



ugr | Universidad
de **Granada**

Grado en Ingeniería Informática. Cuarto.

Práctica 1.

Nombre de la asignatura:

Visión por Computador. Jueves de 10:30 a 12:30.

Realizado por:

Néstor Rodríguez Vico. DNI: 75573052C.

email: nrv23@correo.ugr.es



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS
INFORMÁTICA Y DE TELECOMUNICACIÓN.

Granada, 20 de octubre de 2017.

Índice

1 Ejercicio 1.	4
1.1 Una función que sea capaz de representar varias imágenes con sus títulos en una misma ventana. Usar esta función en todos los demás apartados .	4
1.2 Una función de convolución con máscara Gaussiana de tamaño y sigma variable. Mostrar ejemplos de funcionamiento usando dos tipos de bordes y dos valores distintos de sigma.	5
1.3 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.	6
1.4 Una función de convolución con núcleo de 1 ^a derivada de tamaño variable. Mostrar ejemplos de funcionamiento usando dos tipos de bordes y dos valores distintos de sigma.	9
1.5 Una función de convolución con núcleo de 2 ^a derivada de tamaño variable. Mostrar ejemplos de funcionamiento usando dos tipos de bordes y dos valores distintos de sigma.	14
1.6 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.	15
1.7 Una función que genere una representación en pirámide Gaussiana de 4 niveles de una imagen. Mostrar ejemplos de funcionamiento usando dos tipos de bordes y dos valores distintos de sigma.	17
1.8 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.	18
2 Ejercicio 2: Imágenes híbridas.	20
3 Bonus.	24
3.1 Cálculo del vector máscara Gaussiano: Sea $f(x) = \exp\left(-0,5\frac{x^2}{\sigma^2}\right)$ una función donde σ (sigma) representa un parámetro en unidades píxel. Implementar una máscara de convolución 1D representativa de dicha función. Justificar los pasos dados.	24
3.2 Implementar una función que calcule la convolución de un vector señal 1D con un vector-máscara de longitud interior al de la señas usando condiciones de contorno reflejada. La salida será un vector de igual longitud que el vector señal de entrada.	25
3.3 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.	25

3.4 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 en todas las funciones.	27
3.5 Realizar todas las parejas de imágenes híbridas en su formato a color. . .	31

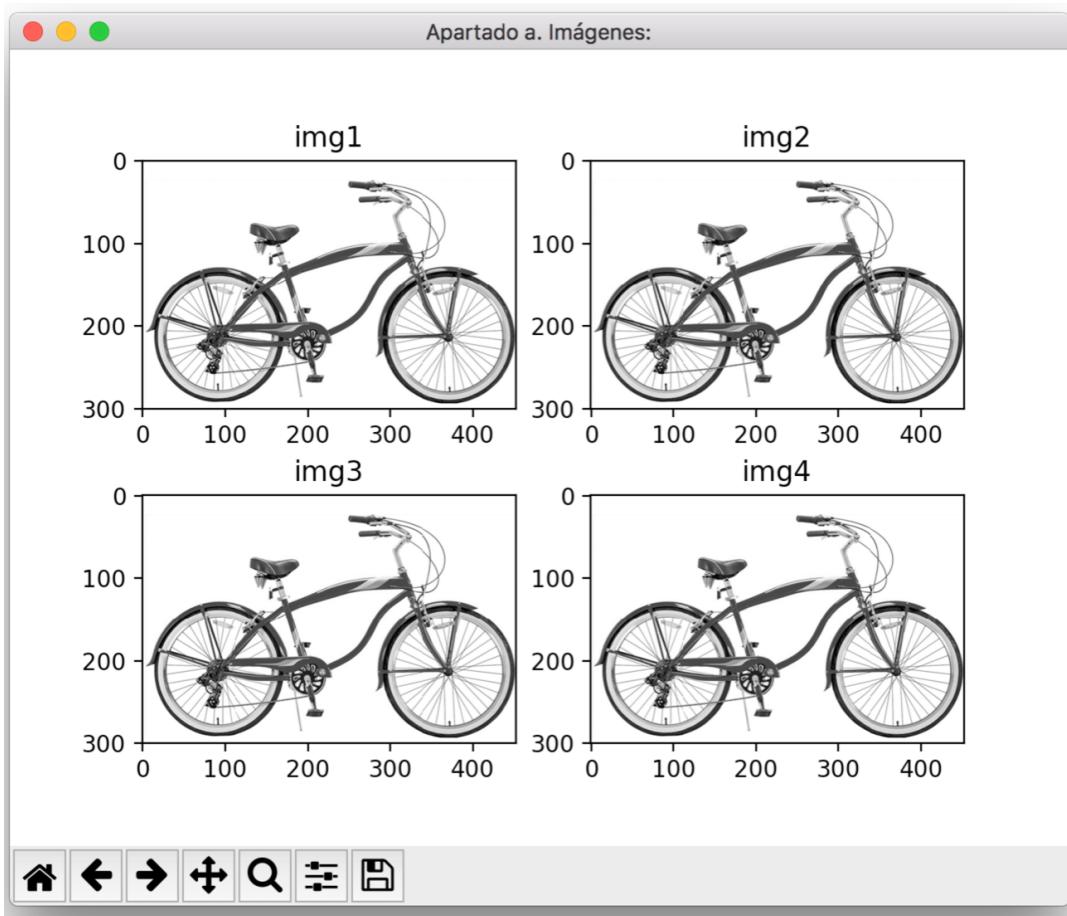
1. Ejercicio 1.

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

Para esta función vamos a hacer uso del módulo matplotlib y de la clase pyplot. La función que nos permite indicar donde queremos pintar es *subplot*. A esta función debemos indicarle donde queremos que pinte y posteriormente llamamos a *imshow* para que pinte la imagen que recibe como argumento. Debemos tener cuidado con el formato y el tipo que van a tener las imágenes que recibe nuestra función. Si la imagen es de color, OpenCV no usa el formato RGB, sino que usa el formato BGR, por eso se hace la conversión en la línea de 19.

```
1 import cv2
2 from matplotlib import pyplot as plt
3 import numpy as np
4 import copy
5 import math
6
7
8 def plot_images(imagenes, nombres, num_filas, num_columnas, name_windows):
9     fig = plt.figure(0)
10    fig.canvas.set_window_title(name_windows)
11
12    for i in range(num_filas * num_columnas):
13        if i < len(imagenes):
14            plt.subplot(num_filas, num_columnas, i + 1)
15
16            if len(np.shape(imagenes[i])) == 3:
17                # Si la imagen tiene tres canales, pintamos en color
18                # Cuidado con que OpenCV no usar RGB
19                img = cv2.cvtColor(np.array(imagenes[i], dtype=np.uint8), cv2.
20                                  COLOR_BGR2RGB)
21                plt.imshow(img)
22            else:
23                # Si tiene un canal, pintamos en gris
24                plt.imshow(imagenes[i], cmap="gray")
25
26            plt.title(nombres[i])
27
plt.show()
```

Por ejemplo, vamos a pintar la imagen de la bicicleta 2 veces, usando dos filas y dos columnas:



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

Por teoría sabemos que el 95 por ciento de la distribución Gaussiana está entre -3σ y 3σ , por lo tanto nuestra máscara tendrá un ancho de $6\sigma+1$ (añadimos 1 por el centro de la máscara). Con esto nos aseguramos un error bajo. Una vez sabemos el tamaño que debe tener nuestra máscara, llamamos a la función *GaussianBlur* de *OpenCV* que implementa un filtrado usando una máscara Gaussiana.

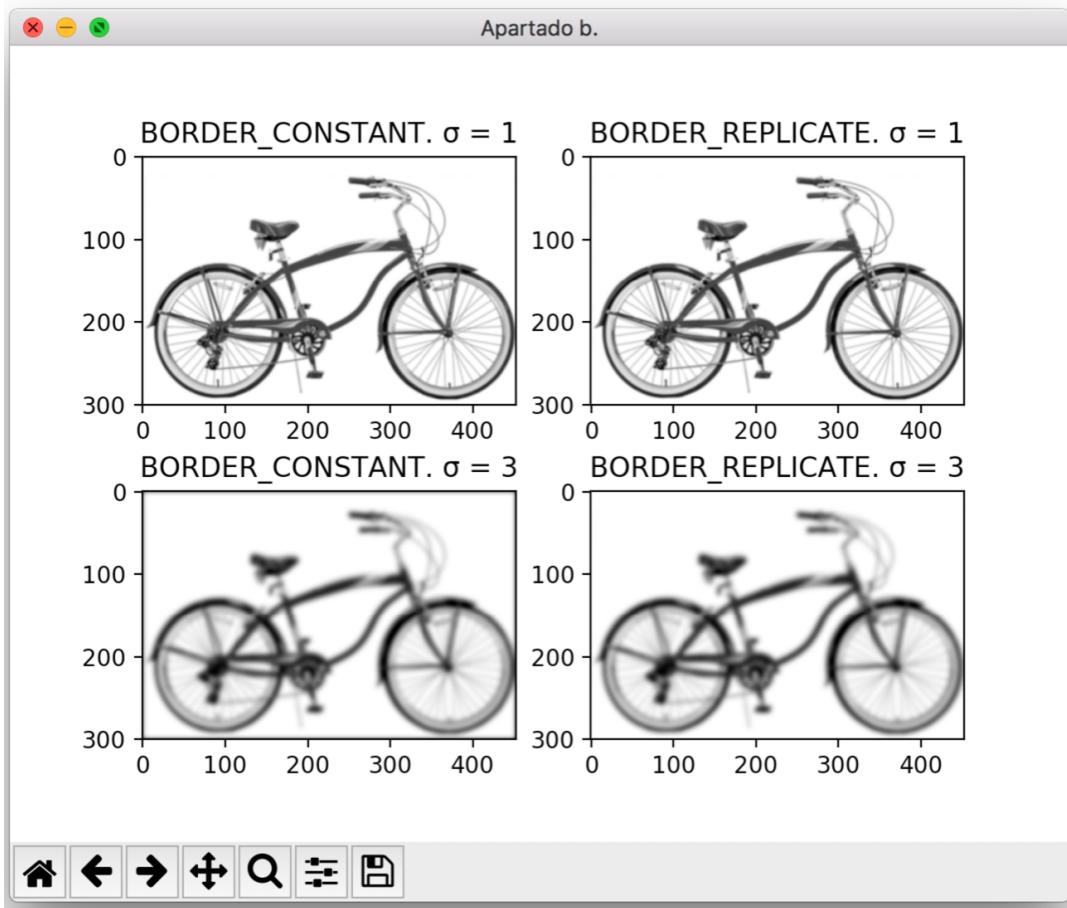
```

1 def convolucion(img , sigma , tamanio=3, borde=0):
2     gaussian_blur = cv2.GaussianBlur(src=img, ksize=(tamanio * 2 * sigma + 1, tamanio *
3         2 * sigma + 1), sigmaX=sigma, borderType=borde)
4
5     return gaussian_blur

```

En la siguiente imagen podemos ver el filtro Gaussiano aplicado usando distintos bordes y distintos valores de sigma. Podemos ver que a mayor nivel de sigma se pierden los detalles de la imagen y se va viendo más borrosa. Esto se debe a que, a la hora de calcular el valor del píxel i, j , al tener una máscara más grande, influyen más píxeles vecinos en el cálculo, lo cual nos lleva a que los píxeles se vayan pareciendo más a

sus vecinos, reduciéndose así los saltos entre los mismos y quedando una imagen más suavizada.



1.3. 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.

Como bien sabemos de teoría, si el núcleo es separable podemos optimizar el tiempo de computo haciendo una convolución separable aprovechando la propiedad asociativa de la convolución. Esto es pasear la máscara que tenemos primero por filas y luego por columnas. También sabemos que el núcleo Gaussiano es el mismo tanto para x como para y, por lo tanto es la misma máscara la que tenemos que pasear por filas y por columnas.

Debemos tener cuidado con la función *filter2D*. Si vemos la documentación de la función vemos que dicha función no hace convolución sino que hacer correlación. Para hacer convolución debemos rotar la máscara tanto en el eje x como en el eje y, tal y como hemos visto en teoría. Si reflexionamos acerca del kernel que nos da la función *getGaussianKernel* nos damos cuenta que sólo tiene una fila y es simétrico, por lo que al rotarlo en los dos ejes nos da el mismo, por eso no es necesario hacerlo. A la función

`getGaussianKernel` le indicamos que el tamaño sea el comentado en el apartado anterior.

La idea de la función es sencilla, primero paseamos la máscara por filas (líneas 17-20), transponemos el resultado obtenido y pasear por columnas la máscara (líneas 32-35), aunque al estar traspuesta en la implementación parece que se pasea por filas de nuevo.

En esta función y en todas las siguientes se contempla que la imagen tenga 3 canales (imagen en color) o 1 canal (imagen en escala de grises) pero en esencia la idea de funcionamiento es igual. También va a haber un argumento, `normalize`, que nos indica si queremos normalizar el resultado o no. Cuando queramos hacer un cálculo, este argumento deberá ser `False` para devolver el resultado tal y como ha sido calculado. Para esta práctica se va a normalizar en la última operación realizada, para poder representar la imagen correctamente.

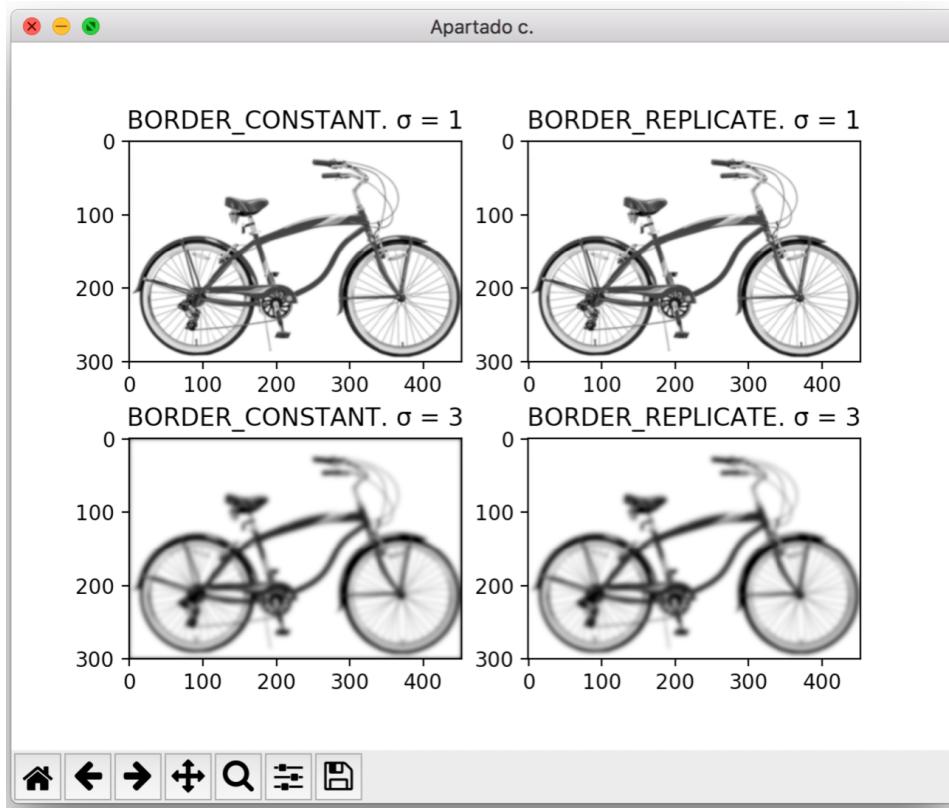
```
1 def convolucion_separable(img, sigma, kernel=None, tamanio=3, borde=0, normalize=False):
2     :
3     if kernel is None:
4         # Si no nos dan el kernel, lo calculamos
5         kernel = cv2.getGaussianKernel(ksize=tamanio * 2 * sigma + 1, sigma=sigma)
6
7     kernel = kernel.flatten()
8
9     if len(np.shape(img)) == 3:
10        # Tratamiento de los tres canales
11        canal_1, canal_2, canal_3 = cv2.split(img)
12
13        auxiliar_1 = np.zeros(shape=np.shape(canal_1), dtype=np.float64)
14        auxiliar_2 = np.zeros(shape=np.shape(canal_2), dtype=np.float64)
15        auxiliar_3 = np.zeros(shape=np.shape(canal_3), dtype=np.float64)
16
17        # Paseamos la máscara por filas
18        for fila_1, fila_2, fila_3, i in zip(canal_1, canal_2, canal_3, range(np.shape(
19            img)[0])):
20            auxiliar_1[i] = cv2.filter2D(src=fila_1, ddepth=-1, kernel=kernel,
21                borderType=borde).flatten()
22            auxiliar_2[i] = cv2.filter2D(src=fila_2, ddepth=-1, kernel=kernel,
23                borderType=borde).flatten()
24            auxiliar_3[i] = cv2.filter2D(src=fila_3, ddepth=-1, kernel=kernel,
25                borderType=borde).flatten()
26
27        # Trasponemos para acceder más fácilmente a las columnas
28        auxiliar_1 = np.array(auxiliar_1).transpose()
29        auxiliar_2 = np.array(auxiliar_2).transpose()
30        auxiliar_3 = np.array(auxiliar_3).transpose()
31
32        # Paseamos la máscara por columnas
33        for fila_1, fila_2, fila_3, i in zip(auxiliar_1, auxiliar_2, auxiliar_3, range(
34            np.shape(auxiliar_1)[0])):
35            salida_1[i] = cv2.filter2D(src=fila_1, ddepth=-1, kernel=kernel, borderType
36                =borde).flatten()
37            salida_2[i] = cv2.filter2D(src=fila_2, ddepth=-1, kernel=kernel, borderType
38                =borde).flatten()
39            salida_3[i] = cv2.filter2D(src=fila_3, ddepth=-1, kernel=kernel, borderType
40                =borde).flatten()
41
42        salida_1 = np.array(salida_1).transpose()
```

```

38     salida_2 = np.array(salida_2).transpose()
39     salida_3 = np.array(salida_3).transpose()
40
41     if normalize:
42         salida_1 = (salida_1 - np.amin(salida_1)) / (np.amax(salida_1) - np.amin(
43             salida_1))
44         salida_2 = (salida_2 - np.amin(salida_2)) / (np.amax(salida_2) - np.amin(
45             salida_2))
46         salida_3 = (salida_3 - np.amin(salida_3)) / (np.amax(salida_3) - np.amin(
47             salida_3))
48
49     return cv2.merge([salida_1, salida_2, salida_3])
50 else:
51     auxiliar = np.zeros(shape=np.shape(img), dtype=np.float64)
52
53     for fila, i in zip(img, range(np.shape(img)[0])):
54         auxiliar[i] = cv2.filter2D(src=fila, ddepth=-1, kernel=kernel, borderType=
55             borde).flatten()
56
57     auxiliar = np.array(auxiliar).transpose()
58
59     salida = np.zeros(shape=np.shape(auxiliar), dtype=np.float64)
60
61     for fila, i in zip(auxiliar, range(np.shape(auxiliar)[0])):
62         salida[i] = cv2.filter2D(src=fila, ddepth=-1, kernel=kernel, borderType=
63             borde).flatten()
64
65     salida = np.array(salida).transpose()
66
67     if normalize:
68         salida = (salida - np.amin(salida)) / (np.amax(salida) - np.amin(salida))
69
70     return salida

```

Veamos un ejemplo de funcionamiento de esta función:



Como podemos ver el resultado obtenido es igual que el producido por la función del apartado anterior que usa la función *GaussianBlur* de *OpenCV*. Al igual que hemos comentado antes, a mayor valor de sigma, la imagen que obtenemos está más suavizada.

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

Para la función de derivada debemos tener cuidado con como está implementada la función *getDerivKernels* de *OpenCV*. Si le indicamos que nos de un kernel de de tamaño 7, el *sigma* que usa es uno. Teniendo en cuenta la propiedad comentada en clase ($G(\sigma_1^2) \otimes G(\sigma_2^2) = G(\sigma_1^2 + \sigma_2^2)$), lo que debemos hacer es convolucionar la imagen con con el kernel obtenido por la función *getDerivKernels* y el resultado convolucionar con un filtro Gaussiano con *sigma* igual al *sigma* deseado menos 1.

Esta función devuelve dos imágenes, la primera de ellas aplicando los núcleos obtenidos al derivar x y la segunda aplicando los núcleos obtenidos al derivar y.

Los núcleos obtenidos por la función *getDerivKernels* no son simétricos, así que debemos invertirlos, tal y como se comentó en el apartado 2, para que la función *filtered2D* haga convolución. Esto lo hacemos en las líneas 12-16. A continuación paseamos las

máscaras obtenidas al derivar con respecto a x (líneas 29-40). Luego trasponemos la imagen original para pasear las máscaras obtenidas al derivar con respecto a y (líneas 55-67) por columnas.

Una vez hemos obtenido la imagen I_1 tras aplicar las máscaras de la derivada en x y la imagen I_2 tras aplicar las máscaras de la derivada en y, lo que debemos hacer es aplicar el filtro Gaussiano pero con *sigma* decrementada en uno (líneas 80-90). Este proceso sólo debemos hacerlo si el valor de *sigma* indicado a nuestra función es mayor que uno.

Finalmente normalizamos si es necesario y juntamos los tres canales.

```

1 def convolucion_derivada(img, sigma, derivada_x=1, derivada_y=1, tamano=3, borde=0,
2 normalize=False):
3     kernel_derivx_x, kernel_derivx_y = cv2.getDerivKernels(dx=derivada_x, dy=0, ksize
4         =7, normalize=True)
5     kernel_derivy_x, kernel_derivy_y = cv2.getDerivKernels(dx=0, dy=derivada_y, ksize
6         =7, normalize=True)
7
8     kernel_derivx_x = kernel_derivx_x.flatten()
9     kernel_derivx_y = kernel_derivx_y.flatten()
10
11    # Invertimos los kernels para hacer convolución
12    kernel_derivx_x = kernel_derivx_x[::-1]
13    kernel_derivx_y = kernel_derivx_y[::-1]
14
15    kernel_derivy_x = kernel_derivy_x[::-1]
16    kernel_derivy_y = kernel_derivy_y[::-1]
17
18 if len(np.shape(img)) == 3:
19     canal_1, canal_2, canal_3 = cv2.split(img)
20     # Paseamos la máscara de la derivada en x por filas
21     auxiliar_x_1 = np.zeros(shape=np.shape(canal_1), dtype=np.float64)
22     auxiliar_x_2 = np.zeros(shape=np.shape(canal_2), dtype=np.float64)
23     auxiliar_x_3 = np.zeros(shape=np.shape(canal_3), dtype=np.float64)
24
25     salida_x_1 = np.zeros(shape=np.shape(canal_1), dtype=np.float64)
26     salida_x_2 = np.zeros(shape=np.shape(canal_2), dtype=np.float64)
27     salida_x_3 = np.zeros(shape=np.shape(canal_3), dtype=np.float64)
28
29 for fila_1, fila_2, fila_3, i in zip(canal_1, canal_2, canal_3, range(np.shape(
30     salida_x_1)[0])):
31     auxiliar_x_1[i] = cv2.filter2D(src=fila_1, ddepth=cv2.CV_16S, kernel=
32         kernel_derivx_x,
33                     borderType=borde).flatten()
34     auxiliar_x_2[i] = cv2.filter2D(src=fila_2, ddepth=cv2.CV_16S, kernel=
35         kernel_derivx_x,
36                     borderType=borde).flatten()
37     auxiliar_x_3[i] = cv2.filter2D(src=fila_3, ddepth=cv2.CV_16S, kernel=
38         kernel_derivx_x,
39                     borderType=borde).flatten()
40
41 for fila_1, fila_2, fila_3, i in zip(auxiliar_x_1, auxiliar_x_2, auxiliar_x_3,
42 range(np.shape(salida_x_1)[0])):
43     salida_x_1[i] = cv2.filter2D(src=fila_1, ddepth=-1, kernel=kernel_derivx_y,
44         borderType=borde).flatten()
45     salida_x_2[i] = cv2.filter2D(src=fila_2, ddepth=-1, kernel=kernel_derivx_y,
46         borderType=borde).flatten()
47     salida_x_3[i] = cv2.filter2D(src=fila_3, ddepth=-1, kernel=kernel_derivx_y,
48         borderType=borde).flatten()

```

```

40     salida_x_3[i] = cv2.filter2D(src=fila_3, ddepth=-1, kernel=kernel_derivx_y,
41                                 borderType=borde).flatten()
42
43     canal_1_transpose = canal_1.transpose()
44     canal_2_transpose = canal_2.transpose()
45     canal_3_transpose = canal_3.transpose()
46
47     auxiliar_y_1 = np.zeros(shape=np.shape(canal_1_transpose), dtype=np.float64)
48     auxiliar_y_2 = np.zeros(shape=np.shape(canal_2_transpose), dtype=np.float64)
49     auxiliar_y_3 = np.zeros(shape=np.shape(canal_3_transpose), dtype=np.float64)
50
51     salida_y_1 = np.zeros(shape=np.shape(canal_1_transpose), dtype=np.float64)
52     salida_y_2 = np.zeros(shape=np.shape(canal_2_transpose), dtype=np.float64)
53     salida_y_3 = np.zeros(shape=np.shape(canal_3_transpose), dtype=np.float64)
54
55     # Paseamos la máscara de la derivada en y por columnas
56     for fila_1, fila_2, fila_3, i in zip(canal_1_transpose, canal_2_transpose,
57                                           canal_3_transpose,
58                                           range(np.shape(auxiliar_y_1)[0])):
59         auxiliar_y_1[i] = cv2.filter2D(src=fila_1, ddepth=cv2.CV_16S, kernel=
59                                         kernel_derivy_x,
60                                         borderType=borde).flatten()
61         auxiliar_y_2[i] = cv2.filter2D(src=fila_2, ddepth=cv2.CV_16S, kernel=
61                                         kernel_derivy_x,
62                                         borderType=borde).flatten()
63         auxiliar_y_3[i] = cv2.filter2D(src=fila_3, ddepth=cv2.CV_16S, kernel=
63                                         kernel_derivy_x,
64                                         borderType=borde).flatten()
65
66     for fila_1, fila_2, fila_3, i in zip(auxiliar_y_1, auxiliar_y_2, auxiliar_y_3,
67                                           range(np.shape(salida_y_1)[0])):
68         salida_y_1[i] = cv2.filter2D(src=fila_1, ddepth=-1, kernel=kernel_derivy_y,
68                                         borderType=borde).flatten()
69         salida_y_2[i] = cv2.filter2D(src=fila_2, ddepth=-1, kernel=kernel_derivy_y,
69                                         borderType=borde).flatten()
70         salida_y_3[i] = cv2.filter2D(src=fila_3, ddepth=-1, kernel=kernel_derivy_y,
70                                         borderType=borde).flatten()
71
72     if normalize:
73         salida_x_1 = (salida_x_1 - np.amin(salida_x_1)) / (np.amax(salida_x_1) - np
73                                         .amin(salida_x_1))
74         salida_x_2 = (salida_x_2 - np.amin(salida_x_2)) / (np.amax(salida_x_2) - np
74                                         .amin(salida_x_2))
75         salida_x_3 = (salida_x_3 - np.amin(salida_x_3)) / (np.amax(salida_x_3) - np
75                                         .amin(salida_x_3))
76         salida_y_1 = (salida_y_1 - np.amin(salida_y_1)) / (np.amax(salida_y_1) - np
76                                         .amin(salida_y_1))
77         salida_y_2 = (salida_y_2 - np.amin(salida_y_2)) / (np.amax(salida_y_2) - np
77                                         .amin(salida_y_2))
78         salida_y_3 = (salida_y_3 - np.amin(salida_y_3)) / (np.amax(salida_y_3) - np
78                                         .amin(salida_y_3))
79
80     salida_x = cv2.merge([salida_x_1, salida_x_2, salida_x_3])
81     salida_y = cv2.merge([salida_y_1.transpose(), salida_y_2.transpose(),
81                         salida_y_3.transpose()])
82
83     if sigma == 1:
84         return salida_x, salida_y
85     else:
86         x = convolucion_separable(img=salida_x, sigma=sigma - 1, kernel=None,
86                                     tamanio=tamanio, borde=borde)
87         y = convolucion_separable(img=salida_y, sigma=sigma - 1, kernel=None,
87                                     tamanio=tamanio, borde=borde)

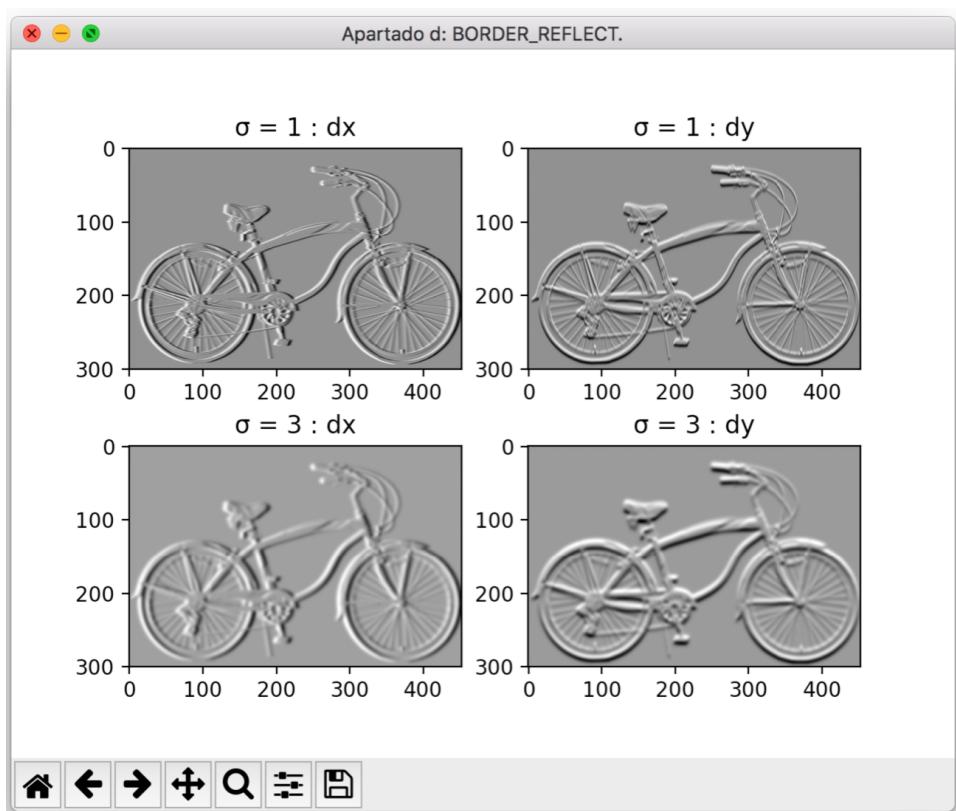
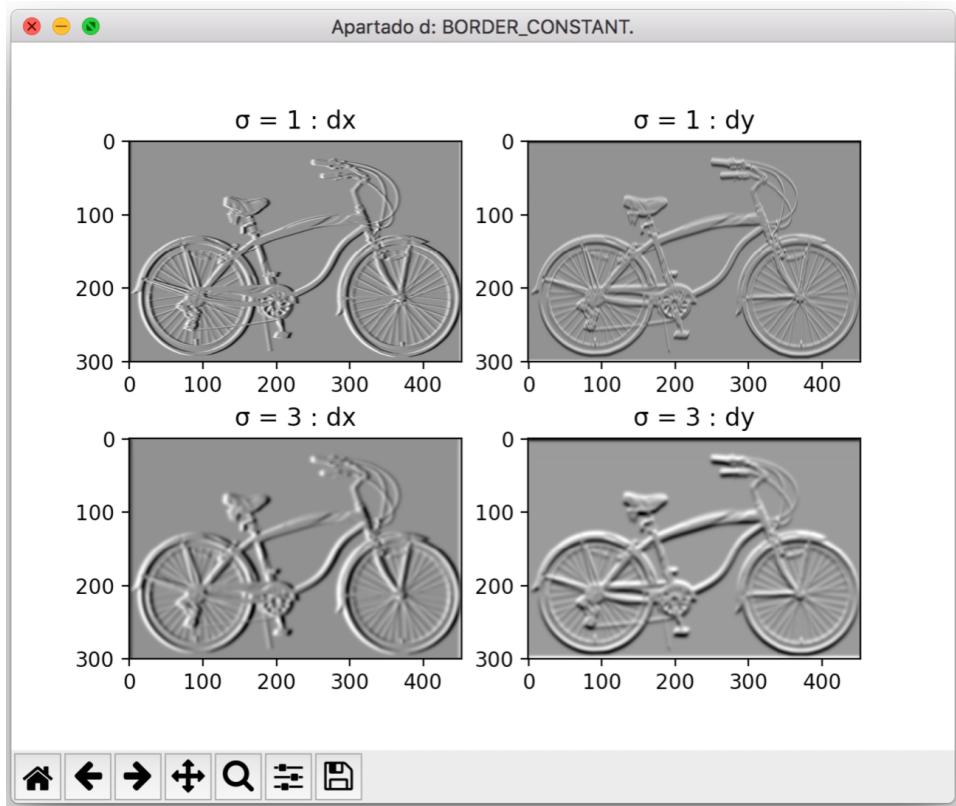

```

```

88         y = (y - np.amin(y)) / (np.amax(y) - np.amin(y))
89
90     return x, y
91 else:
92     auxiliar_x = np.zeros(shape=np.shape(img), dtype=np.float64)
93     salida_x = np.zeros(shape=np.shape(img), dtype=np.float64)
94
95     for fila, i in zip(img, range(np.shape(auxiliar_x)[0])):
96         auxiliar_x[i] = cv2.filter2D(src=fila, ddepth=cv2.CV_16S, kernel=
97             kernel_derivx_x,
98             borderType=borde).flatten()
99
100    for fila, i in zip(auxiliar_x, range(np.shape(salida_x)[0])):
101        salida_x[i] = cv2.filter2D(src=fila, ddepth=-1, kernel=kernel_derivx_y,
102            borderType=borde).flatten()
103
104    img_transpose = img.transpose()
105
106    auxiliar_y = np.zeros(shape=np.shape(img_transpose), dtype=np.float64)
107    salida_y = np.zeros(shape=np.shape(img_transpose), dtype=np.float64)
108
109    for fila, i in zip(img_transpose, range(np.shape(auxiliar_y)[0])):
110        auxiliar_y[i] = cv2.filter2D(src=fila, ddepth=cv2.CV_16S, kernel=
111            kernel_derivy_x,
112            borderType=borde).flatten()
113
114    for fila, i in zip(auxiliar_y, range(np.shape(salida_y)[0])):
115        salida_y[i] = cv2.filter2D(src=fila, ddepth=-1, kernel=kernel_derivy_y,
116            borderType=borde).flatten()
117
118    salida_y = salida_y.transpose()
119
120    if normalize:
121        salida_x = (salida_x - np.amin(salida_x)) / (np.amax(salida_x) - np.amin(
122            salida_x))
123        salida_y = (salida_y - np.amin(salida_y)) / (np.amax(salida_y) - np.amin(
124            salida_y))
125
126    if sigma == 1:
127        return salida_x, salida_y
128    else:
129        x = convolucion_separable(img=salida_x, sigma=sigma - 1, kernel=None,
130            tamano=tamano, borde=borde)
131        y = convolucion_separable(img=salida_y, sigma=sigma - 1, kernel=None,
132            tamano=tamano, borde=borde)
133
134    if normalize:
135        x = (x - np.amin(x)) / (np.amax(x) - np.amin(x))
136        y = (y - np.amin(y)) / (np.amax(y) - np.amin(y))
137
138    return x, y

```

Los resultados los podemos ver a continuación:



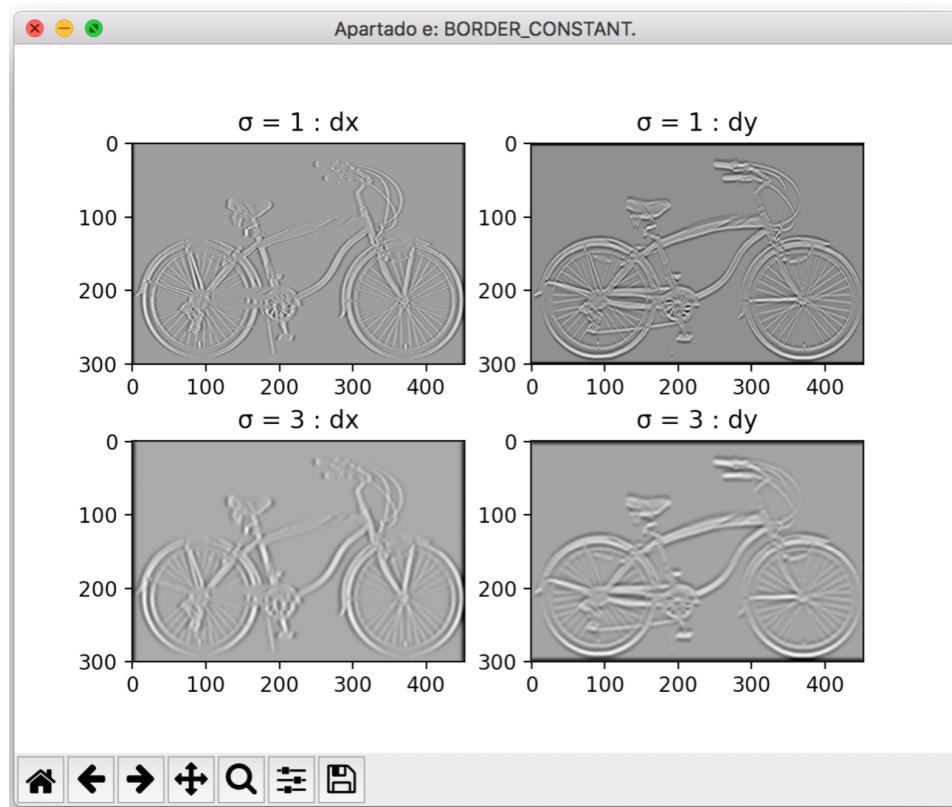
Podemos ver que en las imágenes en las que se aplica la derivada en x se identifican patrones en vertical (como se puede observar en las sombras del sillín o el manillar) mientras que en las de la derivada en y se identifican patrones en horizontal (como se puede ver en el cuerpo de la bici). También podemos ver que las imágenes $\sigma = 3$ se nota el suavizado. Esto se debe a que tras aplicar las máscaras de las derivadas, se les ha aplicado un filtro Gaussiano de $\sigma = 2$, tal y como se ha comentado a principio.

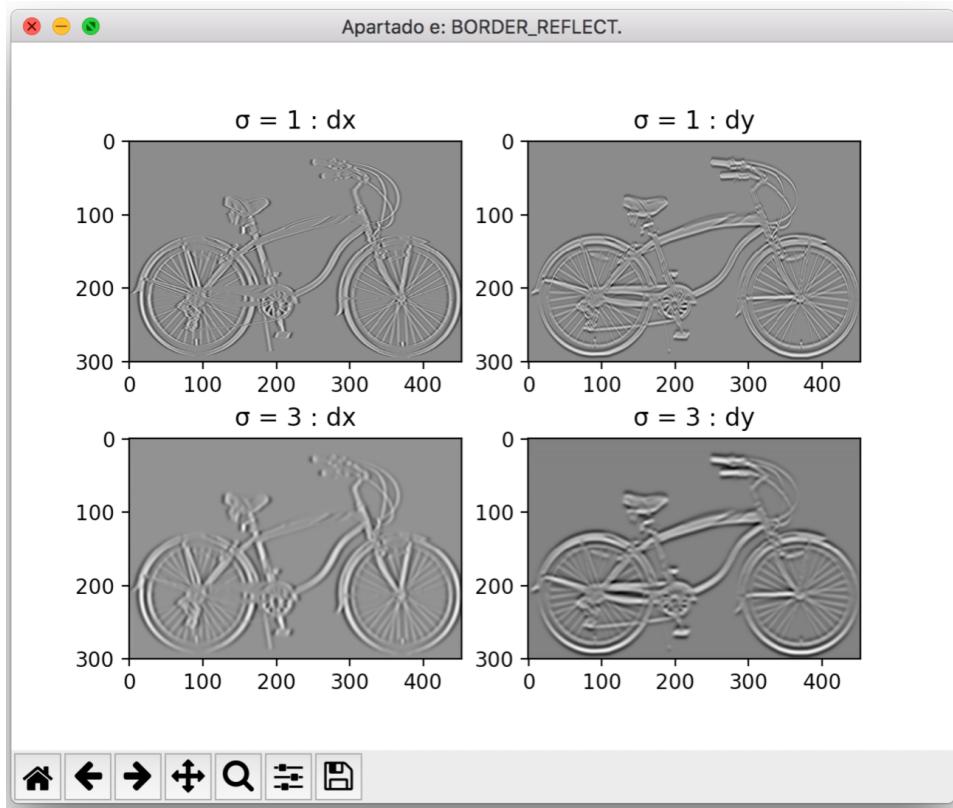
En cuanto a los bordes, podemos ver que las imágenes con borde constante son más oscuras y están menos definidas las siluetas. Si por ejemplo comparamos las imágenes con $\sigma = 1$ en la derivada en y en ambos casos, se puede apreciar dicha diferencia.

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

La función de convolución con núcleo de 2^a derivada es igual que la de 1^a derivada solo que en la llamada a la función indicamos que los argumentos *derivada_x* y *derivada_y* valen 2.

Los resultados los podemos ver a continuación:





Las segundas derivadas tienen una forma muy similar a las primeras, pero los bordes no están tan marcados como en las imágenes del apartado anterior. La mayor diferencia se ve si comparamos las primeras derivadas con las segundas al usar el borde constante.

1.6. 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.

La Laplaciana-de-Gaussiana no es más que la suma de las segundas derivadas de la función Gaussiana. Lo que debemos hacer entonces es llamar a la función convolución derivada y sumar los resultados, sin olvidarnos de normalizar el resultado. Al contrario que la función anterior, la Laplaciana-de-Gaussiana nos devuelve una única imagen, formada por la suma de las imágenes obtenidas aplicando las segundas derivadas. Tal y como nos indican las transparencias de teoría, para normalizar la imagen resultante debemos multiplicar por la varianza, es decir, *sigma*, tal y como se puede ver en la línea 5.

```

1 def laplaciana_gaussiana(img, sigma, tamano=3, borde=0, normalize=False):
2     # La laplaciana es la suma de las segundas derivadas
3     salida_x, salida_y = convolucion_derivada(img=img, sigma=sigma, derivada_x=2,
4                                                 derivada_y=2, tamano=tamano,
4                                                 borde=borde, normalize=normalize)

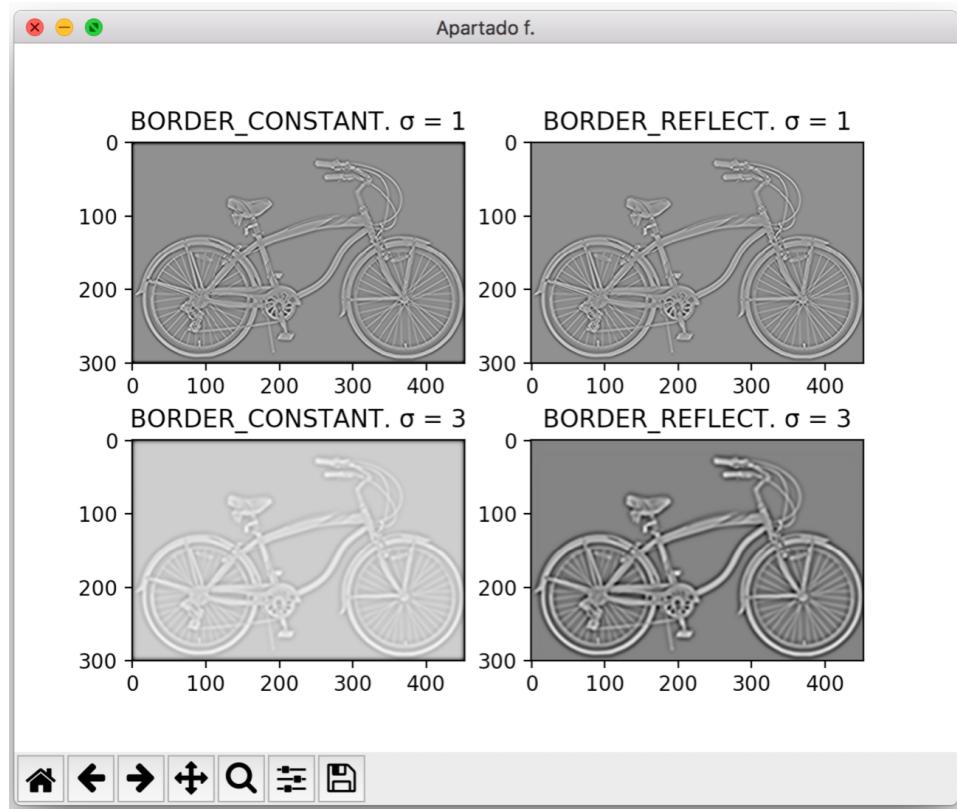
```

```

5      # Multiplicamos por sigma para normalizar
6      salida = sigma * (salida_x + salida_y)
7      if normalize:
8          salida = (salida - np.amin(salida)) / (np.amax(salida) - np.amin(salida))
9
10
11      return salida

```

Podemos ver el resultado a continuación:



En las imágenes podemos ver que tienen una forma similar a las obtenidas con las derivadas (ya que la función Laplaciana-de-Gaussiana es una suma de derivadas) pero en estas podemos ver que los patrones en horizontal y en vertical que obteníamos con las derivadas no se notan. Sin embargo, podemos ver que los contornos de la figura están más marcados y mejor definidos que con las derivadas de los apartados anteriores.

Aunque también se aplica el filtro Gaussiano a las imágenes (antes de ser sumadas para formar la imagen resultante), no se puede apreciar tanto el suavizado en estas imágenes.

En la imagen con borde constante para $\sigma = 3$ parece que se ha perdido un poco el color. Esto se debe al proceso de normalización.

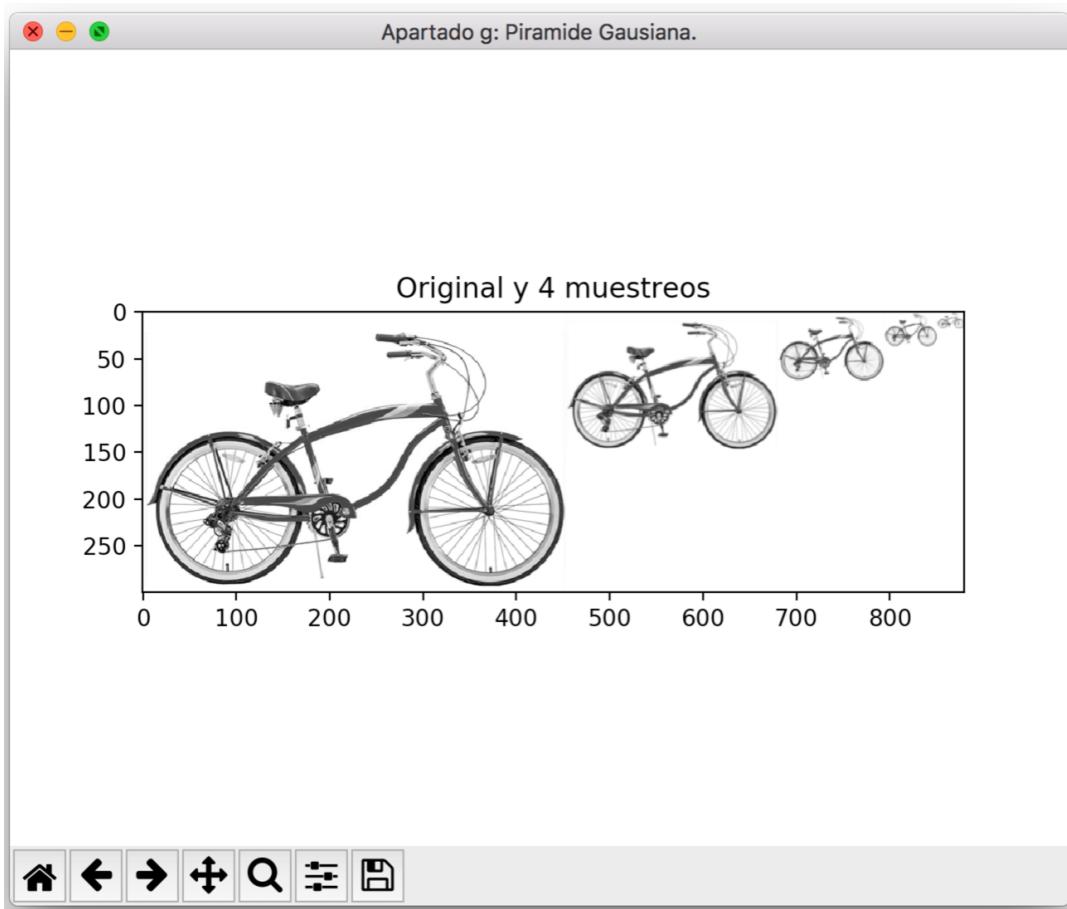
1.7. Una función que genere una representación en pirámide Gaussiana de 4 niveles de una imagen. Mostrar ejemplos de funcionamiento usando dos tipos de bordes y dos valores distintos de sigma.

La piramide Gaussiana nos permite simular como veríamos una imagen si nos fuésemos alejando de ella. Para poder crearla he usado la función *pyrDown* de *OpenCV*. Esta función aplica un filtro Gaussiano a la imagen y posteriormente hace un escalado, reduciendo así el tamaño a la mitad. Una vez tenemos las 4 imágenes que queremos (simplemente debemos hacer 4 llamadas a *pyrDown*) lo que he hecho es montar un collage para visualizar todas las imágenes (incluida la original) de forma conjunta.

```
1 def piramide_gausiana(img, n_imagenes=4, borde=0):
2     aux = copy.deepcopy(img)
3     salida = [img]
4
5     # Obtenemos las n_imagenes imágenes que necesitamos y las guardamos en el vector
6     # salida para luego montar el collage
7     for i in range(n_imagenes):
8         if borde == 0:
9             aux = cv2.pyrDown(src=aux)
10        else:
11            aux = cv2.pyrDown(src=aux, borderType=borde)
12
13    salida.append(aux)
14
15    columnas = 0
16    for elemento in salida:
17        columnas += np.shape(elemento)[1]
18
19    # Creamos el collage de forma manual
20    matriz_final = np.full((np.shape(img)[0], columnas, 3), 255, dtype=np.float64)
21    acumulador = 0
22    for elemento in salida:
23        for i in range(np.shape(elemento)[0]):
24            for j in range(np.shape(elemento)[1]):
25                matriz_final[i][j + acumulador] = elemento[i][j]
26
27    acumulador += np.shape(elemento)[1]
28
29    return matriz_final
```

El resultado lo podemos ver a continuación ¹:

¹La primera imagen mostrada es la original



1.8. 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.

Para esta función he usado las funciones *pyrDown* y *pyrUp*. La idea de la piramide Laplaciana es mostrar la información que se pierde tras muestrear una imagen y compararla con la original. Para ello vamos a aplicar *pyrDown* sobre la imagen I_1 para aplicarle un filtro gaussiano y escalarla y obtener la imagen I_2 , luego aplicamos *pyrUp* para obtener la imagen I_3 . Finalmente comparamos I_1 con I_3 y la diferencia es lo que representamos.

Debemos tener cuidado con los tamaños de las imágenes. Para no tener problemas al reducir y aumentar las imágenes, la imagen original se aumenta de tamaño hasta que el número de filas y columnas alcance la potencia de 2 más cercana (líneas 8-30) y tras el proceso se eliminan dichas filas y columnas extra para mostrar correctamente el resultado (líneas 47-52).

```

1 def piramide_laplaciana(img, n_imagenes=4, borde=0):
2     if len(np.shape(img)) == 3:
3         filas, columnas, _ = np.shape(img)

```

```

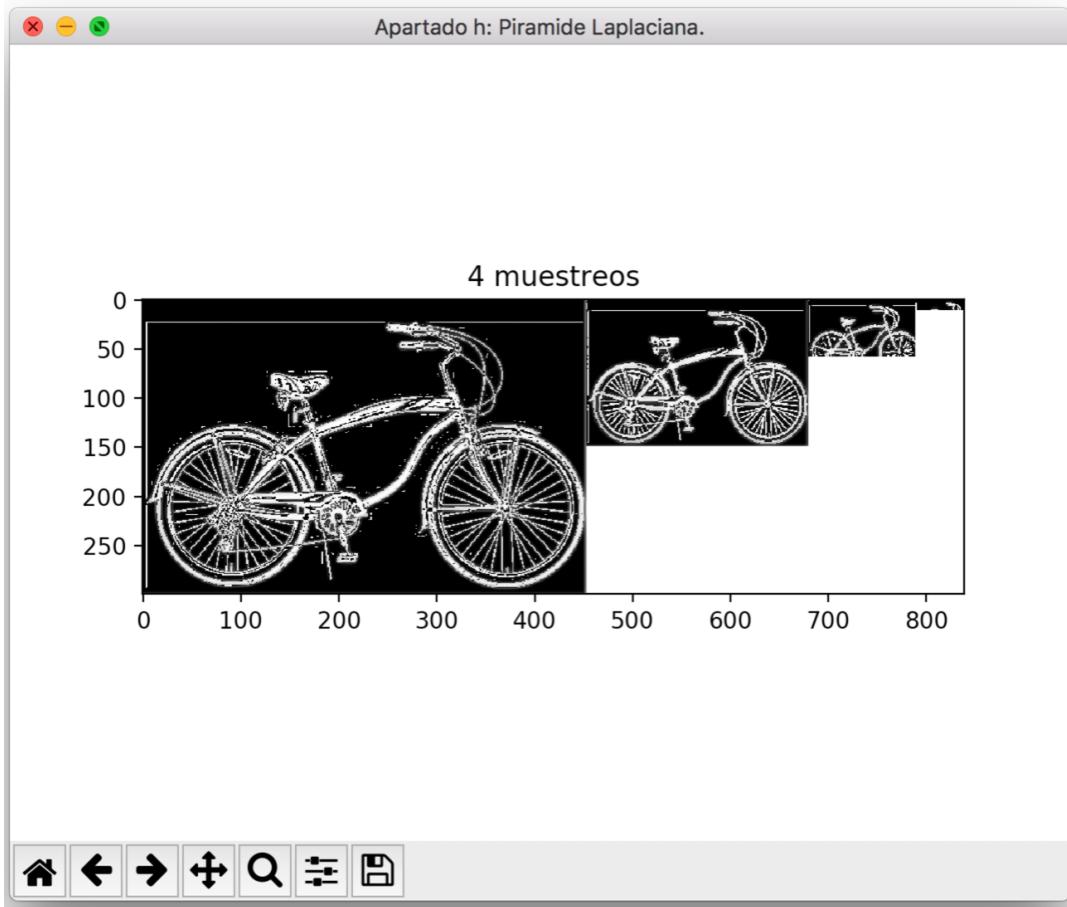
4     else:
5         filas , columnas = np.shape(img)
6
7     # Calculamos cuantas filas faltan hasta alcanzar la potencia de 2 más cercana
8     contador = 0
9     exponente = 0
10    while exponente < filas:
11        exponente = np.power(2, contador)
12        contador += 1
13
14    filas_restanes = exponente - filas
15
16    # Calculamos cuantas columnas faltan hasta alcanzar la potencia de 2 más cercana
17    contador = 0
18    exponente = 0
19    while exponente < columnas:
20        exponente = np.power(2, contador)
21        contador += 1
22
23    columnas_restanes = exponente - columnas
24
25    # Añadimos las columnas y filas restantes para no tener problemas con los escalados
26    img_nueva = np.zeros(shape=(filas + filas_restanes , columnas + columnas_restanes ,
27                          3), dtype=np.array(img).dtype)
28    for i in range(filas):
29        for j in range(columnas):
30            img_nueva[i][j] = img[i][j]
31
32    salida = []
33
34    # Calculamos la piramide Laplaciana
35    for i in range(n_imagenes):
36        if borde == 0:
37            down = cv2.pyrDown(img_nueva)
38            up = cv2.pyrUp(down, dstsize=(np.shape(img_nueva)[0], np.shape(img_nueva)
39                           [1]))
40            salida.append(img_nueva - up)
41            img_nueva = down
42        else:
43            down = cv2.pyrDown(img_nueva, borderType=borde)
44            up = cv2.pyrUp(down, borderType=borde, dstsize=(np.shape(img_nueva)[0], np.
45                           shape(img_nueva)[1]))
46            salida.append(img_nueva - up)
47            img_nueva = down
48
49    # Debemos borrar los zeros extras , para obtener un mejor resultado visual
50    salida_procesada = []
51    contador = 1
52    for elemento in salida:
53        salida_procesada.append(elemento[0:np.shape(elemento)[0] - (filas_restanes //
54                                         contador),
55                                         0:np.shape(elemento)[1] - (columnas_restanes //
56                                         contador)])
57        contador += 1
58
59    columnas = 0
60    for elemento in salida_procesada:
61        columnas += np.shape(elemento)[1]
62
63    # Creamos el collage
64    matriz_final = np.full((np.shape(salida_procesada[0])[0], columnas, 3), 255, dtype=
65                           np.float64)
66    acumulador = 0
67    for elemento in salida_procesada:
68        for i in range(np.shape(elemento)[0]):
69            for j in range(np.shape(elemento)[1]):
```

```

64     matriz_final[i][j + acumulador] = elemento[i][j]
65
66     acumulador += np.shape(elemento)[1]
67
68 return matriz_final

```

El resultado lo podemos ver a continuación:



Podemos ver mucha zona en negro, es decir, con valores 0. Esto representa que en esa zona la diferencia ha sido cero, es decir, que no se ha perdido información. El problema viene en la figura de la bicicleta, donde si podemos observar que se ha perdido más información (píxeles en un color distinto de negro, es decir, con un valor distinto de cero, lo cual representa que hay diferencia entre la imagen original y la filtrada).

2. Ejercicio 2: Imágenes híbridas.

Una imagen híbrida es una imagen compuesta por las frecuencias altas de una primera imagen junto con las frecuencias bajas de una segunda imagen. Para obtener las frecuencias bajas de una imagen debemos aplicar un filtro Gaussiano. En mi caso usaré la función *convolucion_separable* implementada anteriormente. Para obtener las frecuencias altas de una imagen debemos restarle a dicha imagen las frecuencias bajas de la

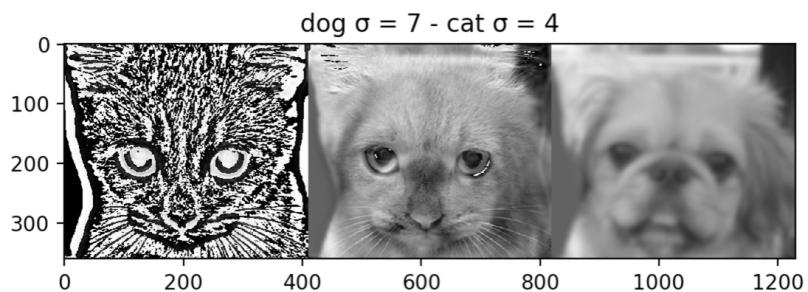
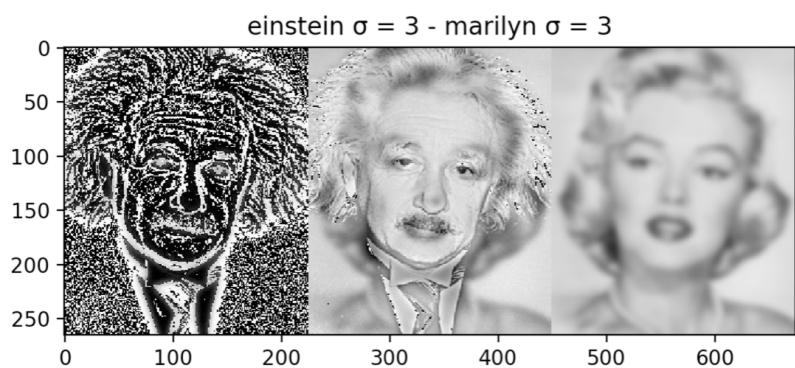
misma. La imagen híbrida es simplemente la suma de ambos rangos de frecuencias.

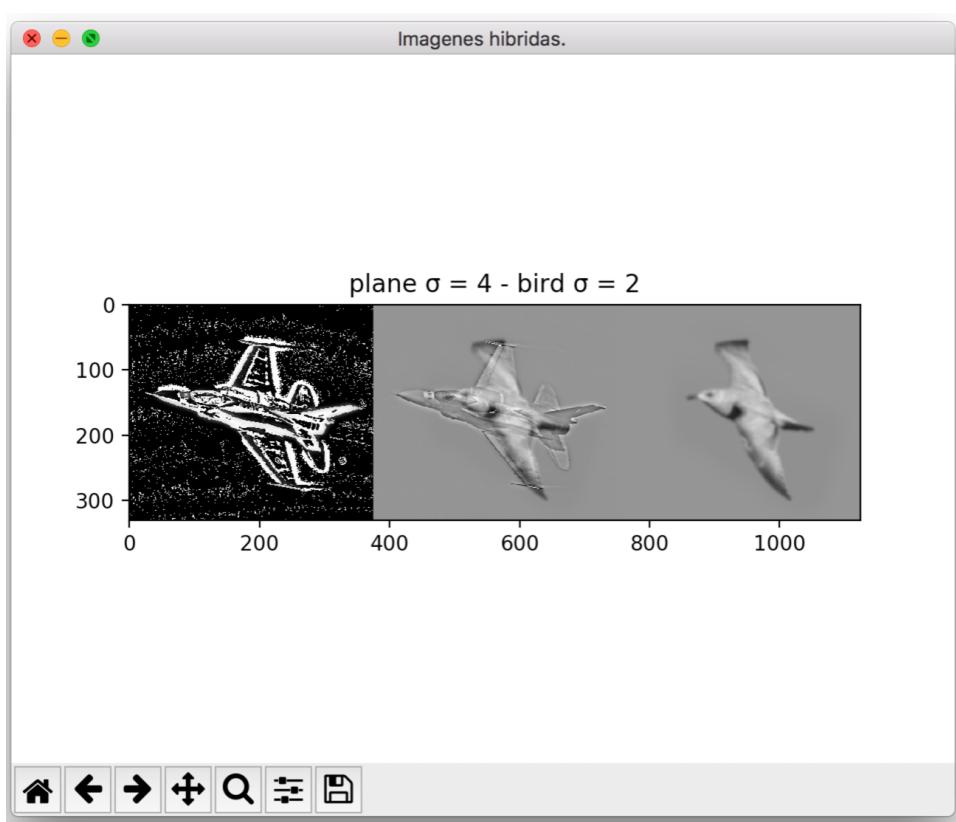
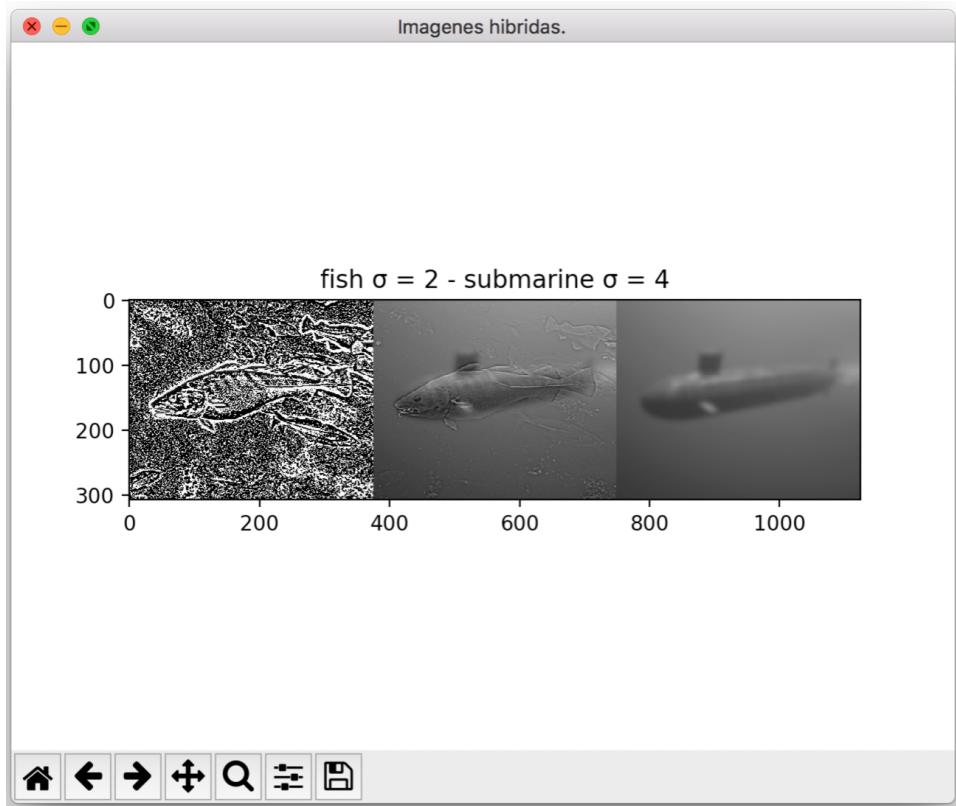
Una vez tenemos las tres imágenes, montamos un collage para poder visualizar las tres imágenes juntas.

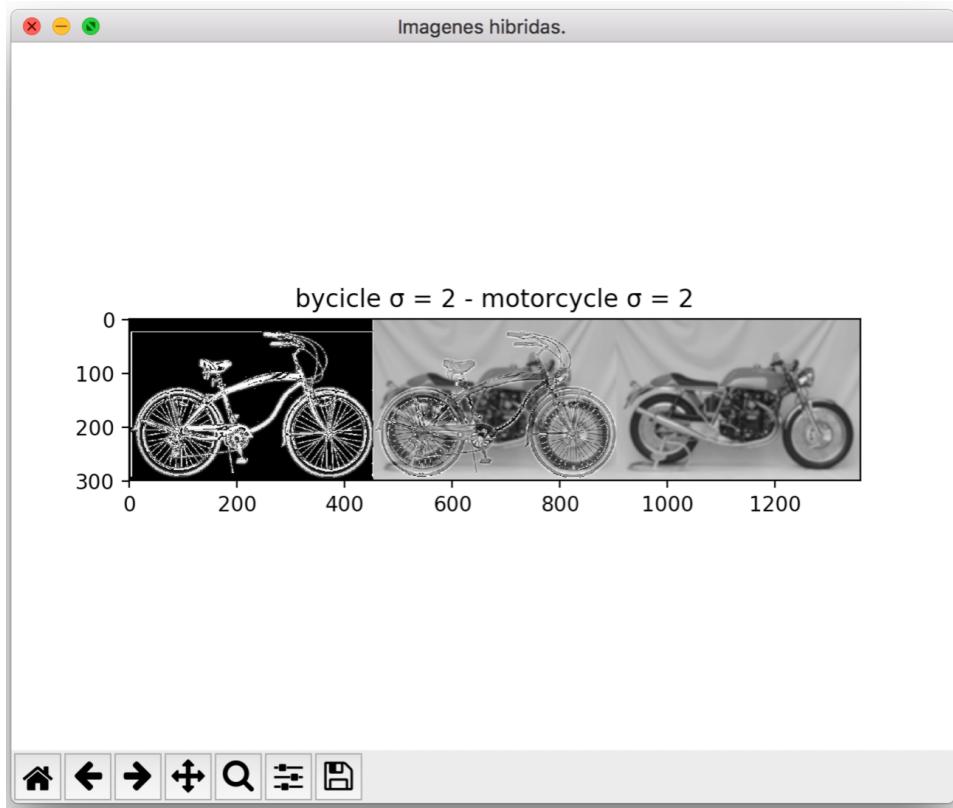
La idea de las imágenes híbridas es ir viendo como, según la cercanía con la que observes la imagen, verás una y otra de las imágenes que la componen.

```
1 def imagen_hibrida(img_1, img_2, sigma_1, sigma_2, tamano=3, borde=0, todas=False):
2     # Frecuencias altas = imagen - frecuencias bajas
3     frecuencias_altas = img_1 - convolucion_separable(img=img_1, sigma=sigma_1, tamano=tamano,
4                                                       borde=borde, normalize=False)
5     frecuencias_bajas = convolucion_separable(img=img_2, sigma=sigma_2, tamano=tamano,
6                                               borde=borde, normalize=False)
7
8     if todas:
9         matriz_final = np.full((np.shape(img_1)[0], np.shape(img_1)[1] * 3, 3), 255,
10                               dtype=np.float64)
11
12         imagenes = [frecuencias_altas, np.array(frecuencias_altas + frecuencias_bajas,
13                                                dtype=np.uint8),
14                     frecuencias_bajas]
15         acumulador = 0
16         for elemento in imagenes:
17             for i in range(np.shape(elemento)[0]):
18                 for j in range(np.shape(elemento)[1]):
19                     matriz_final[i][j + acumulador] = elemento[i][j]
20
21         acumulador += np.shape(elemento)[1]
22
23     return matriz_final
24 else:
25     return np.array(frecuencias_altas + frecuencias_bajas, dtype=np.uint8)
```

En las imágenes mostradas a continuación podemos ver la imagen de frecuencias altas a la izquierda y la imagen de frecuencias bajas a la derecha. En el centro podemos ver la imagen híbrida. Encima de las tres aparecen los sigma usados para cada imagen. Las imágenes híbridas obtenidas son las siguientes:







3. Bonus.

3.1. Cálculo del vector máscara Gaussiano: Sea

$f(x) = \exp\left(-0,5\frac{x^2}{\sigma^2}\right)$ una función donde σ (sigma) representa un parámetro en unidades píxel. Implementar una máscara de convolución 1D representativa de dicha función. Justificar los pasos dados.

Esta función devuelve un máscara Gausianna de $2*tamaño*sigma+1$ elementos. Primero inicializamos un vector, el cual usaremos para el cálculo de $f(x)$. La idea es discretizar la función. Como para nosotros $sigma$ representa un número de píxeles y la máscara representa un vector en tamaño de píxeles, si la máscara fuese de tamaño cero, el contenido del vector sería un único elemento, el 0. Si fuese de tamaño tres, sería el centro en 0 y el píxel a la izquierda y a la derecha, es decir, el -1, el 0 y el 1. Y así sucesivamente. Luego calculamos $f(x)$ para cada elemento y devolvemos el vector normalizado. Al normalizar el vector conseguimos que la suma del mismo de 1, lo cual es lo deseado. Y como $f(x)$ eleva los x al cuadrado, nuestro vector tampoco tendrá valores negativos, que también es lo deseado.

```
1 def mascara_gausiana(sigma, tamaño=3):
2     def fx(x, sigma):
```

```

3     return math.exp(-0.5 * (x ** 2 / sigma ** 2))
4
5 kernel = list(range(-tamanio * sigma, tamanio * sigma + 1))
6 kernel = np.array([fx(m, sigma) for m in kernel])
7 # Lo devolvemos normalizado
8 return np.divide(kernel, np.sum(kernel))

```

3.2. Implementar una función que calcule la convolución de un vector señal 1D con un vector-máscara de longitud interior al de la señas usando condiciones de contorno reflejada. La salida será un vector de igual longitud que el vector señal de entrada.

Esta función recibe la señal y el kernel a convolucionar. Lo primero que hacemos es añadir los bordes reflejados para no tener problemas al convolucionar los píxeles iniciales y finales de la señal (para eso usamos la función *aniadir_borde_reflejado*). Una vez tenemos añadido el borde, sólo debemos pasear el kernel por todo el vector señal, calculando en cada paso el píxel en el cual está centrado el kernel. Una vez lo hemos recorrido entero, devolvemos el vector señal pero sin los bordes.

```

1 def aniadir_borde_reflejado(src, espacio):
2     # Cogemos el trozo izquierdo a reflejar
3     izquierda = src[0:espacio]
4     # Cogemos el trozo derecho a reflejar
5     derecha = (src[-espacio:])[::-1]
6
7     # Concatenamos los 3 trozos y los devolvemos
8     return np.append(np.append(izquierda, src, axis=0), derecha, axis=0)
9
10
11 def convolucion_1d(signal, kernel):
12     signal_bordes = aniadir_borde_reflejado(src=signal, espacio=len(kernel) // 2)
13
14     signal_bordes = signal_bordes.flatten()
15     salida = np.zeros(shape=len(signal_bordes))
16
17     # Paseamos la máscar por el vector señal pero sin contar sus bordes
18     for i in range(len(kernel) // 2, len(salida) - len(kernel) // 2):
19         salida[i] = np.sum(
20             np.multiply(kernel, signal_bordes[i - len(kernel) // 2:i + len(kernel) // 2
21                         + 1]))
22
23     return salida[len(kernel) // 2:-len(kernel) // 2 + 1]

```

3.3. 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.

Esta función sigue la misma filosofía que la implementada en el apartado 3 del primer ejercicio sólo que cuando antes se llamaba a la función *filter2D* ahora hay que llamar a mi función *convolucion_1d*. El código es el siguiente:

```

1 def convolucion_separable_manual(img, sigma, kernel=None, tamanio=3, normalize=False):
2     if kernel is None:
3         kernel = mascara_gausiana(sigma=sigma, tamanio=tamanio)
4
5     kernel = kernel.flatten()
6
7     if len(np.shape(img)) == 3:
8         canal_1, canal_2, canal_3 = cv2.split(img)
9
10    auxiliar_1 = np.zeros(shape=np.shape(canal_1), dtype=np.float64)
11    auxiliar_2 = np.zeros(shape=np.shape(canal_2), dtype=np.float64)
12    auxiliar_3 = np.zeros(shape=np.shape(canal_3), dtype=np.float64)
13
14    filas = np.shape(auxiliar_1)[0]
15
16    for i in range(0, filas):
17        auxiliar_1[i] = convolucion_1d(signal=canal_1[i], kernel=kernel)
18        auxiliar_2[i] = convolucion_1d(signal=canal_2[i], kernel=kernel)
19        auxiliar_3[i] = convolucion_1d(signal=canal_3[i], kernel=kernel)
20
21    auxiliar_1 = np.array(auxiliar_1).transpose()
22    auxiliar_2 = np.array(auxiliar_2).transpose()
23    auxiliar_3 = np.array(auxiliar_3).transpose()
24
25    salida_1 = np.zeros(shape=np.shape(auxiliar_1), dtype=np.float64)
26    salida_2 = np.zeros(shape=np.shape(auxiliar_2), dtype=np.float64)
27    salida_3 = np.zeros(shape=np.shape(auxiliar_3), dtype=np.float64)
28
29    filas = np.shape(auxiliar_1)[0]
30
31    for i in range(0, filas):
32        salida_1[i] = convolucion_1d(signal=auxiliar_1[i], kernel=kernel)
33        salida_2[i] = convolucion_1d(signal=auxiliar_2[i], kernel=kernel)
34        salida_3[i] = convolucion_1d(signal=auxiliar_3[i], kernel=kernel)
35
36    salida_1 = np.array(salida_1).transpose()
37    salida_2 = np.array(salida_2).transpose()
38    salida_3 = np.array(salida_3).transpose()
39
40    if normalize:
41        salida_1 = (salida_1 - np.amin(salida_1)) / (np.amax(salida_1) - np.amin(
42            salida_1))
43        salida_2 = (salida_2 - np.amin(salida_2)) / (np.amax(salida_2) - np.amin(
44            salida_2))
45        salida_3 = (salida_3 - np.amin(salida_3)) / (np.amax(salida_3) - np.amin(
46            salida_3))
47
48    salida = cv2.merge([salida_1, salida_2, salida_3])
49
50    return salida
51 else:
52     auxiliar_1 = np.zeros(shape=np.shape(img), dtype=np.float64)
53
54     filas = np.shape(auxiliar_1)[0]
55
56     for i in range(0, filas):
57         auxiliar_1[i] = convolucion_1d(signal=img[i], kernel=kernel)
58
59     auxiliar_1 = np.array(auxiliar_1).transpose()
60
61     salida_1 = np.zeros(shape=np.shape(auxiliar_1), dtype=np.float64)
62
63     filas = np.shape(auxiliar_1)[0]
64
65     for i in range(0, filas):
66         salida_1[i] = convolucion_1d(signal=auxiliar_1[i], kernel=kernel)

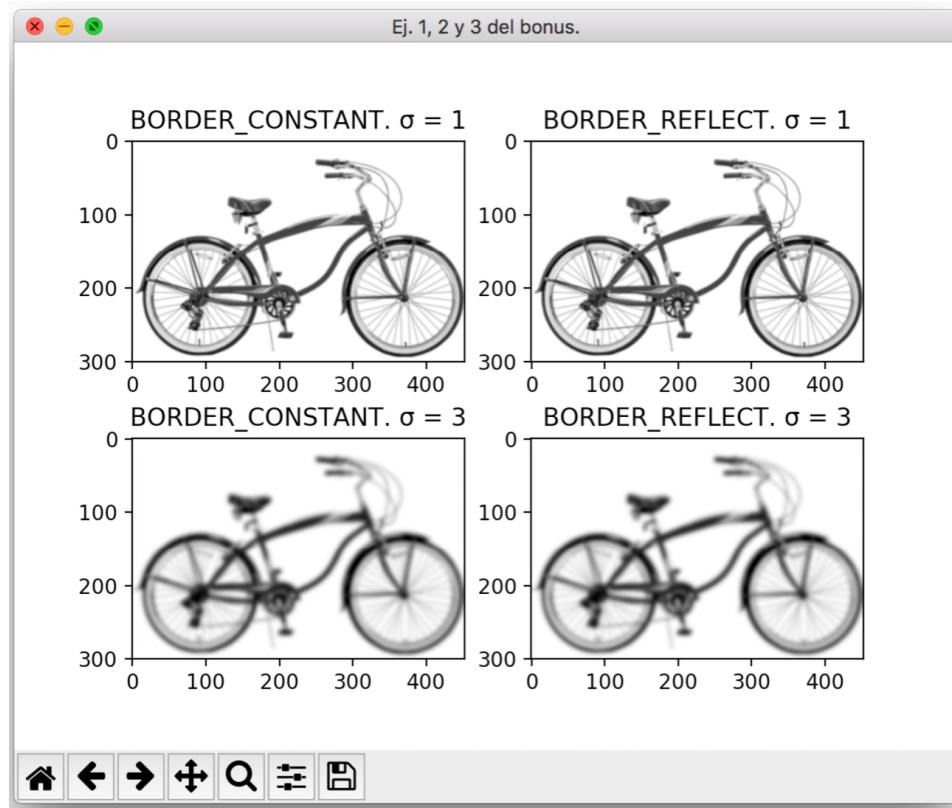
```

```

64     salida = salida_1.transpose()
65     if normalize:
66         salida = (salida - np.amin(salida)) / (np.amax(salida) - np.amin(salida))
67
68
69     return salida

```

El resultado es el siguiente:



Como podemos ver obtenemos el mismo resultado que aplicando las funciones de *OpenCV*.

3.4. 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 en todas las funciones.

Para construir la pirámide Gaussiana se ha usado la misma función que en el ejercicio correspondiente de la parte obligatoria pero cambiando las funciones que eran llamadas. El código es el siguiente:

```

1 def piramide_gausiana_manual(img, n_imagenes=5):
2     aux = copy.deepcopy(img)
3     salida = [img]

```

```

4
5     for i in range(n_imagenes):
6         aux = pyrdown_manual(src=aux)
7         salida.append(aux)
8
9     columnas = 0
10    for elemento in salida:
11        columnas += np.shape(elemento)[1]
12
13    matriz_final = np.full((np.shape(img)[0], columnas, 3), 255, dtype=np.float64)
14    acumulador = 0
15    for elemento in salida:
16        for i in range(np.shape(elemento)[0]):
17            for j in range(np.shape(elemento)[1]):
18                matriz_final[i][j + acumulador] = elemento[i][j]
19
20    acumulador += np.shape(elemento)[1]
21
22    return matriz_final

```

Antes se llamaba a la función *pyrDown* de *OpenCV* y ahora se ha implementado la función a mano:

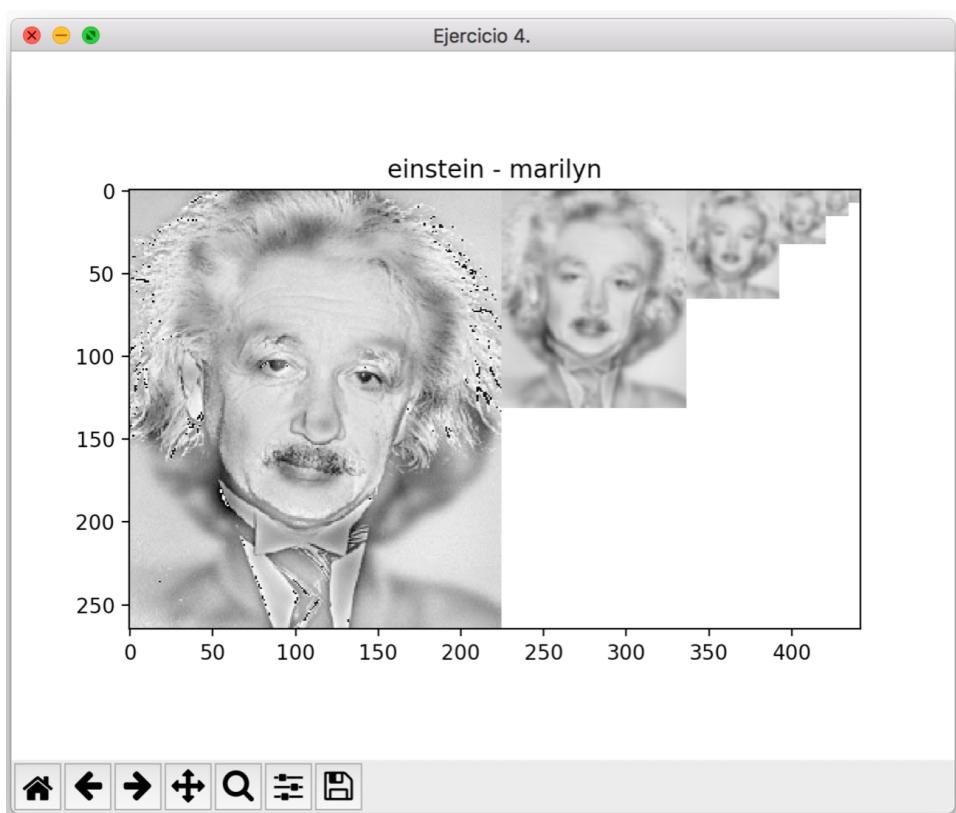
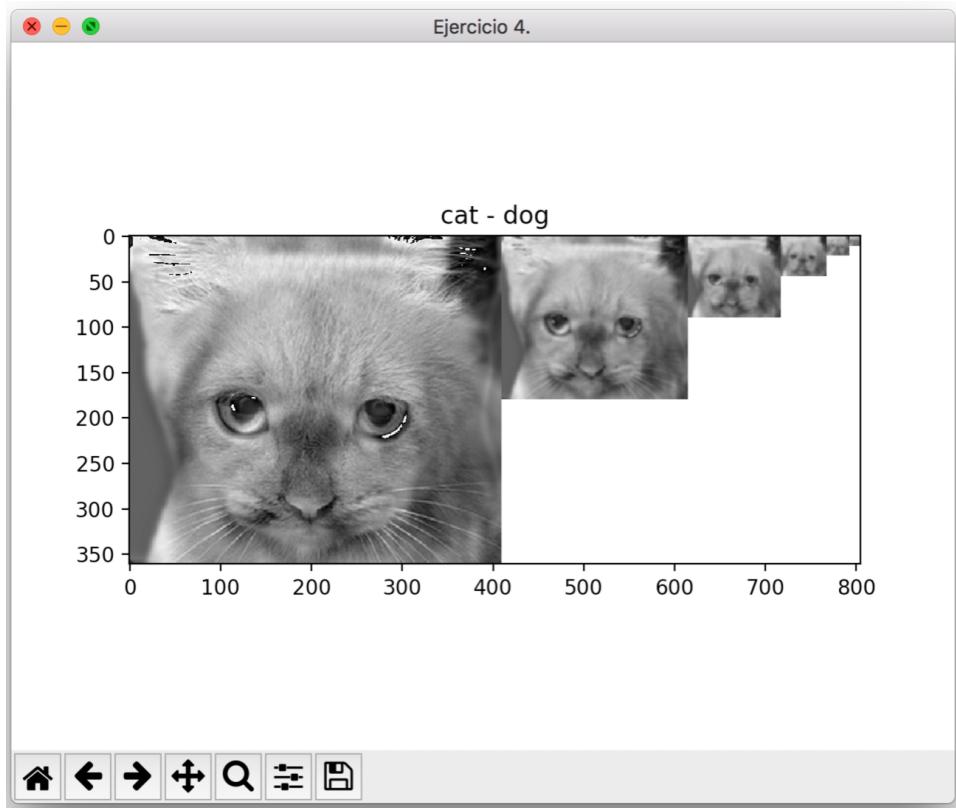
```

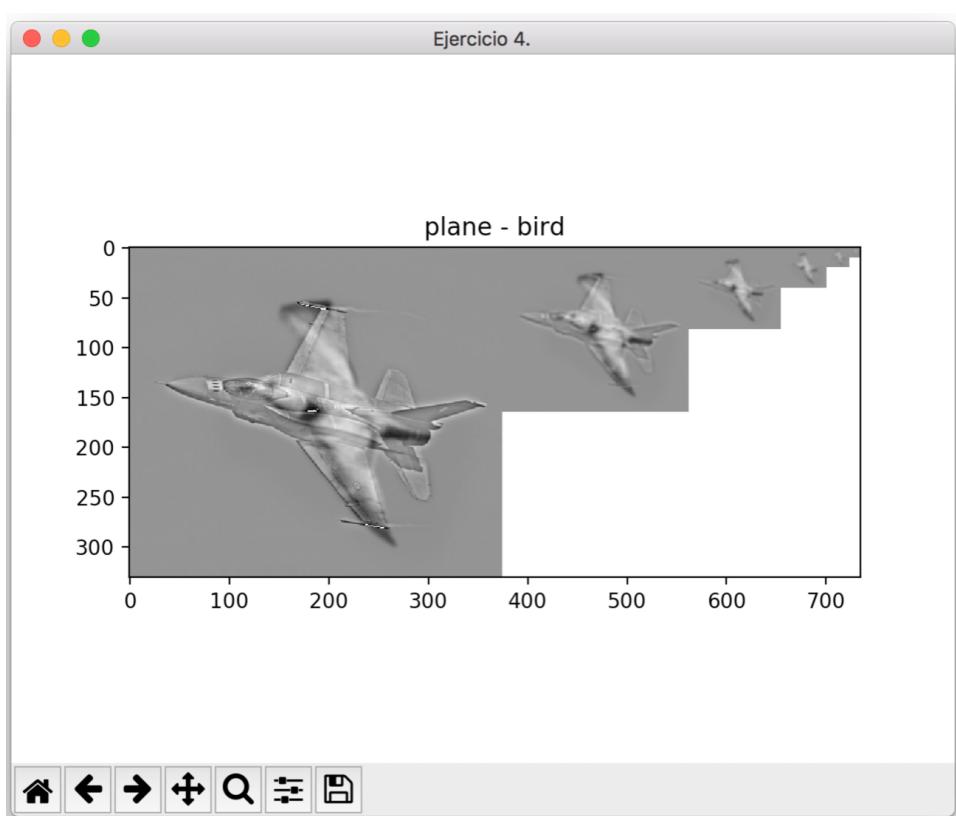
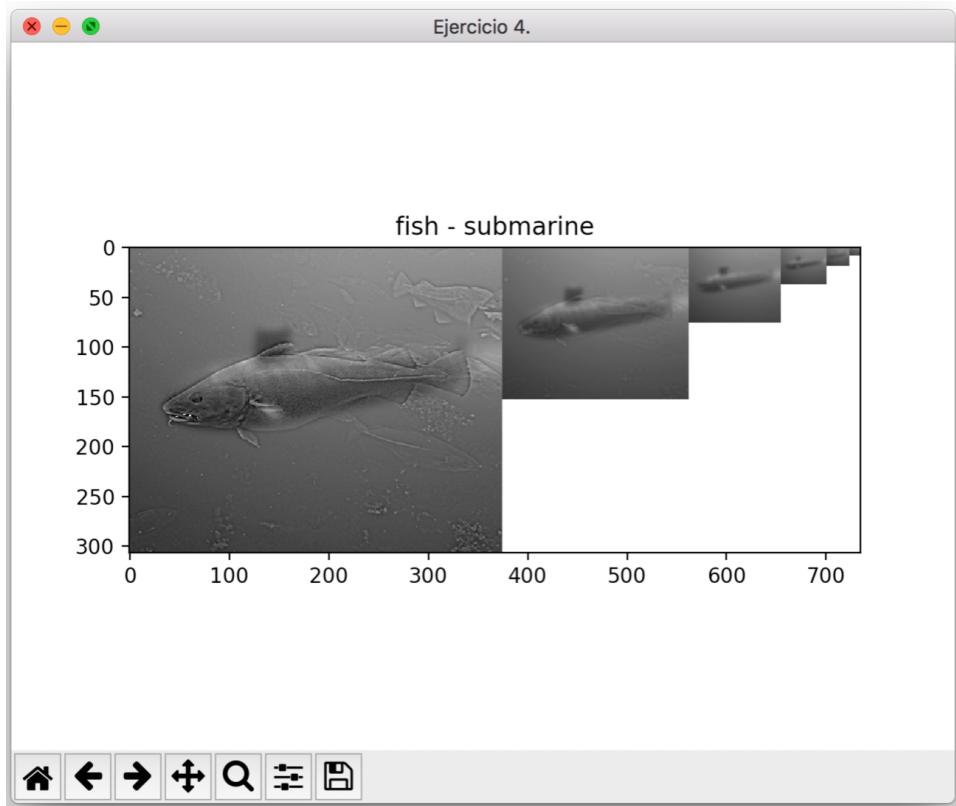
1 def pyrdown_manual(src):
2     src_filtro = convolucion_separable_manual(img=src, sigma=2, kernel=None, tamano=3,
3                                                 normalize=False)
4     # Devolvemos sólo las columnas impares
5     return src_filtro[1::2, 1::2]

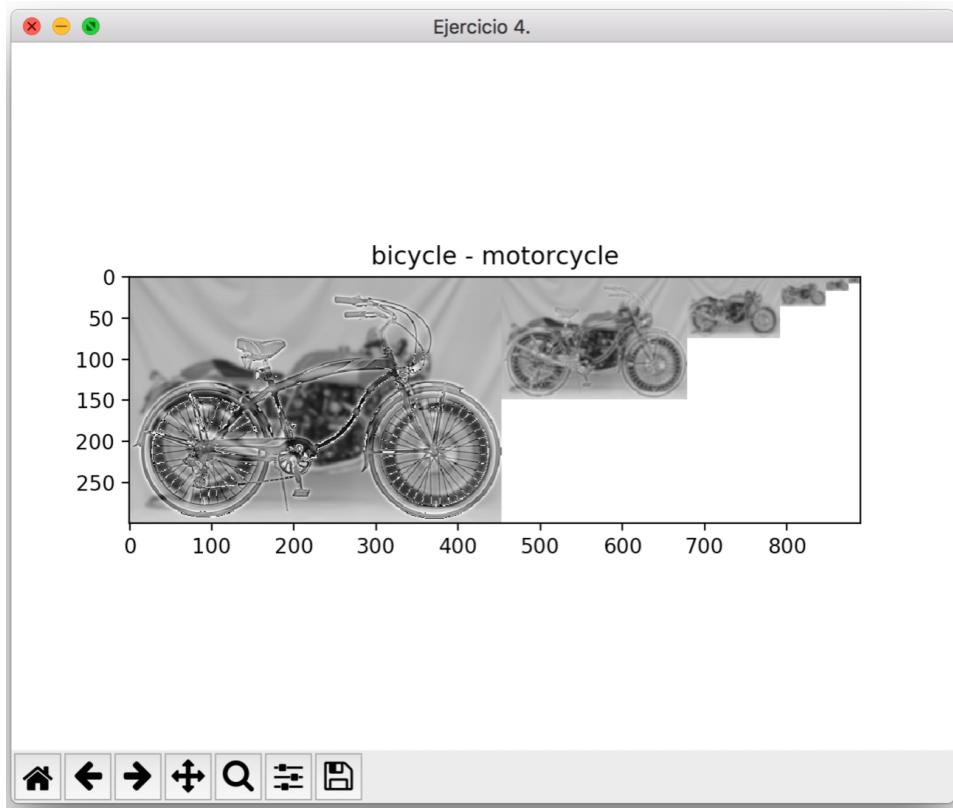
```

Lo único que hay que hacer es aplicar un alisamiento con un filtro Gaussiano y luego un escalado. Para el filtro llamamos a la función implementada anteriormente en el bonus y para el escalado simplemente devolvemos las filas y las columnas impares (línea 3), tal y como hace *pyrDown*.

Las pirámides Gaussianas para las imágenes híbridas son las siguientes:







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

La función desarrollada es igual que la función de la parte obligatoria, sólo que cambia la función de convolución usada para no usar las funciones de *OpenCV*. Debemos recordar que todas las funciones implementadas están preparadas para tratar imágenes con 1 y 3 bandas, por lo tanto no ha hecho falta realizar ningún cambio adicional para obtener las imágenes híbridas correspondientes.

Hasta ahora las imágenes habían sido leídas de la siguiente forma:

```
1 img = cv2.imread('nombre_imagen', cv2.IMREAD_GRAYSCALE)
```

El flag *cv2.IMREAD_GRAYSCALE* nos indica que queremos una única banda, para hacer las imágenes en escala de grises. Para realizar las imágenes híbridas debemos quitar este flag.

```
1 def imagen_hibrida_manual(img_1, img_2, sigma_1, sigma_2, tamano=3, todas=False):
2     # Frecuencias altas = imagen - frecuencias bajas
3     frecuencias_altas = img_1 - convolucion_separable_manual(img=img_1, sigma=sigma_1,
4                                                               tamano=tamano,
5                                                               normalize=False)
6     frecuencias_bajas = convolucion_separable_manual(img=img_2, sigma=sigma_2, tamano=
7                                                       tamano, normalize=False)
8
9     if todas:
```

```

8     matriz_final = np.full((np.shape(img_1)[0], np.shape(img_1)[1] * 3, 3), 255,
9         dtype=np.float64)
10    imagenes = [frecuencias_altas, np.array(frecuencias_altas + frecuencias_bajas,
11        dtype=np.uint8),
12        frecuencias_bajas]
13    acumulador = 0
14    for elemento in imagenes:
15        for i in range(np.shape(elemento)[0]):
16            for j in range(np.shape(elemento)[1]):
17                matriz_final[i][j + acumulador] = elemento[i][j]
18
19    acumulador += np.shape(elemento)[1]
20
21    return matriz_final
22 else:
23     return np.array(frecuencias_altas + frecuencias_bajas, dtype=np.uint8)

```

Las imágenes híbridas son las siguientes:

