Visión por Computador

Trabajo 1

Curso 2017/2018

Francisco Javier Caracuel Beltrán

caracuel@correo.ugr.es

4º - Grado en Ingeniería Informática – CCIA – ETSIIT

Índice

0. Consideraciones previas:

Para el desarrollo de este trabajo se han utilizado los módulos de Python siguientes:

- *cv2*.
- *numpy*.
- *math*.
- *copy*.
- *matplotlib.pyplot*.

Existe una serie de parámetros generales para todo el trabajo:

- *path*: indica la ruta relativa desde el fichero *trabajo1.py*, donde se encuentran las imágenes.
- *continue_text*: mensaje que se muestra para continuar en cada sección.
- *num_cols*: permite especificar el número de columnas que se desea para mostrar las imágenes en cada apartado. Utilizado en el punto 1.A.
- *cmap*: permite modificar el esquema de color utilizado en las imágenes. Por defecto, está establecido en gris (cv2.COLOR_RGB2GRAY).
- *plt.rcParams['image.cmap']*: le indica a plt como debe mostrar las imágenes por pantalla. Por defecto, en gris.

Se han definido tres funciones de ámbito general:

- *set_c_map(imgs)*: recibe una lista de imágenes y las convierte al valor establecido en cmap (escala de grises).
- *power_two(n)*: calcula el logaritmo en base 2 de un número (n), truncando los decimales y devolviendo un número entero.
- *next_power_two(n)*: dado un número (n), calcula el siguiente número mayor que sea potencia de 2.

La estructura del fichero principal (*trabajo1.py*) se puede dividir en dos partes. La primera parte contiene todas las funciones requeridas para realizar el trabajo y la segunda las llamadas a las funciones para ejecutarlas.

La segunda parte viene englobada dentro de *if __name__ == "__main__":* y, al comienzo de esta parte, se permite la inclusión de los apartados que se deseen ejecutar. Por defecto, se ejecutan todos.

Antes de comenzar los cálculos se cargan 6 imágenes que se utilizarán en todo el trabajo. Para su correcto procesamiento se convierten a escala de grises y su representación interna como datos flotantes.

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

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

Se crea una función con la siguiente cabecera:

show_images(imgs, names = list(), cols = num_cols, title = "")

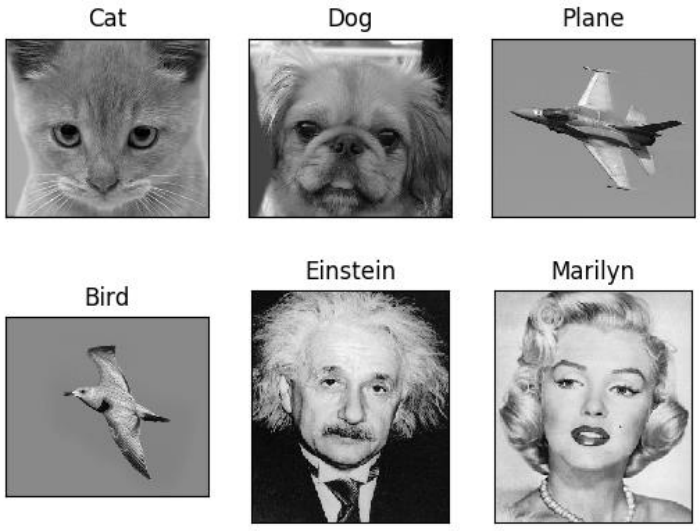
Permite recibir una lista de imágenes (*imgs*), una lista con los títulos de cada imagen (*names*), el número de columnas que se quiere mostrar (*num_cols*) y un prefijo que acompaña a todos los títulos (*title*).

La función muestra por pantalla una tabla con el número de columnas especificado y el número de filas que sean necesarias para que se puedan albergar todas las imágenes.

Se recorre en un bucle cada imagen de la lista y en cada iteración:

- Se cambia la representación interna de la imagen de números reales a enteros.
- Se indica el número de filas y columnas de la tabla y el lugar que ocupa la imagen actual en dicha tabla.
- Se añade el título.
- Se eliminan las marcas de los ejes de coordenadas.
- Se muestra la imagen.

El resultado de ejecutar esta función es:



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

Se crea una función con la siguiente cabecera:

```
convolution_b(img, sigma, mask = -1, border = -1)
```

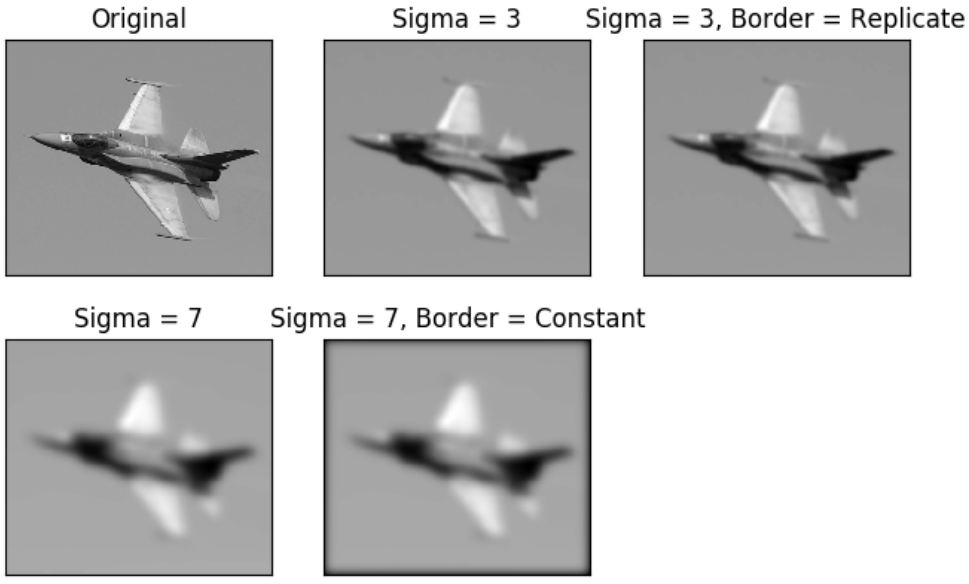
Dada una imagen (*img*), realiza una convolución con máscara gaussiana. El tamaño de la máscara (*mask*) por defecto se calcula a partir del sigma (*sigma*) recibido, aunque tiene como opción especificar otro tamaño de máscara diferente. Si no se especifica tamaño de máscara, se calcula como 6*sigma+1, debido a que el tamaño más óptimo estudiado es crear un intervalo de 3*sigma por cada lado. Esto hace que se tenga 6*sigma y como se debe tener en cuenta el punto intermedio, el resultado final es 6*sigma+1.

Se permite especificar un borde de entre los disponibles en OpenCV. Los bordes que se pueden indicar en el resto de funciones del trabajo son los mismos que se enumeran a continuación, por lo que no se comentarán de nuevo: cv2.BORDER_REPLICATE, cv2.BORDER_REFLECT, cv2.BORDER_REFLECT_101, cv2.BORDER_WRAP, cv2.BORDER_CONSTANT.

Internamente, la función comprueba si *mask* es igual a -1. En este caso, le asigna el valor correspondiente a 6*sigma+1. A continuación, se utiliza la función *cv2.GaussianBlur()*, indicándole la imagen recibida, el tamaño de máscara calculado y el sigma que se desea utilizar.

En el siguiente apartado se implementa manualmente lo que realiza la función *cv2.GaussianBlur()*, por lo que los cálculos internos se explican en él.

El resultado de ejecutar esta función es:



Se puede apreciar como al aumentar *sigma*, la imagen aparece más desenfocada. Esto es debido a que el núcleo aumenta su tamaño y al hacer convolución con la imagen afecta a un mayor número de píxeles, produciendo así un alisamiento mayor (debido también a las características del núcleo que genera la función de OpenCV).

Los bordes solo se aprecian si son constantes. Esto se produce por el tipo de imagen que se está utilizando y el tamaño de la misma. En el borde de tipo *Replicate*, como todo el contorno es similar, se rellena con los mismos píxeles de su extremo y visualmente no se puede observar. El borde *Constant* si añade un borde de *n* píxeles de color negro, por lo que es el único que permite visualizarlo a simple vista. El motivo de añadir bordes al realizar la convolución se explica en el Bonus 2.

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

En este apartado se implementa manualmente lo que realiza *cv2.GaussianBlur()*. La cabecera de esta función es:

```
convolution_c(img , kernel_x = None, kernel_y = None, sigma = 0, border =
cv2.BORDER_DEFAULT, normalize = True, own = False)
```

Dada una imagen (*img*) y dos núcleos (*kernel_x, kernel_y*), realiza una convolución de la imagen utilizando dichos núcleos. En este apartado, *kernel_x* y *kernel_y* es el mismo (el motivo de separarlos es para reutilizar todo el código posible durante el trabajo) y en esta explicación se entenderán como *kernel*.

La imagen que se recibe es 2D y el kernel que se tiene es 1D. La manera de poder hacer convolución con este kernel es a su vez, hacer convolución por filas y con ese resultado, hacer convolución por columnas.

Para hacer convolución se utiliza la función *cv2.filter2D()*, a la que se le envía un vector y un kernel y devuelve un vector del mismo tamaño que el enviado con la convolución realizada.

En este punto el proceso es sencillo. Se deben recorrer todas las filas de la imagen, enviando cada una de ellas junto con el kernel a *cv2.filter2D()* y reemplazarla por el resultado que ofrece la función. Cuando se terminen las filas, se repite el mismo proceso con las columnas y se obtiene la imagen convolucionada.

La explicación más detallada de cómo se hace convolución con un vector 1D y un kernel 1D se encuentra en el Bonus 2.

Un aspecto a tener en cuenta son los valores que se obtienen al hacer la convolución. Estos pueden superar el intervalo [0, 255] que para visualizar la imagen en cualquier pantalla es necesario. Para no tener este problema, siempre que se realiza la convolución, se normalizan los resultados entre dicho intervalo, asegurando que no hay números negativos ni números superiores a 255.

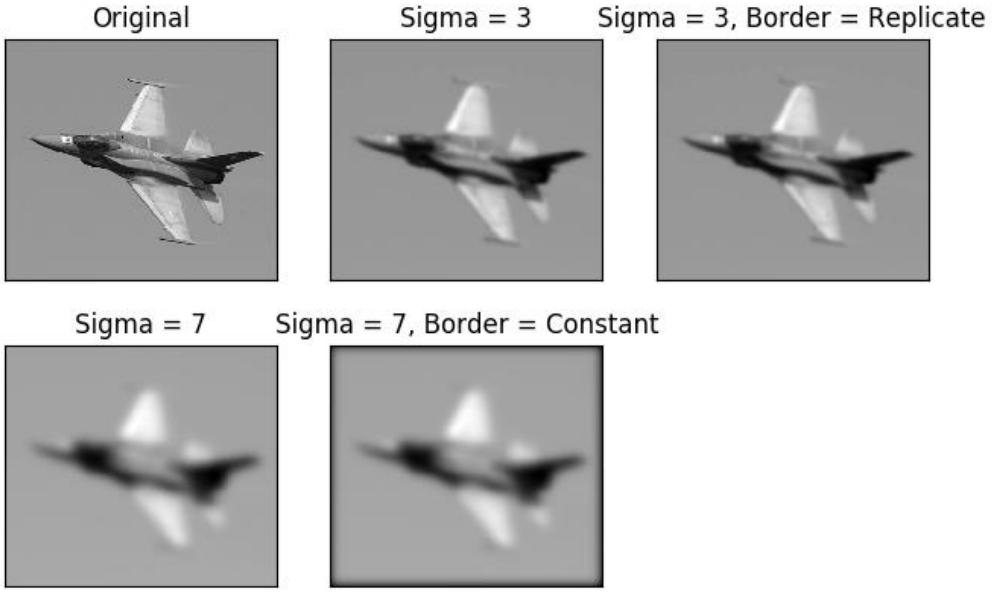
Como aspectos concretos de la implementación realizada, no es necesario especificar ningún kernel para hacer la convolución. En este caso, se genera un kernel gaussiano de un sigma recibido por parámetro y tamaño de máscara 6*sigma+1, utilizando la función *cv2.getGaussianKernel()*.

Como la imagen recibida es en escala de grises, no se debe tener en cuenta la separación de canales para hacer la convolución canal a canal.

La función *cv2.filter2D()* permite aplicar directamente un borde, por lo que se le envía el recibido por parámetro.

Como extensión final, se puede enviar como *True* el parámetro *own* para que haga uso de la función de convolución manual implementada en el Bonus 2.

El resultado de ejecutar esta función es:



Se puede ver gráficamente como el resultado es similar al que hace *cv2.GaussianBlur()*, aunque internamente pueden variar ciertos ajustes que lo hagan más óptimo o mejoren detalles. En definitiva, se comprueba que la función de OpenCV aplica una convolución por filas y columnas de una imagen 2D con un kernel gaussiano 1D.

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

Se crea una función con la siguiente cabecera:

```
convolution_d(img , kernel_x = None, kernel_y = None, ksize = 3, sigma = 0, border = cv2.BORDER_DEFAULT, dx = 1, dy = 1, own = False)
```

Esta función recibe una imagen (*img*) y le aplica la convolución con respecto a x con un kernel (*kernel_x*) y con respecto a y (*kernel_y*). Esto quiere decir que esta función devuelve dos imágenes convolucionadas y cada una es resultado de aplicarle su correspondiente kernel.

En su implementación hace uso de la función del apartado anterior *convolution_c()*.

Los bordes son variaciones fuertes de la intensidad que corresponden a las fronteras de las imágenes. Los máximos de la primera derivada permiten detectar estos bordes al detectar, a su vez, la variación. El máximo del valor absoluto coincide con el punto central del borde. Este hecho es lo que permite obtener todas las fronteras de una imagen.

Para evitar la sobredetectación de los bordes es necesario aplicar, antes de realizar la convolución, un alisamiento de la imagen.

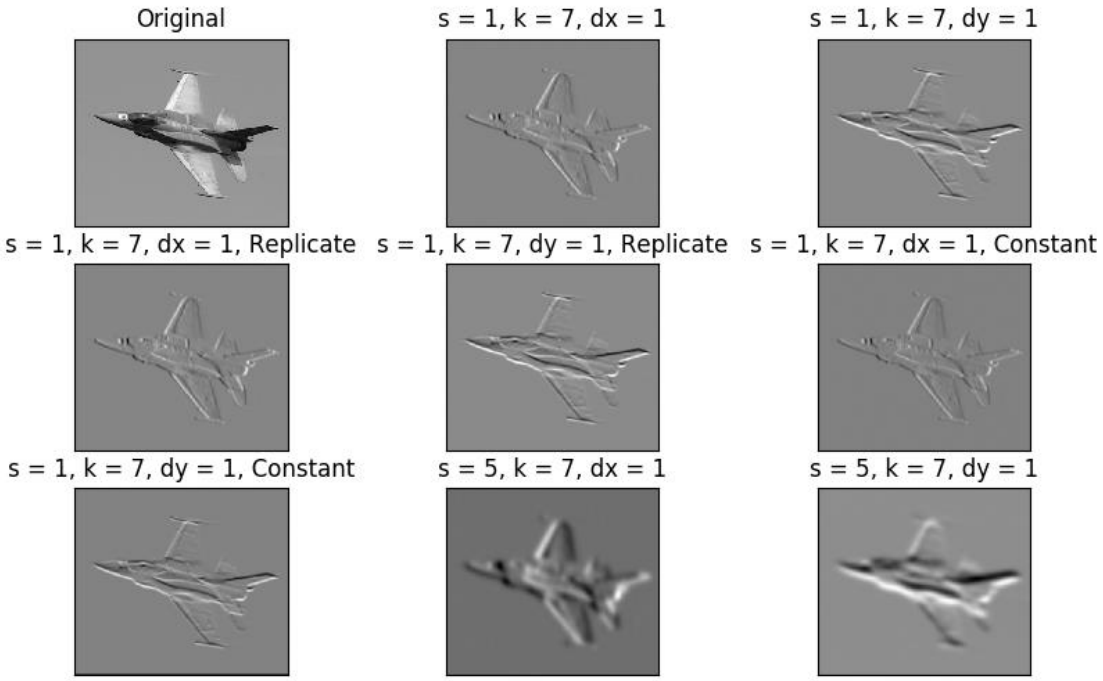
La función de este apartado recibe una imagen y un sigma dado. Antes de realizar cualquier operación, para evitar el ruido, le aplica el alisamiento utilizando dicho sigma y llamando a la función creada en el apartado C *convolution_c()* (como no se le envía ningún kernel, se generará un kernel gaussiano del sigma enviado).

El siguiente paso es generar un kernel de primera derivada. OpenCV dispone de la función *cv2.getDerivKernels()* en el que se le envía que orden de derivada con respecto a x y con respecto a y se quiere y un tamaño de máscara dado. El resultado son los dos vectores de la primera derivada con respecto a x y con respecto a y.

Utilzando *convolution_c()*, se hace la convolución de la imagen con el kernel de primera derivada con respecto a x y con respecto a y. Esta función ya normalizaba, por lo que no es necesario volver a hacerlo.

Finalmente se devuelven las dos imágenes resultado de aplicar los cálculos anteriores.

El resultado de ejecutar esta función es:



Al igual que en los apartados anteriores, los bordes exteriores que se añaden tienen poco efecto en las imágenes.

La utilización de un sigma cada vez más grande para alisar y evitar ruido hace que los bordes que se detectan sean de mayor tamaño. La utilización de un sigma u otro depende de cada imagen. Si no es necesario alisar porque la operación no genere mucho ruido, se obtendrán mejores resultados.

Cuando se aplica un kernel de primera derivada con respecto a x, los bordes que se detectan son los verticales, siendo los horizontales los que se detectan cuando se utiliza la primera derivada con respecto a y.

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

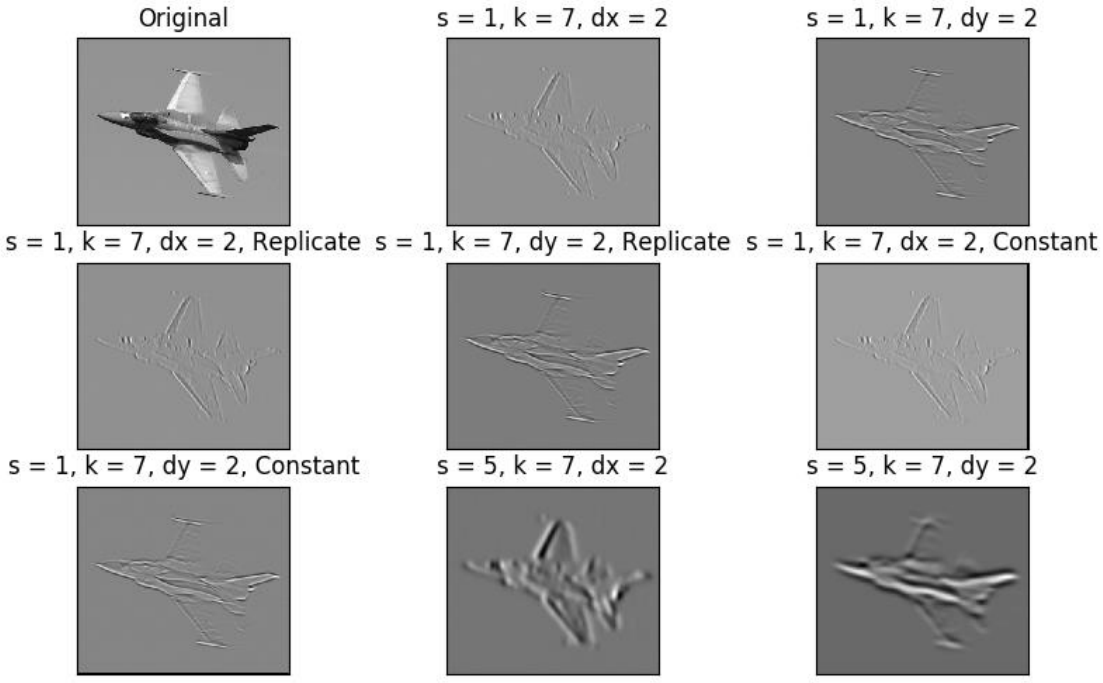
Se crea una función con la siguiente cabecera:

```
convolution_e(img , kernel_x = None, kernel_y = None, ksize = 3, sigma = 0, border = cv2.BORDER_DEFAULT, dx = 2, dy = 2, own = False)
```

La función de este apartado llama directamente a la función del apartado anterior *convolution_d()*, ya que el proceso es el mismo. La única diferencia es que cuando se llama se envía como parámetro dx = 2 y dy = 2 para indicar que se va a realizar la segunda derivada.

La diferencia de la segunda derivada con respecto a la primera es que la segunda derivada calcula los cruces por cero para detectar los bordes. Detecta los cambios en la pendiente y los cambios de la primera derivada. Los pasos por cero coinciden con el centro del borde.

El resultado de ejecutar esta función es:



Los cambios en los bordes son más sutiles que en la primera derivada al quedarse con los cruces en los ceros. Al igual que en el apartado anterior, un sigma mayor que alise más provoca que los bordes sean más notables.

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

Se crea una función con la siguiente cabecera:

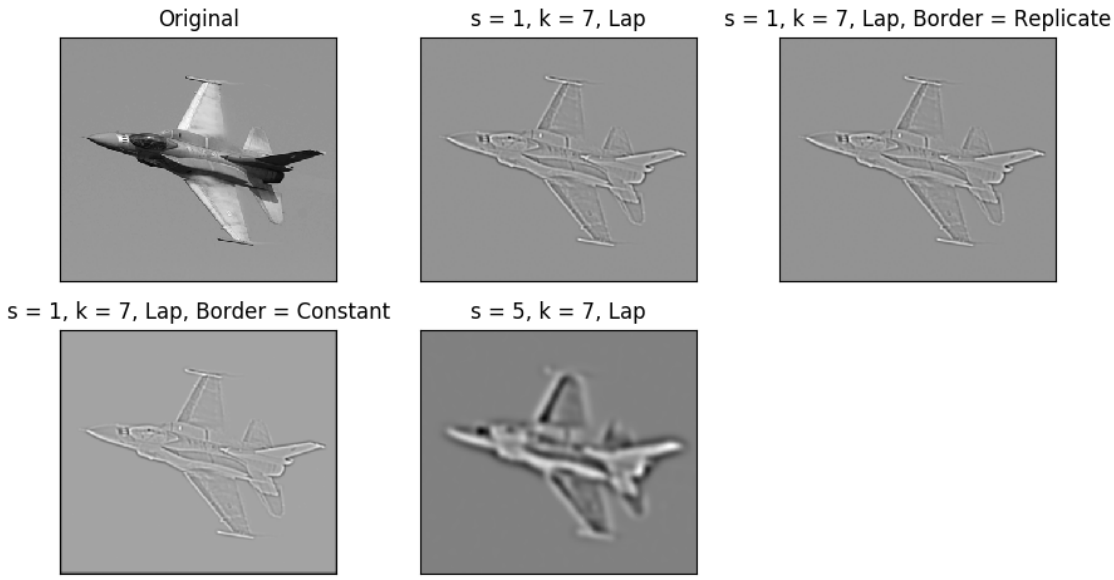
convolution_f(img , kernel_x = None, kernel_y = None, ksize = 3, sigma = 0, border = cv2.BORDER_DEFAULT, dx = 2, dy = 2, own = False)

La Laplaciana Gaussiana es un proceso por el que primero se suaviza mediante la convolución con una gaussiana y posteriormente se hace convolución con un núcleo de segunda derivada con respecto a x y con respecto a y. El resultado final será la suma de ambas convoluciones, es decir, de ambos núcleos.

Gracias a la propiedad distributiva que tiene esta operación, se puede calcular primero las derivadas segundas con la gaussiana y después hacer convolución de ese resultado con la imagen.

En esta función se ha optado por la primera opción, de manera que aprovechando la función creada en el apartado anterior *convolution_e()*, lo que devuelve la función de este apartado es la suma de las dos imágenes que devuelve la función *convolution_e()*, es decir, devuelve la suma de las imágenes al realizar una convolución con un núcleo de segunda derivada tras aplicar un alisamiento.

El resultado de ejecutar esta función es:



Realiza una detección más nítida de todos los bordes de la imagen ya que las respuestas que genera son positivas, negativas o nulas. La posición de los bordes es justo la posición donde se produce el cruce por cero.

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

Se crea una función con la siguiente cabecera:

show_pyr(imgs)

Esta función se utilizará cada vez que se quiera pintar una pirámide. A través del parámetro *imgs* recibe una lista con las imágenes que se quieren utilizar. Como requisito para su correcto funcionamiento es que esas imágenes deben encontrarse correctamente escaladas.

El proceso para generar la pirámide consiste en crear un canvas de la misma altura que la primera imagen (imagen de mayor tamaño) y de anchura, la anchura de la primera imagen más la mitad de ella misma.

El proceso una vez creado el canvas es copiar la primera imagen en la esquina superior izquierda, para comenzar un bucle con el resto de imágenes.

Se debe llevar un contador donde se indica a partir de que fila se debe comenzar a copiar la imagen. Este contador se aumentará en cada iteración con la altura de la imagen que se ha copiado. La columna desde la que se parte será la anchura de la primera imagen + 1.

El último paso es devolver el canvas y mostrarlo con *show_images()*.

Se crea una función con la siguiente cabecera:

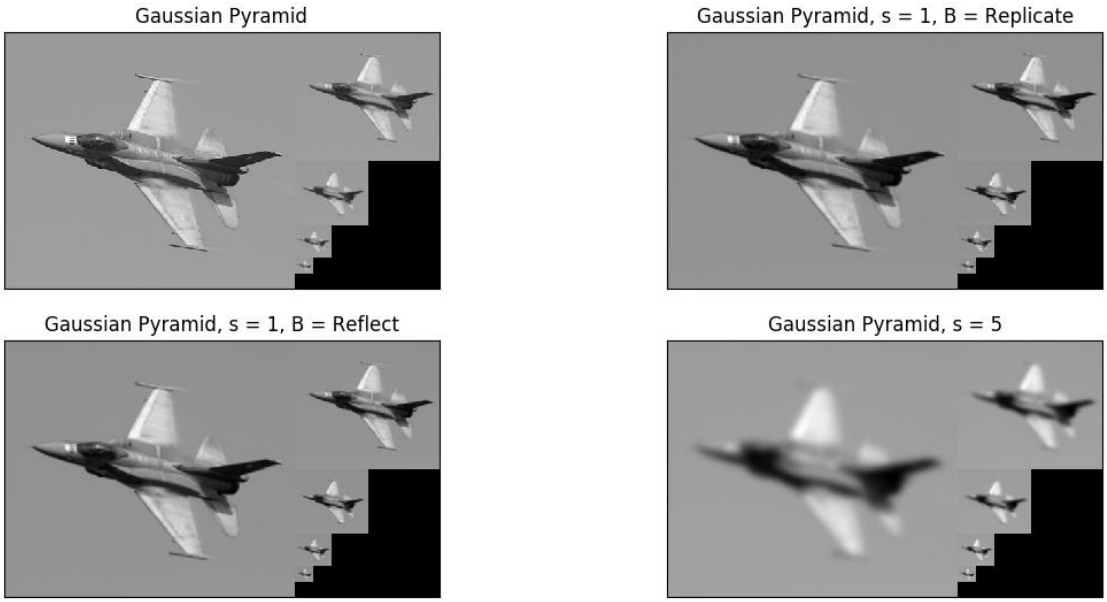
generate_gaussian_pyr_imgs(img, n = 4, sigma = 0, sigma_down = 1, border = cv2.BORDER_DEFAULT, own = False)

Esta función recibe una imagen (*img*) y un número (*n*), de manera que se crea un bucle de n iteraciones.

Lo primero que realiza la función es un alisamiento con el sigma recibido llamando a *convolution_c()*, después, comienza un bucle de *n* iteraciones, llamando a la función de OpenCV *cv2.pyrDown()* en cada una y reduciendo así el tamaño de la imagen. Finalmente, se inserta cada imagen generada en una lista y se devuelve.

Como *cv2.pyrDown()* ya hace alisamiento cada vez que reduce la imagen, no es necesario aplicar ningún tratamiento.

El resultado de ejecutar esta función es:



El borde Constant no se puede aplicar en *cv2.pyrDown()* y como es habitual, los otros dos tipos de bordes no se aprecian.

Cuando se hace una pirámide Gaussiana, al reducir se pierde información, porque el proceso consiste en eliminar las filas y columnas intermedias y hacer alisamiento. Al mostrar la imagen en un tamaño más pequeño, esa pérdida de información no se aprecia visualmente.

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

Se crea una función con la siguiente cabecera:

```
generate_laplacian_pyr_imgs(img, n = 4, sigma = 0, border =
cv2.BORDER_DEFAULT, own = False)
```

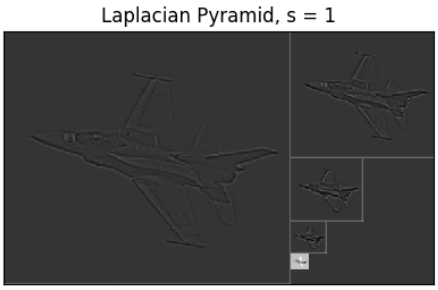
La idea de generar una pirámide Laplaciana es la misma que en la pirámide Gaussiana. La diferencia es que en este caso, las imágenes que se muestran son la diferencia entre la imagen original y la imagen producto de reducir y aumentarla, es decir, lo que se muestra es la pérdida de información que se produce en cada escalado.

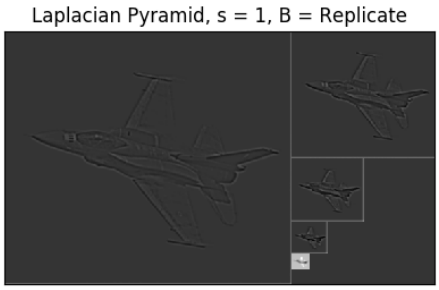
La implementación básica es similar a la de la pirámide Gaussiana, por lo que se explica los aspectos peculiares de la pirámide Laplaciana.

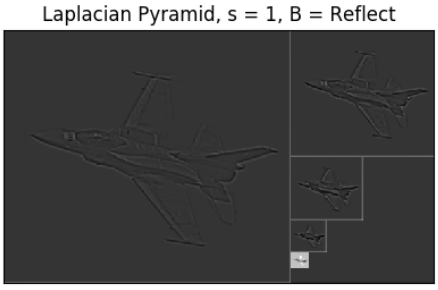
Cuando se reduce una imagen a la mitad y el número de filas o columnas es impar, se produce una imagen con un número de filas o columnas que, si se vuelve a duplicar su tamaño, ya no coincide con la imagen original. Este hecho hace que no se pueda hacer la diferencia entre ambas imágenes.

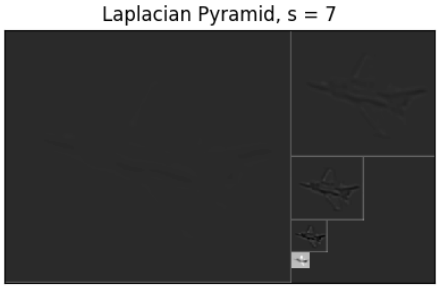
La solución para este hecho es utilizar un canvas con el número de filas y columnas de tamaño a la potencia de 2 más próxima. Si se tiene una imagen de 500x500, se crea un canvas de 512x512 y se copia la imagen original en él. De este modo se pueden hacer todas las operaciones sin problema y cuando haya que devolver la imagen calculada, es suficiente con recuperar el trozo de imagen que ocupaba la imagen original.

El resultado de ejecutar esta función es:









La pirámide Laplaciana muestra la pérdida que se produce en cada escalado. Como en el último nivel se tiene la imagen original, si se duplica su tamaño y se le suma la imagen del nivel n+1, se tiene la imagen en dicha escala sin pérdida de información. De este modo se puede redimensionar una imagen a su tamaño original y que sea exactamente igual que en su inicio.

2. Imágenes Híbridas:

1. Implementar una función que genera las imágenes de baja y alta frecuencia a partir de las parejas de imágenes. El valor de sigma más adecuado para cada pareja habrá que encontrarlo por experimentación.

Se crea una función con la siguiente cabecera:

```
generate_low_high_imgs(img1, img2, sigma1 = 0, sigma2 = 0, own = False)
```

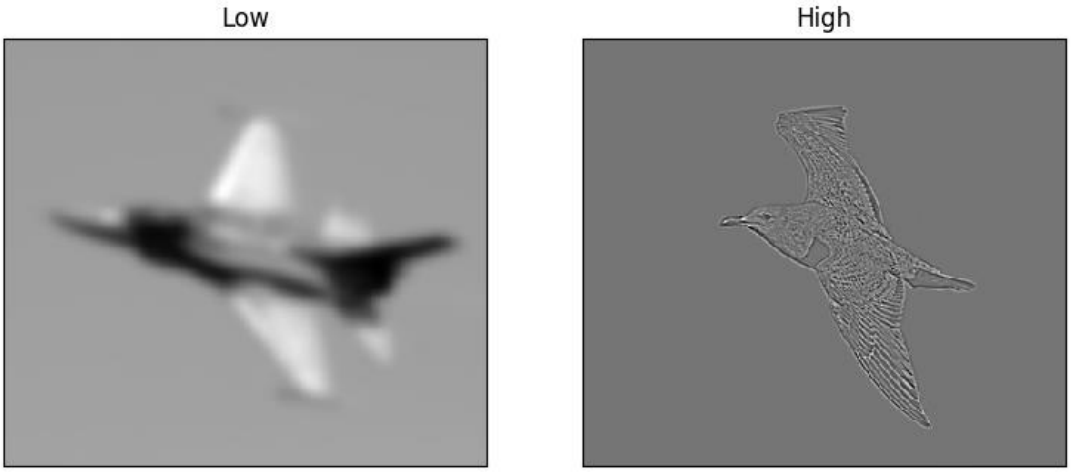
Las frecuencias bajas son las que el ojo humano percibe cuando se encuentra lejos del objeto. En cuanto a las imágenes, se consiguen las frecuencias bajas aplicando un alisamiento.

Las frecuencias altas son las que el ojo humano percibe cuando se encuentra cerca del objeto. En cuanto a las imágenes, se consiguen restando a la imagen original, las frecuencias bajas.

Basándonos en lo anterior, lo que realiza esta función es eso mismo. Recibe dos imágenes (*img1, img2*) con un sigma cada una (*sigma1, sigma2*) y devuelve otras dos imágenes, la imagen con las frecuencias bajas de la primera y la imagen con las frecuencias altas de la segunda.

Para conseguir las frecuencias bajas se hace uso de la función *convolution_c()* y para conseguir las frecuencias altas, se resta la imagen original a la imagen que devuelve la función *convolution_c()*.

El resultado de ejecutar la función es:



La imagen del avión es como la ya mostrada en los primeros apartados y contiene las frecuencias bajas. Se ha utilizado un sigma de 5 para generarla.

La imagen del pájaro contiene las frecuencias altas y se ha utilizado sigma 1.

El resultado que se puede apreciar es como desde una distancia lejana se puede ver el avión con un buen detalle y, a medida que se observa desde cerca, pierde calidad. El pájaro a cierta distancia deja de verse al contener solo las frecuencias altas, pero desde cerca se puede ver muy bien su contorno.

2. Escribir una función que muestre las tres imágenes (alta, baja e híbrida) en una misma ventana.

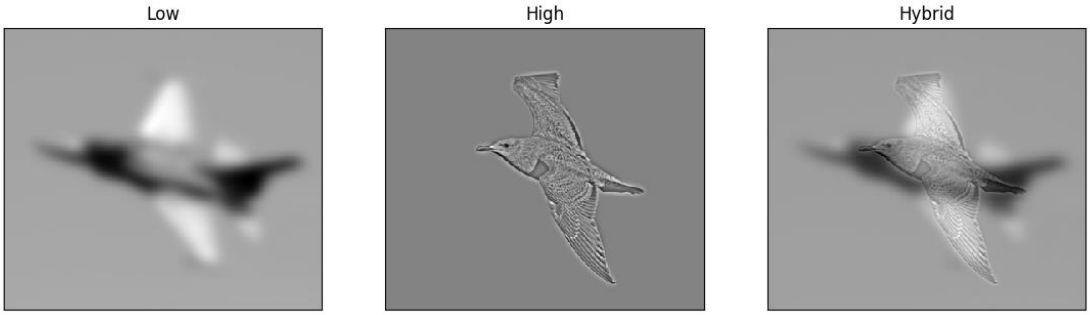
Se crean dos funciones con la siguiente cabecera:

```
show_hybrid(img1, img2, sigma1 = 0, sigma2 = 0, own = False)
generate_low_high_hybrid_imgs(img1, img2, sigma1 = 0, sigma2 = 0, own = False)
```

Estas funciones reciben dos imágenes y *generate_low_high_hybrid_imgs()* devuelve una imagen con las frecuencias bajas de la primera, las frecuencias altas de la segunda y la imagen híbrida de las dos anteriores. *show_hybrid()* muestra por pantalla las imágenes que ha generado la función anterior.

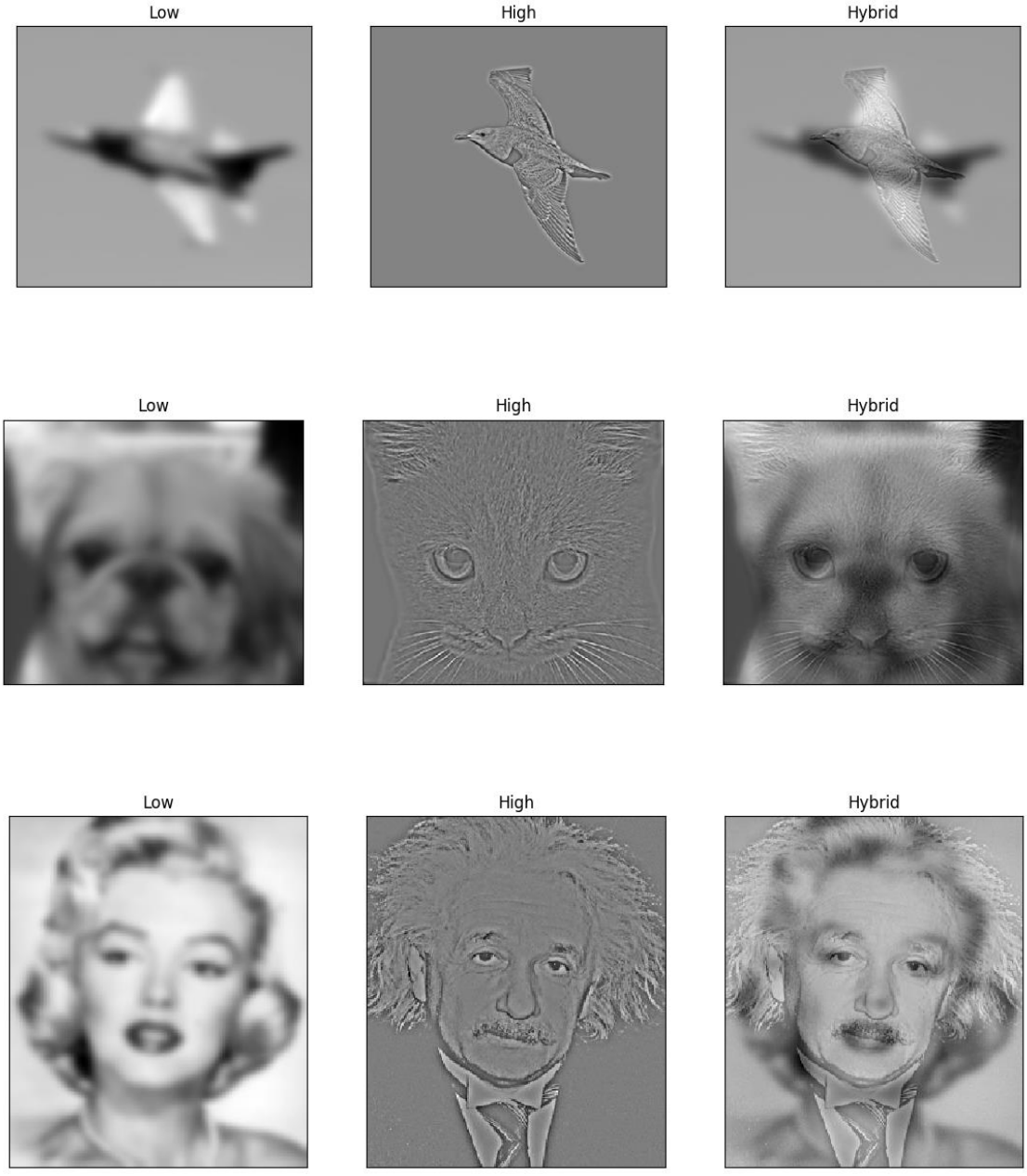
Como ya se tiene implementada la función que genera las imágenes en el apartado anterior, se llama para obtener las dos de las frecuencias. La híbrida se calcula simplemente sumando las dos imágenes.

El resultado de ejecutar esta función es:



Ahora, fijándose en la imagen híbrida, se puede apreciar como desde una distancia cercana se aprecian las frecuencias altas y aparece el pájaro y desde una distancia más lejana se ven las frecuencias bajas y aparece el avión.

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



3. Bonus:

1. Cálculo del vector máscara Gaussiano:

Se crea una función con la siguiente cabecera:

f_gaussian(x, sigma)

Esta función dado un x y devuelve el valor correspondiente a la siguiente fórmula:

$$f(x) = exp^{(-0.5\frac{x^2}{\sigma^2})}$$

Se crea una función con la siguiente cabecera:

get_gaussian_kernel(sigma = 0)

Esta función se encarga de generar un kernel similar al que calcula *cv2.getGaussianKernel()*.

Lo primero que se debe calcular es el tamaño. Este tamaño será de 6*sgima+1 y se debe inicializar en un intervalo cuyo valor central sea 0. Es decir, para un sigma de 1, se genera un vector con los siguientes valores [-3, -2, -1, 0, 1, 2, 3].

El siguiente paso consiste en recorrer cada elemento del vector y aplicarle la fórmula de la máscara Gaussiana especificada anteriormente.

Cuando ya se han realizado todos los cálculos, se debe normalizar el vector, ya que la suma de todos los elementos del kernel debe ser 1.

El resultado de ejecutar la función y *cv2.getGaussianKernel()* es (sigma = 3):

```
Función creada:
 [[ 0.00147945]
 [ 0.00380424]
 [ 0.00875346]
 [ 0.01802341]
 [ 0.03320773]
 [ 0.05475029]
 [ 0.08077532]
 [ 0.106639  ]
 [ 0.12597909]
 [ 0.133176  ]
 [ 0.12597909]
 [ 0.106639  ]
 [ 0.08077532]
 [ 0.05475029]
 [ 0.03320773]
 [ 0.01802341]
 [ 0.00875346]
 [ 0.00380424]
 [ 0.00147945]]
cv2.getGaussianKernel():
 [[ 0.00147945]
 [ 0.00380424]
 [ 0.00875346]
 [ 0.01802341]
 [ 0.03320773]
 [ 0.05475029]
 [ 0.08077532]
 [ 0.106639  ]
 [ 0.12597909]
 [ 0.133176  ]
 [ 0.12597909]
 [ 0.106639  ]
 [ 0.08077532]
 [ 0.05475029]
 [ 0.03320773]
 [ 0.01802341]
 [ 0.00875346]
 [ 0.00380424]
 [ 0.00147945]]
```

El resultado que ofrecen ambas funciones es el mismo, incluida la estructura en la que se devuelve, por lo que es posible utilizar una u otra indistintamente.

2. Implementar una función que calcule la convolución de un vector señal 1D con un vector-máscara de longitud inferior al de la señal usando condiciones de contorno reflejada.

Se crea una función con la siguiente cabecera:

filter_2d(signal, kernel)

Esta función recibe un vector y un kernel y realiza la convolución. Es igual que la función *cv2.filter2D()*.

La convolución consiste en iterar por todos los elementos del vector señal y modificar el valor del elemento en el que se está iterando en cada momento con una operación.

Si se tiene *l* como la longitud del kernel, la operación que se debe realizar es multiplicar un intervalo de tamaño *l* del vector señal (tomando como punto central el elemento que se está calculando) por el kernel. El resultado de multiplicar cada elemento del intervalo del vector señal con su correspondiente del kernel, se debe sumar y éste será el nuevo valor que tenga el elemento en el que se centra la iteración.

El problema de la convolución está en los extremos. Para calcular los *l* /2 elementos primeros o los *l*/2 elementos últimos, se debe extender el vector señal.

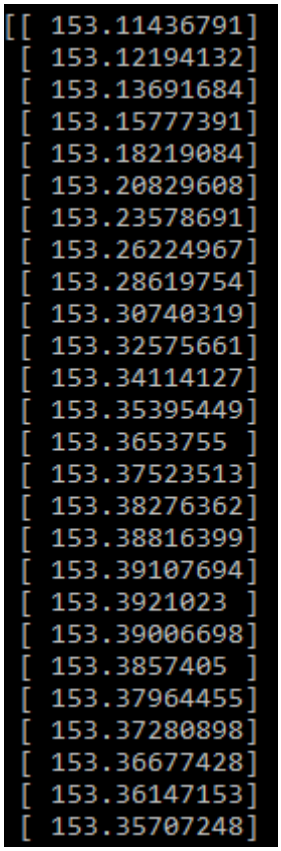
En este apartado se indica que se haga con condiciones de contorno reflejado, es decir, si el vector comienza en [1,2,3,4,5,…] y se debe extender 3 posiciones, el nuevo vector comenzará con [3,2,1,1,2,3,4,5,..]. Este mismo razonamiento se aplica para el extremo final.

El resultado de ejecutar esta función y cv2.filter2D() es (tomando como vector señal la primera fila de la imagen *plane.bmp* y sigma = 5):

- *filter_2d()* (la imagen se ha acortado por motivos de espacio)*:*

```
[ 153.11436791  153.12194132  153.13691684  153.15777391  153.18219084
  153.20829608  153.23578691  153.26224967  153.28619754  153.30740319
  153.32575661  153.34114127  153.35395449  153.3653755   153.37523513
  153.38276362  153.38816399  153.39107694  153.3921023   153.39006698
  153.3857405   153.37964455  153.37280898  153.36677428  153.36147153
  153.35707248  153.35479633  153.35459508  153.35644114  153.3607027
  153.36861677  153.37974306  153.39562979  153.41414171  153.43583401
  153.4607179   153.48516868  153.50820517  153.52805584  153.54294118
```

- *cv2.filter2D()* (la imagen se ha acortado por motivos de espacio)*:*



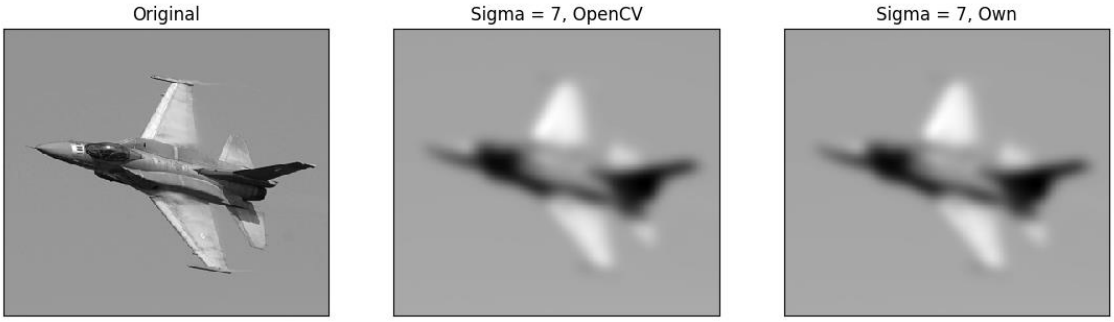
Los resultados son iguales aunque su representación no. No importa que su representación no sea igual ya que solo es necesario tratarlos de diferente manera.

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

La función creada en el Bonus 2 y *cv2.filter2D()* es la misma, por lo que se puede reutilizar en la función *convolution_c()* del apartado 1.C.

Lo único que se debe tener en cuenta es la estructura que devuelve una y otra. Por este motivo todas las funciones de la práctica pueden recibir el parámetro *own*. Si *own* es *True*, la función que se utiliza para hacer la convolución es la propia *filter_2d()* en lugar de la de OpenCV.

El resultado de ejecutar la función con el procesamiento propio y el procesamiento de OpenCV en un alisamiento es:



El procesamiento es mucho más lento en la función propia al hacer uso de un doble for que en Python es excesivamente lento. Por lo demás, se puede apreciar visualmente que los resultados son similares.

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

Durante todo el trabajo se ha hecho una implementación que permite especificar un parámetro *own = True* para indicar que la convolución y las operaciones que se realicen son las implementadas manualmente, por lo que solo se debe llamar a las funciones que generan las imágenes híbridas y la pirámide Gaussiana con este parámetro a *True.*

El único aspecto a tener en cuenta es la función *cv2.pyrDown()*, que si realiza la operación de escalado por ella misma.

Se crea una función con la siguiente cabecera:

```
pyr_down(img, sigma = 1, own = False)
```

Esta función recibe una imagen (*img*) y la reduce a la mitad, devolviéndola.

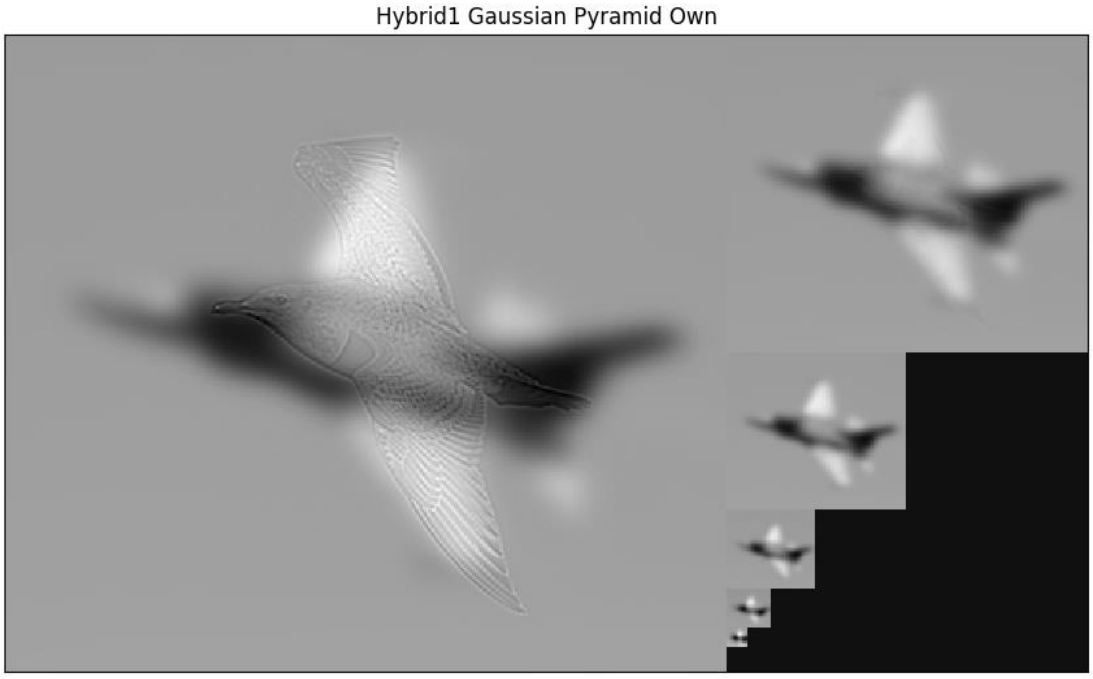
Cuando se reduce una imagen a la mitad, se eliminan la filas y columnas intermedias y se realiza un suavizado para eliminar los saltos pronunciados. Esto es lo que realiza *cv2.pyrDown()* y lo que se implementa en esta función.

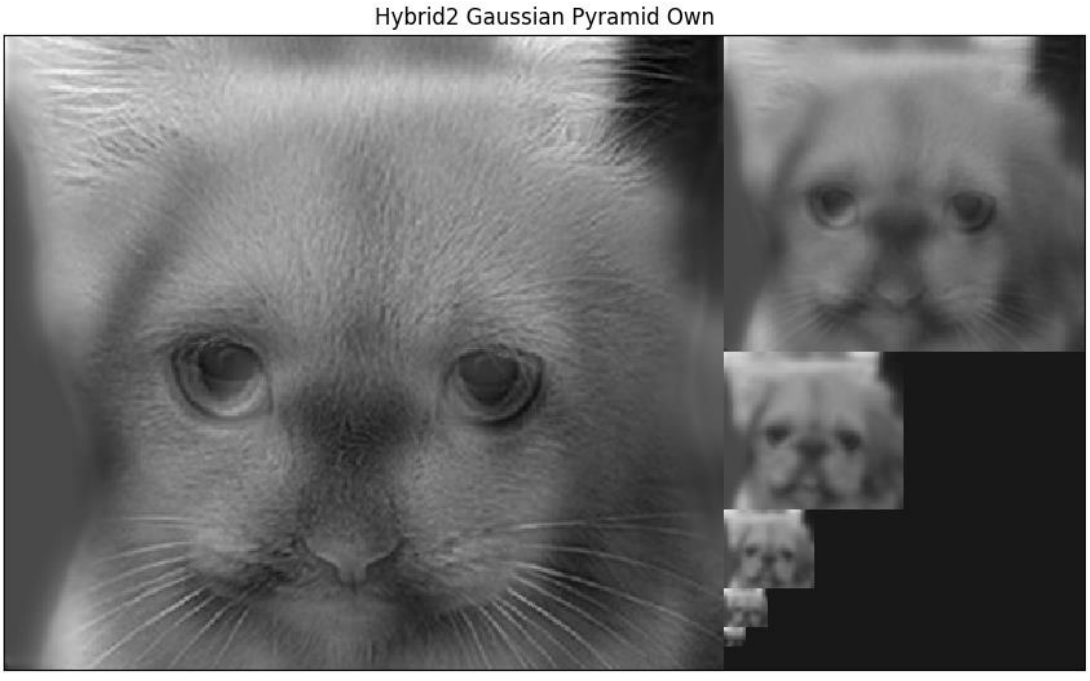
Para evitar problemas con imágenes de tamaño impar, se utiliza un canvas de tamaño potencia de 2 superior a la imagen, como ya se ha hecho en la pirámide Laplaciana.

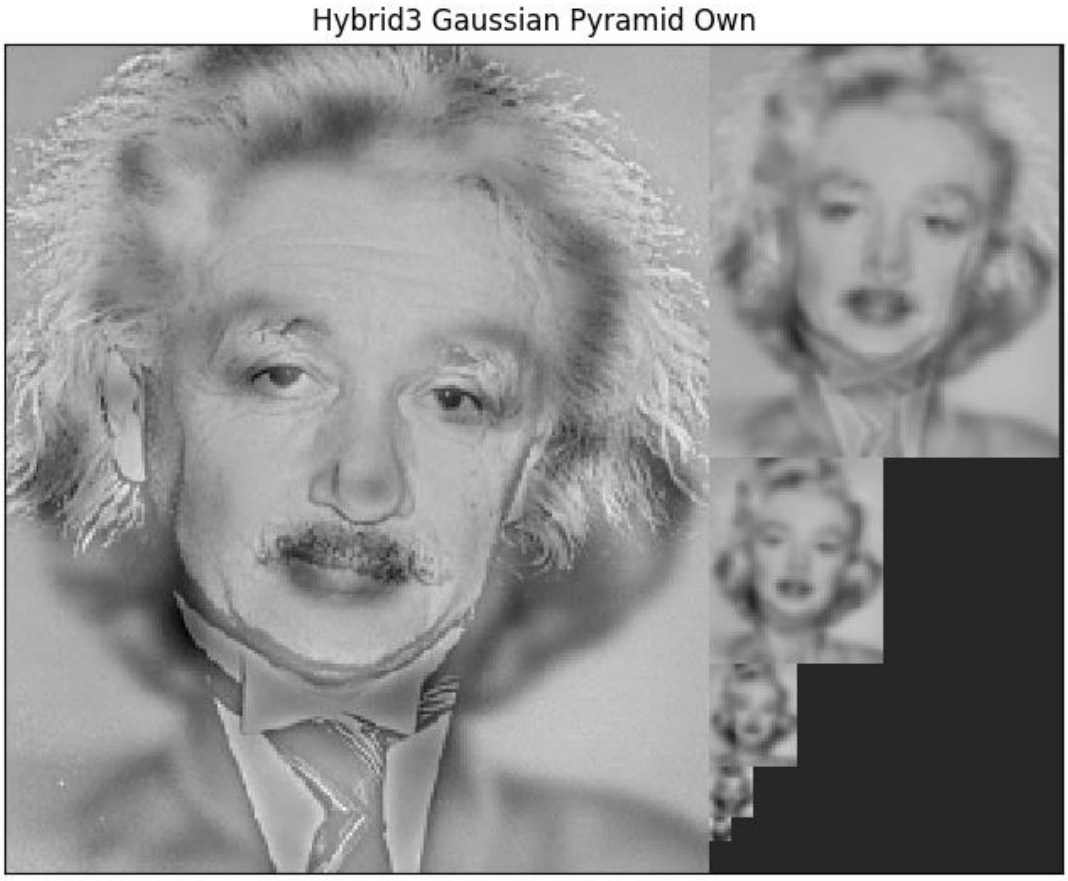
El resultado de ejecutar esta función y *cv2.pyrDown()* es:

El resultado de hacer el escalado a la mitad es prácticamente el mismo, aunque la función propia realiza más alisamiento, probablemente porque el valor de sigma utilizado no es el mismo.

El resultado de ejecutar las imágenes híbridas y mostrarlas como una pirámide Gaussiana es:







Se puede observar como la disminución del tamaño de las imágenes permite apreciar las frecuencias altas en las iniciales y las bajas en las finales.

4. Bibliografía

- Computer Vision: Local transformations. Apuntes Tema 1. Nicolás Pérez de la Blanca Capilla.
- http://www.sc.ehu.es/ccwgrrom/transparencias/pdf-vision-1-transparencias/capitulo-6.pdf
- http://www.lcc.uma.es/~munozp/documentos/procesamiento_de_imagenes/temas/pi_cap6.pdf