hecho una traslación de k columnas.

```
def correlation1D(image, kernel):
    mitad = int(len(kernel)/2)
    salida = np.zeros(image.shape)

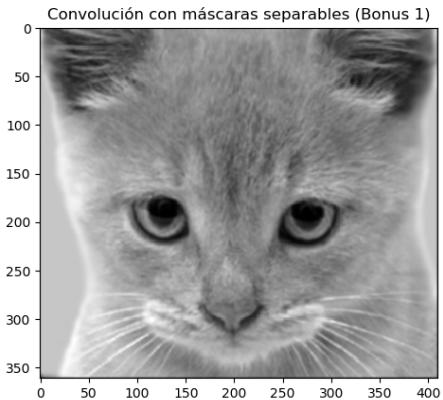
    if len(image.shape) == 2:
        im = np.zeros((image.shape[0], image.shape[1] + 2*mitad))
        im[:, mitad:im.shape[1]-mitad] = image

        for i in range(0, image.shape[0]):
            for j in range(0, image.shape[1]):
                for n in range(-mitad, mitad+1):
                    salida[i][j] += im[i][j+mitad+n] * kernel[n+mitad]

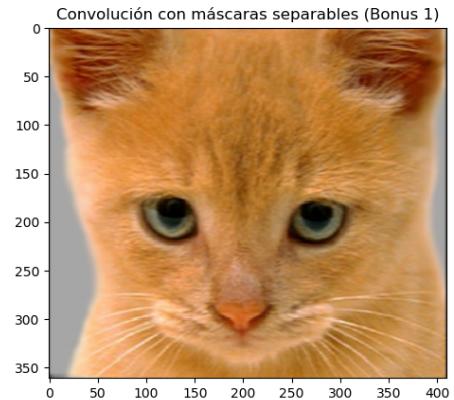
    elif len(image.shape) == 3:
        # similar a lo anterior con las tres bandas

return salida
```

He probado con varias máscaras gaussianas:

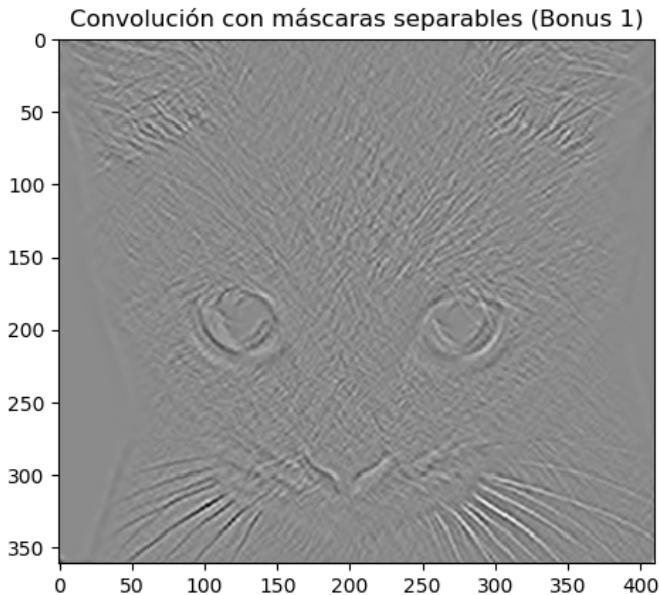


Convolución 2D máscara gaussiana (Gris)



Convolución 2D máscara gaussiana (Color)

Por último pruebon con una máscara dada por `cv2.getDerivKernels` con $(dx, dy) = (1, 1)$ y `ksize = 7`.

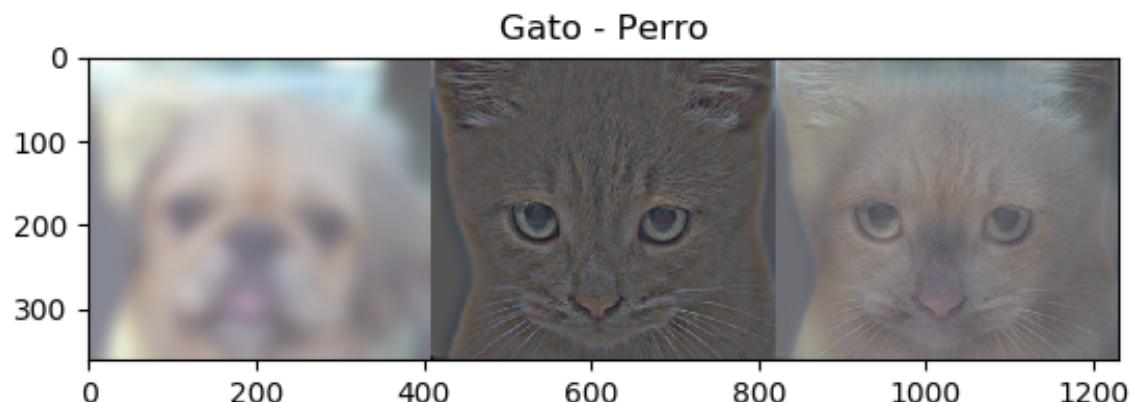
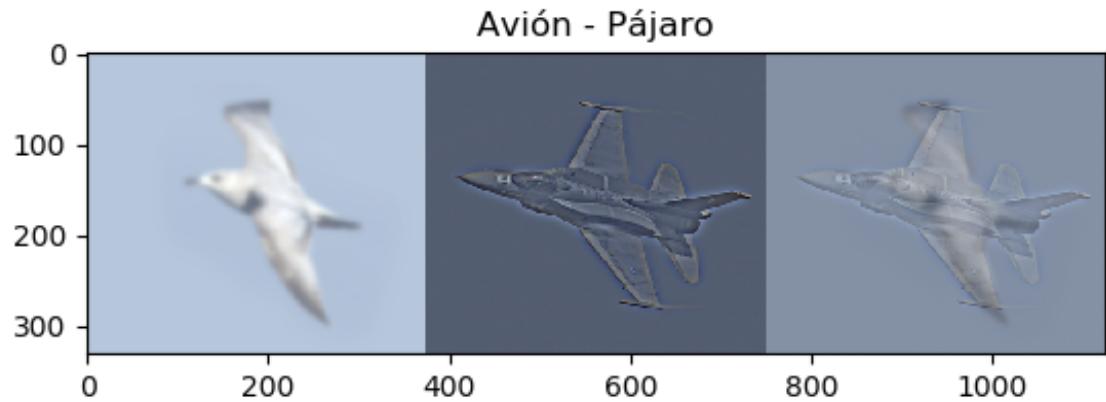


Convolución 2D máscara de derivada

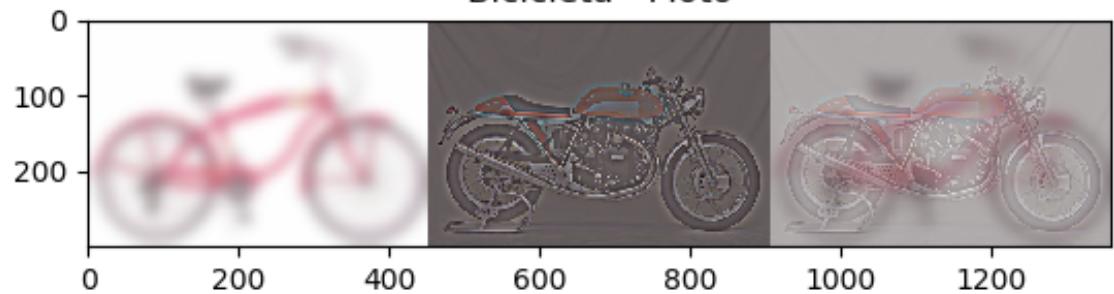
Bonus 2

Realizar todas las parejas de imágenes híbridas en su formato a color

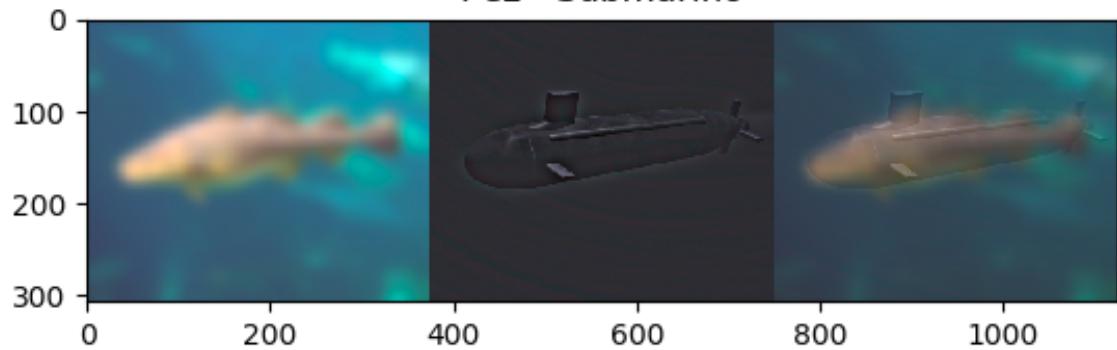
Tenemos programado todo lo necesario así que nos limitamos a leer todas las imágenes a color, hibridar e imprimir los resultados en pantalla. Obtenemos lo esperado.



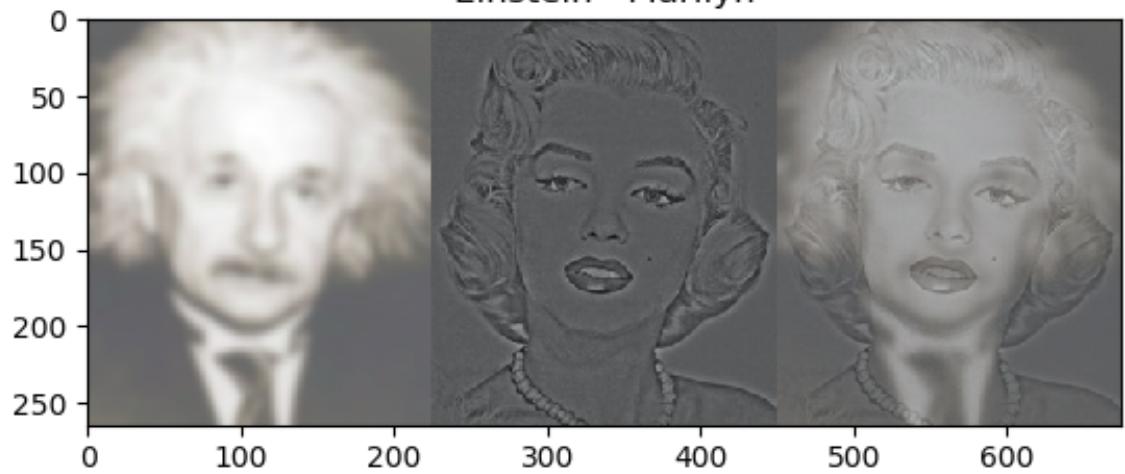
Bicicleta - Moto



Pez - Submarino



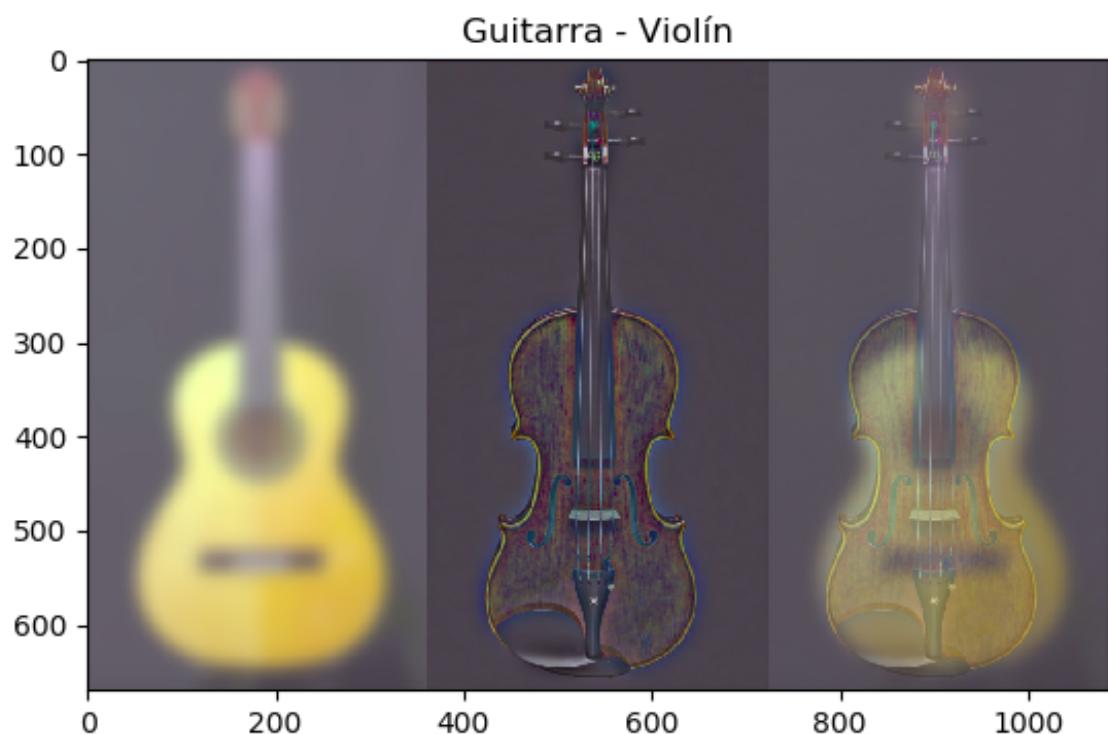
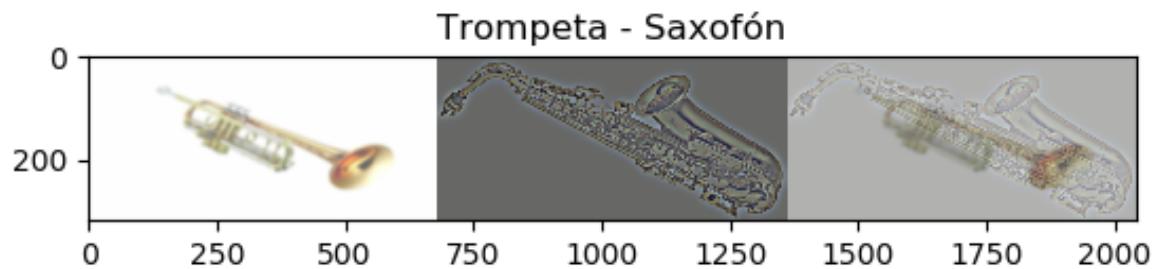
Einstein - Marilyn



Bonus 3

Realizar una imagen híbrida con al menos una pareja de imágenes de su elección que hayan sido extraídas de imágenes más grandes. Justifique la elección y todos los pasos que realiza.

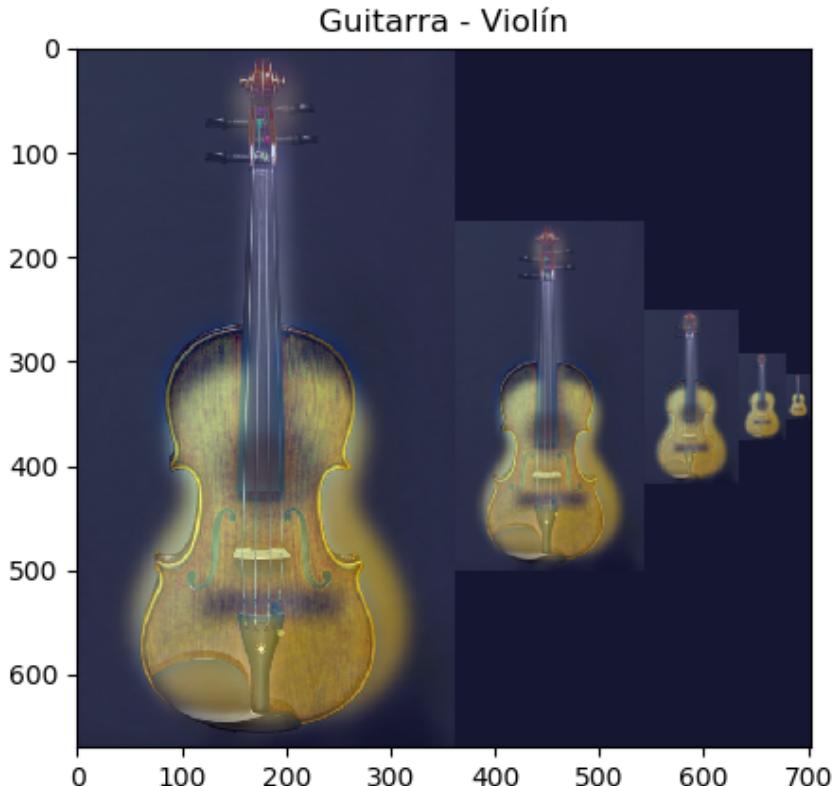
Mi idea ha sido hibridar instrumentos musicales. Empecé probando con un saxofón y una trompeta pero como las dimensiones no coincidían muy bien busqué otra mejor. Esta nueva idea fue hibridar un violín y una guitarra. Y los resultados son más satisfactorios.



Es fundamental que las dos imágenes tengan el mismo tamaño. Yo las recorté

lo más parecidas posibles y que además la guitarra y el vioín se superpusieran lo mejor posible. Finalmente hago un `cv2.resize` a los mínimos de ancho y alto, lo cual va a variar muy poco la imagen ya que tenían casi el mismo número de píxeles de partida. Muestro el código:

```
def bonus_3(im_1, im_2, sigma_1, sigma_2, flag_color = 1, image_title = "Imagen"):
    # Calculo ímnimos de ancho y alto
    min_alt = min(im_1.shape[0], im_2.shape[0])
    # Hago resize a los ímnimos de ambas áimenes.
    min_anc = min(im_1.shape[1], im_2.shape[1])
    im_1 = cv2.resize(im_1, (min_anc, min_alt), im_1, interpolation = cv2.INTER_CUBIC)
    im_2 = cv2.resize(im_2, (min_anc, min_alt), im_2, interpolation = cv2.INTER_CUBIC)
    # Hibrido y muestro las áimenes
    vim = hybridize_images(im_1, im_2, sigma_1, sigma_2)
    muestraMI(vim, flag_color, image_title) # Mostramos la óhbridacin
    gau_pyr = gaussian_pyramid(vim[2], 4) # Construimos las ápirmides gaussianas
    muestraMI(gau_pyr, 0, image_title) # Imprimimos las ápirmides gaussianas
```



He construído también la pirámide gaussiana de la imagen hibridada, donde en la primera imagen vemos el violín y a partir de la tercera ya sólo la guitarra.