

---

Curso: 2019 - 2020

# Práctica 1

Filtrado y detección de regiones

José Javier Alonso Ramos



**UNIVERSIDAD  
DE GRANADA**

## Índice

Bibliotecas . . . . .	3
<b>Funciones de lectura y mostrado de imágenes</b>	<b>3</b>
Función para imprimir imágenes de opencv con matplot . . . . .	3
Función para la lectura de imágenes . . . . .	4
<b>Funciones que implementan el funcionamiento de los distintos ejercicios</b>	<b>4</b>
Gaussiana 1D . . . . .	4
Gaussiana 2D . . . . .	5
Sobel (getDerivKernels) . . . . .	5
Laplaciana de Gaussiana . . . . .	5
Subsample . . . . .	6
Upsample . . . . .	6
Pyramid . . . . .	6
Hibridación . . . . .	7
Imprime híbrida . . . . .	7
Blob detection . . . . .	8
Dibuja círculos . . . . .	8
<b>Ejercicio 1.</b>	<b>9</b>
1a. Máscara Gaussiana aplicada con máscaras 1D y 2D comparadas con la función Gaussian-Blur. Máscara de Sobel (derivadas - resaltado de bordes) en x e y . . . . .	9
1b. Cálculo de una convolución 2D con una máscara <i>Laplaciana de Gaussiana</i> . . . . .	30
<b>Ejercicio 2.</b>	<b>42</b>
2a. Pirámide Gaussiana de 4 niveles . . . . .	42
2b. Pirámide Laplaciana de 4 niveles . . . . .	45
2c. Blob Detection . . . . .	48
<b>Ejercicio 3.</b>	<b>53</b>
3. Imágenes Híbridas . . . . .	53
<b>Bonus.</b>	<b>60</b>
Bonus3. Hibridación con imágenes propias . . . . .	60

## Bibliotecas

```
1 import sys
2 import cv2
3 import matplotlib.pyplot as plt
4 import numpy as np
```

## Funciones de lectura y mostrado de imágenes

### Función para imprimir imágenes de opencv con matplotlib

Deja preparada la imagen *im*, ya sea a color o en escala de grises, para que al hacer *plt.show()* la muestre por pantalla con el título *title*. Normaliza → convierte a una escala de color reconocible → muestra

```
1 def plt_imshow(im, title = ''):
2
3     # NORMALIZAMOS
4     # Si es a color
5     if len(im.shape) == 3:
6         im[:, :, 0] = (im[:, :, 0] - np.min(im[:, :, 0])) / (np.max(im
7             [:, :, 0]) - np.min(im[:, :, 0]))*255
8         im[:, :, 1] = (im[:, :, 1] - np.min(im[:, :, 1])) / (np.max(im
9             [:, :, 1]) - np.min(im[:, :, 1]))*255
10        im[:, :, 2] = (im[:, :, 2] - np.min(im[:, :, 2])) / (np.max(im
11            [:, :, 2]) - np.min(im[:, :, 2]))*255
12
13    # SI es en ByN normalizamos la imagen entera (solo tiene un canal)
14    else:
15        im[:, :] = (im[:, :] - np.min(im[:, :])) / (np.max(im[:, :]) - np.
16            min(im[:, :]))*255
17
18    # Si la imagen es a color
19    if len(im.shape) == 3:
20        im_plt = cv2.cvtColor(im, cv2.COLOR_BGR2RGB) # La cambiamos a
21            RGB
22        plt.imshow(im_plt) # mostramos
23
24    else: # Si es gris
25        plt.imshow(im, cmap='gray') # Mostramos en escala de grises
26        plt.title(title) # añadimos un título
```

## Función para la lectura de imágenes

Recibe como parámetro el *path* hasta la imagen y el modo de color de opencv a emplear. Devuelve la imagen en *float64* para trabajar con ella.

```
1 def leer_imagen(path, color = None):
2
3     # Si queremos leer la imagen en BGR
4     if color == None:
5         im_cv = cv2.imread(path)
6
7     # Si queremos leer la imagen en ByN
8     elif color == 0:
9         im_cv = cv2.imread(path, 0)
10
11    # Devolvemos la imagen para cv y para plt
12    return np.float64(im_cv)
```

## Funciones que implementan el funcionamiento de los distintos ejercicios

Para no sobrecargar la memoria con código solo se muestran las cabeceras de las funciones para ver los parámetros que tienen y, también, para que sea más fácil buscarlas en el archivo de código.

### Gaussiana 1D

Obtenemos el kernel de una función Gaussiana de tamaño *kernel\_size*,  $\sigma = \text{sigma}$  y bordes de tipo *borde* y lo aplicamos por filas y columnas a la imagen pasada como parámetro.

```
1 def Gaussiana1D(im, kernel_size, sigma, borde):
2
3     ...
4
5     return im2
```

## Gaussiana 2D

Obtenemos el kernel de una función Gaussiana de tamaño  $kernel\_size$ ,  $\sigma = sigma$  y bordes de tipo  $borde$ . Tras esto multiplicamos el kernel obtenido por su traspuesta para obtener una máscara 2D para aplicarla a la imagen.

```
1 def Gaussiana2D(im, kernel_size, sigma, borde):  
2  
3     ...  
4  
5     return im2
```

## Sobel (getDerivKernels)

Aplicamos Sobel de tamaño  $kernel\_size$  en  $x$  e  $y$  a la imagen  $im$  con configuración de bordes =  $borde$ . Sobel suaviza la imagen y después calcula su derivada en un determinado eje con la intención de detectar los bordes de la imagen o, más concretamente, los cambios de frecuencia (intensidad de color).

Obtenemos los kernels de ambos ejes y los multiplicamos matricialmente para obtener una máscara 2D la cual tendremos que voltear en ambos ejes para que al aplicarla realicemos una convolución. Tras esto aplicamos estas máscaras por separado obteniendo dos imágenes una con un resalto de bordes en el eje horizontal y otra en el vertical.

```
1 def gradiente(im, kernel_size, borde):  
2  
3     ...  
4  
5     return im2, im3
```

## Laplaciana de Gaussiana

La Laplaciana de Gaussiana es lo mismo que la segunda derivada de la Gaussiana.

Recibe como parámetro una imagen  $im$  a la que aplicar el filtro, un tamaño de máscara  $kernel\_size$ , un valor de  $\sigma = sigma$  para calcular la Gaussiana y un tipo de  $borde$  a aplicar.

Primero obtenemos la gaussiana de la imagen y después la derivamos en ambos ejes obteniendo una derivada en el eje  $x$  y otra derivada en el eje  $y$ . La imagen que queremos es la suma de estas dos.

```
1 def LoG(im, kernel_size, sigma, borde):  
2  
3     ...  
4  
5     return img
```

## Subsample

Reducimos la imagen pasada como parámetro suprimiendo las filas y columnas pares.

```
1 def imageSubsample(im):  
2  
3     ...  
4  
5     return im2
```

## Upsample

Aumentamos el tamaño de la imagen pasada como argumento insertando cada fila y columna dos veces

```
1 def imageUpsample(im):  
2  
3     ...  
4  
5     return im2
```

## Pyramid

Recibe como parámetro un vector de 5 imágenes de manera que  $v[i+1]$  es el subsample de  $v[i]$ . Crea una imagen en forma de “pirámide” compuesta por las imágenes del vector  $v\_img$  con la intención de representarlas de una forma más vistosa y útil que permite compararlas.

En primer lugar crea una serie de imágenes en blanco que actuarán como fondo de las pasadas como parámetro y las complementan para formar una nueva imagen de un tamaño específico que encaje en la composición de la pirámide. Todo el algoritmo está dedicado a ajustar las imágenes con su fondo correspondiente y a concatenarlas entre sí para que al final tenga forma de “pirámide”.

```
1 def pyramid(v_img):  
2  
3     ...  
4  
5     return final
```

## Hibridación

Recibe como parámetro las dos imágenes a hibridar ( $im1, im2$ ), así como el tamaño de máscara y el valor de sigma ( $kernel1, kernel2, sigma1, sigma2$ ) que se utilizará para calcular la gaussiana de cada una de ellas.

A la primera imagen solo le calcularemos su Gaussiana, mientras que a la segunda le calcularemos su Gaussiana únicamente como paso intermedio para obtener las frecuencias altas. Una vez tenemos las frecuencias bajas de la primera imagen (su Gaussiana) y las frecuencias altas de la segunda (original - su Gaussiana) sumamos las dos imágenes resultantes para obtener la imagen híbrida. Además, se devuelve junto con la imagen híbrida, una imagen que muestra las dos imágenes procesadas por separado junto con la híbrida.

```
1 def hybrid(im1, im2, kernel1, kernel2, sigma1, sigma2):  
2  
3     ...  
4  
5     return fin, hyb
```

## Imprime híbrida

Recibe como parámetro los mismos argumentos que la función *hybrid* ya que serán estos mismos los que serán pasados.

Esta función solo se encarga de leer las dos imágenes que se le pasarán como argumento a la función *hybrid*, llamar a dicha función y mostrar el resultado.

```
1 def imprimeHybrid(im1, im2, k1, k2, s1, s2):  
2  
3     ...  
4  
5     return hyb
```

## Blob detection

Recibe como parámetros el número de escalas a realizar (cuántas veces modificaremos sigma), la imagen en la que queremos encontrar los *blobs*, un valor inicial de sigma, un tamaño para el kernel y un umbral a partir del cual consideraremos importante el blob detectado.

Creamos una imagen *fin* de 0s del mismo tamaño que la original

Para cada una de las escalas:

- Calcularemos la Laplaciana de Gaussiana de la imagen, la normalizaremos multiplicándola por  $\sigma^2$  y elevaremos el resultado al cuadrado.
- Creamos una matriz (imagen) a la que llamaremos Z del mismo tamaño que la original con todos sus valores a 0 (negro) en la que iremos guardando los blobs detectados.
- Recorremos la Laplaciana de Gaussiana pixel a pixel comprobando que sea el máximo de su vecindario  $3 \times 3$  (es un blob). Si lo es, guardamos ese pixel en la matriz de 0s. - Una vez hayamos detectado los blobs de la imagen en esa escala de  $\sigma$ , normalizamos Z y dibujamos un círculo en todos aquellos blobs que superen el valor de umbral pasado. - Sumamos *fin* + Z - Aumentamos el valor de sigma  $\sigma = 14 * \sigma$

Devolvemos la imagen *fin* que contiene todos los círculos de las diferentes escalas que marcan zonas de interés en la imagen.

```
1 def blob(escalas, im, sigma, kernel, umbral):  
2  
3     ...  
4  
5     return fin
```

## Dibuja círculos

Toma como parámetros una imagen *z* en la que se pintarán los círculos, un umbral *u* que marcará en qué puntos se dibujan los círculos, y un sigma *s* que marcará el tamaño de esos círculos.

Primero nos hacemos con las columnas y filas en las que los valores de la imagen superan el umbral, agrupamos los valores por parejas para pasárselo a la función *circle* a la que le pasaremos también la imagen *z*, el tamaño del radio que irá definido por el valor de sigma (lo multiplicamos por 15 para que tome un tamaño visible) y el color del que colorearemos el círculo, en este caso blanco (255).

```
1 def drawCircles(z, u, s):  
2  
3     ...
```

## Ejercicio 1.

### 1a. Máscara Gaussiana aplicada con máscaras 1D y 2D comparadas con la función GaussianBlur. Máscara de Sobel (derivadas - resaltado de bordes) en x e y

#### ■ Qué vamos a calcular

Vamos a aplicar un filtro de Gaussiana a una imagen de tres maneras distintas: mediante máscaras de una dimensión, máscaras de dos dimensiones y con la función propia de OpenCV *GaussianBlur()*. También aplicaremos el filtro de Sobel en ambos ejes de la imagen.

#### ■ Cómo lo hacemos

Los filtros de Sobel los aplicamos con la función *gradiente()*. Para aplicar la Gaussiana utilizaremos las funciones *Gaussiana1D()*, *Gaussiana2D()* y *cv2.GaussianBlur()*.

#### ■ Funcionamiento de *gradiente()*

Esta función suaviza la imagen y después calcula su derivada en ambos ejes con la intención de detectar los bordes de la imagen o, más concretamente, los cambios de frecuencia (intensidad de color). En ella obtenemos dos parejas de kernels (una pareja para cada eje) y multiplicamos los miembros de las parejas matricialmente para obtener dos máscaras 2D las cuales tendremos que voltear en ambos ejes para que al aplicarlas realicemos una convolución. Tras esto aplicamos estas máscaras por separado obteniendo dos imágenes una con un resalto de bordes en el eje horizontal y otra en el vertical.

#### ■ Funcionamiento *Gaussiana1D()*

Obtenemos el kernel de una función Gaussiana de tamaño *kernel\_size*,  $\sigma = \text{sigma}$  y bordes de tipo *borde* y lo aplicamos una vez por filas y otra vez por columnas a la imagen pasada como parámetro.

#### ■ Funcionamiento *Gaussiana2D()*

Obtenemos el kernel de una función Gaussiana de tamaño *kernel\_size*,  $\sigma = \text{sigma}$  y bordes de tipo *borde*. Tras esto multiplicamos el kernel obtenido por su traspuesta para obtener una máscara 2D para así aplicarla una única vez a la imagen.

#### ■ Parámetros

>**kernels = [3,15,31]**: El tamaño máximo de máscara en las funciones Gaussianas es 31 por lo que se intenta escoger valores que representen todo el dominio viable pero sin que sean demasiados como para que haya muchas ejecuciones.

**k\_size = [1,5,7]**: Tenemos algo parecido al caso anterior salvo que el valor máximo en la función *gradiente()* es 7.

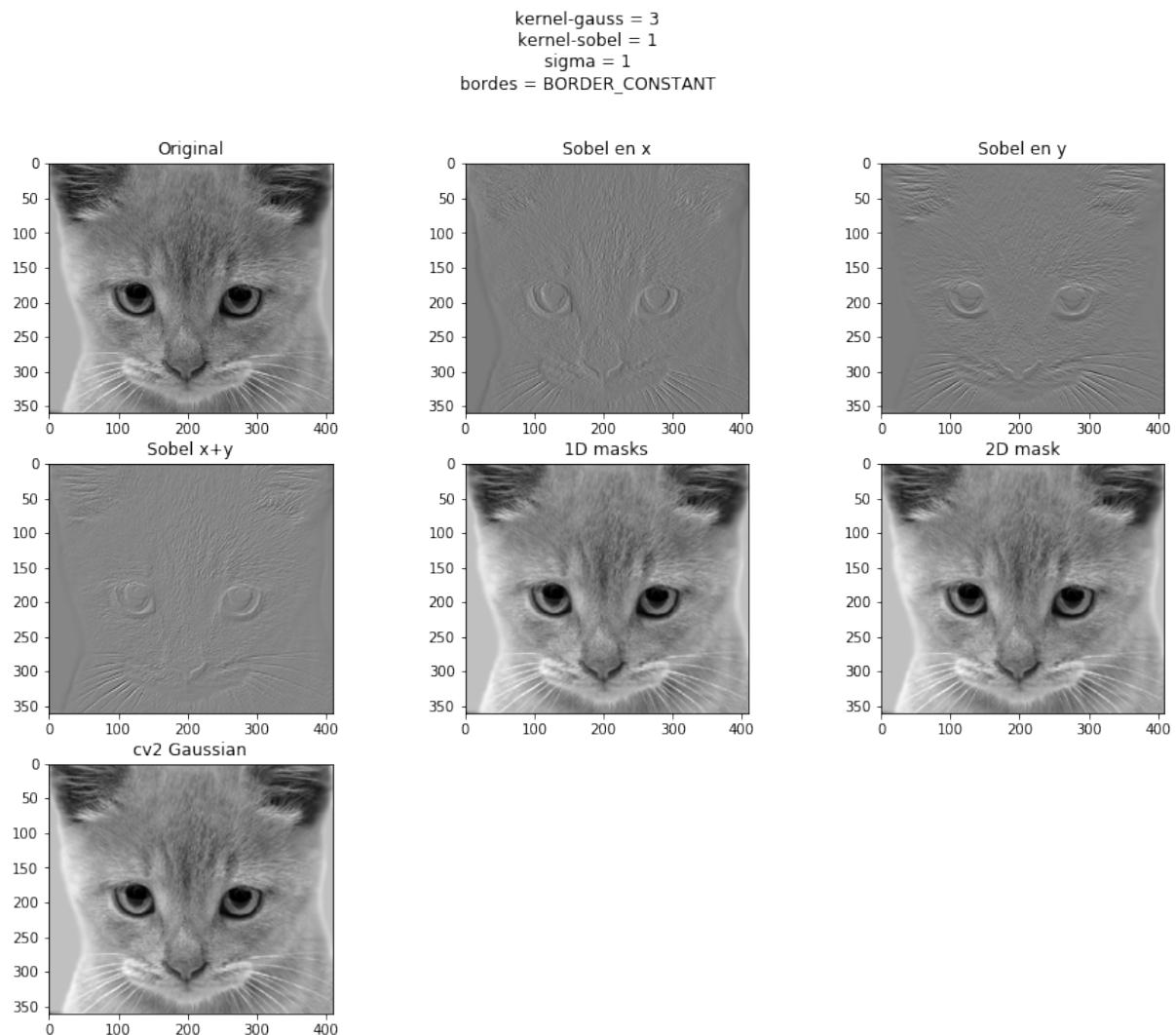
**sigmas = [1, 3, 9]**: Con estos tres valores de sigma acompañados de los valores de kernel se ve perfectamente como a medida que aumentamos el valor de  $\sigma$  y *kernel* el difuminado es mayor.

No hace falta irnos a valores de  $\sigma$  muy alto para obtener un buen emborronamiento. **bordes** = [cv2.BORDER\_CONSTANT, cv2.BORDER\_DEFAULT]: De entre todos los tipos de bordes que ofrece OpenCV estos dos muestran diferencias apreciables a simple vista. El borde constante rellena la imagen con un número fijo (por defecto 0) mientras que el borde por defecto refleja la imagen para completar los bordes.

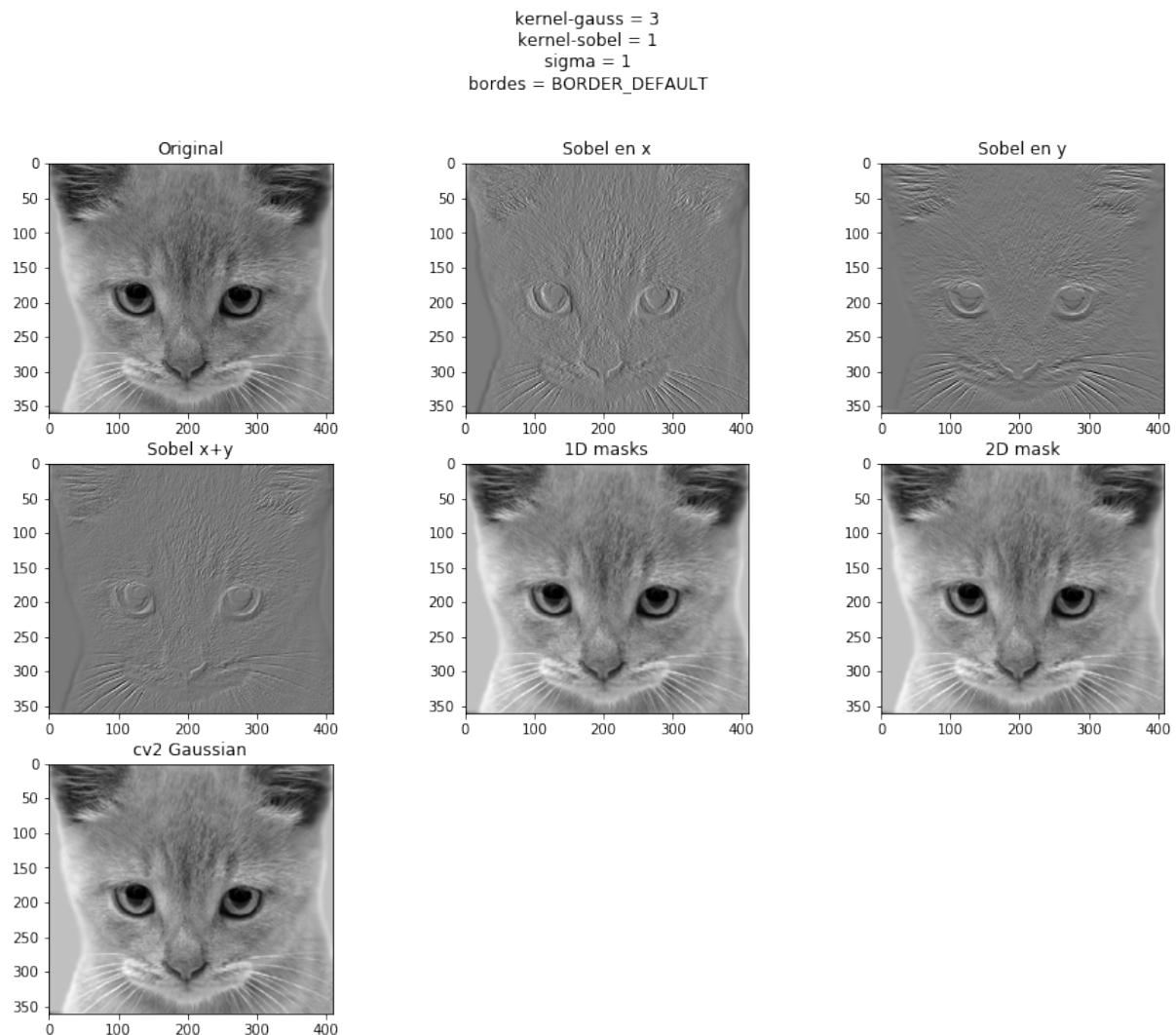
```
1 def ejer1a():
2     # Leemos imagen en ByN
3     im = leer_imagen('imagenes/cat.bmp', 0)
4     im_orig = np.copy(im)
5
6     # Predefinimos los valores del tamaño de la máscara (para
7     # gaussianas y para Sobel),
8     # el valor de sigma para las Gaussianas y
9     # el tipo de bordes
10    kernels = [3,15,31]
11    k_size = [1,5,7]
12    sigmas = [1, 3, 9]
13
14    # Borde constante: rellena con 0s
15    # Borde reflect: expande en espejo los bordes de la imagen --> abcd
16    # | dcba
17    # Borde default: expande en espejo los bordes de la imagen pero no
18    # replica el píxel del límite --> abcd | cba
19    bordes = [cv2.BORDER_CONSTANT, cv2.BORDER_DEFAULT]
20    str_bordes = ['BORDER_CONSTANT', 'BORDER_DEFAULT']
21
22    for i in range(len(kernels)):
23        for j in sigmas:
24            for k in range(len(bordes)):
25                #input('Intro para kernel-gauss = ' + str(kernels[i]) +
26                #      '\nkernel-sobel = ' +
27                #      str(k_size[i]) + '\nsigma = ' + str(j) + '\
28                nbordes = ' + str_bordes[k])
29
30                im_grad1, im_grad2 = gradiente(im, k_size[i], bordes[k])
31                im_1d = GaussianalD(im, kernels[i], j, bordes[k])
32                im_2d = Gaussiana2D(im, kernels[i], j, bordes[k])
33                im_gauss = cv2.GaussianBlur(im, (kernels[i],kernels[i]),
34                                            j, borderType=k)
```

```
30         plt.figure(num=None, figsize=(15,15)) # Establecemos el
31             tamaño de la imagen
32 # Ponemos un título general
33 plt.suptitle('kernel-gauss = ' + str(kernels[i]) + '\
34             nkernel-sobel = ' +
35                 str(k_size[i]) + '\nsigma = ' + str(j) + \
36                     '\nbordes = ' + str_bordes[k])
37 # Realizamos un subplot y para cada sub-imagen añadimos
38             un título
39 plt.subplot(4,3,1)
40 plt.imshow(im_orig, 'Original')
41 plt.subplot(4,3,2)
42 plt.imshow(im_grad1,'Sobel en x')
43 plt.subplot(4,3,3)
44 plt.imshow(im_grad2, 'Sobel en y')
45 plt.subplot(4,3,4)
46 plt.imshow(im_grad1 + im_grad2, 'Sobel x+y')
47 plt.subplot(4,3,5)
48 plt.imshow(im_1d, '1D masks')
49 plt.subplot(4,3,6)
50 plt.imshow(im_2d, '2D mask')
51 plt.subplot(4,3,7)
52 plt.imshow(im_gauss, 'cv2 Gaussian')
53 plt.show()
```

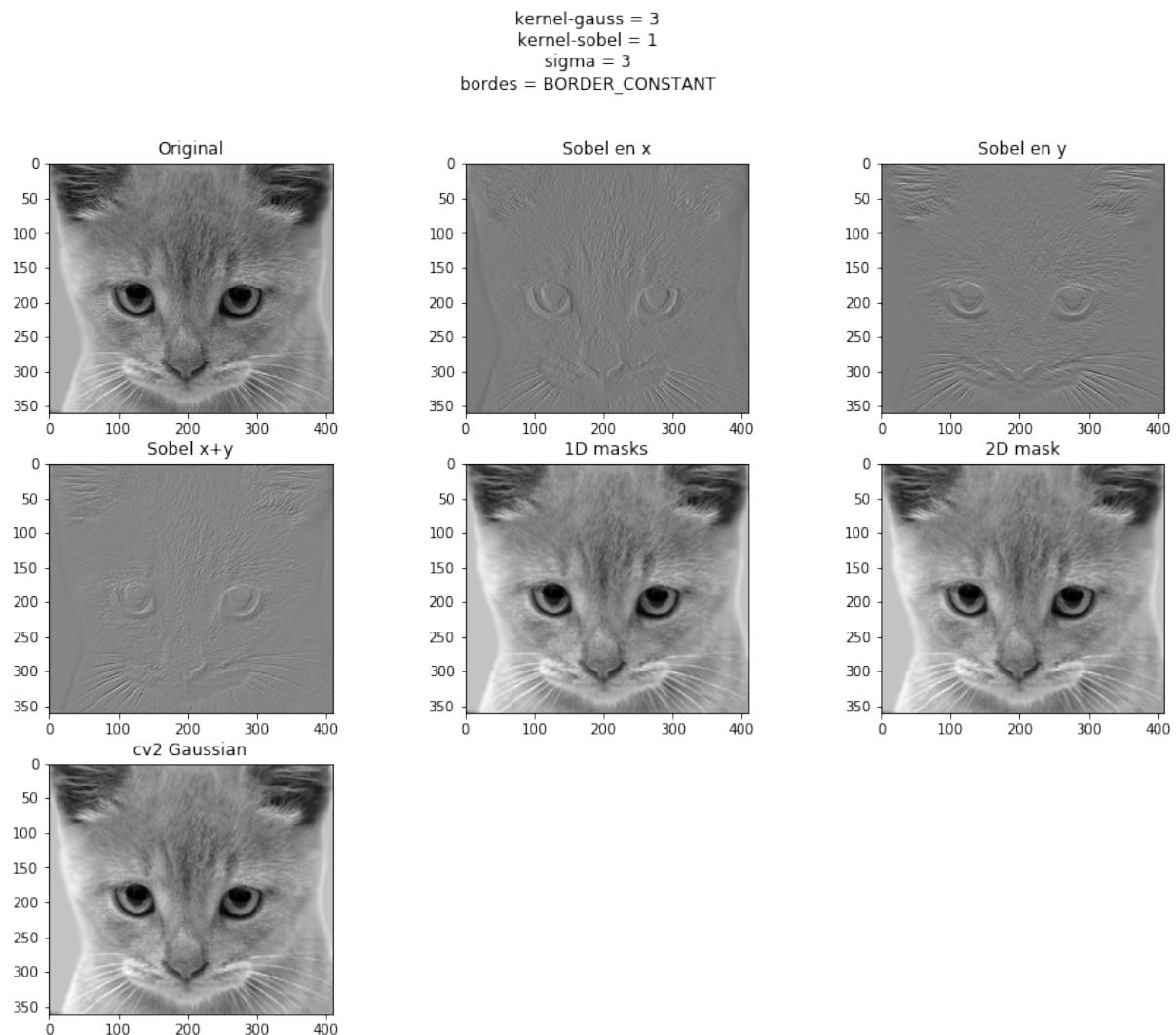
```
1 ejer1a()
```



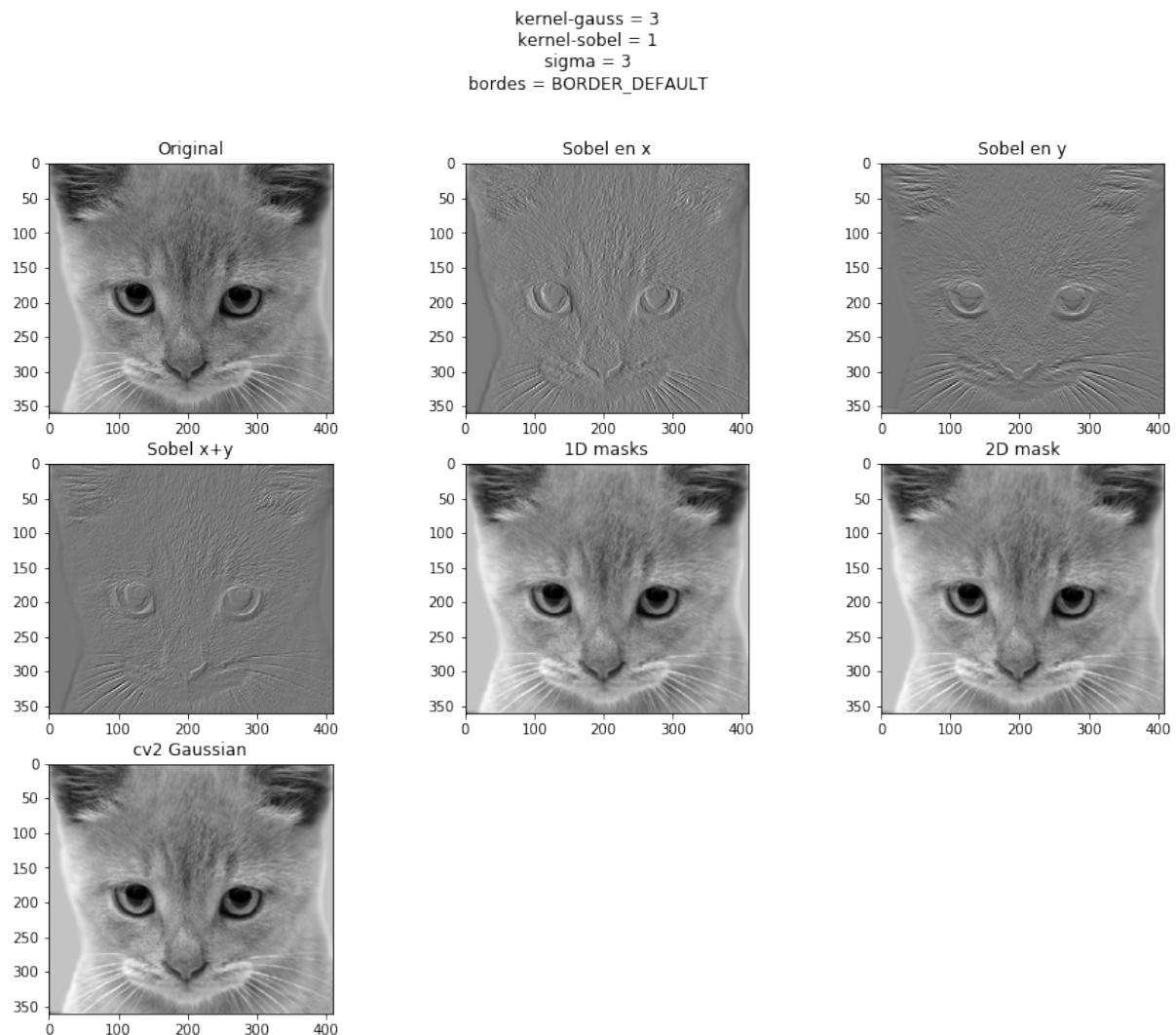
**Figura1:** png



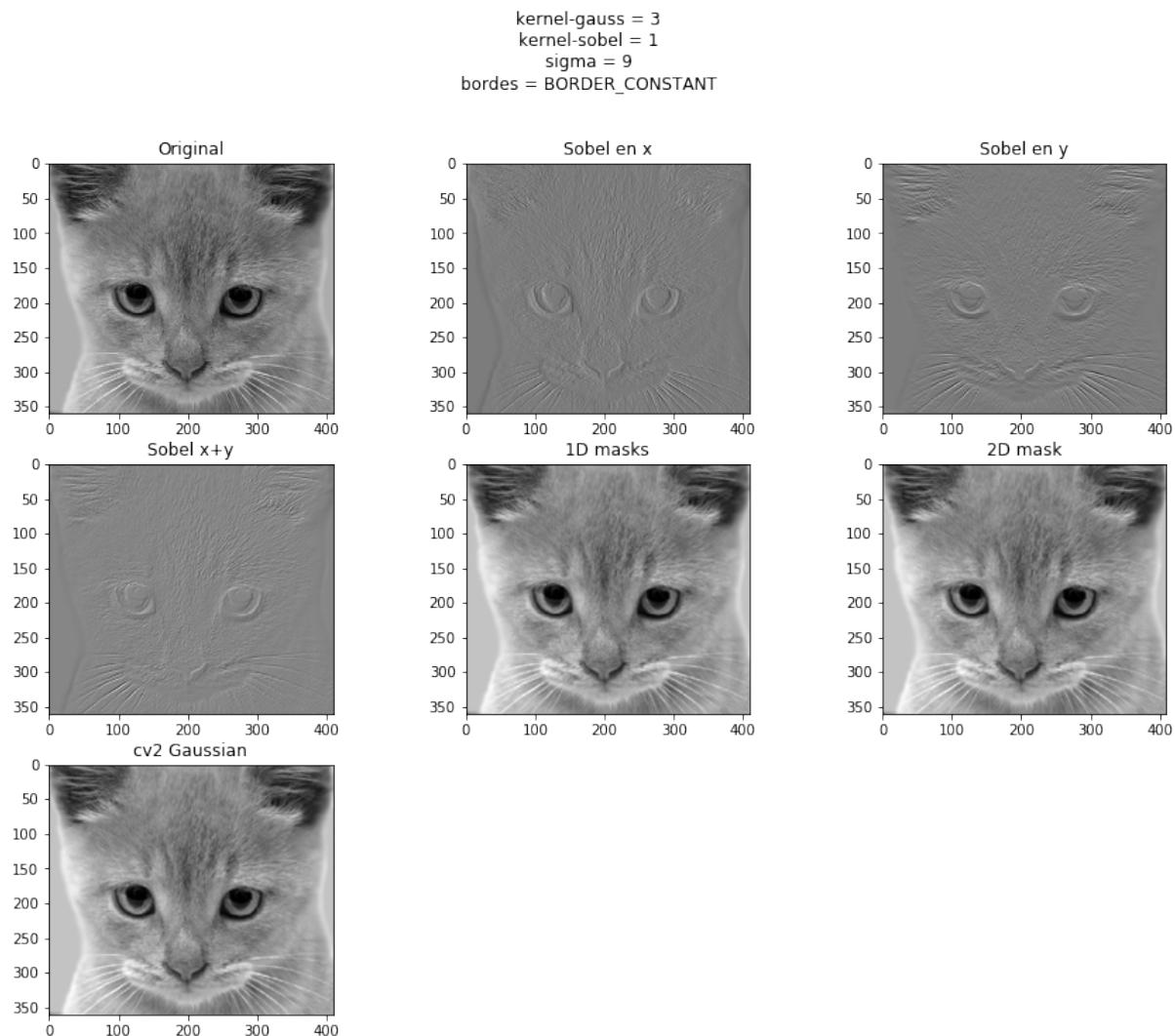
**Figura2:** png



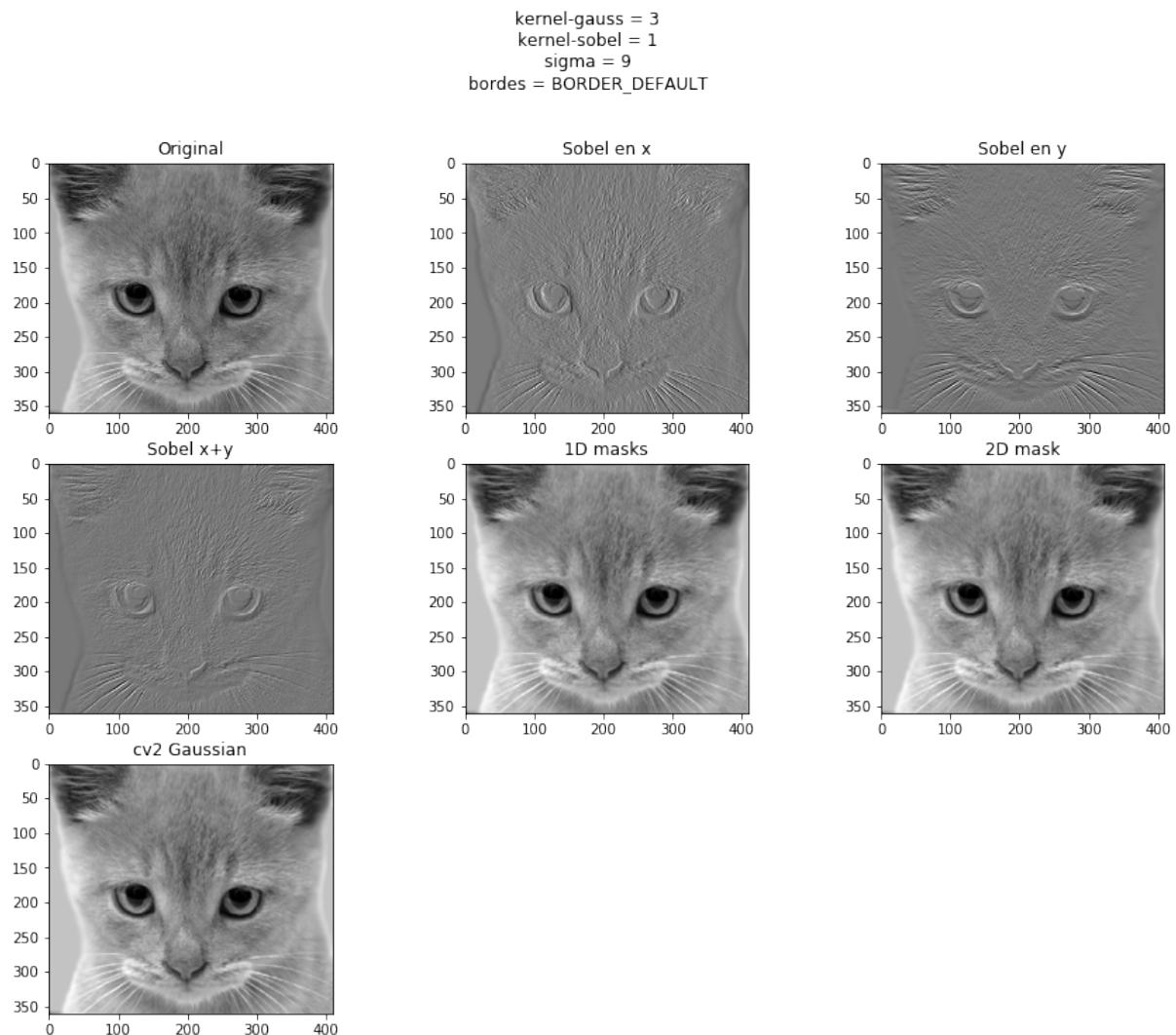
**Figura3:** png



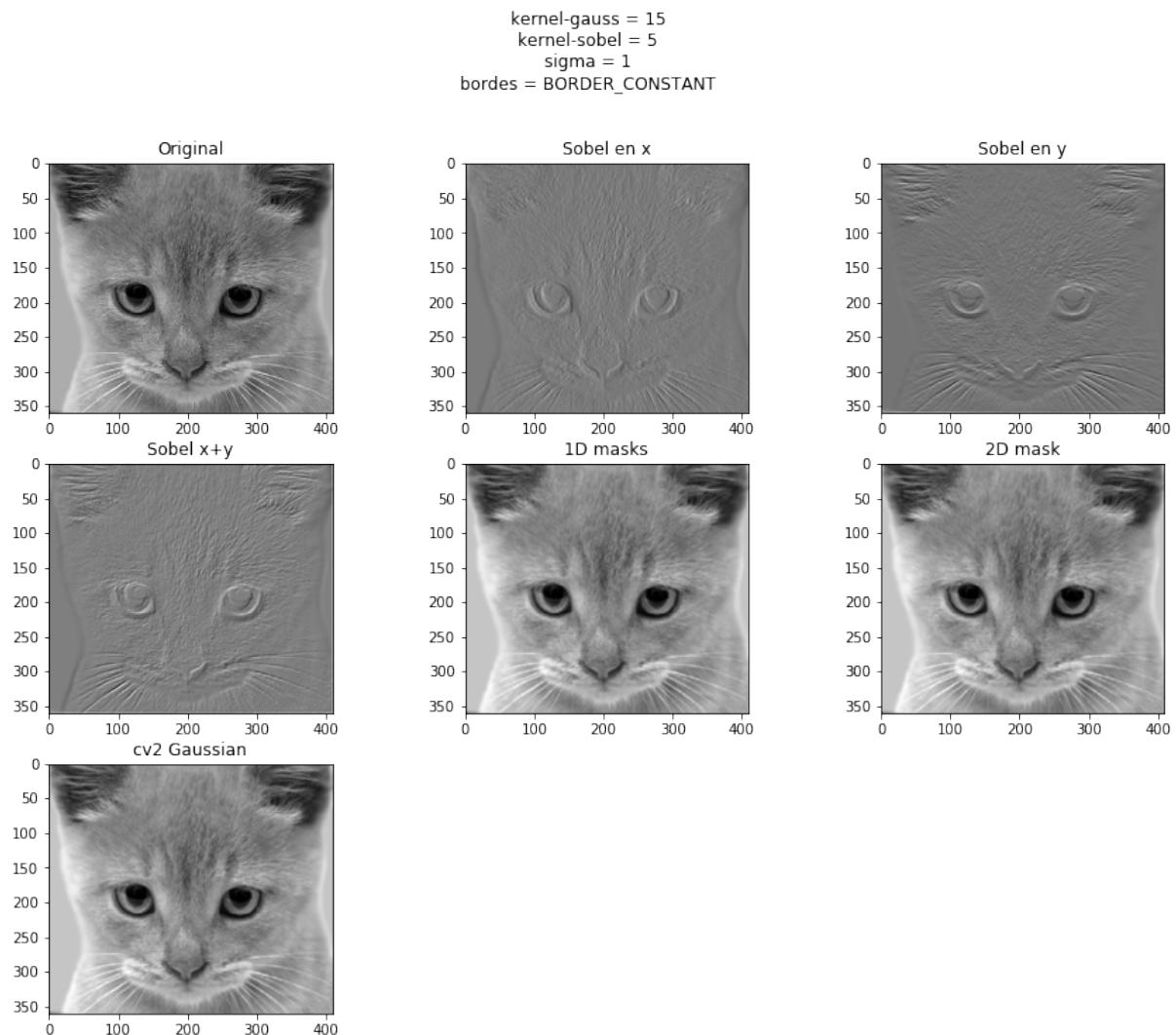
**Figura4:** png



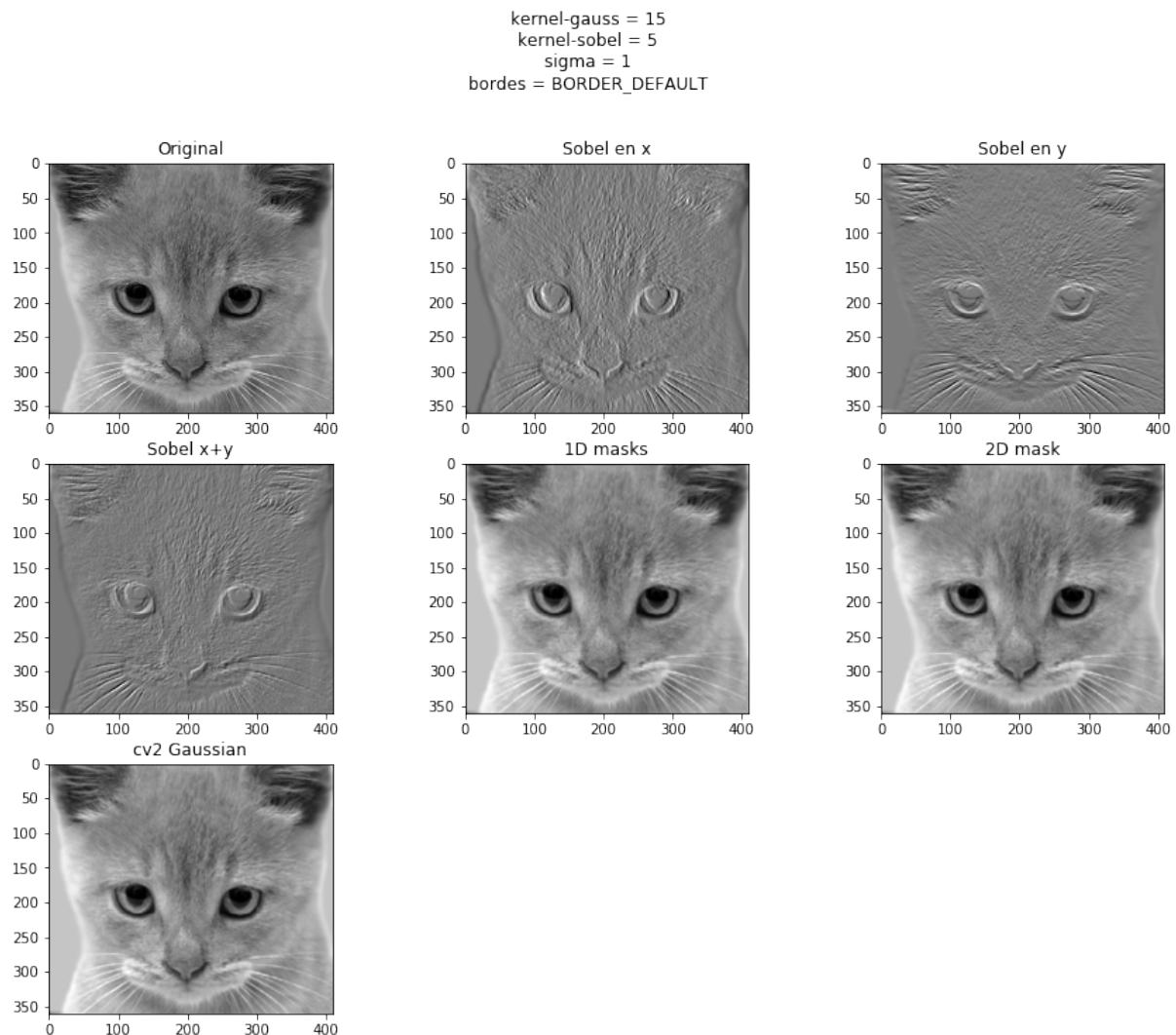
**Figura5:** png



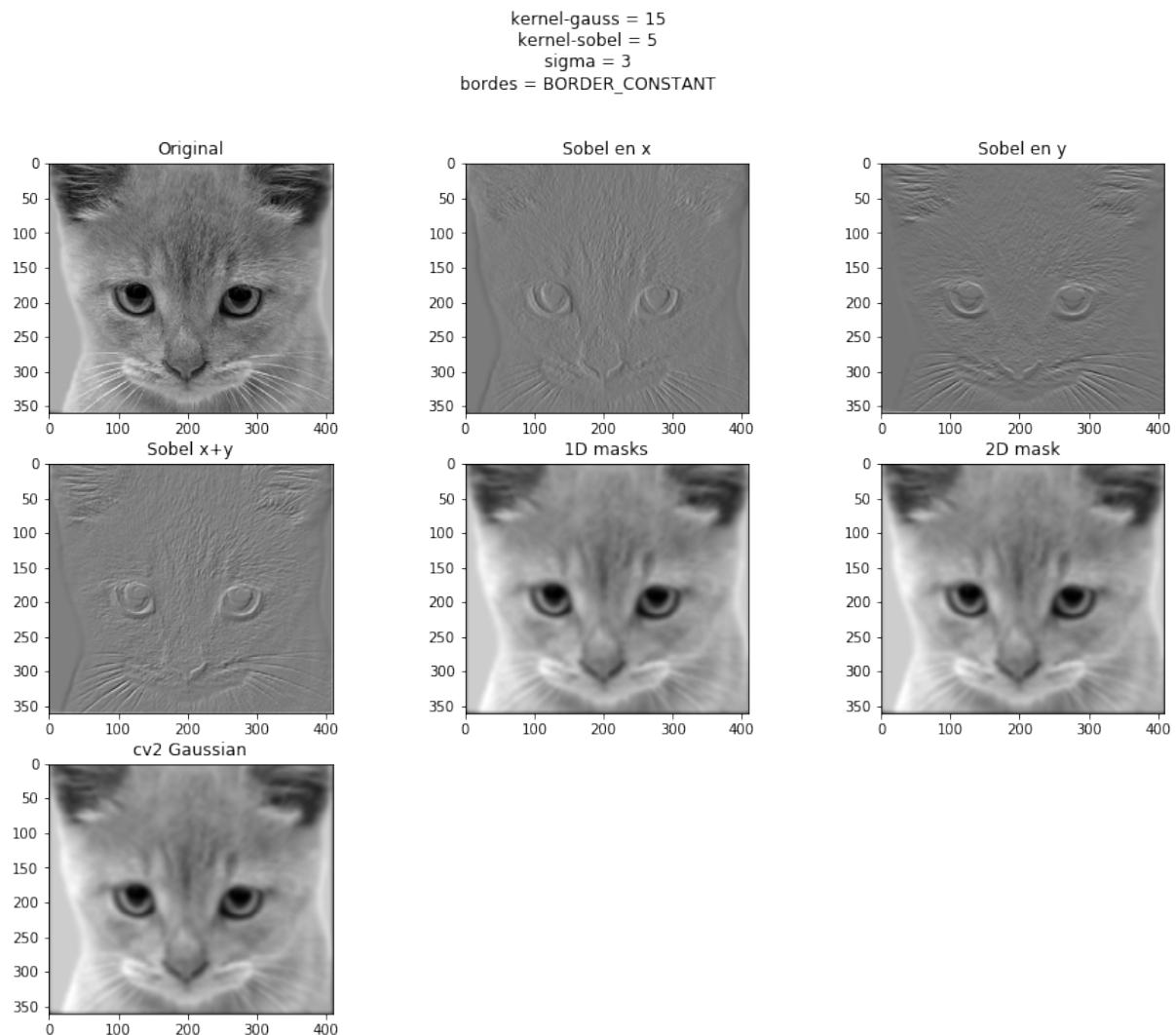
**Figura6:** png



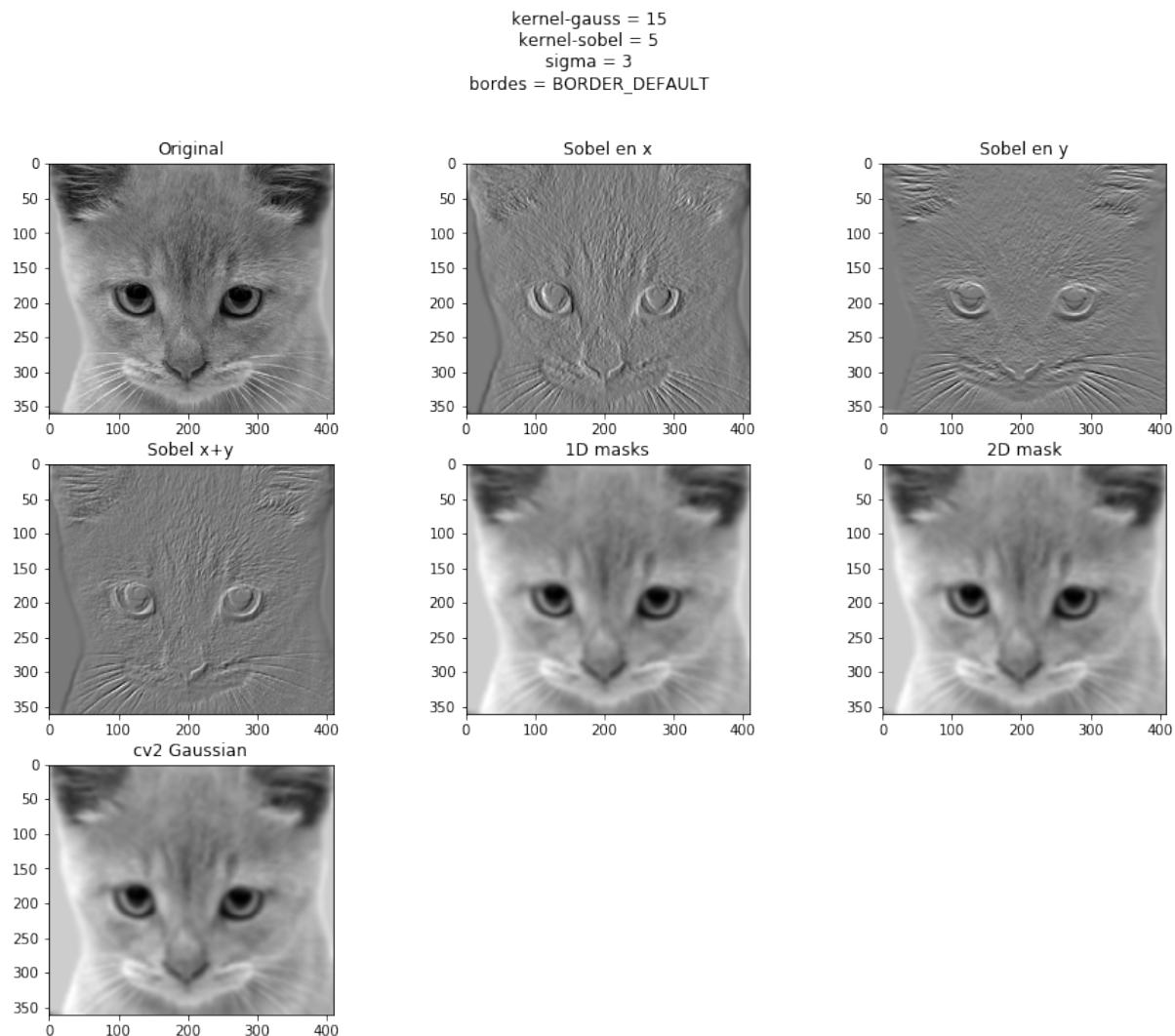
**Figura7:** png



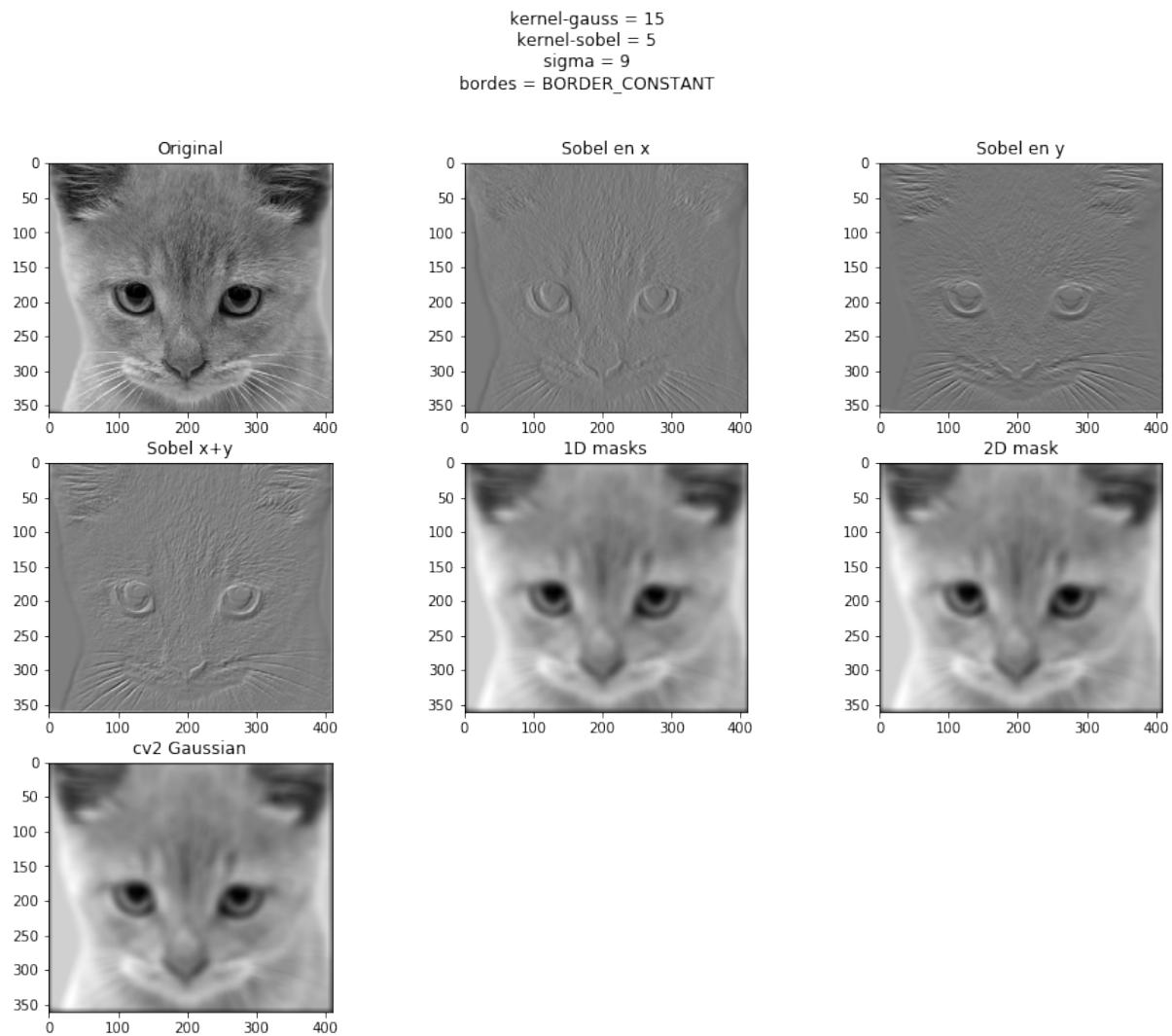
**Figura8:** png



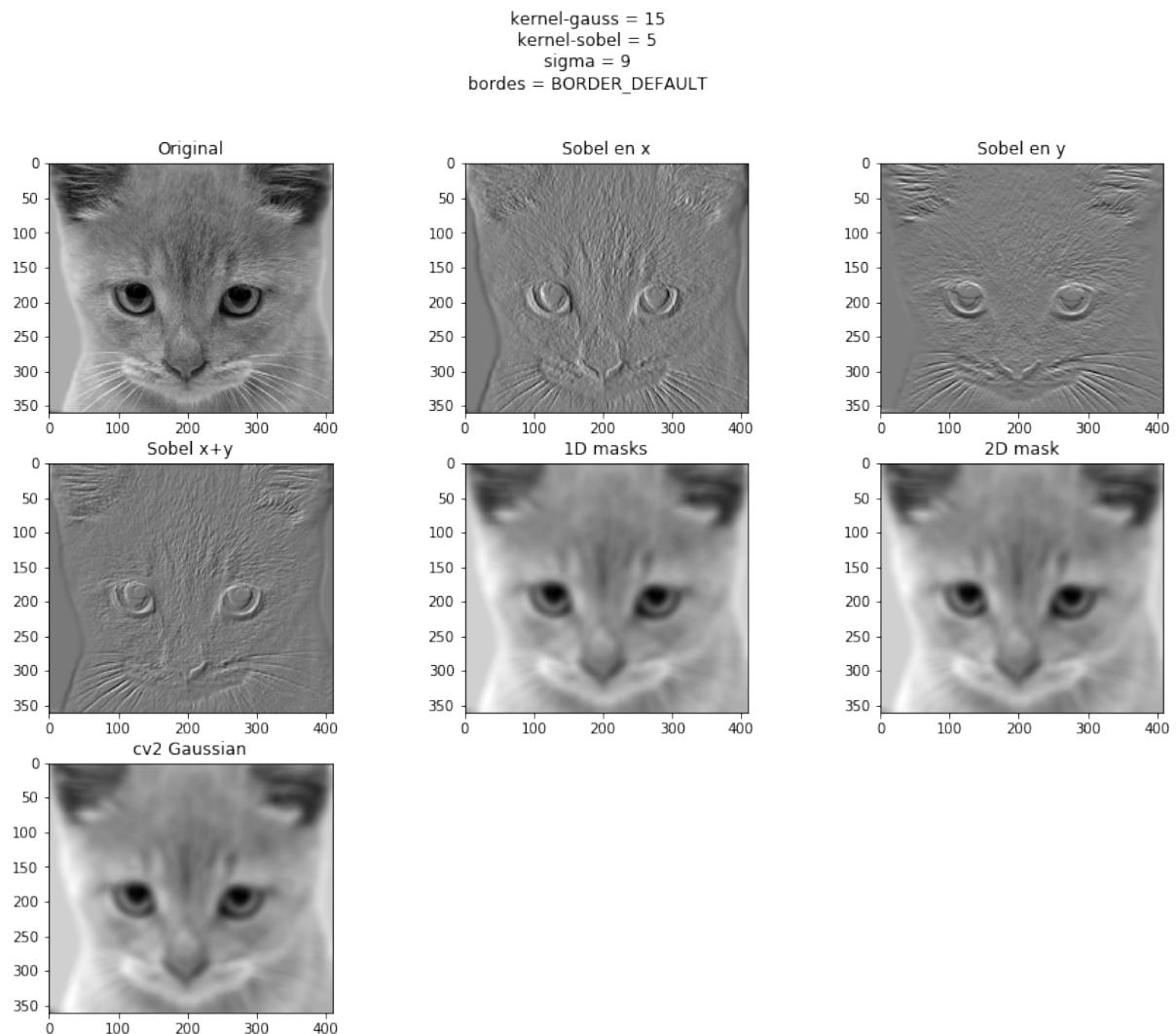
**Figura9:** png



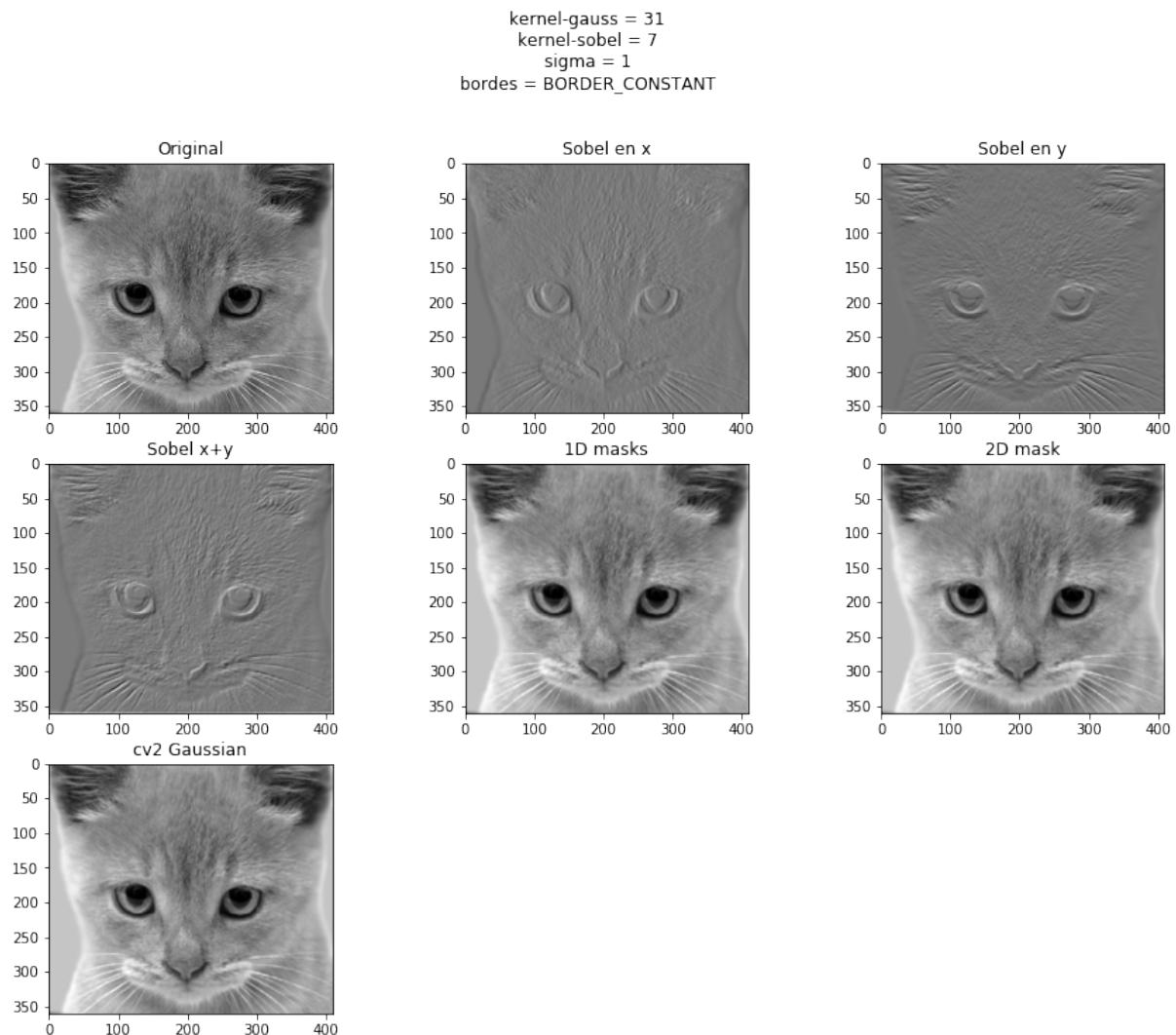
**Figura10:** png



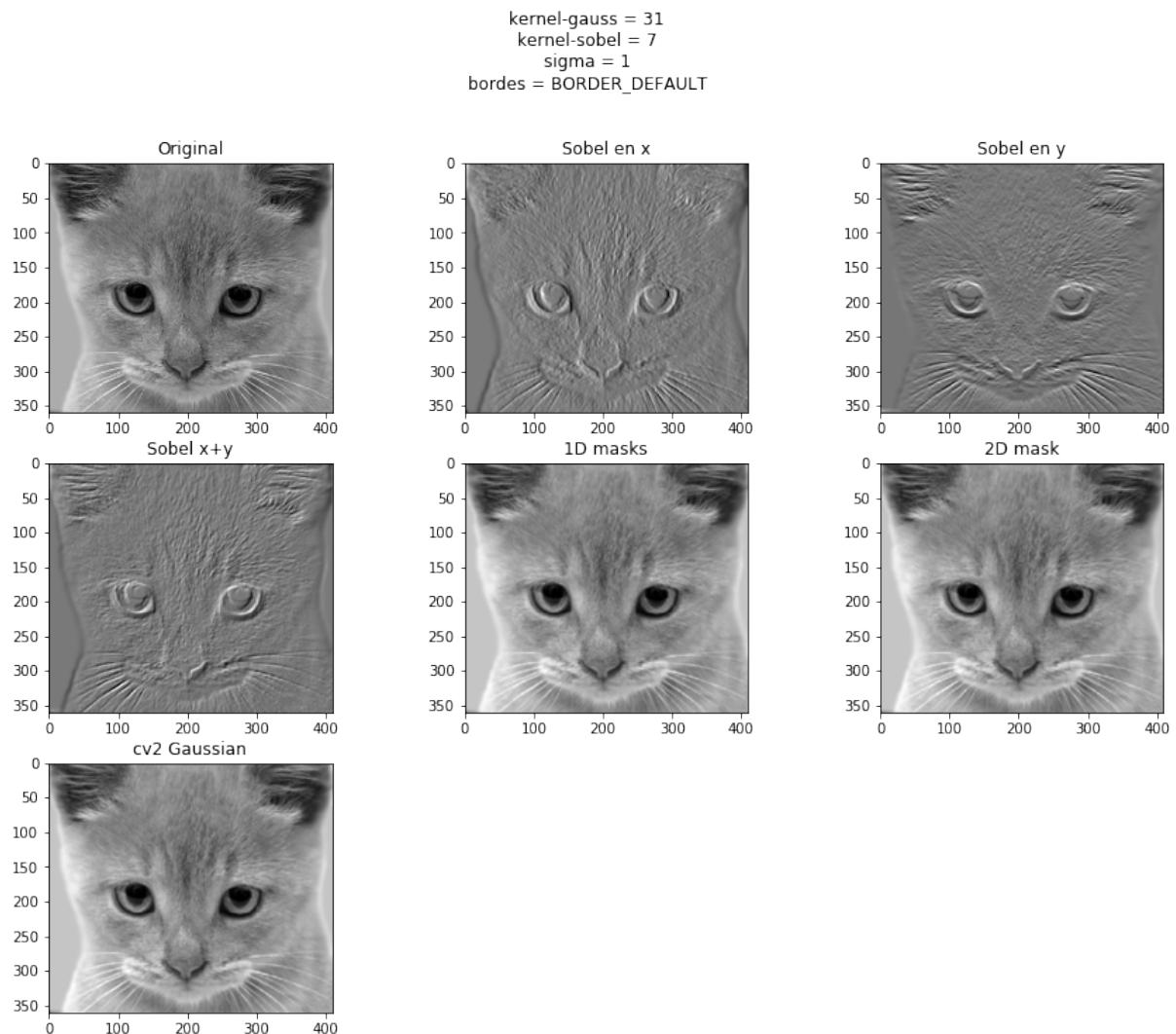
**Figura11:** png



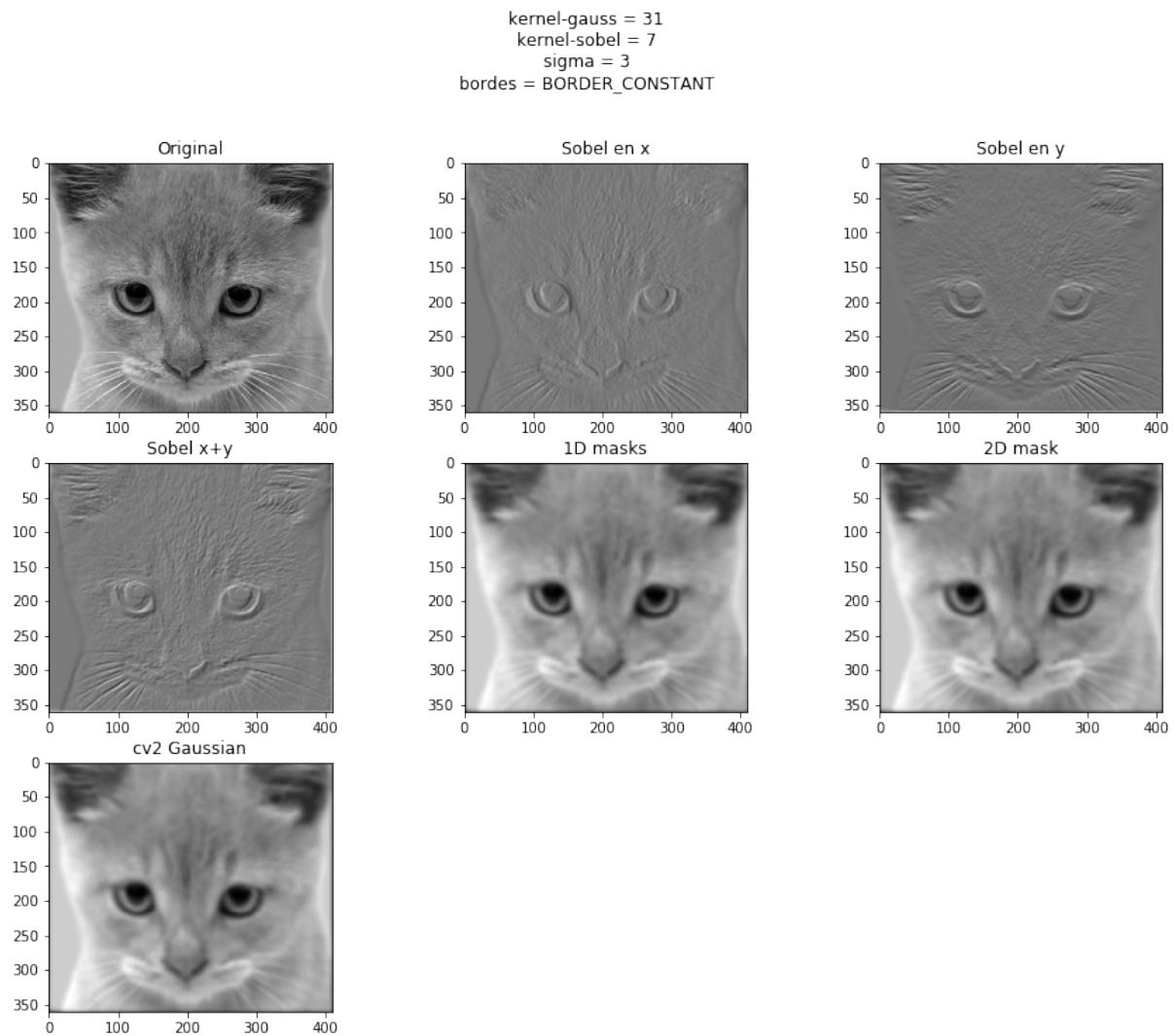
**Figura12:** png



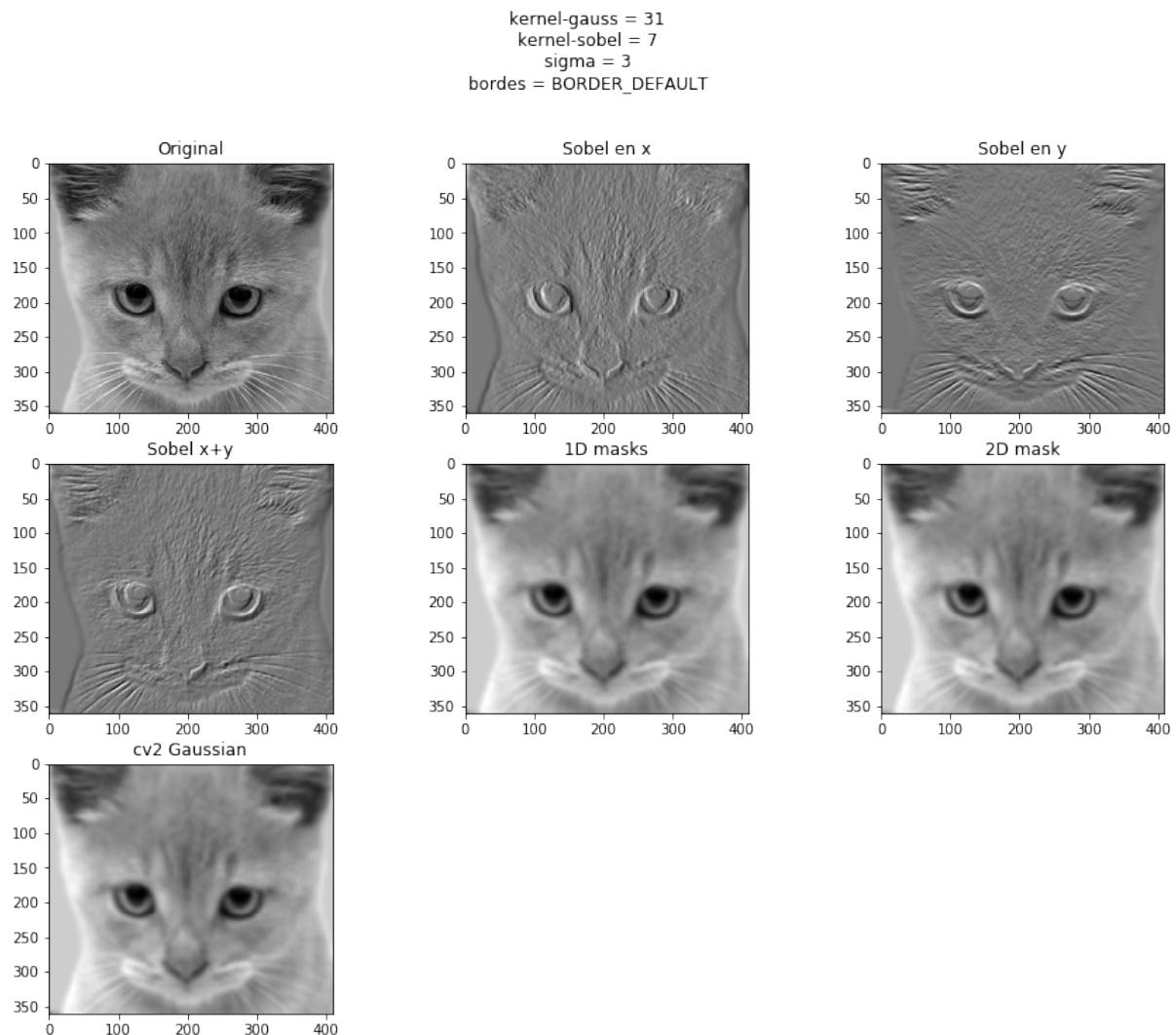
**Figura13:** png



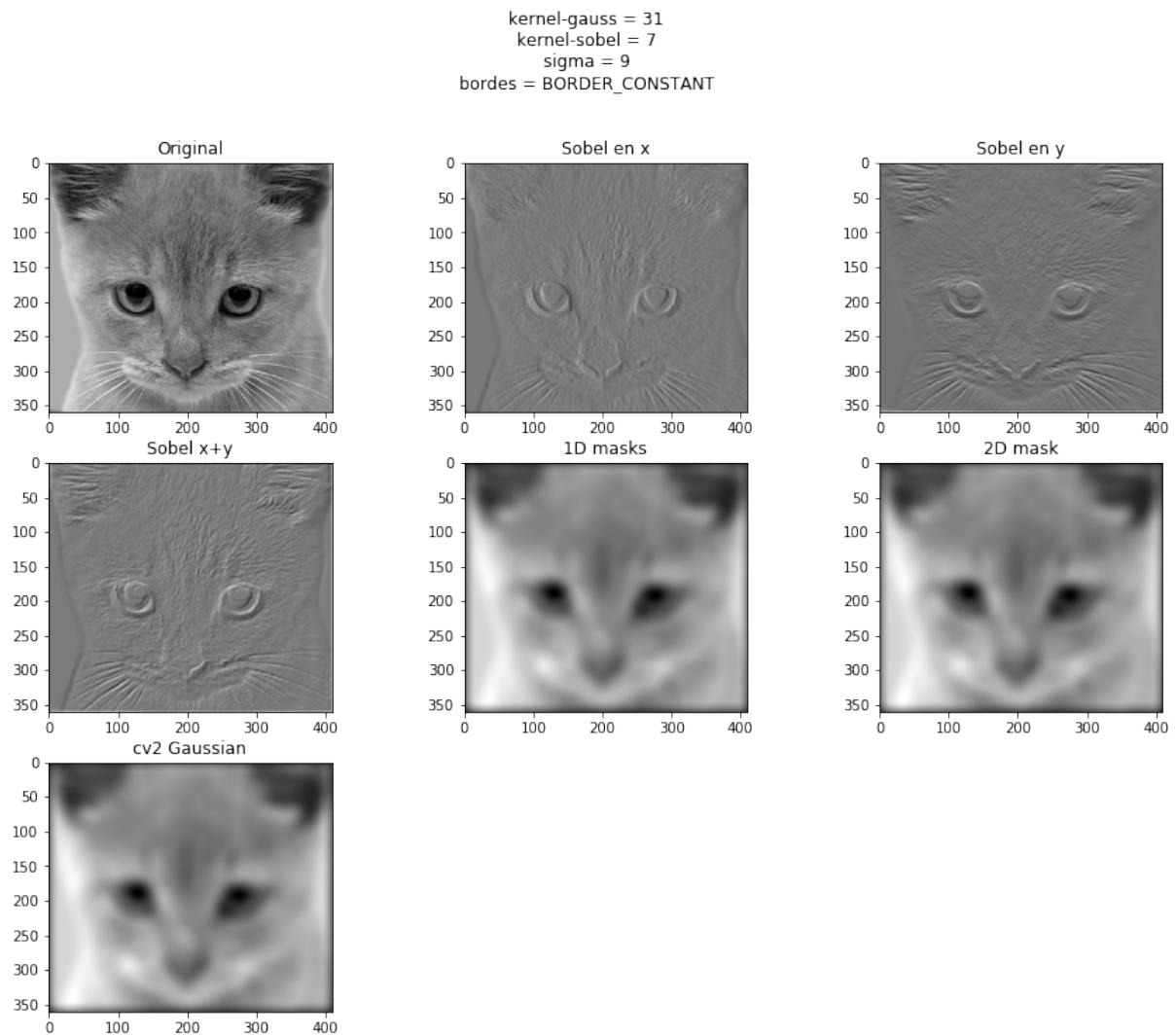
**Figura14:** png



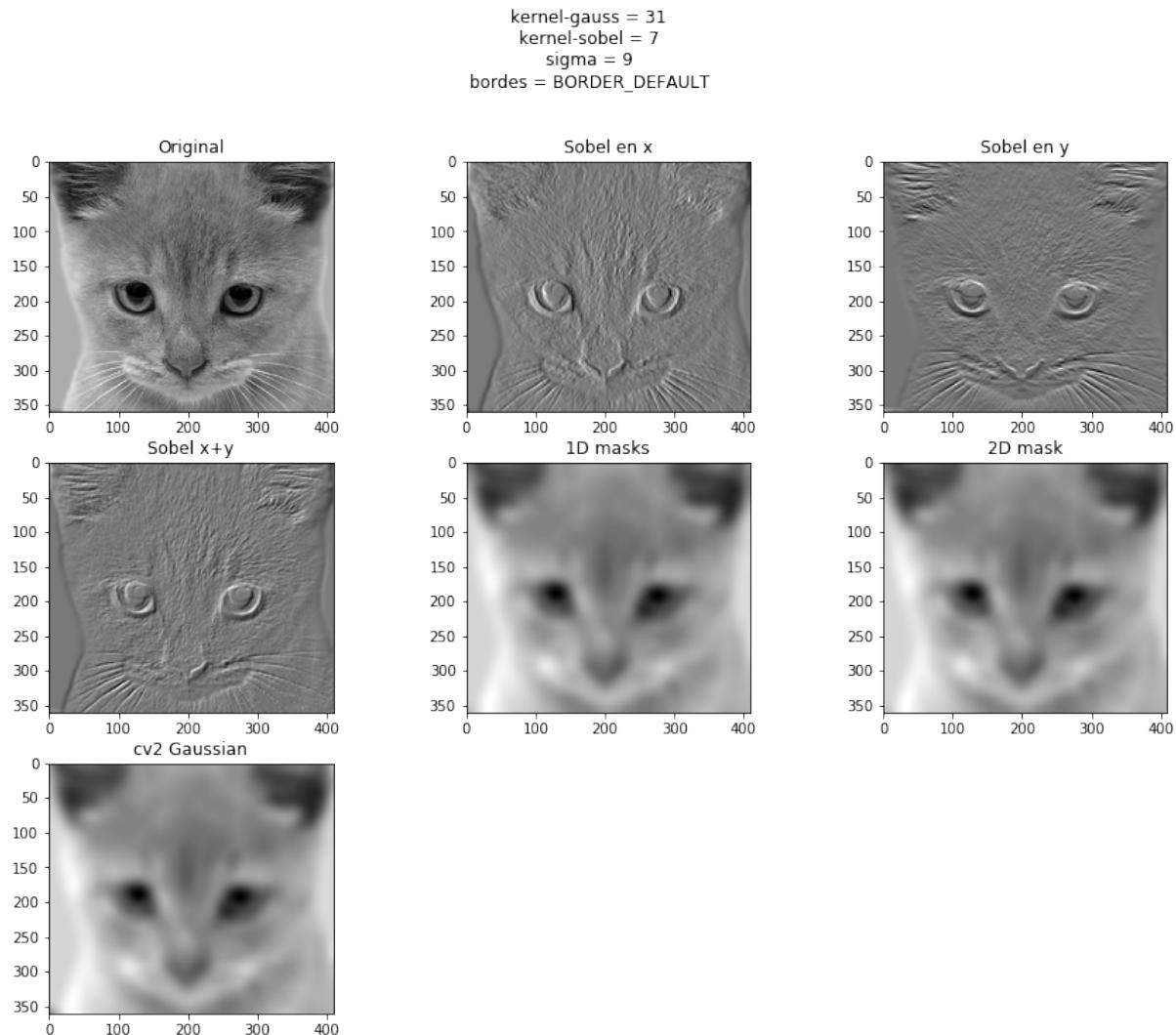
**Figura15:** png



**Figura16:** png



**Figura17:** png

**Figura18:** png

En las imágenes podemos notar que a mayor  $\sigma$  y a mayor tamaño de máscara más se emborrona las imágenes con filtro de Gaussiana. Además si el tipo de borde es CONSTANT en las imágenes con filtro de Gaussiana y tamaño de kernel grande se aprecia un borde negro en los marcos de la imagen.

En las imágenes con filtro Sobel vemos como sólo afecta el tipo de borde y el tamaño del kernel. Las imágenes están mucho más definidas a mayor tamaño de máscara y cuando el tipo de borde definido es DEFAULT.

## 1b. Cálculo de una convolución 2D con una máscara *Laplaciana de Gaussiana*

### ■ Qué vamos a calcular

Vamos a aplicar un filtro de Laplaciana de Gaussiana a una imagen.

### ■ Cómo lo hacemos

El filtro lo aplicaremos con la función *LoG()*. La Laplaciana de Gaussiana es lo mismo que la segunda derivada de la Gaussiana. Primero obtenemos la gaussiana de la imagen y después la derivamos en ambos ejes obteniendo una derivada en el eje x y otra derivada en el eje y. La imagen que queremos es la suma de estas dos.

### ■ Parámetros

**>kernels = [3,15,31]**: El tamaño máximo de máscara en las funciones Gaussianas es 31 por lo que se intenta escoger valores que representen todo el dominio viable pero sin que sean demasiados como para que haya muchas ejecuciones.

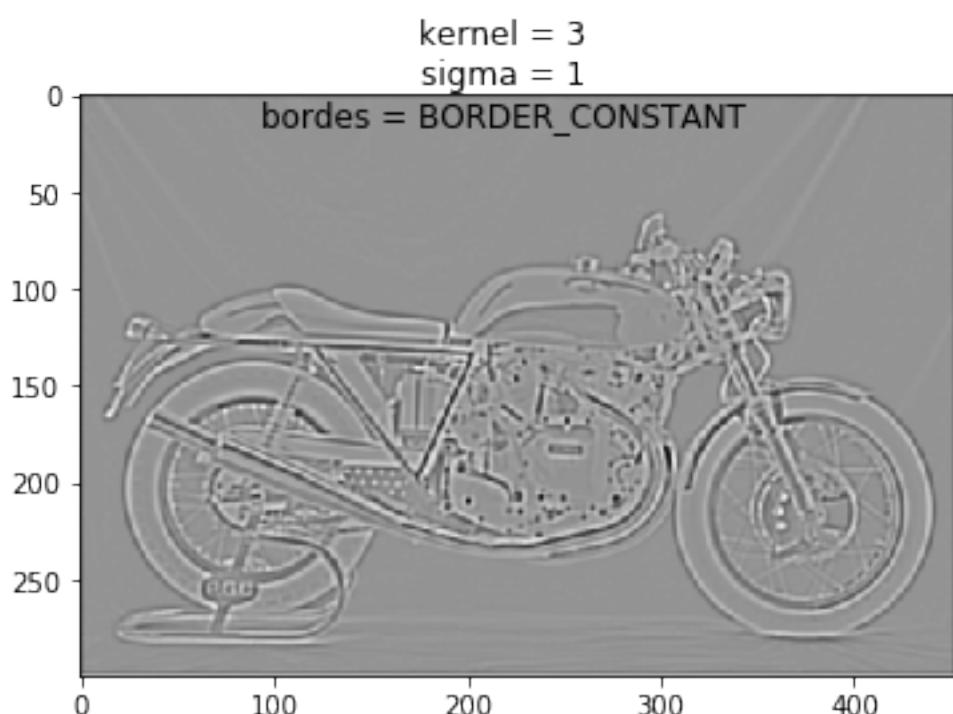
**sigmas = [1, 3, 9]**: Con estos tres valores de sigma acompañados de los valores de kernel se ve perfectamente como a medida que aumentamos el valor de  $\sigma$  y *kernel* el difuminado es mayor.

No hace falta irnos a valores de  $\sigma$  muy alto para obtener un buen emborronamiento. **bordes = [cv2.BORDER\_CONSTANT, cv2.BORDER\_DEFAULT]**: De entre todos los tipos de bordes que ofrece OpenCV estos dos muestran diferencias apreciables a simple vista. El verde constante rellena la imagen con un número fijo (por defecto 0) mientras que el borde por defecto refleja la imagen para completar los bordes.

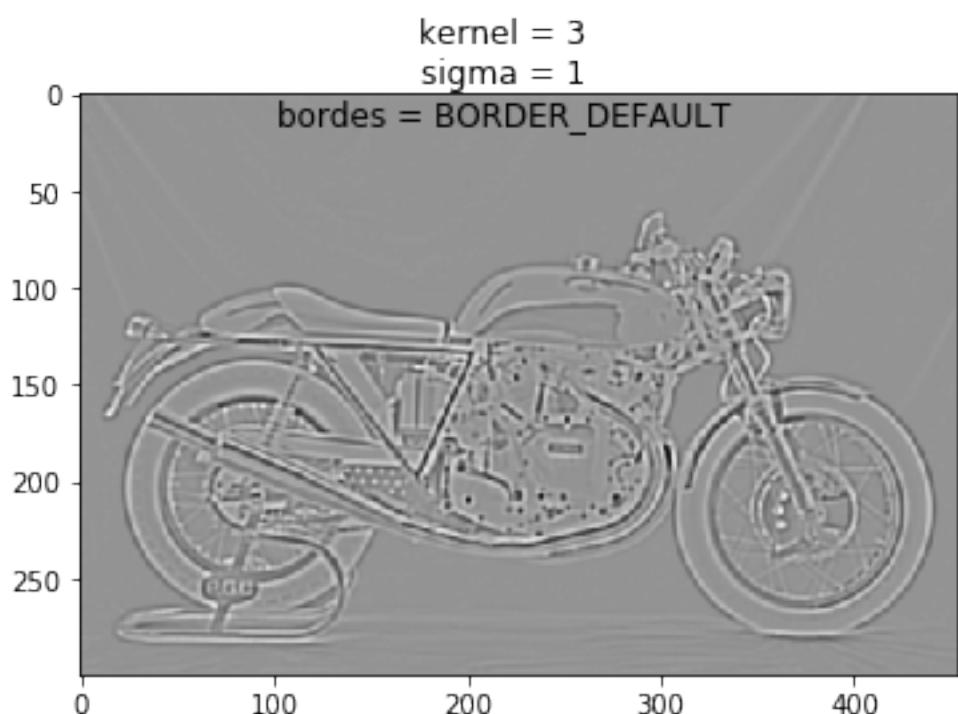
```
1 def ejer1b():
2     im = leer_imagen('imagenes/motorcycle.bmp', 0)
3
4     kernels = [3,15,31]
5     sigmas = [1, 3]
6
7     # Borde constante: rellena con 0s
8     # Borde reflect: expande en espejo los bordes de la imagen --> abcd
9         | dcba
10    # Borde default: expande en espejo los bordes de la imagen pero no
11        replica el píxel del límite --> abcd | cba
12    bordes = [cv2.BORDER_CONSTANT, cv2.BORDER_DEFAULT]
13    str_bordes = ['BORDER_CONSTANT', 'BORDER_DEFAULT']
14
15    for i in range(len(kernels)):
16        for j in sigmas:
17            for k in range(len(bordes)):
18                #input('kernel = ' + str(kernels[i]) + '\nsigma = ' +
19                     str(j) + '\nbordes = ' + str_bordes[k])
```

```
17
18     img_lap = LoG(im,kernels[i],j,bordes[k])
19
20     plt.suptitle('kernel = ' + str(kernels[i]) + '\nsigma =
21                 ' + str(j) + '\nbordes = ' + str_bordes[k])
22     plt.title('Laplacian')
23     plt_imshow(img_lap)
24     plt.show()
```

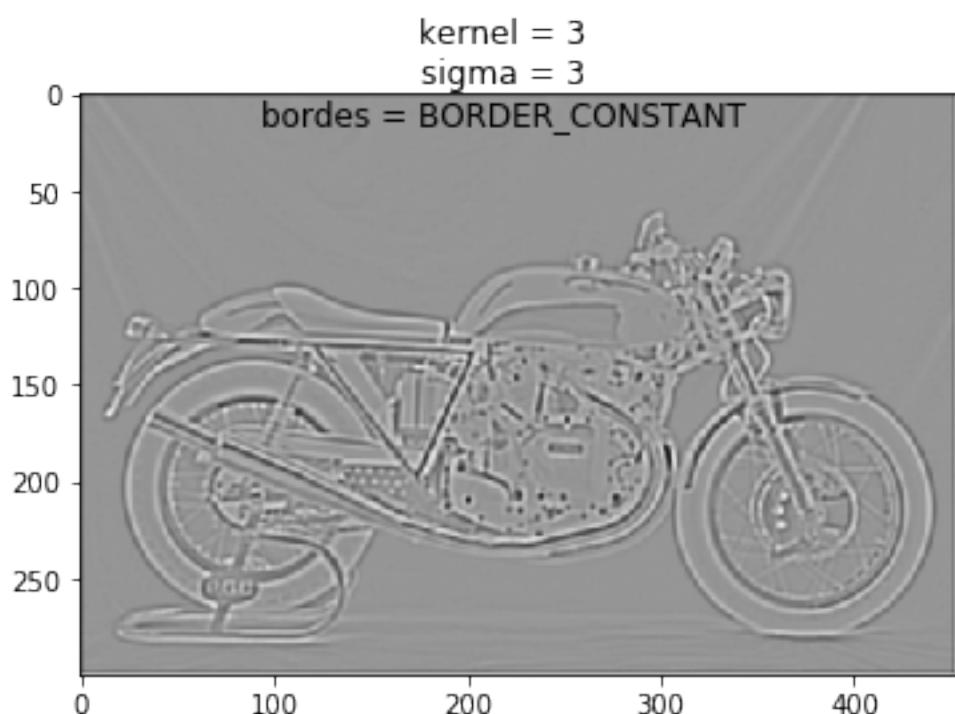
```
1 ejer1b()
```



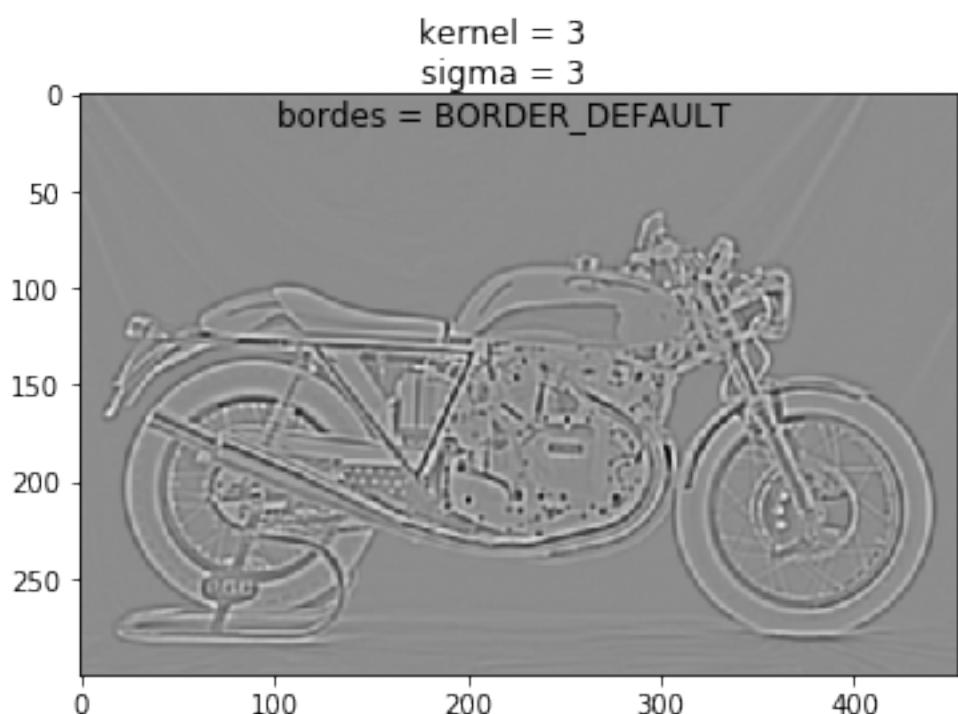
**Figura19:** png



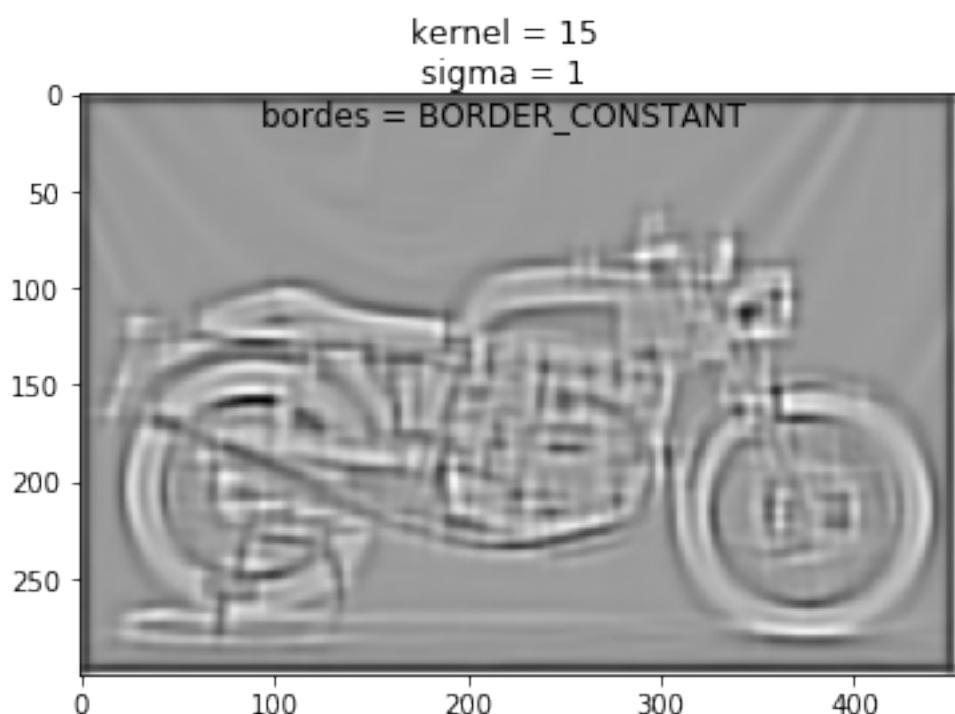
**Figura20:** png



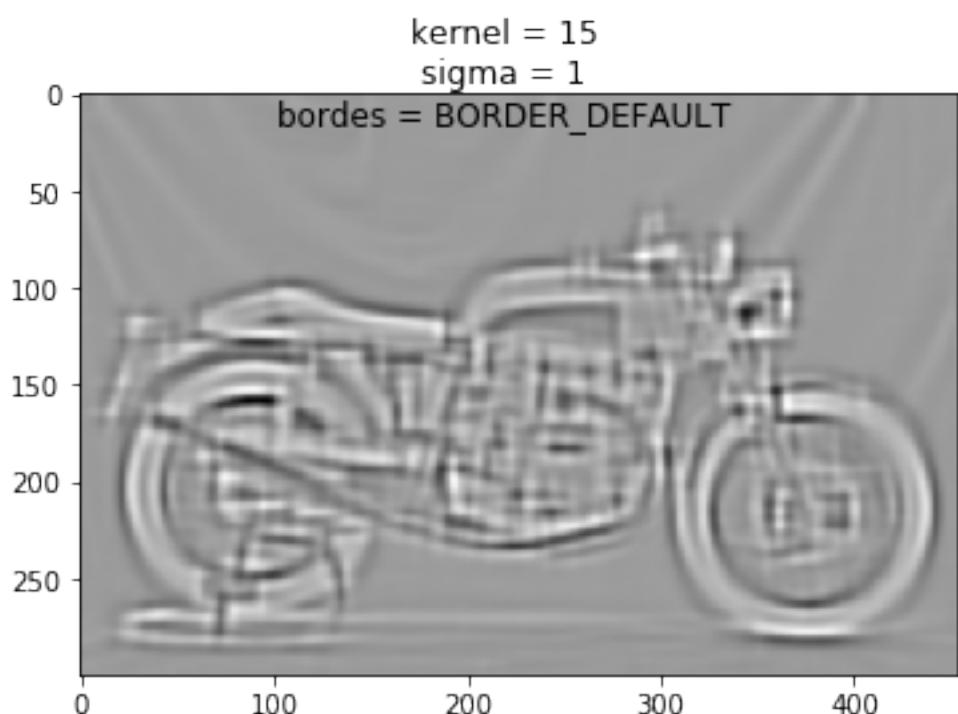
**Figura21:** png



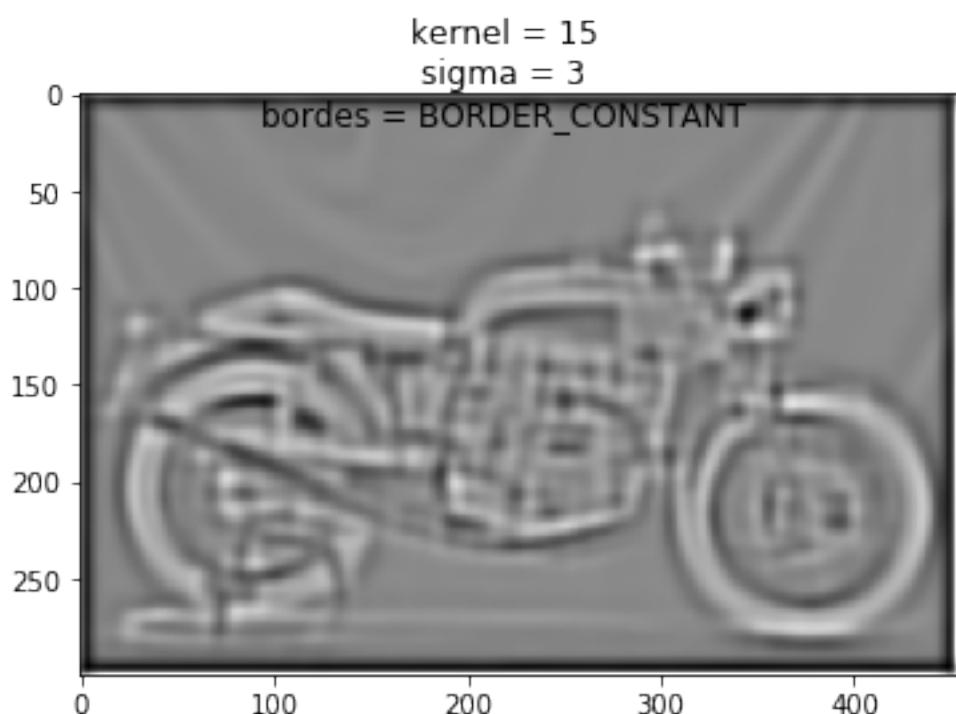
**Figura22:** png



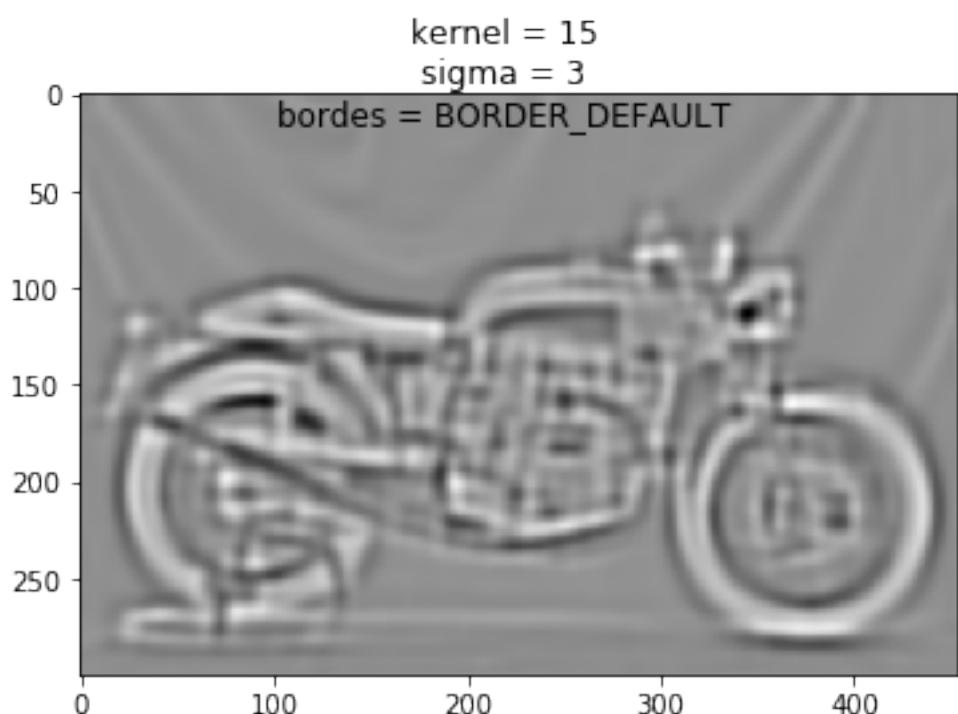
**Figura23:** png



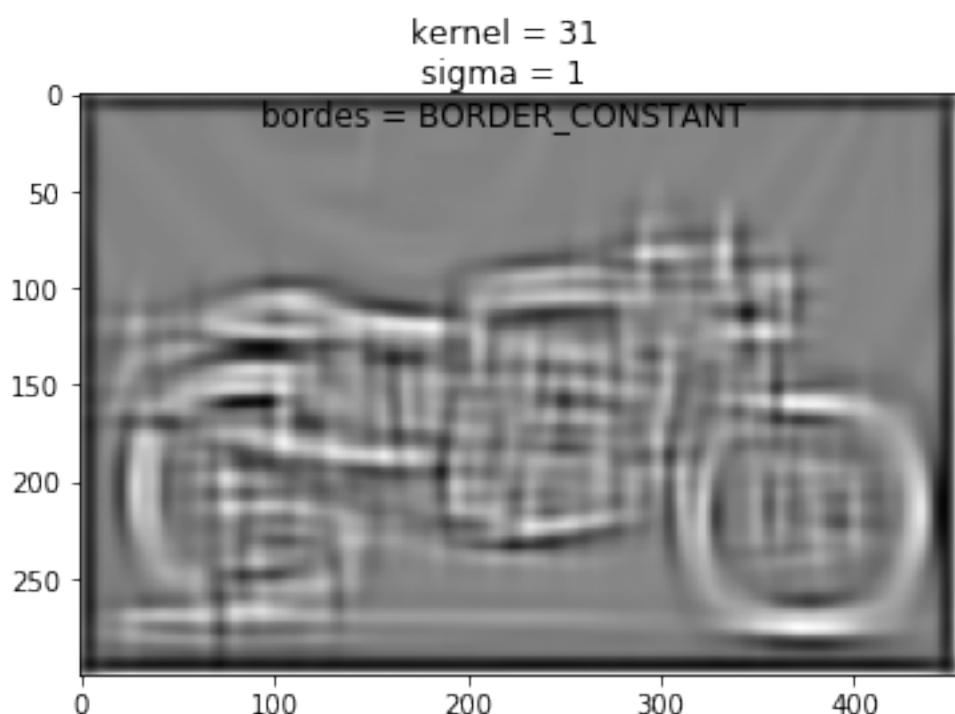
**Figura24:** png



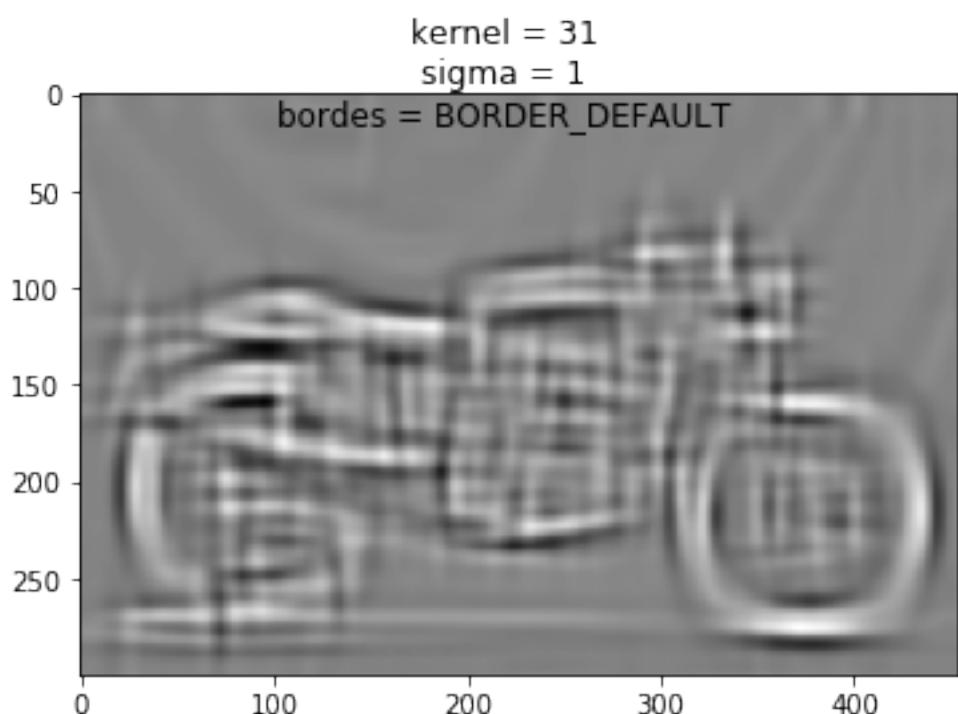
**Figura25:** png



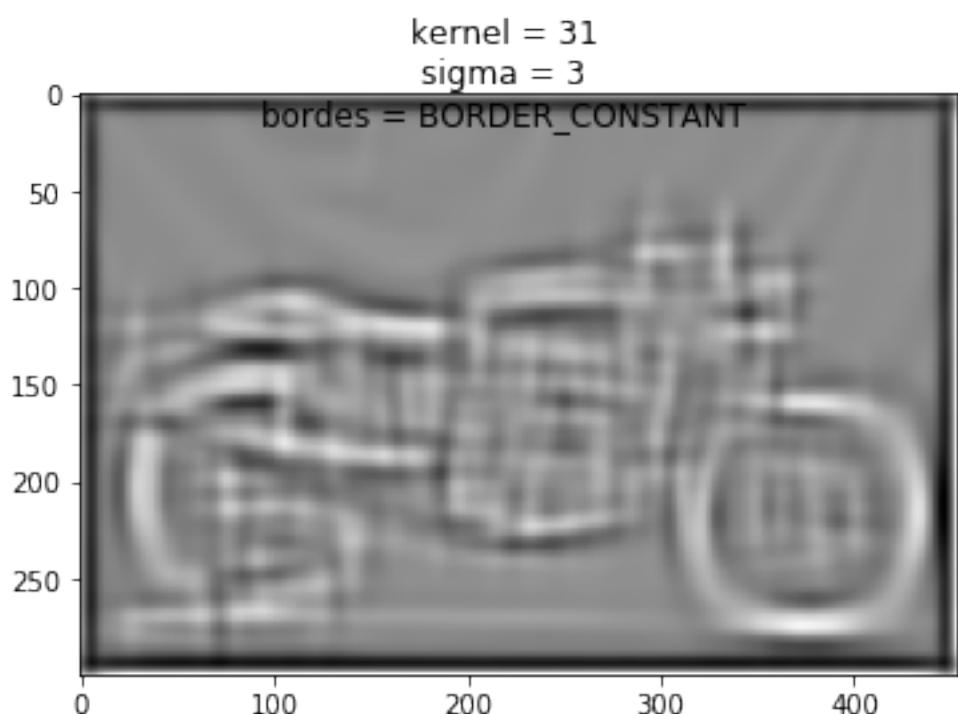
**Figura26:** png



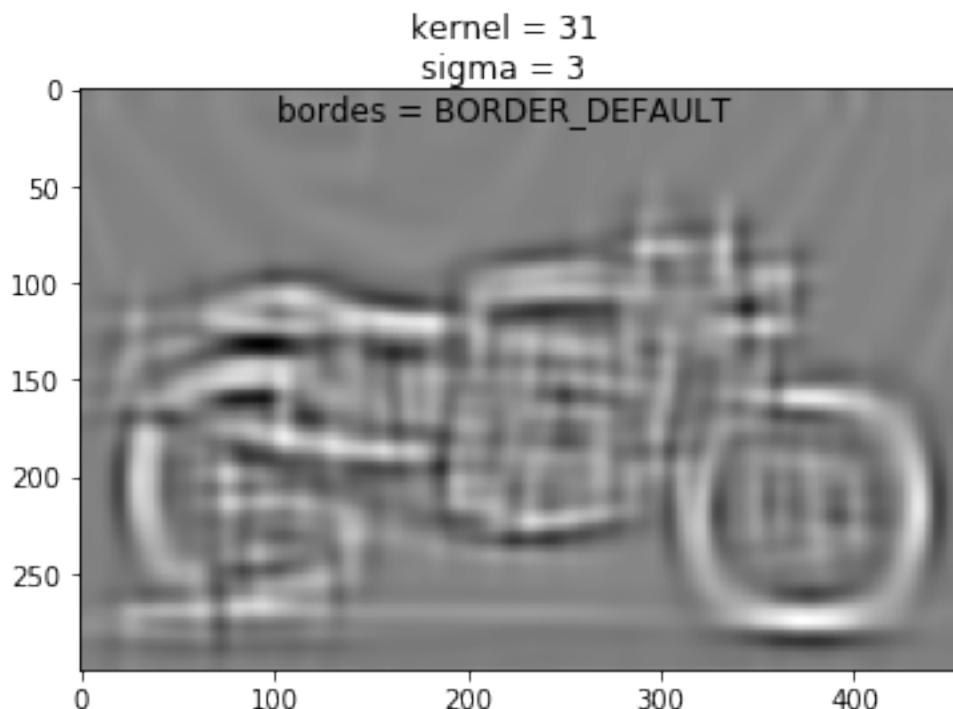
**Figura27:** png



**Figura28:** png



**Figura29:** png



**Figura30:** png

Como antes, si el tamaño de máscara es suficientemente grande y el tipo de borde es CONSTANT aparece un marco negro en la imagen.

Vemos que la Laplaciana de Gaussiana muestra mejores resultados, al menos en este caso, cuando el kernel y  $\sigma$  son lo suficientemente pequeños para difuminar los bordes más “afilados” que son los que luego mostrarán las imágenes pero no bordes menos importantes que luego destorsionarán la imagen.

## Ejercicio 2.

### 2a. Pirámide Gaussiana de 4 niveles

#### ■ Qué vamos a calcular

Vamos a realizar una pirámide Gaussiana de 4 niveles (5 imágenes suponiendo como nivel 0 el tamaño original).

#### ■ Cómo lo hacemos

Para hacer la pirámide Gaussiana nos ayudaremos de tres funciones: *Gaussian1D()*, *image-Subsample()* y *pyramid()*. Primero aplicamos la Gaussiana a la imagen y la guardamos como un

nivel de la pirámide, disminuimos la imagen con *imageSubsample()* y repetimos hasta cubrir los 4 niveles (5 contando el 0). Una vez tengamos todas las imágenes *pyramid()* se encargará de ordenarlas adecuadamente y mostrarlas en forma de pirámide.

#### ■ Funcionamiento *Gaussiana1D()*

Obtenemos el kernel de una función Gaussiana de tamaño *kernel\_size*,  $\sigma = \text{sigma}$  y bordes de tipo *borde* y lo aplicamos una vez por filas y otra vez por columnas a la imagen pasada como parámetro.

#### ■ Funcionamiento *imageSubsample()*

La imagen que se le pasa como parámetro es reducida a la mitad de columnas y filas.

#### ■ \*\*Funcionamiento *pyramid()*\_\*\*

Crea una imagen en forma de “pirámide” compuesta por las imágenes pasadas como parámetro con la intención de representarlas de una forma más vistosa y útil que permite compararlas. En primer lugar crea una serie de imágenes en blanco que actuarán como fondo de las pasadas como parámetro y las complementan para formar una nueva imagen de un tamaño específico que encaje en la composición de la pirámide. Todo el algoritmo está dedicado a ajustar las imágenes con su fondo correspondiente y a concatenarlas entre sí para que al final tenga forma de “pirámide”.

#### ■ Parámetros

**>bordes = [cv2.BORDER\_CONSTANT, cv2.BORDER\_DEFAULT]:** De entre todos los tipos de bordes que ofrece OpenCV estos dos muestran diferencias apreciables a simple vista. El verde constante rellena la imagen con un número fijo (por defecto 0) mientras que el borde por defecto refleja la imagen para completar los bordes.

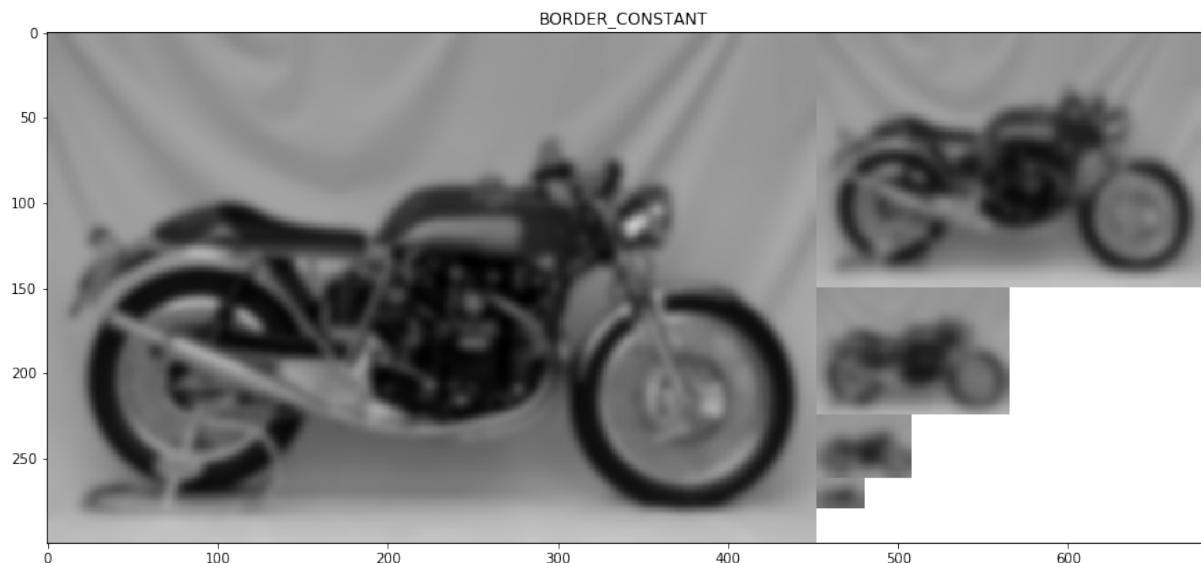
**sigma = 3:** Es un valor suficiente para conseguir emborronamiento.

**kernel = 19:** Por la teoría de calidad Six Sigma ( $19 = 6 * \sigma + 1$ )

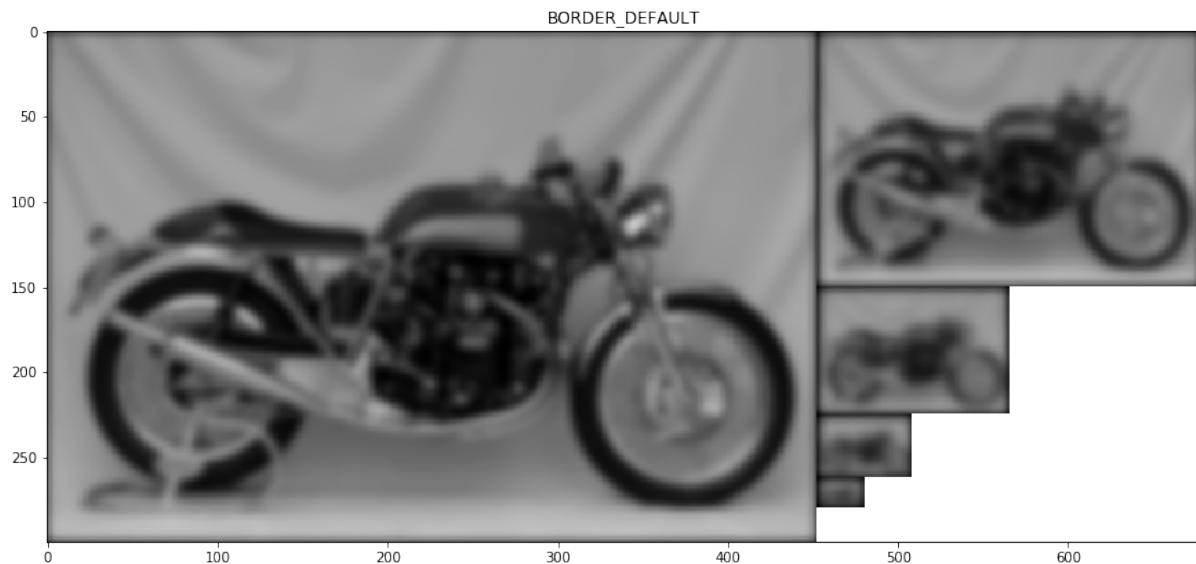
```
1 def ejer2a():
2     im = leer_imagen('imagenes/motorcycle.bmp', 0)
3     bordes = [cv2.BORDER_DEFAULT, cv2.BORDER_CONSTANT]
4     str_bordes = ['BORDER_CONSTANT', 'BORDER_DEFAULT']
5
6     for b in range(len(bordes)):
7         im2 = np.copy(im)
8         v_imgs = []
9         for i in range(5):
10             im2 = Gaussiana2D(im2, 19, 3, bordes[b])
11             v_imgs.append(im2)
12             im2 = imageSubsample(im2)
13     p = pyramid(v_imgs)
```

```
14     plt.figure(num=None, figsize=(15,15)) # Establecemos el tamaño  
15         de la imagen  
16     plt_imshow(p)  
17     plt.title(str_bordes[b])  
18     plt.show()
```

```
1 ejer2a()
```



**Figura31:** png



**Figura32:** png

De nuevo vemos la presencia del marco negro cuando utilizamos borde CONSTANT.

## 2b. Pirámide Laplaciana de 4 niveles

### ■ Qué vamos a calcular

Vamos a realizar una pirámide Laplaciana de 4 niveles (5 imágenes suponiendo como nivel 0 el tamaño original).

### ■ Cómo lo hacemos

Para hacer la pirámide Laplaciana nos ayudaremos de cuatro funciones: *Gaussian1D()*, *imageSubsample()* , *imageUpsample()* y *pyramid()*. Primero aplicamos la Gaussiana a la imagen, la disminuimos con *imageSubsample()*, la aumentamos con *imageUpsample()*, ajustamos las dimensiones para que vuelvan a ser las originales y restamos la imagen modificada a la original para obtener la laplaciana. Guardadamos este resultado y repetimos hasta cubrir los 4 niveles (5 contando el 0). Una vez tengamos todas las imágenes *pyramid()* se encargará de ordenarlas adecuadamente y mostrarlas en forma de pirámide.

### ■ Funcionamiento *Gaussian1D()*

Obtenemos el kernel de una función Gaussiana de tamaño *kernel\_size* ,  $\sigma = \text{sigma}$  y bordes de tipo *borde* y lo aplicamos una vez por filas y otra vez por columnas a la imagen pasada como parámetro.

### ■ Funcionamiento *imageSubsample()*

La imagen que se le pasa como parámetro es reducida a la mitad de columnas y filas.

#### ■ Funcionamiento *imageUpsample()*

Aumentamos el tamaño de la imagen pasada como argumento insertando cada fila y columna dos veces

#### ■ \*\*Funcionamiento *pyramid()*\_\*\*

Crea una imagen en forma de “pirámide” compuesta por las imágenes pasadas como parámetro con la intención de representarlas de una forma más vistosa y útil que permite compararlas. En primer lugar crea una serie de imágenes en blanco que actuarán como fondo de las pasadas como parámetro y las complementan para formar una nueva imagen de un tamaño específico que encaje en la composición de la pirámide. Todo el algoritmo está dedicado a ajustar las imágenes con su fondo correspondiente y a concatenarlas entre sí para que al final tenga forma de “pirámide”.

#### ■ Parámetros

**>bordes = [cv2.BORDER\_CONSTANT, cv2.BORDER\_DEFAULT]:** De entre todos los tipos de bordes que ofrece OpenCV estos dos muestran diferencias apreciables a simple vista. El verde constante rellena la imagen con un número fijo (por defecto 0) mientras que el borde por defecto refleja la imagen para completar los bordes.

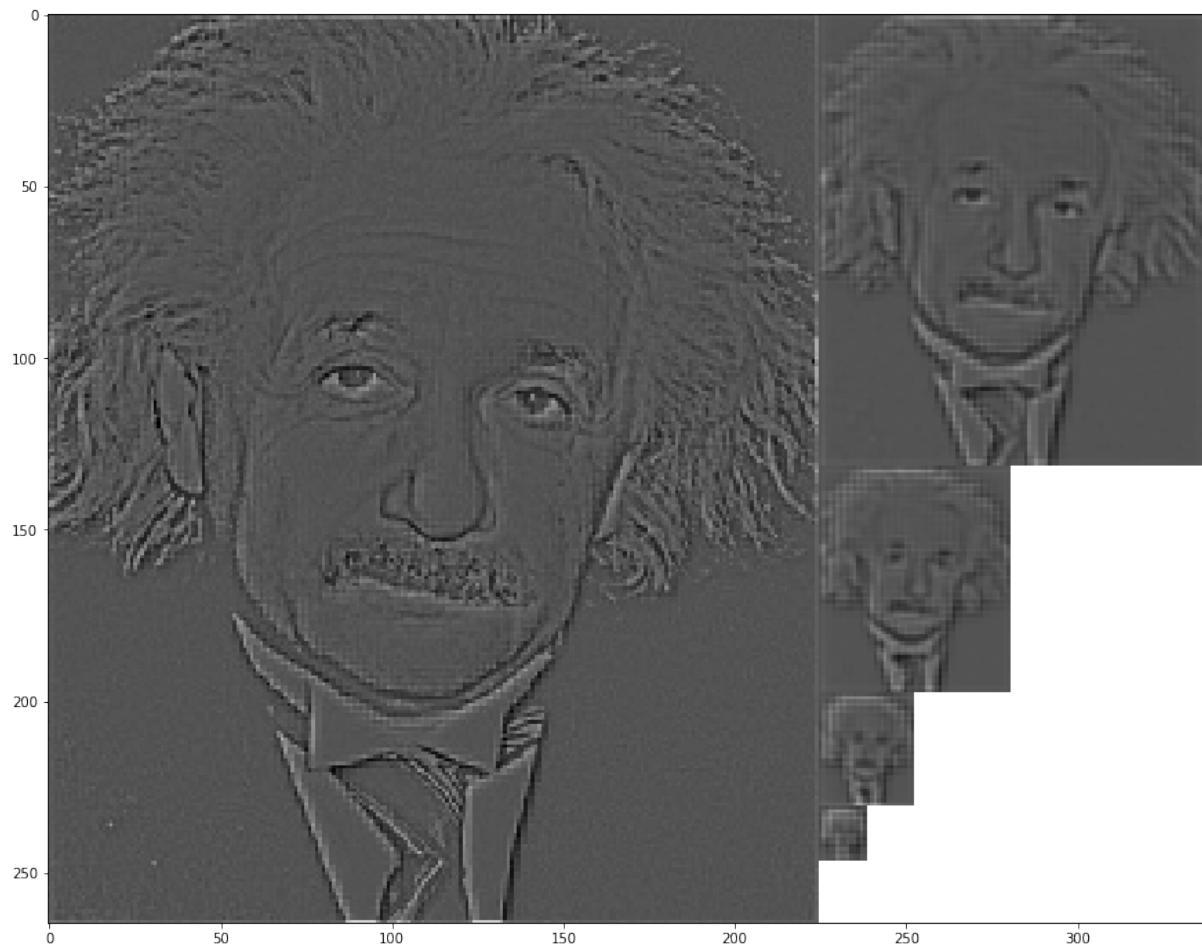
**sigma = 3:** Es un valor suficiente para conseguir emborronamiento.

**kernel = 19:** Por la teoría de calidad *Six Sigma* ( $19 = 6 * \sigma + 1$ )

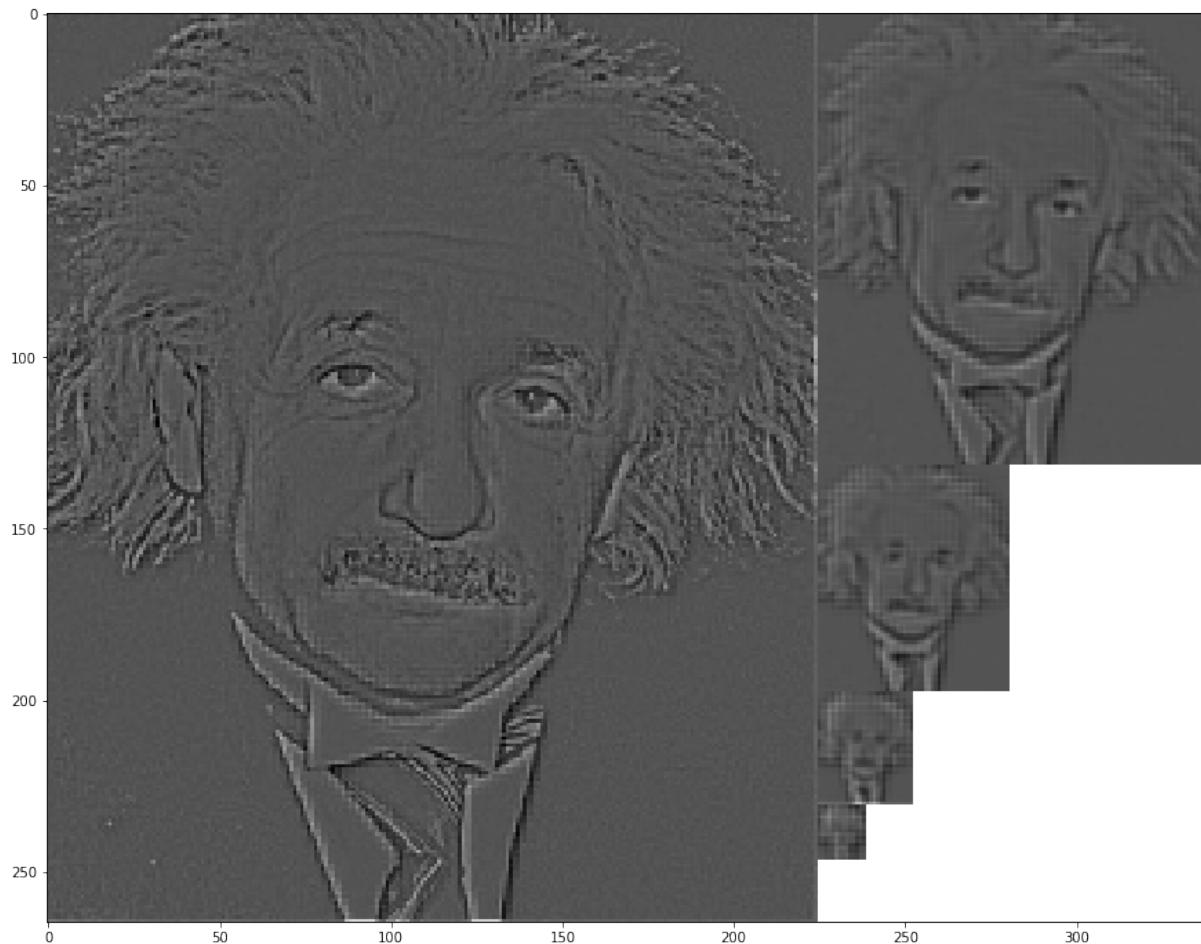
```
1 def ejer2b():
2     im = leer_imagen('imagenes/einstein.bmp', 0)
3     bordes = [cv2.BORDER_CONSTANT, cv2.BORDER_REFLECT]
4     str_bordes = ['BORDER_CONSTANT', 'BORDER_DEFAULT']
5
6     for b in bordes:
7         im2 = np.copy(im)
8         v_imgs = []
9         for i in range(5):
10            im3 = Gaussiana1D(im2, 5, 3, b)
11            im3 = imageSubsample(im3)
12            im4 = imageUpsample(im3)
13
14            if im2.shape[1] != im4.shape[1]:
15                dif = im2.shape[1] - im4.shape[1]
16                relleno = np.zeros((im4.shape[0], dif))
17                im4 = np.concatenate((im4, relleno), axis=1)
18            if im2.shape[0] != im4.shape[0]:
19                dif = im2.shape[0] - im4.shape[0]
```

```
20             relleno = np.zeros((dif, im2.shape[1]))
21             im4 = np.concatenate((im4, relleno), axis=0)
22
23             v_imgs.append(im2-im4)
24             im2 = im3
25             p = pyramid(v_imgs)
26             plt.figure(num=None, figsize=(15,15)) # Establecemos el tamaño
27                           de la imagen
27             plt.title('')
28             plt_imshow(p)
29             plt.show()
```

```
1 ejer2b()
```



**Figura33:** png



**Figura34:** png

En este caso el usar distintos tipos de bordes no afecta al resultado, al menos a simple vista.

## 2c. Blob Detection

- **Qué vamos a calcular**

Vamos a realizar la detección de puntos de interés de una imagen

- **Cómo lo hacemos**

Este problema lo resolveremos con la función *blob()*

- **Funcionamiento *blob()***

Recibe como parámetros el número de escalas a realizar (cuántas veces modificaremos sigma), la imagen en la que queremos encontrar los *blobs*, un valor inicial de sigma, un tamaño para el kernel y un umbral a partir del cual consideraremos importante el blob detectado.

Creamos una imagen *fin* de 0s del mismo tamaño que la original

Para cada una de las escalas:

- Calcularemos la Laplaciana de Gaussiana de la imagen, la normalizaremos multiplicándola por  $\sigma^2$  y elevaremos el resultado al cuadrado.
- Creamos una matriz (imagen) a la que llamaremos Z del mismo tamaño que la original con todos sus valores a 0 (negro) en la que iremos guardando los blobs detectados.
- Recorremos la Laplaciana de Gaussiana pixel a pixel comprobando que sea el máximo de su vecindario  $3 \times 3$  (es un blob). Si lo es, guardamos ese pixel en la matriz de 0s. - Una vez hayamos detectado los blobs de la imagen en esa escala de  $\sigma$ , normalizamos Z y dibujamos un círculo en todos aquellos blobs que superen el valor de umbral pasado. - Sumamos *fin* + Z - Aumentamos el valor de sigma  $\sigma = 14 * \sigma$

Devolvemos la imagen *fin* que contiene todos los círculos de las diferentes escalas que marcan zonas de interés en la imagen.

#### ■ Parámetros

>**escalas = 3**: Con 3 escalas distintas se reconocen los principales puntos de interés de la imagen en cuestión. Con mayor número de escalas satura la imagen con círculos que, subjetivamente, aportan poco.

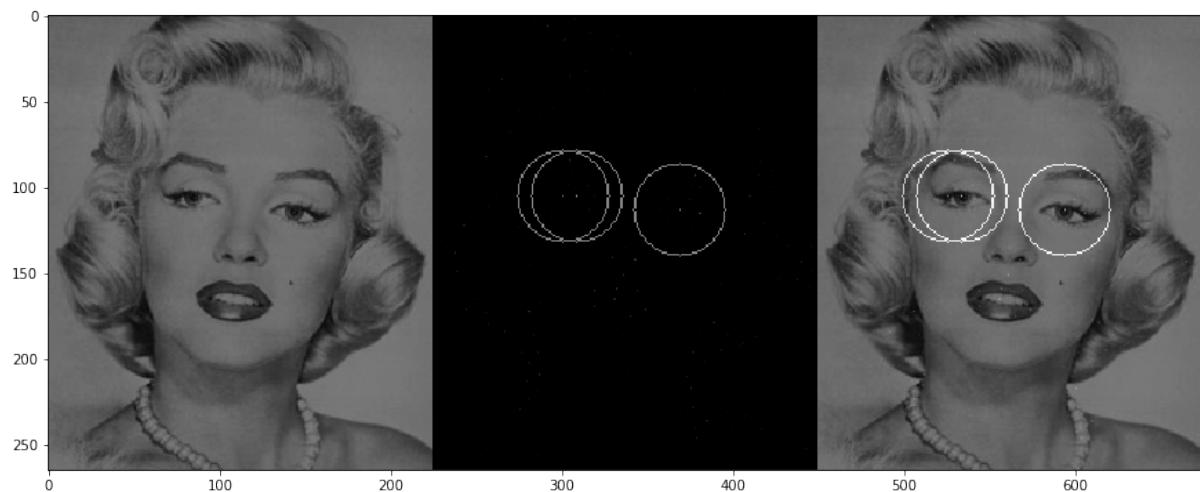
**sigma = 1.8**: A base de prueba y error es un valor que da buenos resultados.

**kernel = 15**: A base de prueba y error es un valor que da buenos resultados.

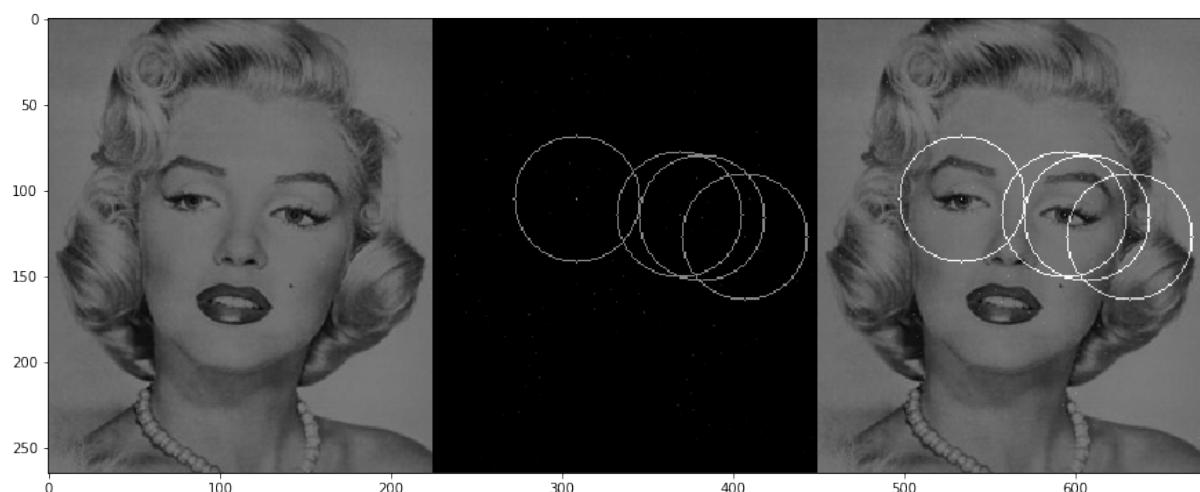
**umbral = 120**: A base de prueba y error es un valor que da buenos resultados.

```
1 def ejer2c():
2     im = leer_imagen('imagenes/marilyn.bmp', 0)
3     puntos_calientes = blob(3,im,1.8,15, 120)
4     plt.figure(num=None, figsize=(15,15)) # Establecemos el tamaño de
        la imagen
5     plt_imshow(puntos_calientes+im)
6     plt.show()
```

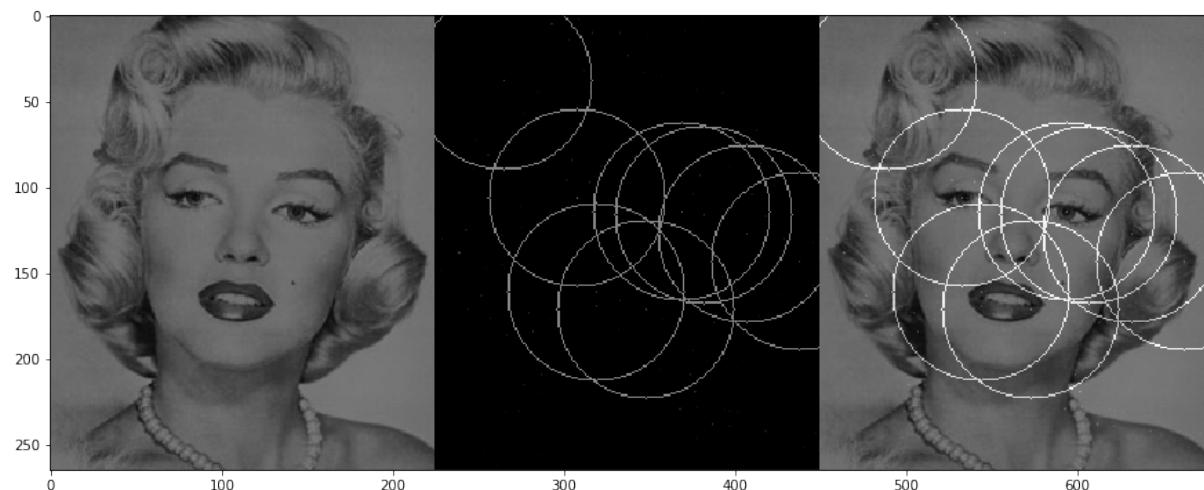
```
1 ejer2c()
```



**Figura35:** png



**Figura36:** png



**Figura37:** png



**Figura38:** png

Vemos que a mayor escala de sigma son más los puntos de interés que encuentra el algoritmo y que la importancia (subjetiva) de los mismos parece ser inversa a la escala en la que aparecen. Desde la

escala 1 hasta la 3 se identifican los ojos como zonas de interés.

## Ejercicio 3.

### 3. Imágenes Híbridas

#### ■ Qué vamos a calcular

Vamos a realizar la hibridación de dos imágenes. Una de altas frecuencias y otra de bajas para comprobar que al aumentar o disminuir la distancia con la que las observamos somos capaces de ver una u otra.

#### ■ Cómo lo hacemos

Este problema lo resolveremos con la función *hybrid()* la cual es llamada por *imprimeHybrid()*. Estas funciones nos devuelven las imágenes híbridas que luego convertiremos a pirámide Gaussiana con el mismo procedimiento que en el ejercicio 2a.

#### ■ Funcionamiento *hybrid()*

Recibe como parámetro las dos imágenes a hibridar (*im1, im2*), así como el tamaño de máscara y el valor de sigma (*kernel1, kernel2, sigma1, sigma2*) que se utilizará para calcular la gaussiana de cada una de ellas.

A la primera imagen solo le calcularemos su Gaussiana, mientras que a la segunda le calcularemos su Gaussiana únicamente como paso intermedio para obtener las frecuencias altas. Una vez tenemos las frecuencias bajas de la primera imagen (su Gaussiana) y las frecuencias altas de la segunda (original - su Gaussiana) sumamos las dos imágenes resultantes para obtener la imagen híbrida.

#### ■ Funcionamiento *imprimeHybrid()*

Recibe como parámetro los mismos argumentos que la función *hybrid()* ya que serán estos mismos los que serán pasados.

Esta función solo se encarga de leer las dos imágenes que se le pasarán como argumento a la función *hybrid*, llamar a dicha función y mostrar el resultado.

#### ■ Parámetros

>Para cada pareja de imágenes se han escogido distintos sigmas y tamaños de máscara pero de todos se ha partido probando con  $\sigma = 3$  y máscara =  $6 * \sigma + 1$ . A partir de ahí se ha ido jugando con los valores según requiriesen mayor o menor difuminación.

<sigma img1, sigma img2, kernel\_size img1, kernel\_size img2> **Pareja 1:** 19,17,3,5

**Pareja 2:** 15,7,5,5

**Pareja 3:** 13,61,3,15

**Pareja 4:** 13,20,2,7

**Pareja 5:** 31,61,5,10

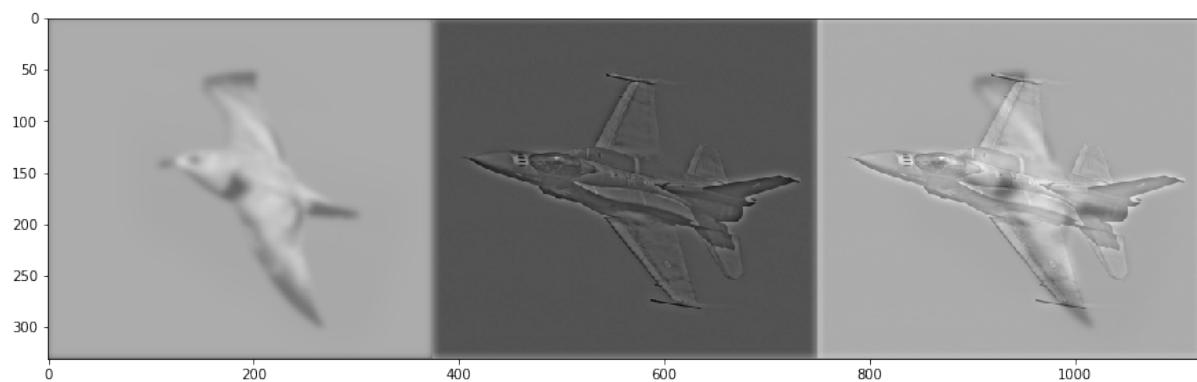
Para la pirámide Gaussiana:

**sigma = 3:** A base de prueba y error es un valor que da buenos resultados.

**kernel = 5:** A base de prueba y error es un valor que da buenos resultados.

```
1 def ejer3():
2     v_img = []
3
4     im1 = imprimeHybrid('imagenes/bird.bmp', 'imagenes/plane.bmp',
5                           19,17,3,5)
6     im2 = imprimeHybrid('imagenes/motorcycle.bmp', 'imagenes/bicycle.
7                           bmp', 15,7,5,5)
8     im3 = imprimeHybrid('imagenes/dog.bmp', 'imagenes/cat.bmp',
9                           13,61,3,15)
10    im4 = imprimeHybrid('imagenes/einstein.bmp', 'imagenes/marilyn.bmp'
11                           , 13,20,2,7)
12    im5 = imprimeHybrid('imagenes/fish.bmp', 'imagenes/submarine.bmp',
13                           31,61,5,10)
14    plt.show()
15
16    v_img.append(im1)
17    v_img.append(im2)
18    v_img.append(im3)
19    v_img.append(im4)
20    v_img.append(im5)
21
22    for img in v_img:
23        v_pyramid = []
24        im = img
25        for i in range(5):
26            im = Gaussiana1D(im, 5, 3, cv2.BORDER_REFLECT)
27            v_pyramid.append(im)
28            im = imageSubsample(im)
29
30        p = pyramid(v_pyramid)
31        plt.figure(num=None, figsize=(15,15)) # Establecemos el tamaño
32                                         de la imagen
33        plt_imshow(p)
34        plt.show()
```

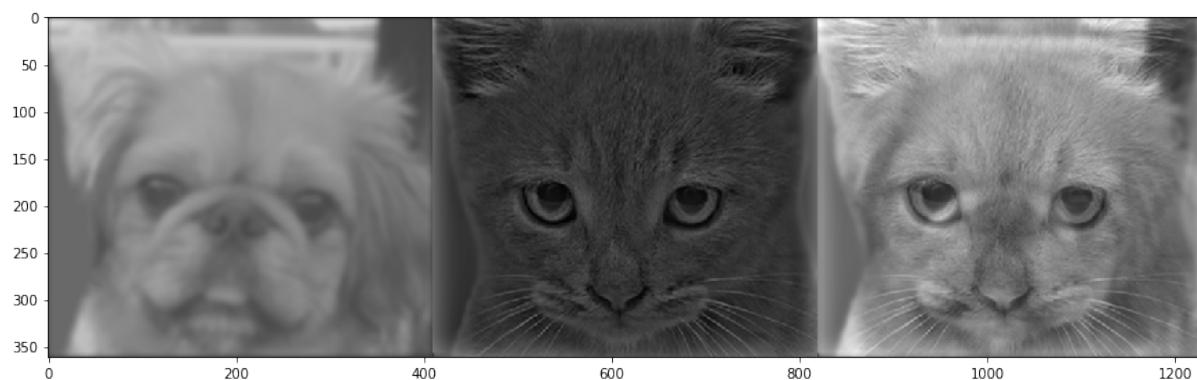
```
1 ejer3()
```



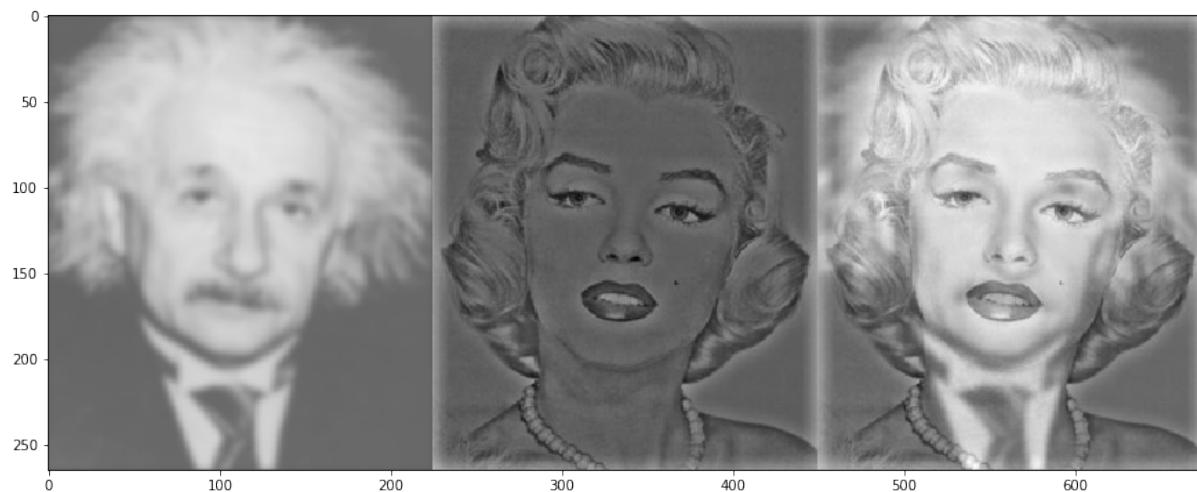
**Figura39:** png



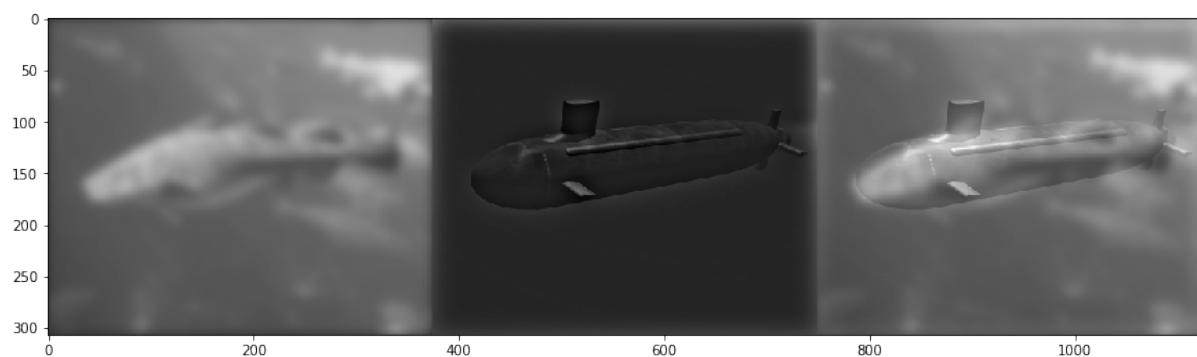
**Figura40:** png



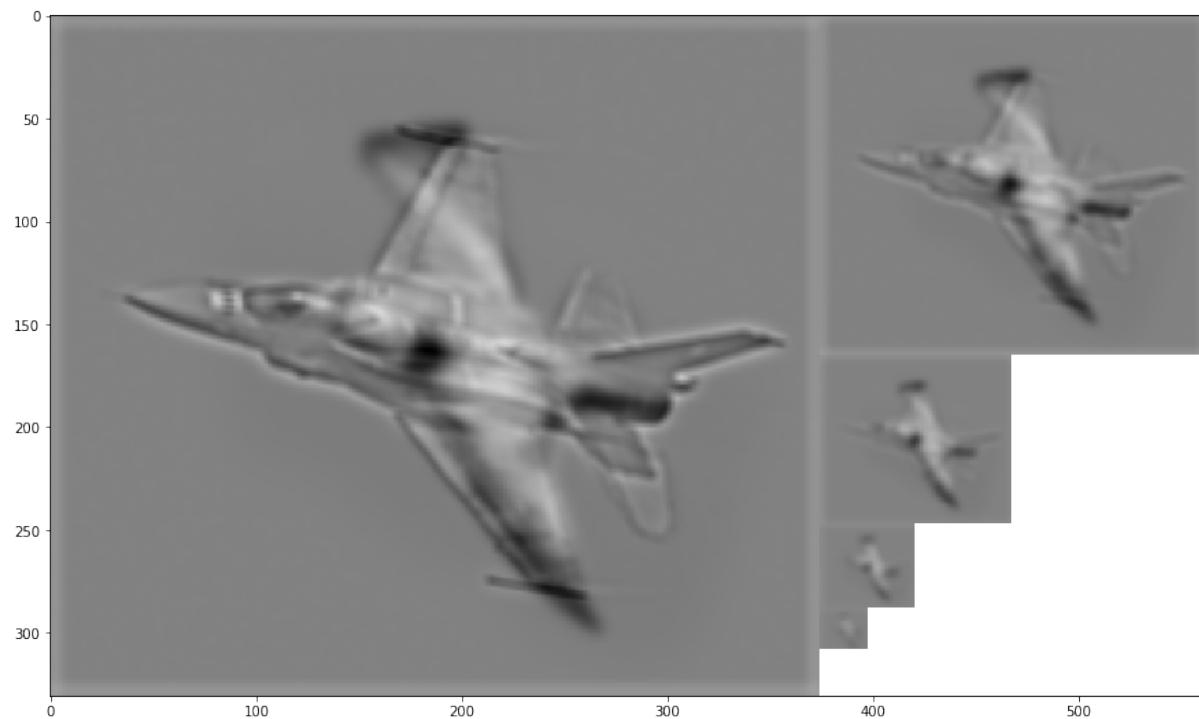
**Figura41:** png



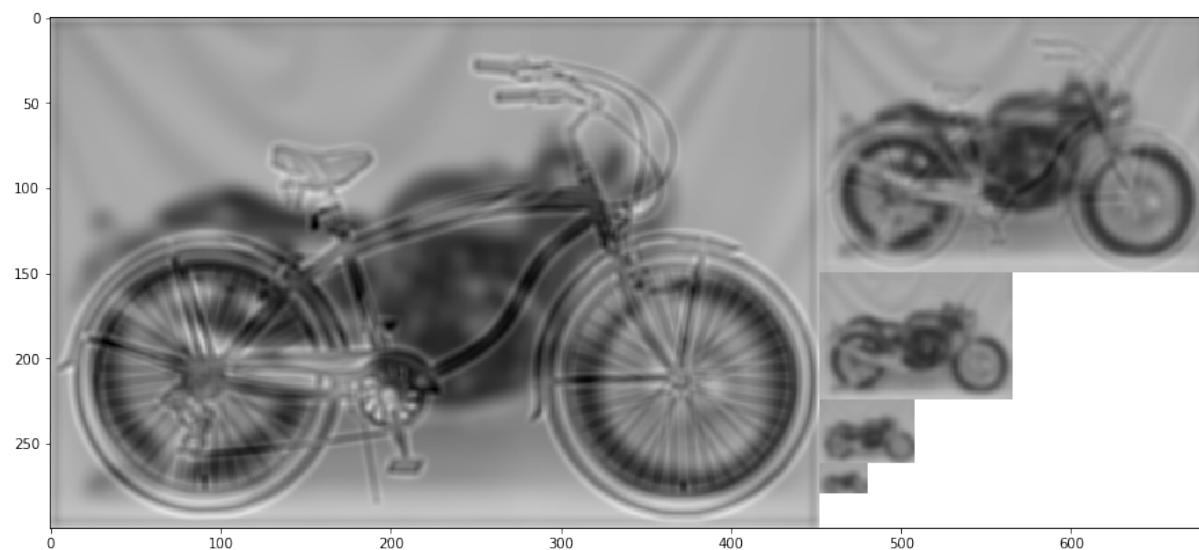
**Figura42:** png



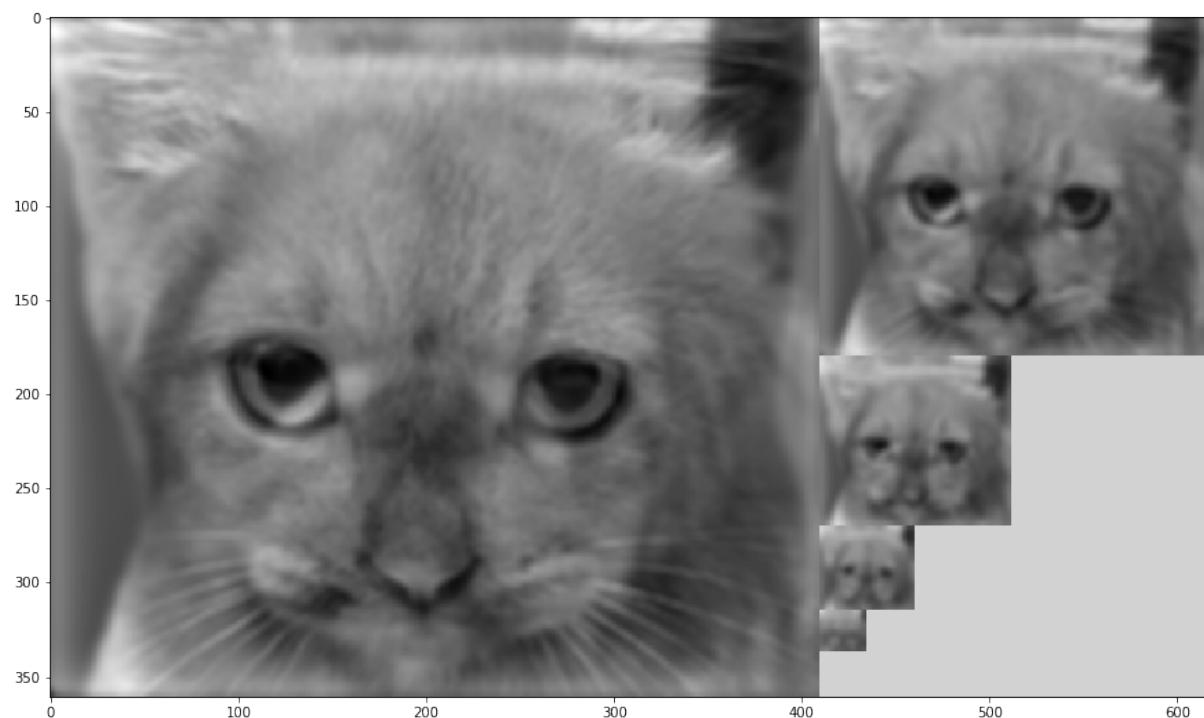
**Figura43:** png



**Figura44:** png



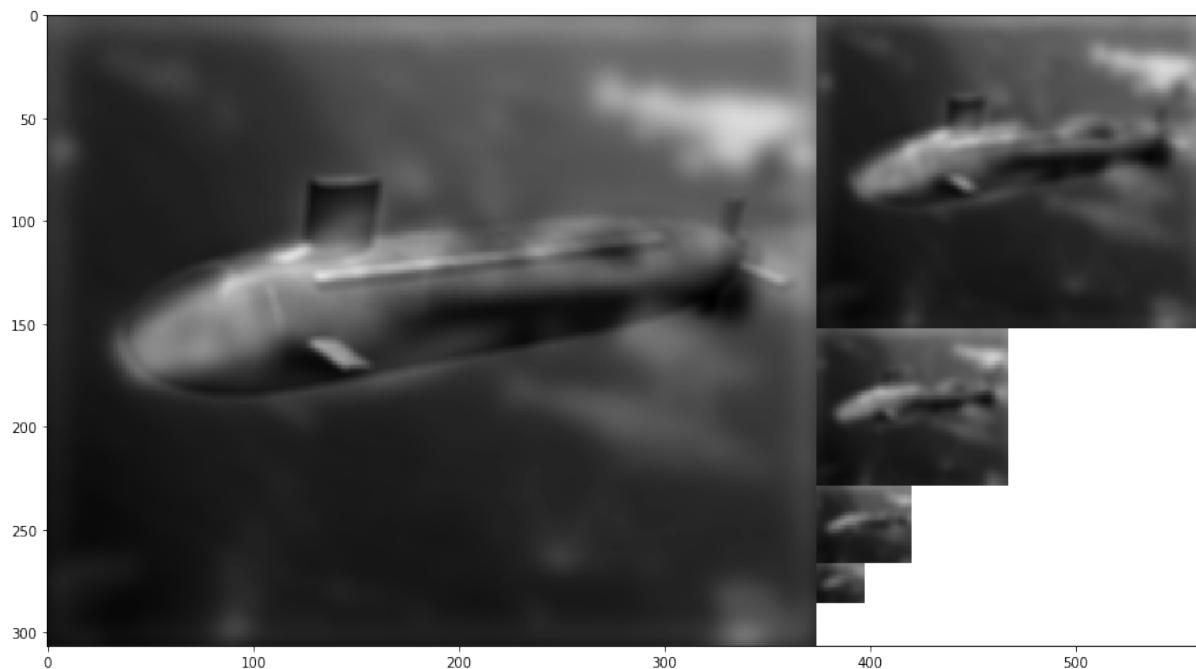
**Figura45:** png



**Figura46:** png



**Figura47:** png



**Figura48:** png

Con este ejercicio queda claro que hay imágenes más fáciles de hibridar que otras. Por ejemplo perro/gato y pez/submarino son imágenes complicadas, pero pájaro/avión o bici/moto son más fáciles de combinar.

Con las pirámides vemos que en la imagen más grande es más reconocible la imagen de frecuencias altas y a medida que disminuye toma fuerza la imagen de frecuencias bajas.

## Bonus.

### Bonus3. Hibridación con imágenes propias

#### ■ Qué vamos a calcular

Vamos a hacer lo mismo que en el ejercicio 3 con la diferencia de que las imágenes son escogidas por mí y no coinciden en dimensiones.

#### ■ Cómo lo hacemos

El procedimiento es el mismo que en el ejercicio 3 salvo que antes de llamar a las funciones de hibridación debemos asegurarnos de que las dimensiones de ambas imágenes son la misma. Para ello restamos a la imagen más grande tantas filas y columnas como sean necesarias hasta ajustarse a la imagen más pequeña.

**■ Parámetros**

><sigma img1, sigma img2, kernel\_size img1, kernel\_size img2> **Pareja:** 25, 31, 4, 5

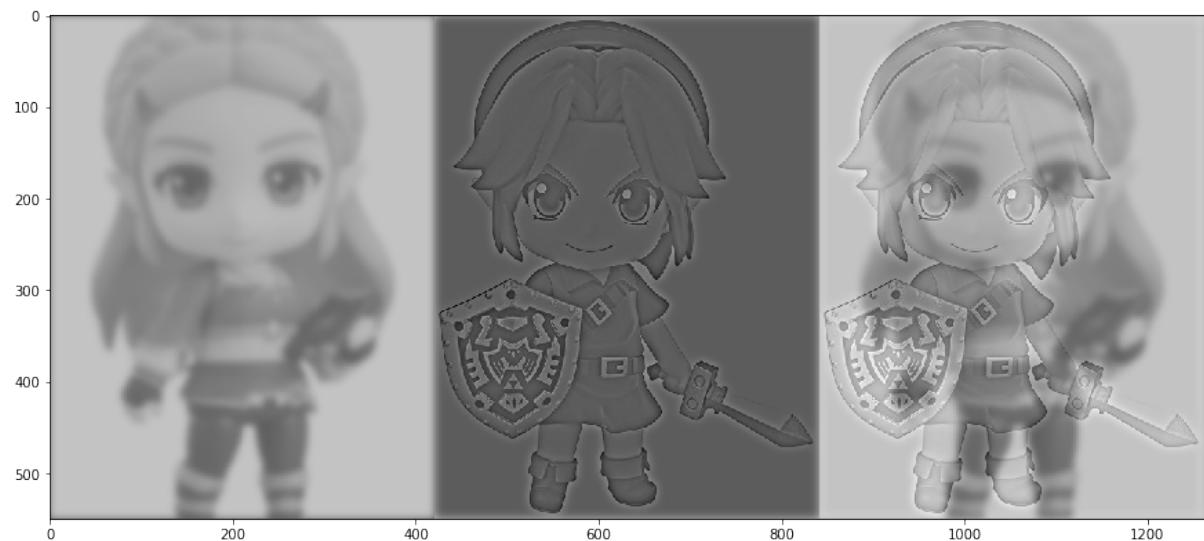
Para la pirámide Gaussiana:

**sigma = 3:** A base de prueba y error es un valor que da buenos resultados.

**kernel = 5:** A base de prueba y error es un valor que da buenos resultados.

```
1 def bonus3():
2     zelda = leer_imagen('imagenes/zelda.png', 0)
3     link = leer_imagen('imagenes/link.png', 0)
4
5     dif1 = zelda.shape[0] - link.shape[0]
6     dif2 = zelda.shape[1] - link.shape[1]
7
8     l11 = int(dif1 // 2)
9     l12 = int(dif1 - l11)
10
11    l21 = int(dif2 // 2)
12    l22 = int(dif2 - l21)
13
14    new_zelda = zelda[l11 : zelda.shape[0] - l12, l21 : zelda.shape[1]
15                  - l22]
16
17    mix, hyb = hybrid(new_zelda, link, 25, 31, 4, 5)
18    plt.figure(num=None, figsize=(15,15)) # Establecemos el tamaño de
19          # la imagen
20    plt_imshow(mix)
21
22    v_pyramid = []
23    im = hyb
24    for i in range(5):
25        im = Gaussiana1D(im, 5, 3, cv2.BORDER_REFLECT)
26        v_pyramid.append(im)
27        im = imageSubsample(im)
28
29    p = pyramid(v_pyramid)
30    plt.figure(num=None, figsize=(15,15)) # Establecemos el tamaño de
31          # la imagen
32    plt_imshow(p)
33    plt.show()
```

```
1 bonus3()
```



**Figura49:** png



**Figura50:** png