

Practical-2020

October 30, 2020

1 Procesamiento de Imágenes Digitales

Visión Computacional 2020-21 Practica 1. 6 de octubre de 2020

Autor1: Alejandro Muñoz Navarro Autor2: Fernando Planes Ruiz

1.1 Objetivos

Los objetivos de esta práctica son:

- * Repasar algunos conceptos de filtrado de imágenes y programar algunas rutinas para suavizado y extracción de bordes.
- * Implementar un algoritmo de segmentación de imágenes y otro de extracción de líneas mediante la transformada de Hough.

1.2 Requerimientos

Para esta práctica es necesario disponer del siguiente software:

- * Python 2.7 ó 3.X, preferiblemente el segundo
- * Jupyter <http://jupyter.org/>.
- * Los paquetes `python-pip` y/o `python-pip3` y el paquete “PyMaxFlow”
- * Las librerías científicas de Python: NumPy (`python-numpy`), SciPy (`python-scipy`) y Matplotlib (`python-matplotlib`).
- * El paquete `python-pygame`
- * La librería OpenCV, que puedes instalar desde el paquete `python-opencv`.

Las versiones preferidas del entorno de trabajo puedes consultarlas en el Aula Virtual en el archivo “ConfiguracionPC2018.txt”.

El material necesario para la práctica se puede descargar del Aula Virtual.

1.3 Condiciones

- La fecha límite de entrega será el lunes 26 de octubre a las 23:55.
- La entrega consiste en dos archivos con el código, resultados y respuestas a los ejercicios:
 1. Un “notebook” de Jupyter con el fuente y los resultados (ejecuta “Restart & Run all” antes de guardar)
 2. Un documento “pdf” generado a partir del fuente de Jupyter, por ejemplo usando el comando `jupyter nbconvert --execute --to pdf notebook.ipynb`. Asegúrate de que el documento “pdf” contiene todos los resultados correctamente ejecutados (previamente ejecuta en el menú “Kernel” la opción “Restart & Run All”).
- Las respuestas a los ejercicios debes introducirlas en tantas celdas de código o texto como creas necesarias, insertadas inmediatamente después de un enunciado y antes del siguiente.
- La puntuación del ejercicio 7 es el triple que el resto.
- Las prácticas puede realizarse en parejas. Sólo es necesario que uno de los miembros del equipo entregue la práctica.

1.4 Instala el entorno de trabajo

1. Instala el entorno de trabajo.

```
apt install python apt install python-scipy apt install python-numpy apt  
install python-matplotlib apt install python-opencv apt install jupyter  
apt install jupyter-nbconvert
```

Para para trabajar con la versión 3.X de Python, basta sustituir la palabra “python” por “python3” en los comandos anteriores.

2. Instala el paquete PyMaxflow

```
pip install PyMaxflow o pip3 install PyMaxflow
```

Si no tienes el paquete “pip” debes instalarlo: apt install python-pip o apt install python3-pip

3. Instala el paquete “pygame”

```
apt install python-pygame
```

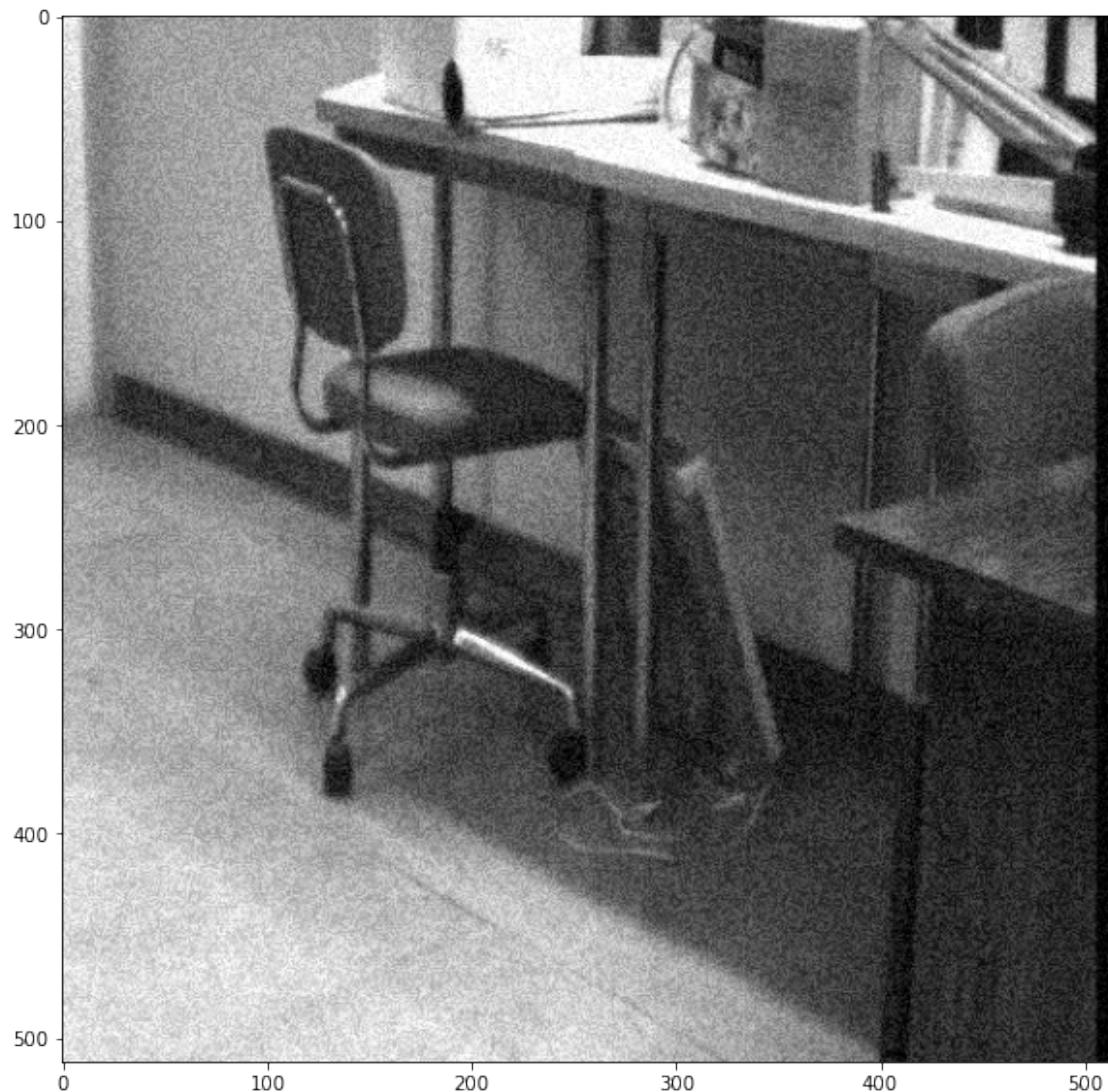
En Python 3.X, la versión 18.04 de Ubuntu no tiene el paquete “python3-pygame” pero puedes instalarlo con la herramienta pip: pip install pygame

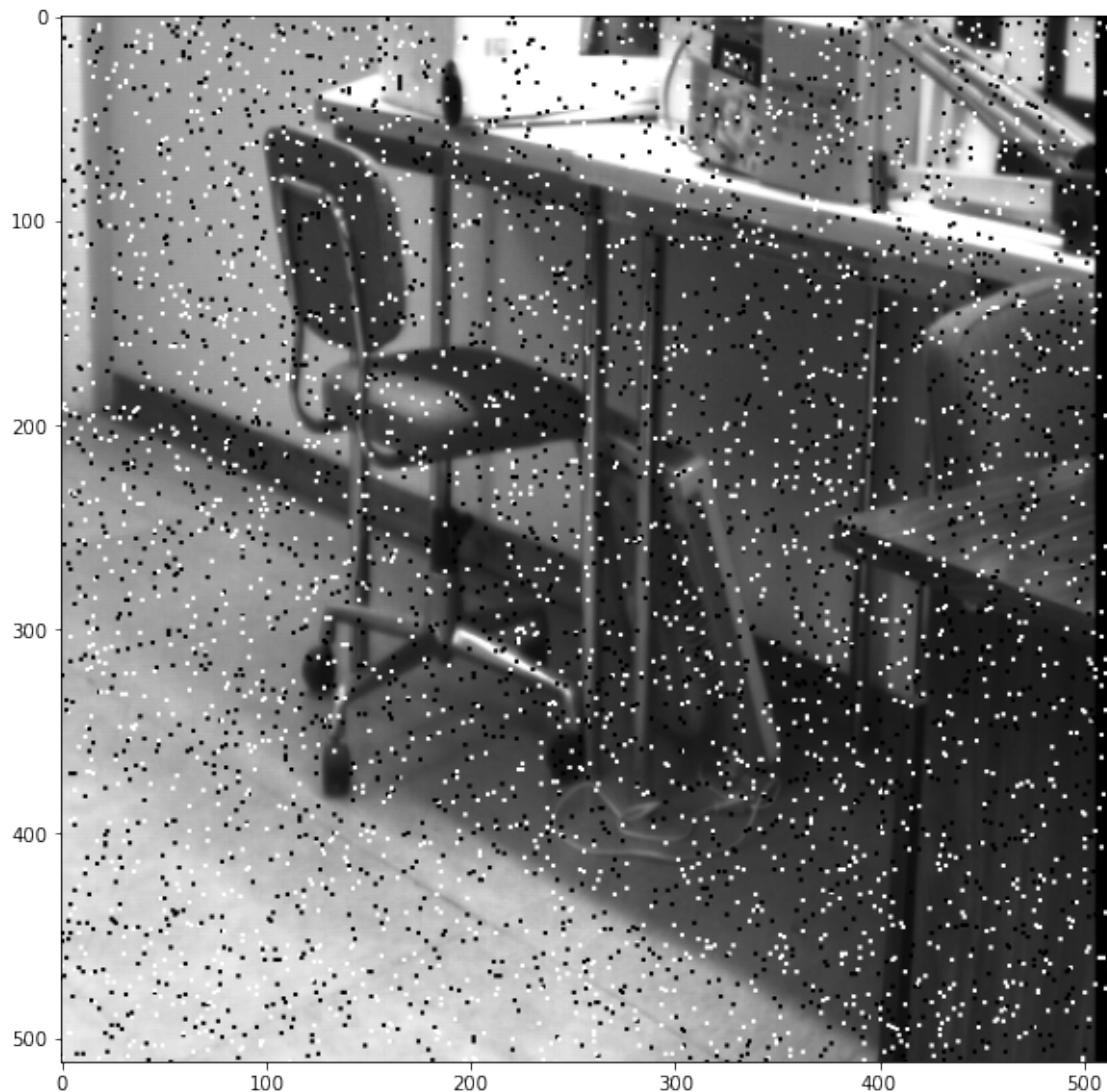
1.5 Filtrado

Para realizar las convoluciones utiliza la función convolve o convolve1d de `scipy.ndimage`.

Carga y muestra las imágenes `escgaus.bmp` y `escimp5.bmp` que están contaminadas respectivamente con ruido de tipo gaussiano e impulsional. En los siguientes ejercicios también puedes utilizar otras imágenes que te parezcan interesantes.

```
[1]: from matplotlib import pyplot as plt  
import cv2  
  
imgaus = cv2.cvtColor(cv2.imread("./imagenes/escgaus.bmp"), cv2.COLOR_BGR2GRAY)  
plt.figure(figsize = (10,10))  
plt.imshow(imgaus, cmap='gray')  
plt.show()  
  
imimp = cv2.cvtColor(cv2.imread("./imagenes/escimp5.bmp"), cv2.COLOR_BGR2GRAY)  
plt.figure(figsize = (10,10))  
plt.imshow(imimp, cmap='gray')  
plt.show()
```





Ejercicio 1. Escribe una función `masc_gaus(sigma, n)` que construya una máscara de una dimensión de un filtro gaussiano de tamaño n y varianza σ^2 . Filtra las imágenes anteriores con filtros gaussianos bidimensionales de diferentes tamaños de n , y/o σ^2 .

En este ejercicio tenéis que implementar vosotros la función que construye la máscara. No podéis usar funciones que construyan la máscara o realicen el filtrado automáticamente.

Muestra cómo afecta este filtrado a los dos tipos de ruido que contaminan las imágenes anteriores y discute los resultados. Pinta alguna de las máscaras utilizadas.

```
[2]: import numpy as np
from scipy.ndimage import convolve, convolve1d

def masc_gaus(sigma, n):
```

```

g = np.exp(-(np.arange(n) - (n - 1) / 2) ** 2 / (2 * sigma ** 2))
return np.array([g / g.sum()])

```

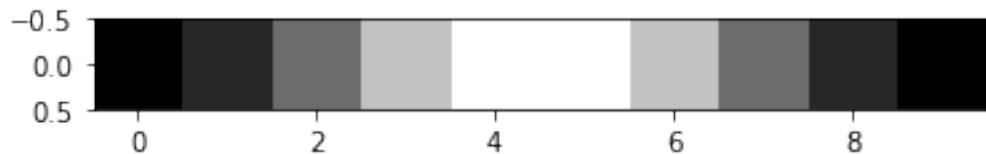
Dibujamos una máscara unidimensional para comprobar

```

[3]: sigma = 2
n = 10

plt.imshow(masc_gaus(sigma, n), cmap='gray')
plt.show()

```



A continuación escribimos el código para obtener máscaras bidimensionales

```

[4]: def masc_gaus_bi(sigma, n):
    h = masc_gaus(sigma, n)
    return h.T * h

```

Dibujamos un par de máscaras a modo de comprobación

```

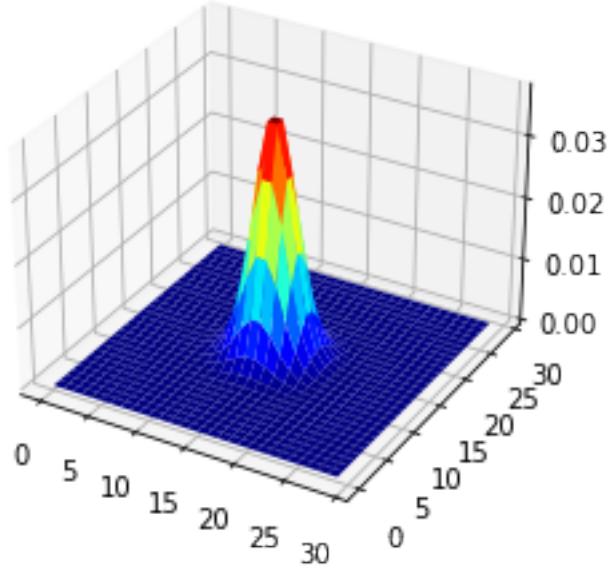
[5]: def plot_gaus_bi_Kernel(sigma, n):
    fig = plt.figure()
    ax = fig.gca(projection = '3d')
    jet = plt.get_cmap('jet')
    Z = masc_gaus_bi(sigma, n)

    aux = n // 2
    x, y = np.mgrid[0 :n, 0 :n]
    surf = ax.plot_surface(x, y, Z, rstride = 1, cstride = 1, cmap = jet, u
    ↪linewidth = 0)
    ax.set_zlim3d(Z.min(), Z.max())
    ax.set_title('Sigma = ' + str(sigma) + '; n = ' + str(n))
    plt.show()

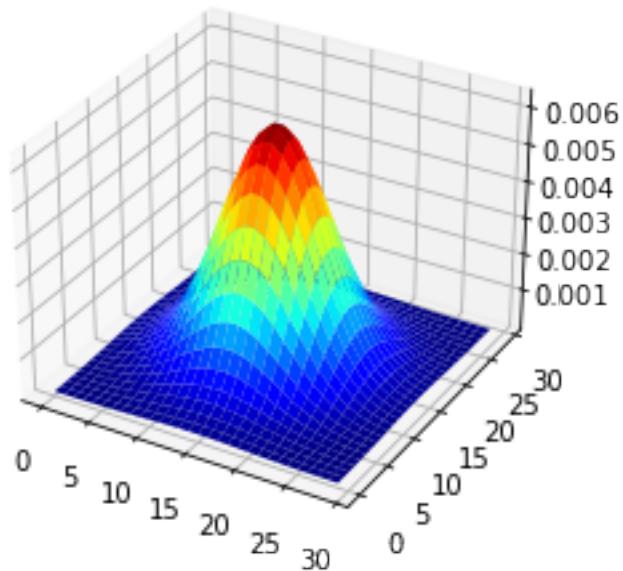
plot_gaus_bi_Kernel(2, 30)
plot_gaus_bi_Kernel(5, 30)

```

$\text{Sigma} = 2; n = 30$



$\text{Sigma} = 5; n = 30$

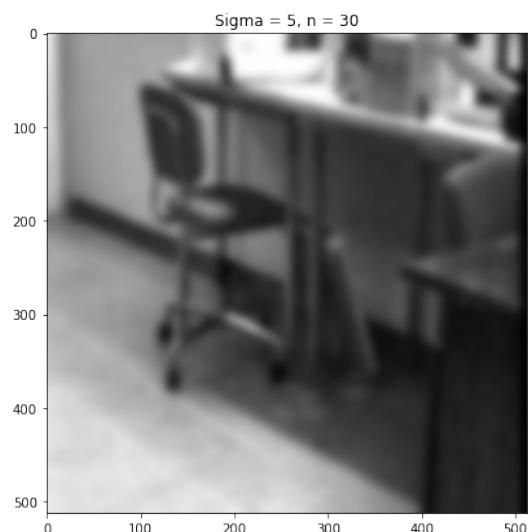
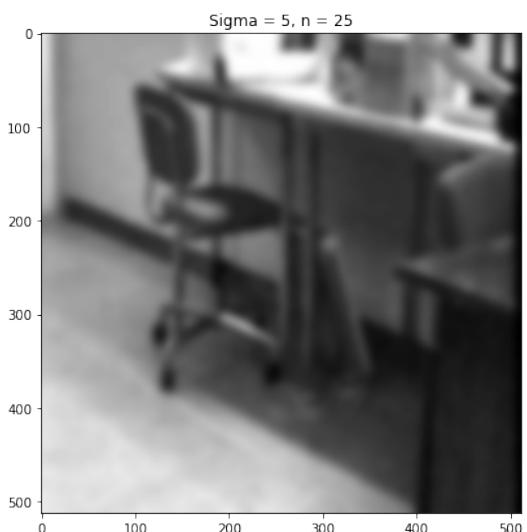
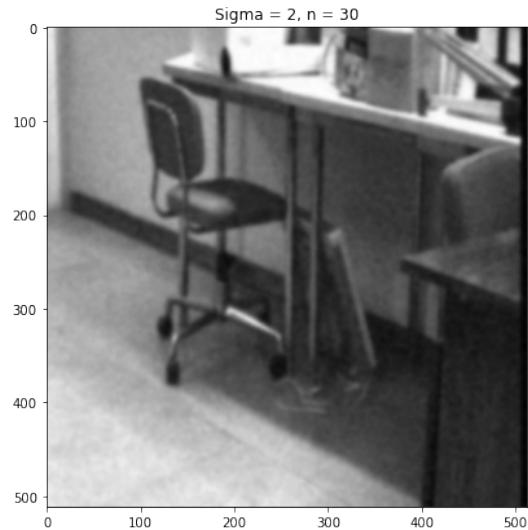
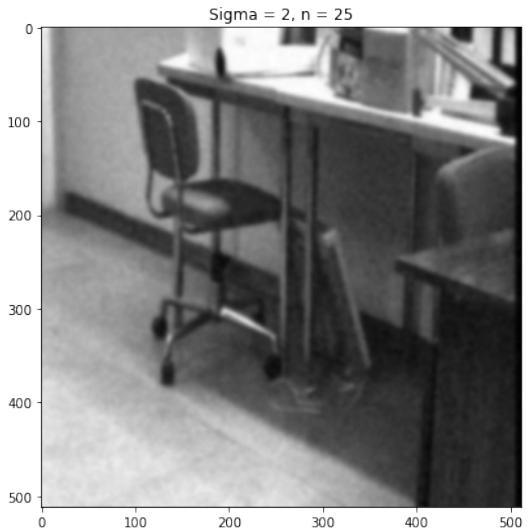


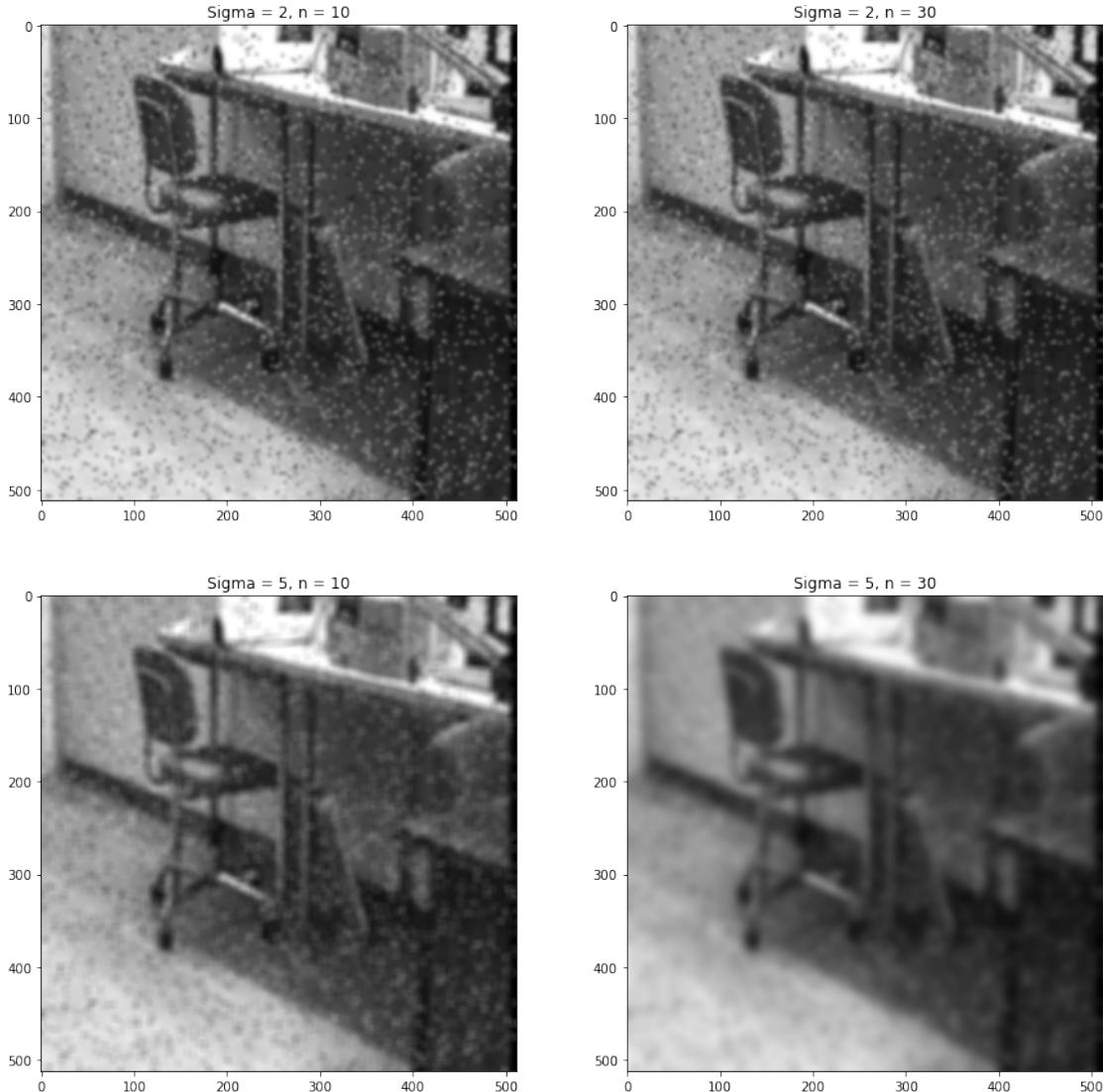
Usamos las máscaras bidimensionales para filtrar las imágenes probando distintos parámetros

```
[6]: def filtrado_gaussiano_bidimensional(img, sigma, n):
    h = masc_gaus_bi(sigma, n)
    return convolve(img, h)

def plot_filtrado_gaussiano_bidimensional(img, sigmas, ns, width, height):
    fig, axs = plt.subplots(len(sigmas), len(ns), figsize = (width, height))
    for sigma in range(len(sigmas)):
        for n in range(len(ns)):
            img_filtrada = filtrado_gaussiano_bidimensional(img, sigmas[sigma], ns[n])
            axs[sigma, n].imshow(img_filtrada, cmap='gray')
            axs[sigma, n].set_title('Sigma = ' + str(sigmas[sigma]) + ', n = ' + str(ns[n]))
    plt.show()

plot_filtrado_gaussiano_bidimensional(imgaus, [2, 5], [25, 30], 15, 15)
plot_filtrado_gaussiano_bidimensional(imimp, [2, 5], [10, 30], 15, 15)
```





Discusión de resultados: Como cabía esperar, en la imagen contaminada con ruido gaussiano si que somos capaces de atenuarlo pero sin embargo, en la imagen con ruido impulsivo no, pues más que atenuar ese ruido, estaremos manchando los píxeles de alrededor de los contaminados.

Ejercicio 2. Escribe una función `masc_deriv_gaus(sigma, n)` que construya una máscara de una dimensión de un filtro derivada del gaussiano de tamaño n y varianza σ^2 . Convierte de RGB a escala de grises y filtra la imagen `telefonica.jpg` con filtros bidimensionales de derivada del gaussiano para extraer los bordes de la imagen. Prueba con diferentes valores de n y/o σ^2 .

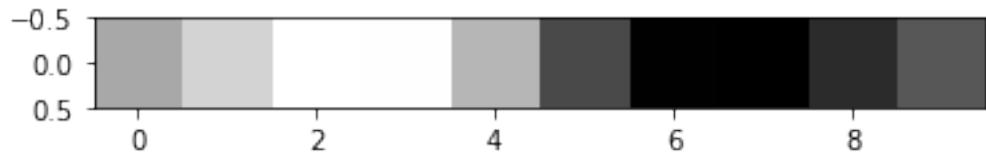
Muestra y discute los resultados. Pinta alguna de las máscaras construidas.

```
[7]: def masc_deriv_gaus(sigma, n):
    t = (np.arange(n) - (n - 1) / 2) / sigma
    return np.array([(-t / sigma) * np.exp(-1 / 2 * t ** 2)])
```

Dibujamos una máscara unidimensional para comprobar

```
[8]: sigma = 2
n = 10

plt.imshow(masc_deriv_gaus(sigma, n), cmap='gray')
plt.show()
```



A continuación escribimos el código para obtener máscaras bidimensionales

```
[9]: def masc_gaus_grad_bi_Dx(sigma, n):
    hd = masc_deriv_gaus(sigma, n)
    h = masc_gaus(sigma, n)
    return hd * h.T

def masc_gaus_grad_bi_Dy(sigma, n):
    hd = masc_deriv_gaus(sigma, n)
    h = masc_gaus(sigma, n)
    return hd.T * h
```

Dibujamos algunas de las máscaras bidimensionales para comprobar

```
[10]: def plot_gaus_grad_bi_Kernel(sigma, n):
    fig = plt.figure()
    ax = fig.gca(projection = '3d')
    jet = plt.get_cmap('jet')
    Z = masc_gaus_grad_bi_Dx(sigma, n)
    aux = n // 2
    x, y = np.mgrid[0 :n, 0 :n]
    surf = ax.plot_surface(x, y, Z, rstride = 1, cstride = 1, cmap = jet, linewidth = 0)
    ax.set_zlim3d(Z.min(), Z.max())
    ax.set_title('Sigma = ' + str(sigma) + '; n = ' + str(n))
    plt.show()
    plt.imshow(Z, cmap='gray')
    plt.show()
    fig = plt.figure()
    ax = fig.gca(projection = '3d')
    jet = plt.get_cmap('jet')
    Z = masc_gaus_grad_bi_Dy(sigma, n)
```

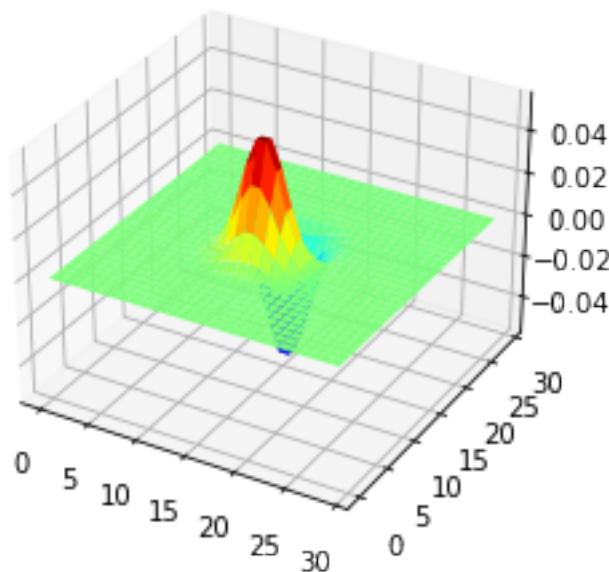
```

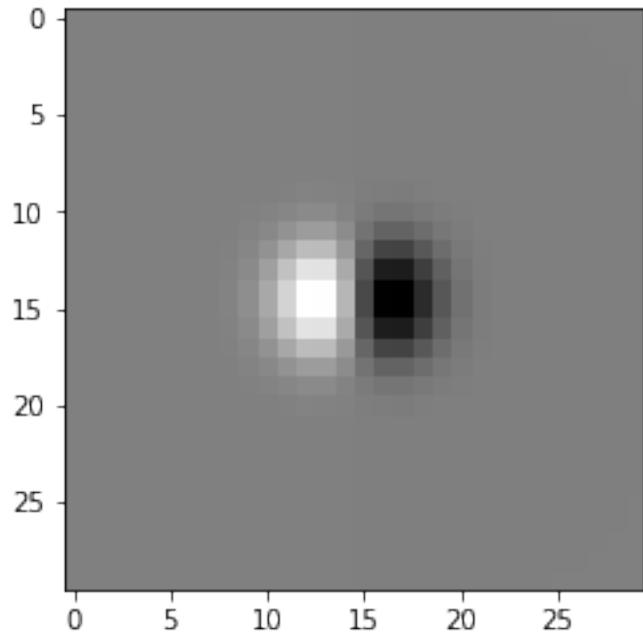
aux = n // 2
x, y = np.mgrid[0 :n, 0 :n]
surf = ax.plot_surface(x, y, Z, rstride = 1, cstride = 1, cmap = jet, u
→ linewidth = 0)
ax.set_zlim3d(Z.min(), Z.max())
ax.set_title('Sigma = ' + str(sigma) + '; n = ' + str(n))
plt.show()
plt.imshow(Z, cmap='gray')
plt.show()

plot_gaus_grad_bi_Kernel(2, 30)
plot_gaus_grad_bi_Kernel(5, 30)

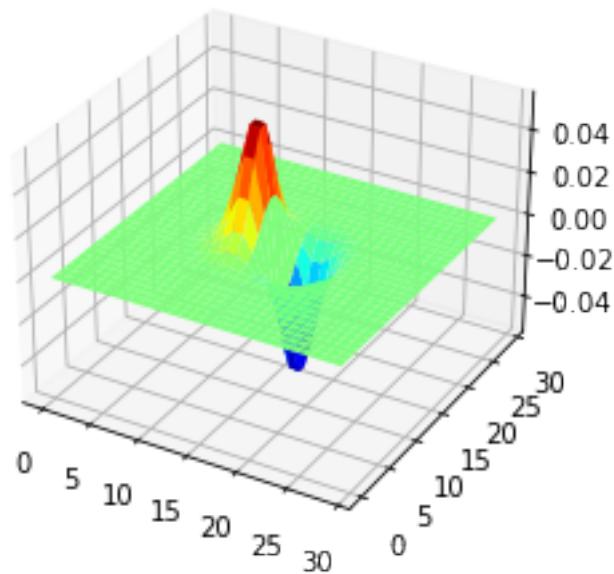
```

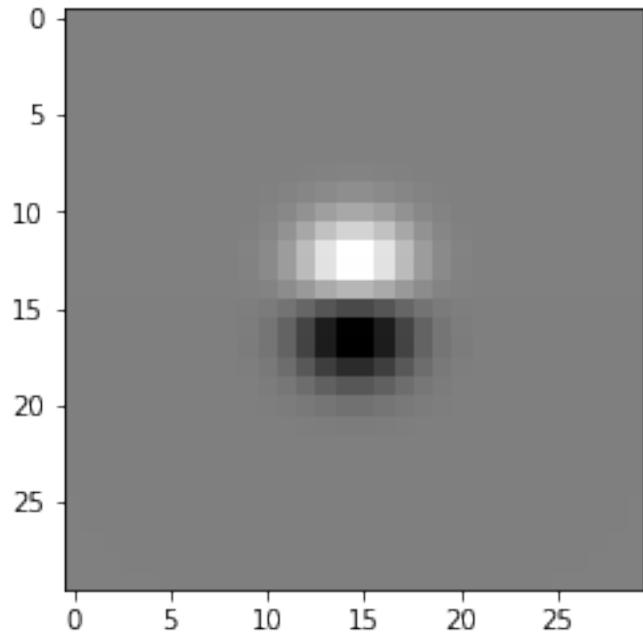
Sigma = 2; n = 30



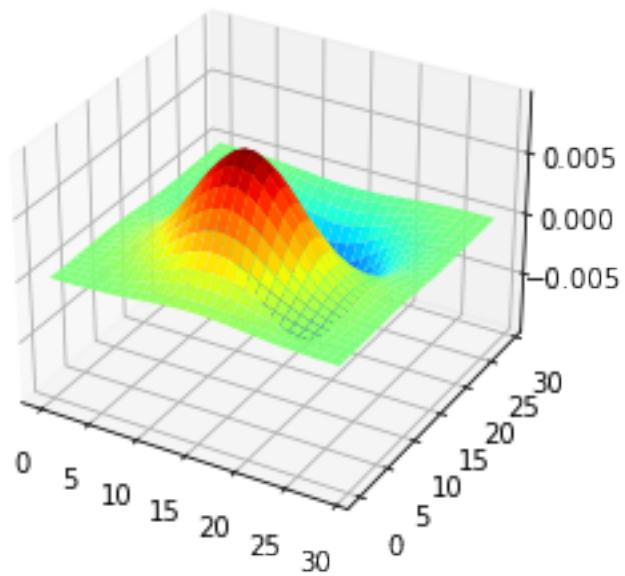


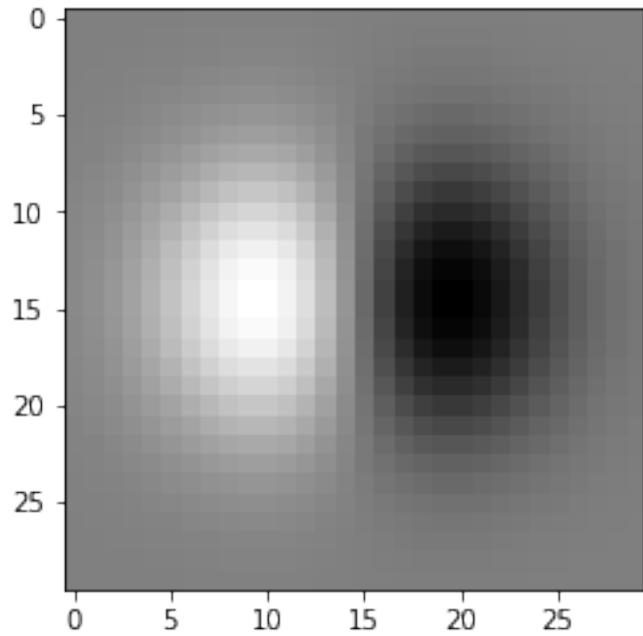
Sigma = 2; n = 30



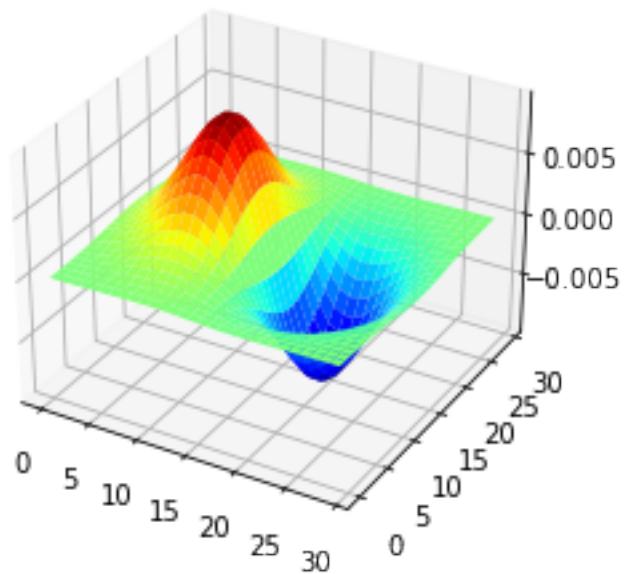


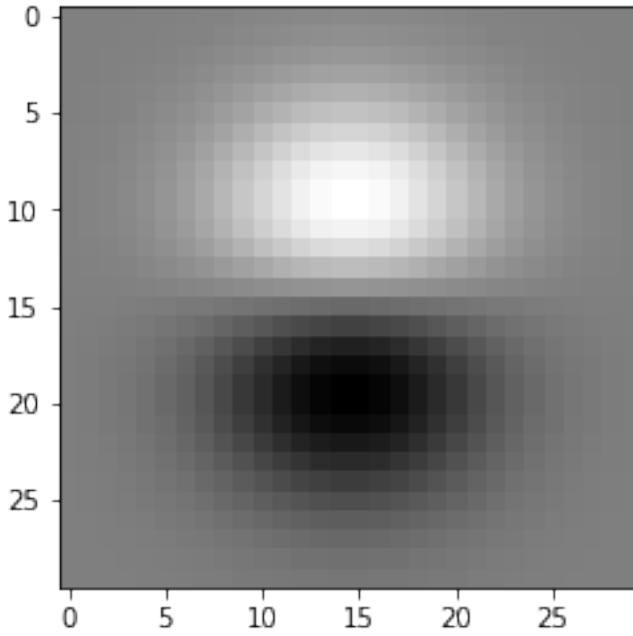
Sigma = 5; n = 30





Sigma = 5; n = 30





Hacemos algunas pruebas de extracción de bordes en la imagen telefonica.jpg

```
[11]: imgTlf = cv2.imread('./imagenes/telefonica.jpg')
grayTlf = cv2.cvtColor(imgTlf, cv2.COLOR_BGR2GRAY)

def filtrado_deriv_x_gaussiano_bidimensional(img, sigma, n):
    h = masc_gaus_grad_bi_Dx(sigma, n)
    return convolve(img, h)

def filtrado_deriv_y_gaussiano_bidimensional(img, sigma, n):
    h = masc_gaus_grad_bi_Dy(sigma, n)
    return convolve(img, h)

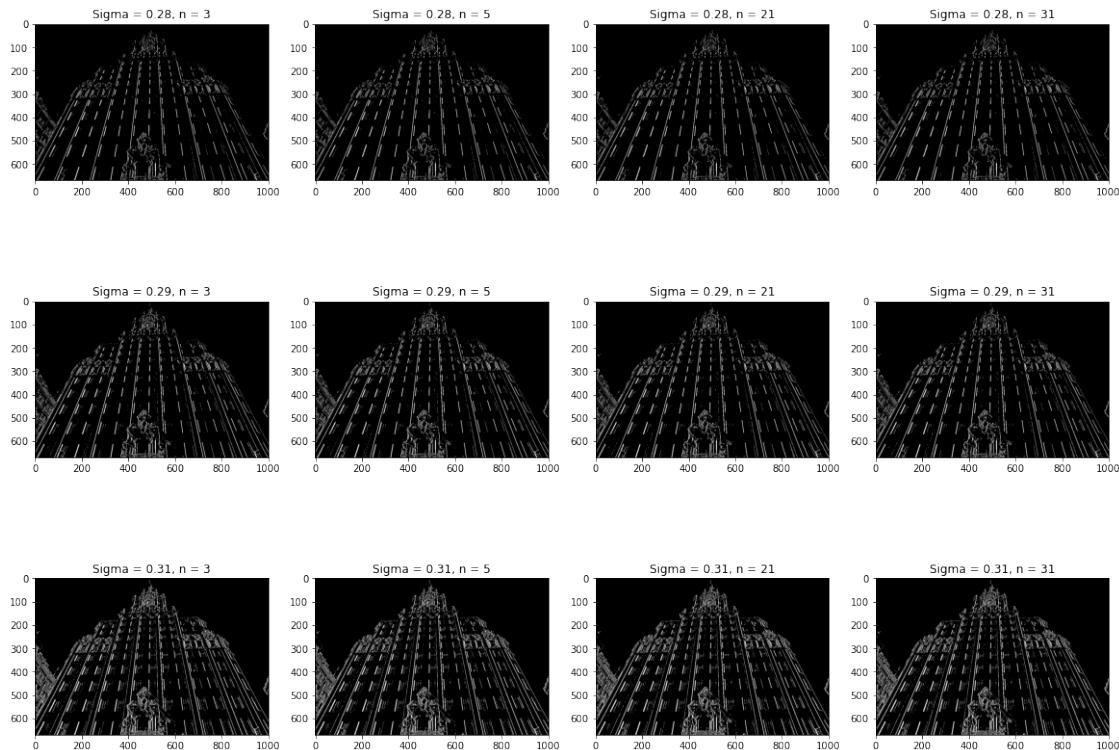
def plot_filtrado_deriv_x_gaussiano_bidimensional(img, sigmas, ns, width, height):
    fig, axs = plt.subplots(len(sigmas), len(ns), figsize = (width, height))
    for sigma in range(len(sigmas)):
        for n in range(len(ns)):
            img_filtrada = filtrado_deriv_x_gaussiano_bidimensional(img, sigmas[sigma], ns[n])
            axs[sigma, n].imshow(img_filtrada, cmap='gray')
            axs[sigma, n].set_title('Sigma = ' + str(sigmas[sigma]) + ', n = ' + str(ns[n]))
    plt.show()
```

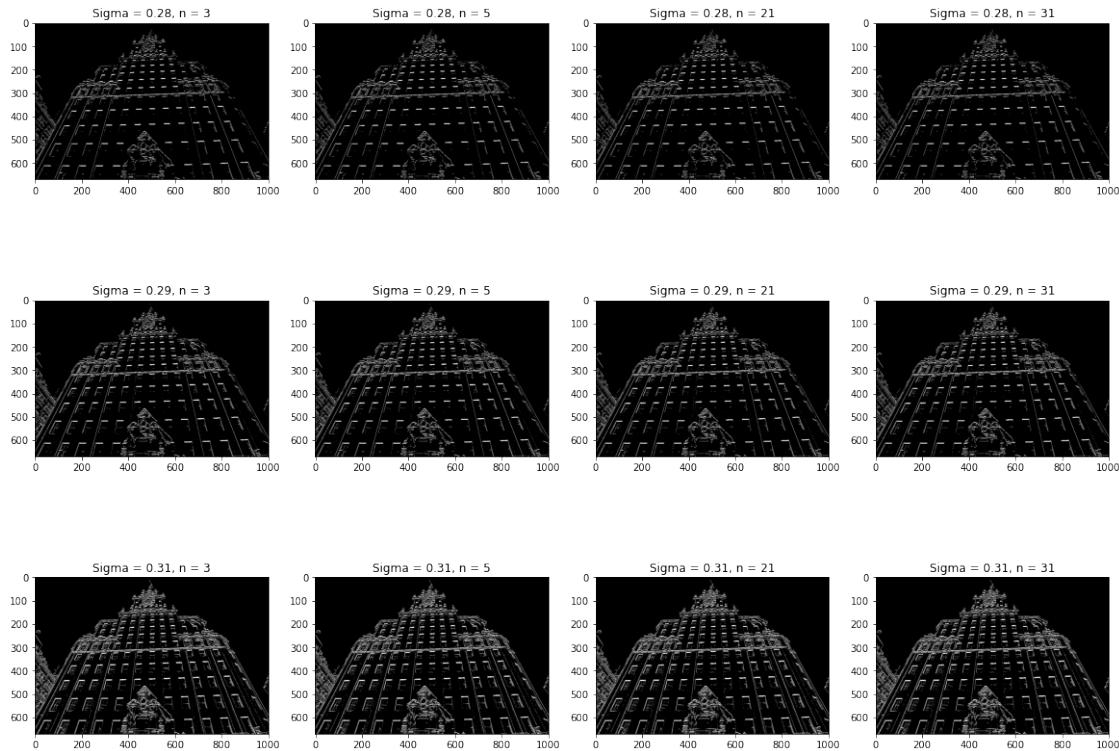
```

def plot_filtrado_deriv_y_gaussiano_bidimensional(img, sigmas, ns, width, height):
    fig, axs = plt.subplots(len(sigmas), len(ns), figsize = (width, height))
    for sigma in range(len(sigmas)):
        for n in range(len(ns)):
            img_filtrada = filtrado_deriv_y_gaussiano_bidimensional(img, sigmas[sigma], ns[n])
            axs[sigma, n].imshow(img_filtrada, cmap='gray')
            axs[sigma, n].set_title('Sigma = ' + str(sigmas[sigma]) + ', n = ' + str(ns[n]))
    plt.show()

plot_filtrado_deriv_x_gaussiano_bidimensional(grayTlf, [0.28, 0.29, 0.31], [3, 5, 21, 31], 20, 15)
plot_filtrado_deriv_y_gaussiano_bidimensional(grayTlf, [0.28, 0.29, 0.31], [3, 5, 21, 31], 20, 15)

```



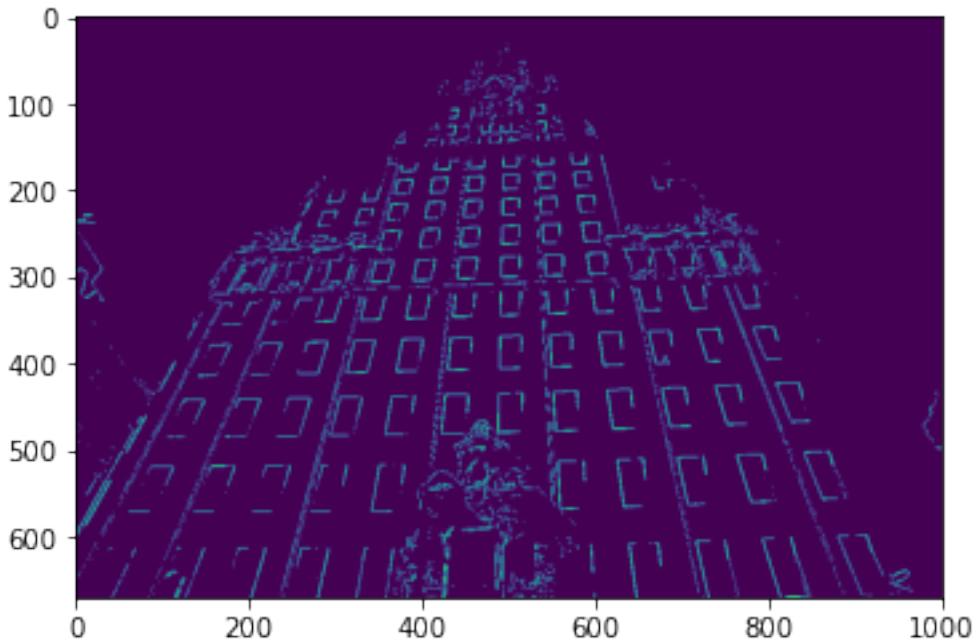


Extraemos los bordes de la imagen.

```
[12]: def obtener_bordes(img, sigma, n):
    bord = abs(filtrado_deriv_x_gaussiano_bidimensional(img, sigma, n)) + abs(filtrado_deriv_y_gaussiano_bidimensional(img, sigma, n))
    bord[bord > 0] = 255
    return bord

plt.imshow(obtener_bordes(grayTlf, 0.2565, 3))
```

```
[12]: <matplotlib.image.AxesImage at 0x7fd8c2290650>
```



Discusión de resultados Tras varias pruebas nos dimos cuenta de que para la extracción de bordes de la imagen convenía usar un valor de n menor que 1, en concreto cercano a 0.3. Con un valor tan pequeño no tiene sentido utilizar una n grande, pues los píxeles vecinos que aporten algo estarán muy cerca del píxel central. A la vez que resolviamos este ejercicio fuimos probando el de Hough y por ello nuestra función *obtener_bordes* suma los bordes verticales y horizontales y luego realza los píxeles con valores distintos de 0. Esto nos permite utilizar un valor de n un poco más pequeño dando lugar a menos píxeles identificados como bordes, pero más seguros, lo que ayuda a que el algoritmo de Hough detecte mejor las líneas que nos interesan y evite aquellas que son más bien resultado del “ruido” que aparecía en nuestra extracción de bordes. Probamos también a utilizar valores distintos de n para obtener los bordes horizontales y verticales, pero obtuvimos los mejores resultados utilizando la misma n , por ello finalmente se dejó como aparece en la función.

Ejercicio 3. Utiliza la función `median_filter` del paquete `scipy.ndimage` que realiza el filtrado de la imagen con un filtro de la mediana de tamaño $n \times n$.

Muestra y discute los resultados para diferentes valores del parámetro n en ambas imágenes. Comáralos con los obtenidos en el Ejercicio 1.

```
[13]: from scipy.ndimage import median_filter

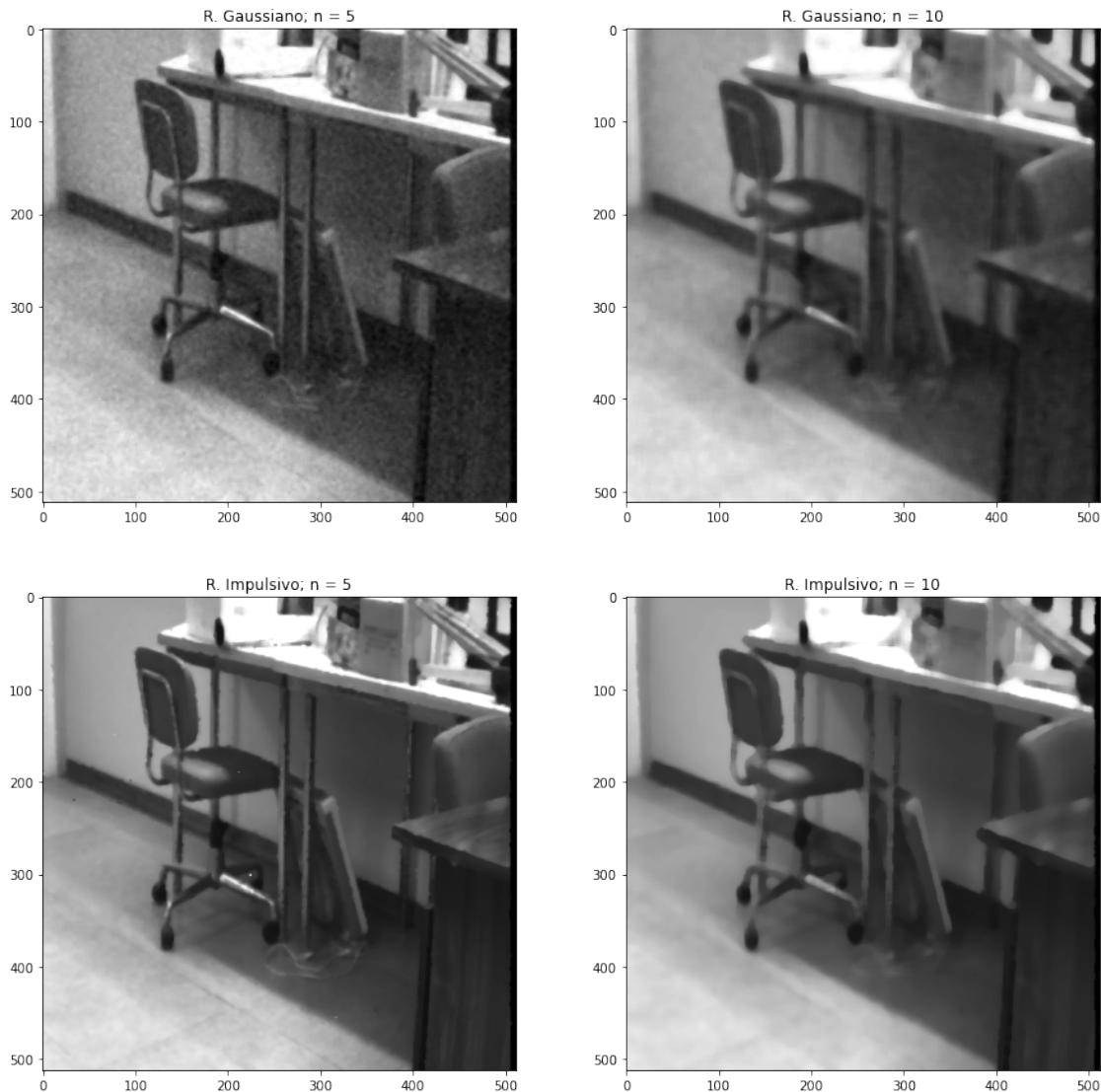
def plot_median_filter(img, titles, ns, width, height):
    fig, axs = plt.subplots(len(img), len(ns), figsize = (width, height))
    for im in range(len(img)):
        for n in range(len(ns)):
            img_filtrada = median_filter(img[im], ns[n])
            axs[im, n].imshow(img_filtrada, cmap='gray')
            axs[im, n].set_title(titles[im] + ' ; n = ' + str(ns[n]))
```

```

plt.show()

plot_median_filter([imgaus, imimp], ['R. Gaussiano', 'R. Impulsivo'], [5, 10],
                   ↪15, 15)

```



Discusión de resultados El filtro de la mediana elimina los píxeles con valores más atípicos o extremos, por lo que consigue un muy buen trabajo en la imagen con ruido impulsivo. Sin embargo, en la imagen con ruido gaussiano no es muy bueno pues la mayor parte de los píxeles contaminados no llegan a tener valores tan extremos como para que el filtro de la mediana les afecte de manera considerable.

Ejercicio 4. Utiliza la función `cv2.bilateralFilter()` de OpenCV para realizar el filtrado bilateral de una imagen. Selecciona los parámetros adecuados y aplícalo a las imágenes `tapiz.jpg`, `escgaus.bmp` y `escimp5.bmp` y otras que elijas tú.

Si llamamos σ_r a la varianza de de la gaussiana que controla la ponderación debida a la diferencia entre los valores de los píxeles y σ_s a la varianza de la gaussiana que controla la ponderación debida a la posición de los píxeles. Responde a la siguientes preguntas:

- * ¿Cómo se comporta el filtro bilateral cuando la varianza σ_r es muy alta? ¿En este caso qué ocurre si σ_s es alta o baja?
- * ¿Cómo se comporta si σ_r es muy baja? ¿En este caso cómo se comporta el filtro dependiendo si σ_s es alta o baja?

Muestra y discute los resultados para distintos valores de los parámetros, tanto para las imágenes contaminadas con ruido gaussiano como impulsivo. Compáralos con los obtenidos en el Ejercicio 1.

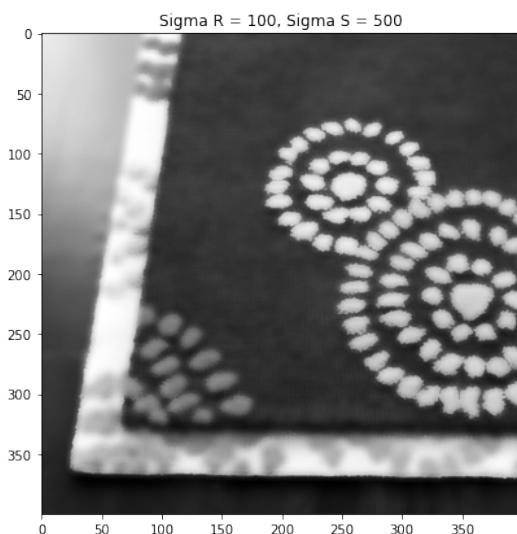
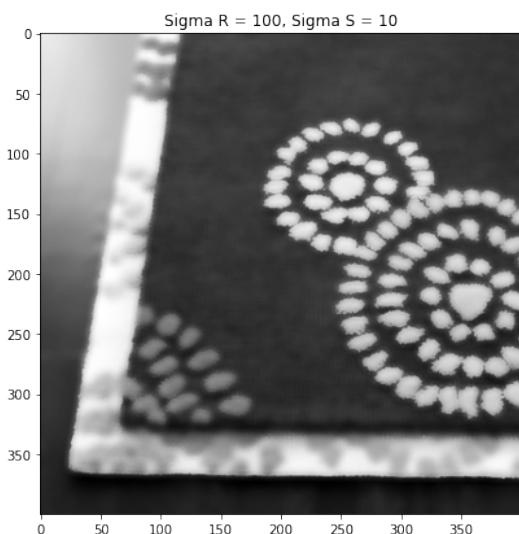
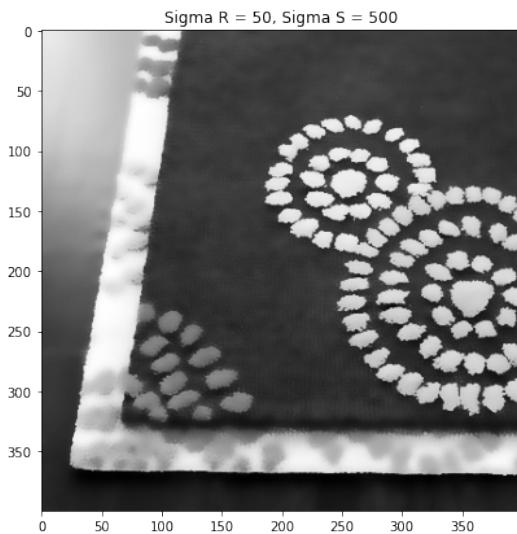
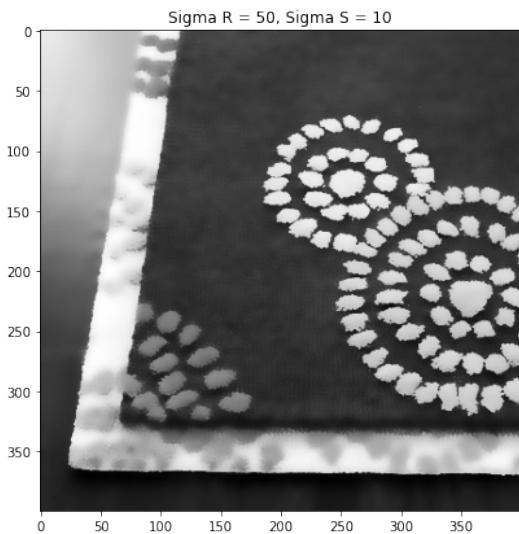
```
[14]: import cv2 as cv

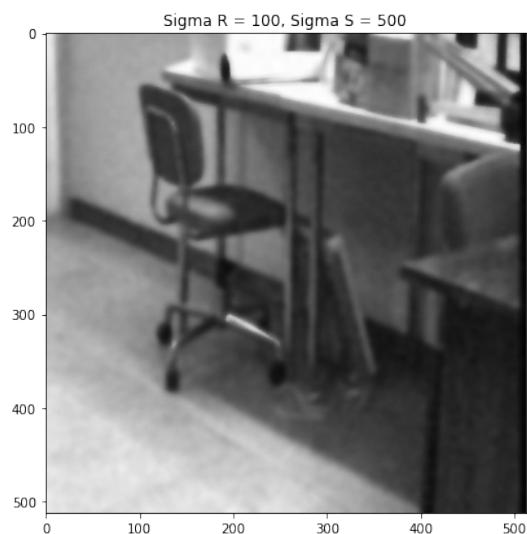
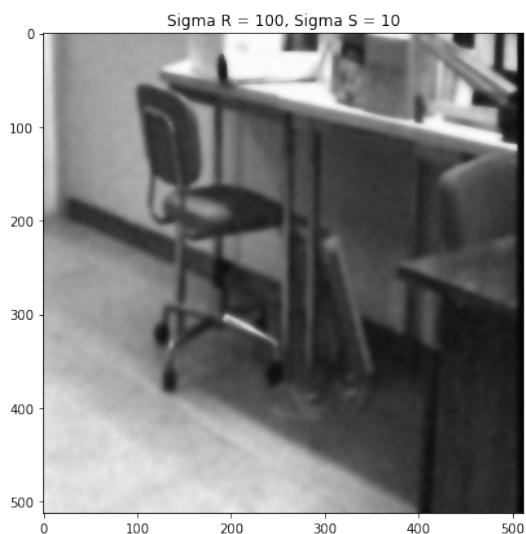
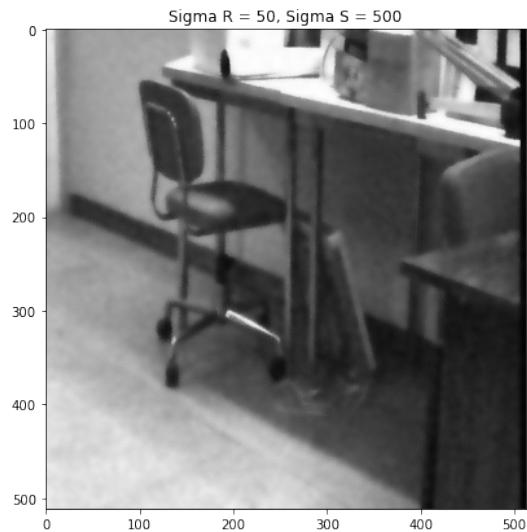
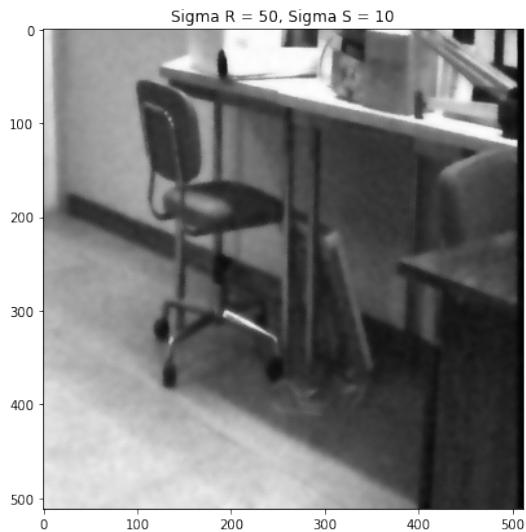
def plot_bilateralFilter(img, n, sigmaR, sigmaS, width, height):
    fig, axs = plt.subplots(len(sigmaR),len(sigmaS), figsize = (width, height))

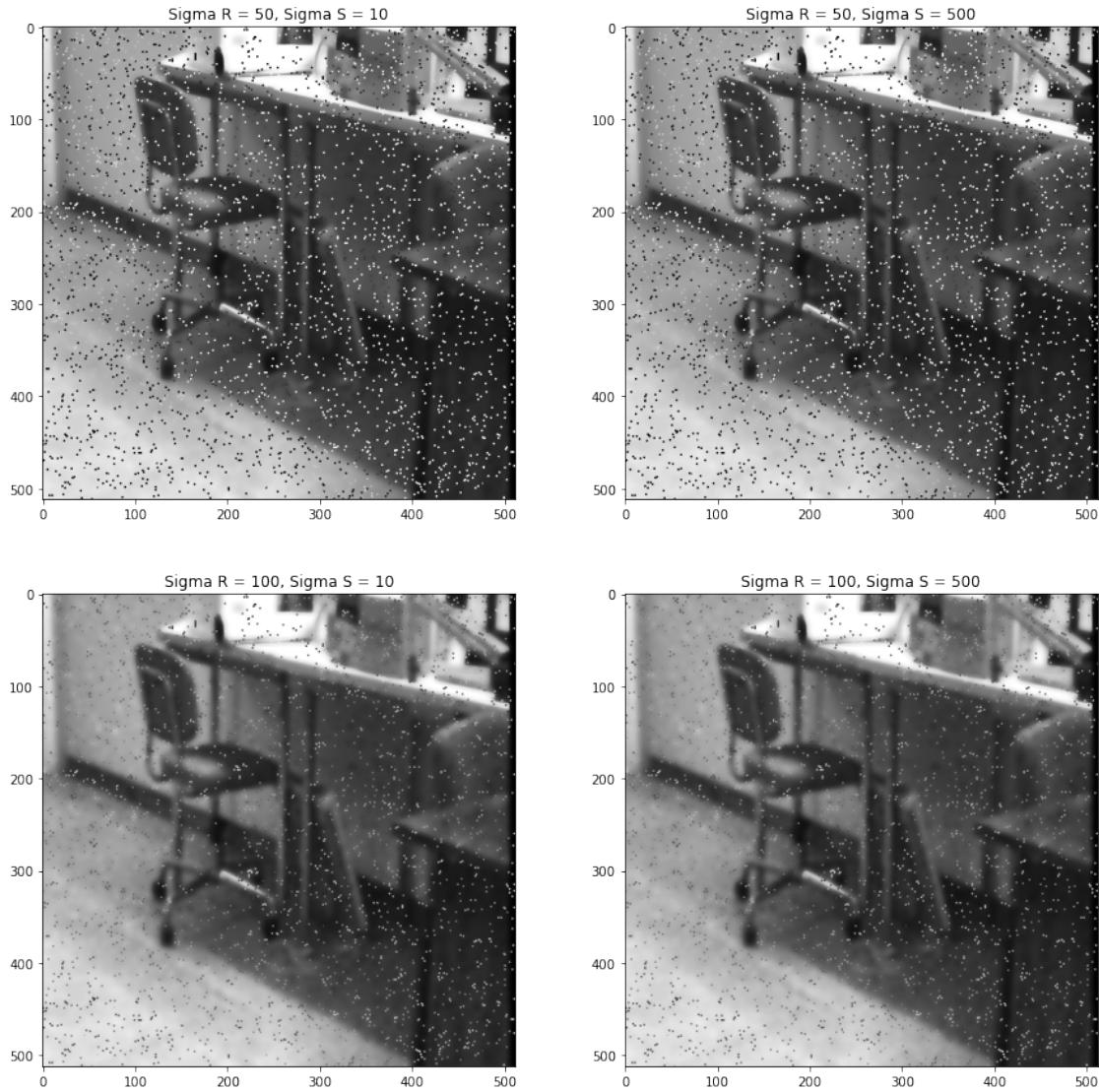
    for r in range(len(sigmaR)):
        for s in range(len(sigmaS)):
            img_filtrada = cv.bilateralFilter(img, n, sigmaR[r], sigmaS[s])
            axs[r,s].imshow(img_filtrada, cmap='gray')
            axs[r,s].set_title('Sigma R = '+str(sigmaR[r])+', Sigma S ='+str(sigmaS[s]))
    plt.show()

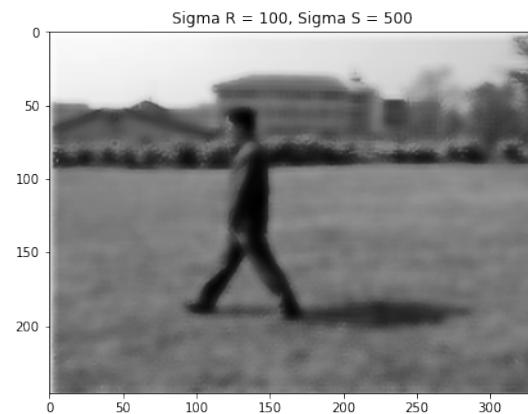
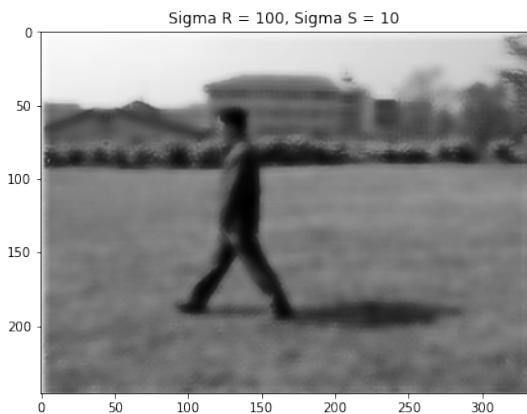
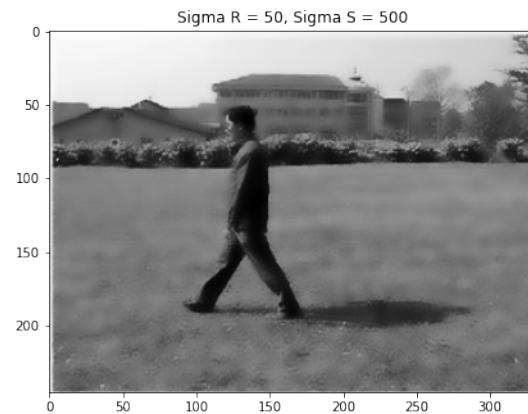
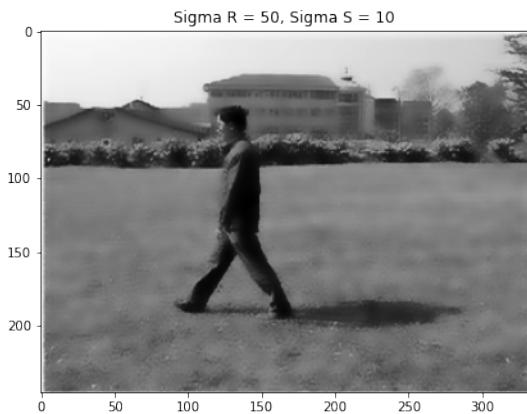
im1 = cv2.cvtColor(cv2.imread("./imagenes/tapiz.png"), cv2.COLOR_BGR2GRAY)
im2 = cv2.cvtColor(cv2.imread("./imagenes/escgaus.bmp"), cv2.COLOR_BGR2GRAY)
im3 = cv2.cvtColor(cv2.imread("./imagenes/escimp5.bmp"), cv2.COLOR_BGR2GRAY)
im4 = cv2.cvtColor(cv2.imread("./imagenes/persona.png"), cv2.COLOR_BGR2GRAY)
im5 = cv2.cvtColor(cv2.imread("./imagenes/horse.jpg"), cv2.COLOR_BGR2GRAY)

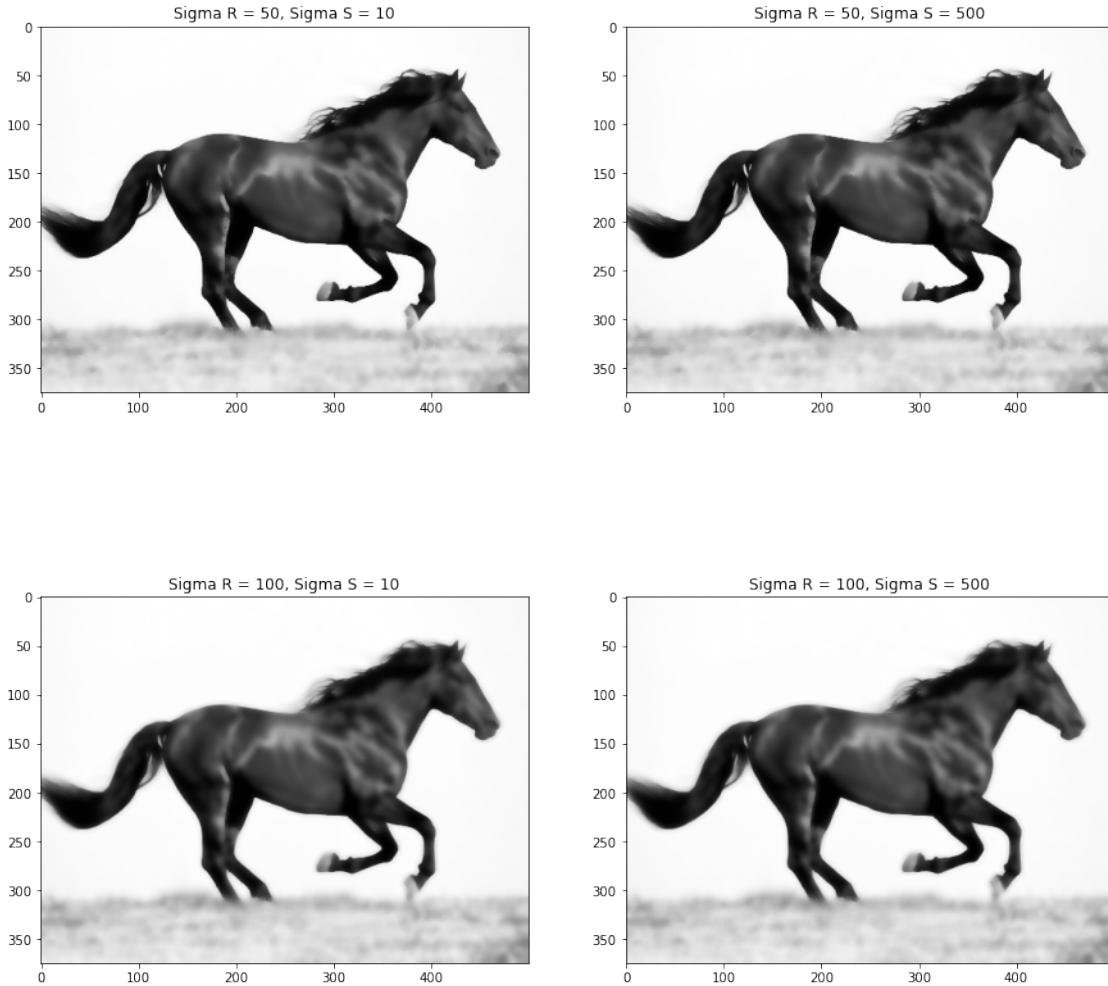
plot_bilateralFilter(im1,10,[50,100],[10,500], 15,15)
plot_bilateralFilter(im2,10,[50,100],[10,500], 15,15)
plot_bilateralFilter(im3,10,[50,100],[10,500], 15,15)
plot_bilateralFilter(im4,10,[50,100],[10,500], 15,15)
plot_bilateralFilter(im5,10,[50,100],[10,500], 15,15)
```





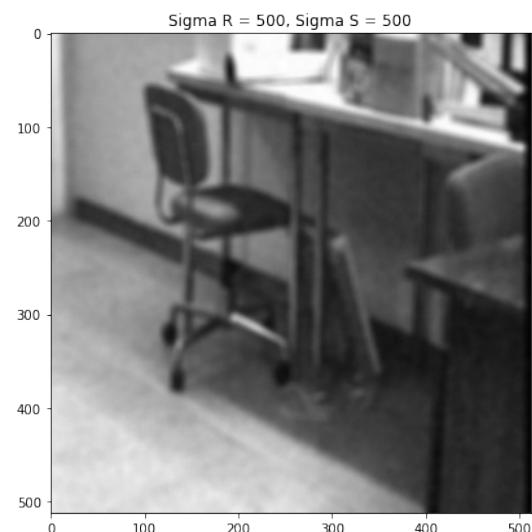
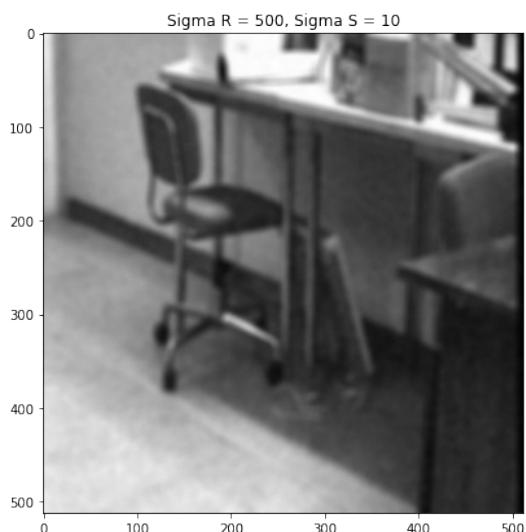
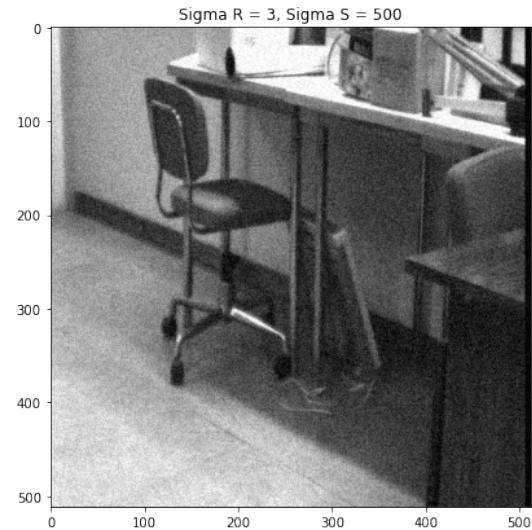
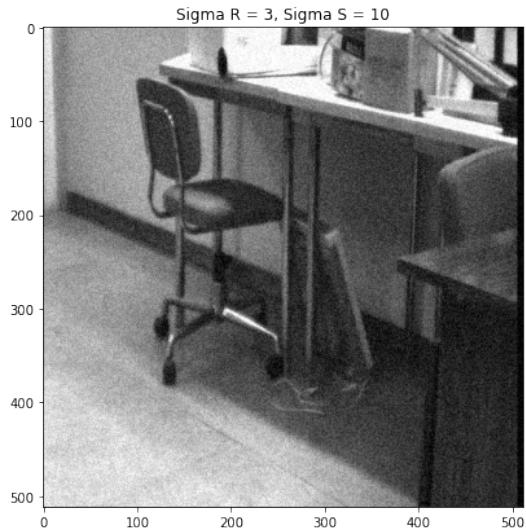


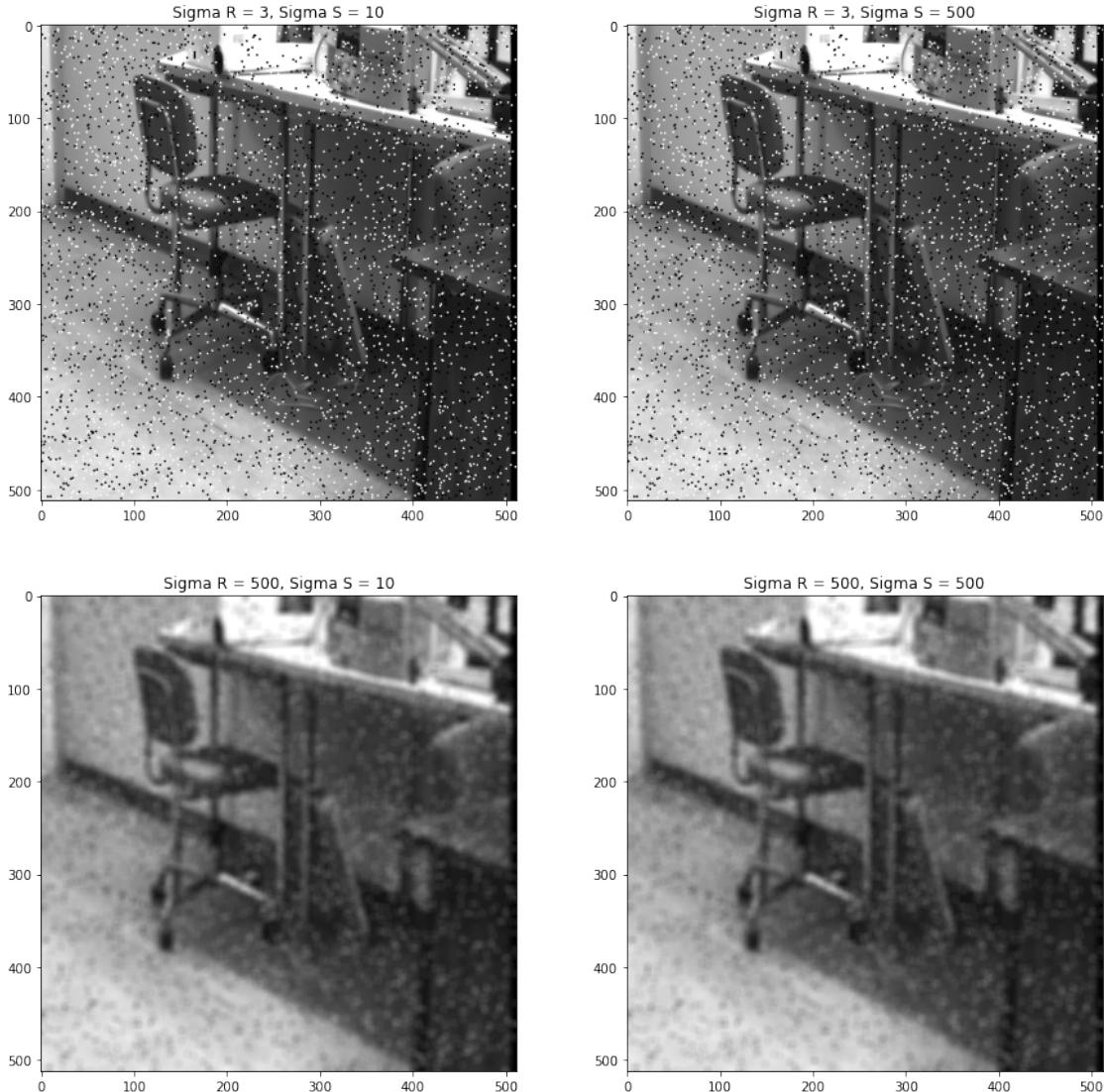




Respuesta a las preguntas 1. ¿Cómo se comporta el filtro bilateral cuando la varianza es muy alta? ¿En este caso qué ocurre si es alta o baja? Cuando la es muy alta, el filtro mezcla todos los píxeles aunque la diferencia de color sea muy diferente. Sin embargo, la delimita que cantidad de píxeles se mezclan entre sí. Si la es muy alta tendiendo a infinito, tendríamos un filtro gaussiano, pero si la es muy baja tendiendo a cero, se mezcla un número pequeño de vecinos. 2. ¿Cómo se comporta si es muy baja? ¿En este caso cómo se comporta el filtro dependiendo si es alta o baja? Cuando la es muy baja, el filtro sólo mezcla aquellos píxeles cuya diferencia de intensidades no sea muy diferente. Además, en este caso, la delimitará el número de vecinos que se mezclan entre sí si cumplen la anterior propiedad. Por otro lado, a diferencia del anterior caso, siempre conservaremos los bordes.

```
[15]: plot_bilateralFilter(imgaus,10,[3,500],[10,500], 15,15)
plot_bilateralFilter(imimp,10,[3,500],[10,500], 15,15)
```





Discusión de resultados El filtro de bilateral es muy parecido al filtro gaussiano, filtra los píxeles de una imagen mediante una sigma la cual determina la distancia de píxeles filtrados. Sin embargo, este filtro contiene una sigma que hace que se tenga en cuenta la diferencia de intensidades. De esta forma se filtrarán píxeles dentro de una distancia y con una diferencia de intensidades determinada. En el caso de la imagen con un ruido gaussiano, podemos apreciar como el filtrado bilateral al aplicar unas sigmas muy altas, logra filtrar la imagen y elimina el ruido aunque también hace que se pierda el detalle. Sin embargo, para una imagen con ruido impulsivo, el resultado no es bueno pues a diferencia de lo que por ejemplo haría un filtro de la mediana (los valores más atípicos no se tendrían en cuenta), el filtro bilateral si utilizará (en mayor o menor medida, según sean las sigmas) los valores atípicos, emborronando la imagen.

1.6 Transformada de Hough

Ejercicio 5. Utiliza la función `cv2.HoughLines()` de OpenCV para encontrar líneas en la imagen `telefonica.jpg` y `urjc.jpg`. Para extraer los bordes de la imagen utiliza las funciones escritas más arriba.

Discute cómo has realizado la conversión de RGB a niveles de gris, el funcionamiento para distintos valores de los parámetros de la función, así como de los filtros utilizados para extraer los bordes de la imagen. Pinta los resultados sobre la imagen (te proporcionamos algo de código por si fuese útil).

```
[16]: import cv2
import matplotlib.pyplot as plt

def draw_lines(img, lines, color=(0, 0, 255), thickness=2):
    """
    Draws a set of lines detected using the OpenCV Hough transform
    :param img: An input image in BGR format of type np.int8
    :param lines: List or Numpy array containing the parameters of the
    ↪homogeneous line as: ax + by + c = 0
    :param color: The color used to draw the lines. Red by default.
    :param thickness: The thickness of the lines to be drawn
    """
    if lines is not None:
        for i in range(len(lines)):
            eq = lines[i]
            rho = -eq[2]
            a = eq[0]
            b = eq[1]
            x0 = a * rho
            y0 = b * rho
            x1 = int(x0 - 10000 * b)
            y1 = int(y0 + 10000 * a)
            x2 = int(x0 + 10000 * b)
            y2 = int(y0 - 10000 * a)

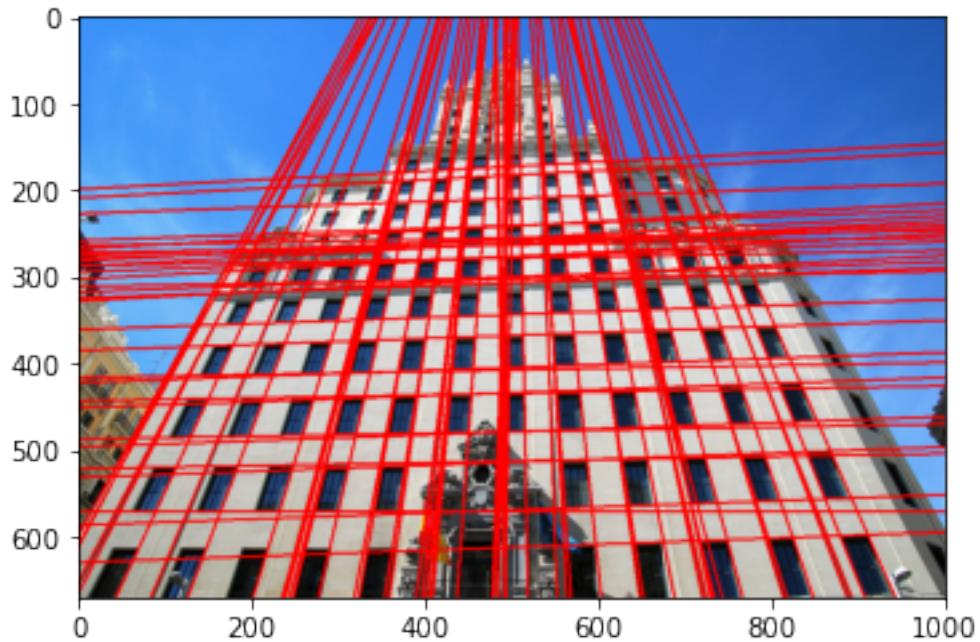
            cv2.line(img, (x1, y1), (x2, y2), color, thickness)

#####
img = cv2.imread('./imagenes/telefonica.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
edges = obtener_bordes(gray, 0.2565, 3)

lines = np.squeeze(cv2.HoughLines(edges, 1, np.pi/180, 80))
# Convert the lines to homogeneous coordinates
lines = np.array([np.cos(lines[:, 1]), np.sin(lines[:, 1]), -lines[:, 0]]).T
```

```
# Draw andshow the lines
draw_lines(img, lines)
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
```

[16]: <matplotlib.image.AxesImage at 0x7fd8c2775550>



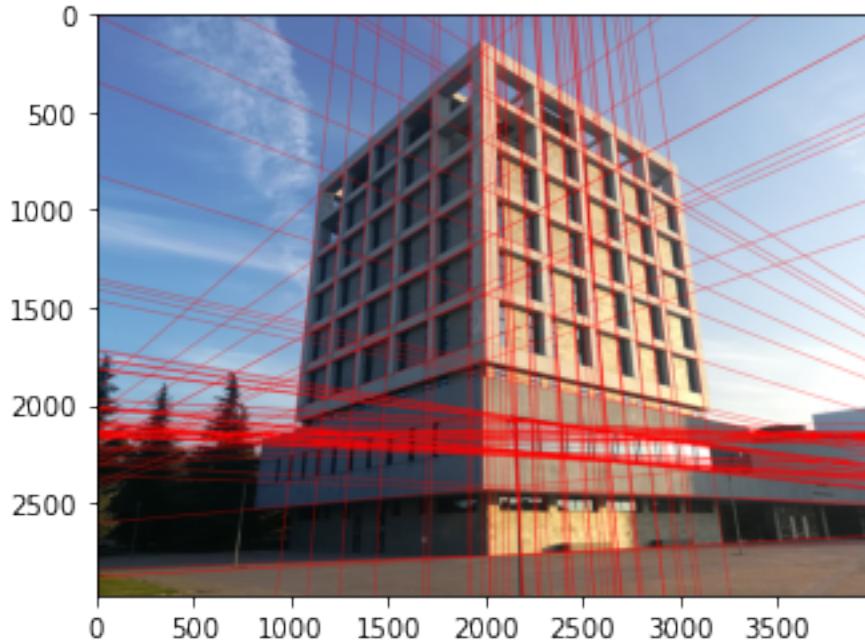
```
[17]: imgUrjc = cv2.imread('./imagenes/urjc.jpg')
grayUrjc = cv2.cvtColor(imgUrjc, cv2.COLOR_BGR2GRAY)
edges = obtener_bordes(grayUrjc, 0.309, 3)

imgRGB = cv2.cvtColor(imgUrjc, cv2.COLOR_BGR2GRAY)

lines = np.squeeze(cv2.HoughLines(edges, 1, np.pi/180, 650))
# Convert the lines to homogeneous coordinates
lines = np.array([np.cos(lines[:, 1]), np.sin(lines[:, 1]), -lines[:, 0]]).T

# Draw andshow the lines
draw_lines(imgUrjc, lines)
plt.imshow(cv2.cvtColor(imgUrjc, cv2.COLOR_BGR2RGB))
```

[17]: <matplotlib.image.AxesImage at 0x7fd8be34b8d0>



Discusión de resultados El cambio a escala de grises lo hemos hecho directamente con la función `cvtColor` de OpenCV. En cuanto a los parámetros, primero descubrimos que una sigma menor que 1 generaba buenos bordes, de hecho alrededor de 0.3 obteníamos los mejores valores. Dado que la sigma es tan pequeña, no es necesario utilizar un kernel con una n mayor que 3, ya que lo único que haría sería gastar procesamiento sin modificar los resultados. En la imagen `telefonica.jpg` fuimos capaces de conseguir unos bordes bastante limpios, por lo que el umbral de la función `cv2.HoughLines` lo pudimos bajar bastante. En la imagen `urjc.jpg` utilizamos un valor algo más alto para la sigma del detector de bordes, lo que nos llevó a tener que subir el umbral de `cv2.HoughLines`. También probamos a utilizar ángulos distintos en la función, pero encontramos que no mejoraba nuestros resultados.

1.7 Segmentación

Ejercicio 6. Escribe una función que segmente el objeto central de una imagen a partir de una segmentación manual inicial realizada por el usuario.

Puedes utilizar, a modo de ejemplo, el código proporcionado en el archivo `segm.py`. El archivo `select_pixels.py` contiene código para pintar sobre la imagen.

En esta primera versión 1. toma como afinidad entre una pareja de píxeles la diferencia en sus valores de color y ; 2. sólo establece los términos unitarios de los píxeles marcados por el usuario.

Aplícalo al menos a la imagen `horse.jpg` aunque también debes intentar segmentar `flower.png`, `persona.png`, `car.jpg` y `mit2.png`, que son progresivamente más complejas. Muestra y discute los resultados. Justifica los errores. En el ejercicio siguiente tendrás la oportunidad de mejorar tu algoritmo para que pueda resolver casos más difíciles.

```
[18]: # %load ./code/select_pixels.py
#####
# This code lets you paint on top of an image and returns the painted image
# it can be used to select pixels somehow in an image
#
# It requires that you install "python-pygame" and "python-opencv"
#
# The interesting function here is "select_fg_bg" read documentation below
#####
import pygame
import numpy as np
import cv2

def roundline(srf, color, start, end, radius=1):
    dx = end[0]-start[0]
    dy = end[1]-start[1]
    distance = max(abs(dx), abs(dy))
    for i in range(distance):
        x = int( start[0]+float(i)/distance*dx)
        y = int( start[1]+float(i)/distance*dy)
        pygame.draw.circle(srf, color, (x, y), radius)

def select_fg_bg(img, radio=2):
    """
    Shows image img on a window and lets you mark in red, green and blue
    pixels in the image.
    img: numpy array with the image to be labeled
    radio: is the radio of the circumference used as brush
    returns: a numpy array that is the image painted
    """
    # Creates the screen where the image will be displayed
    # Shapes are reversed in img and pygame screen
    screen = pygame.display.set_mode(img.shape[-2::-1])

    # imgpyg=pygame.image.load(imgName)
    imgpyg=pygame.image.frombuffer(img,img.shape[-2::-1], 'RGB')
    screen.blit(imgpyg,(0,0))
    pygame.display.flip() # update the display

    draw_on = False
    last_pos = (0, 0)
    color_red = (255, 0, 0)
    color_green = (0,255,0)
    color_blue = (0,0,255)

    while True:
        e = pygame.event.wait()
        if e.type == pygame.QUIT:
```

```

        break;
if e.type == pygame.MOUSEBUTTONDOWN:
    if pygame.mouse.get_pressed()[0]:
        color=color_red
    elif pygame.mouse.get_pressed()[2]:
        color=color_green
    else:
        color=color_blue
    pygame.draw.circle(screen, color, e.pos, radio)
    draw_on = True
if e.type == pygame.MOUSEBUTTONUP:
    draw_on = False
if e.type == pygame.MOUSEMOTION:
    if draw_on:
        pygame.draw.circle(screen, color, e.pos, radio)
        roundline(screen, color, e.pos, last_pos, radio)
    last_pos = e.pos
pygame.display.flip()

imgOut=np.ndarray(shape=img.shape[:2]+(4,),dtype='u1',buffer=screen.
←get_buffer().raw)
pygame.quit()

return(cv2.cvtColor(imgOut[:, :, :3],cv2.COLOR_BGR2RGB))

```

pygame 2.0.0 (SDL 2.0.12, python 3.7.7)
Hello from the pygame community. <https://www.pygame.org/contribute.html>

[19]:

```

#####
# Segmentacion de imagen a la "Grab Cut" simplificado
# por Luis Baumela. UPM. 15-10-2015
# Vision por Computador. Master en Inteligencia Artificial
#####

import numpy as np
#from scipy.misc import imread
import maxflow
import matplotlib.pyplot as plt

imgName='./imagenes/horse.jpg'

img = cv2.imread(imgName)

# Marco algunos pixeles que pertenecen el objeto y el fondo
markedImg = select_fg_bg(img)

# Create the graph.

```

```

g = maxflow.Graph[float]()

# Add the nodes. nodeids has the identifiers of the nodes in the grid.
nodeids = g.add_grid_nodes(img.shape[:2])

# Calcula los costes de los nodos no terminales del grafo
# Pasamos de 3 canales de color a 1 de intensidades. Al ser la intensidad la
→ media de los canales de color,
# la diferencia entre intensidades será proporcional a la diferencia de los
→ canales de color.
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Estos son los costes de los vecinos horizontales (Restamos las diferencias
→ entre píxeles a 255 para obtener costes)
exp_aff_h = np.append(np.zeros((gray.shape[0], 1)), 255 - abs(np.diff(gray)), axis = 1) # Añadimos una columna de 0s a la izquierda que no computarán
# Estos son los costes de los vecinos verticales
exp_aff_v = np.append(np.zeros((1, gray.shape[1])), 255 - abs(np.diff(gray, axis = 0)), axis = 0) # Añadimos una fila de 0s encima que no computarán

# Construyo el grafo
# Para construir el grafo relleno las estructuras
hor_struc = np.array([[0, 0, 0], [1, 0, 0], [0, 0, 0]])
ver_struc = np.array([[0, 1, 0], [0, 0, 0], [0, 0, 0]])

# Construyo el grafo
g.add_grid_edges(nodeids, exp_aff_h, hor_struc, symmetric = True)
g.add_grid_edges(nodeids, exp_aff_v, ver_struc, symmetric = True)

# Leo los píxeles etiquetados
# Los marcados en rojo representan el objeto
pts_fg = np.transpose(np.where(np.all(np.equal(markedImg, (255, 0, 0)), 2)))
# Los marcados en verde representan el fondo
pts_bg = np.transpose(np.where(np.all(np.equal(markedImg, (0, 255, 0)), 2)))

# Incluyo las conexiones a los nodos terminales
# Pesos de los nodos terminales
g.add_grid_tedges(nodeids[pts_fg[:, 0], pts_fg[:, 1]], 255, 0) # Hacemos
→ coincidir los costes máximos con el valor máximo de la intensidad
g.add_grid_tedges(nodeids[pts_bg[:, 0], pts_bg[:, 1]], 0, 255)

# Find the maximum flow.
g.maxflow()

# Get the segments of the nodes in the grid.
sgm = g.get_grid_segments(nodeids)

```

```

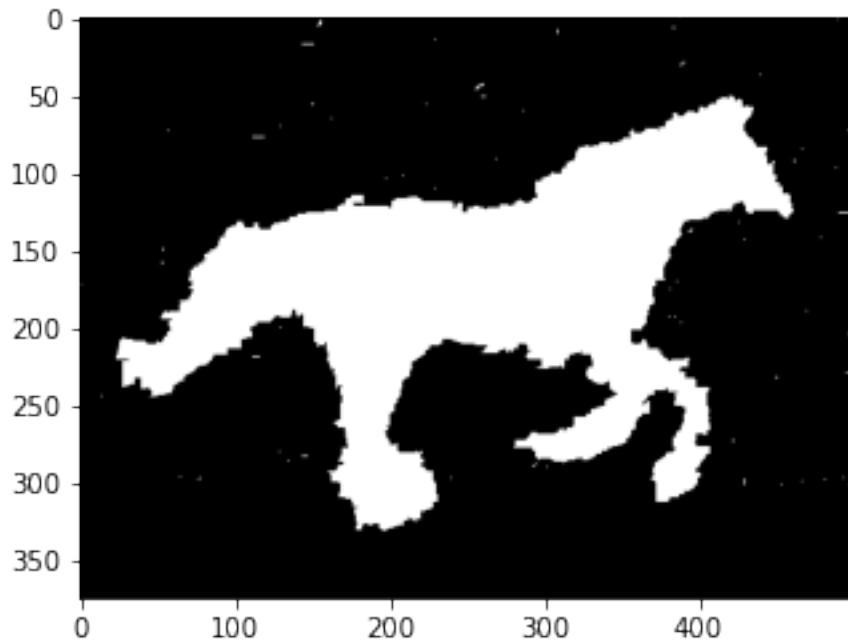
# Muestro el resultado de la segmentacion
plt.figure()
plt.imshow(np.uint8(np.logical_not(sgm)), cmap = 'gray')
plt.show()

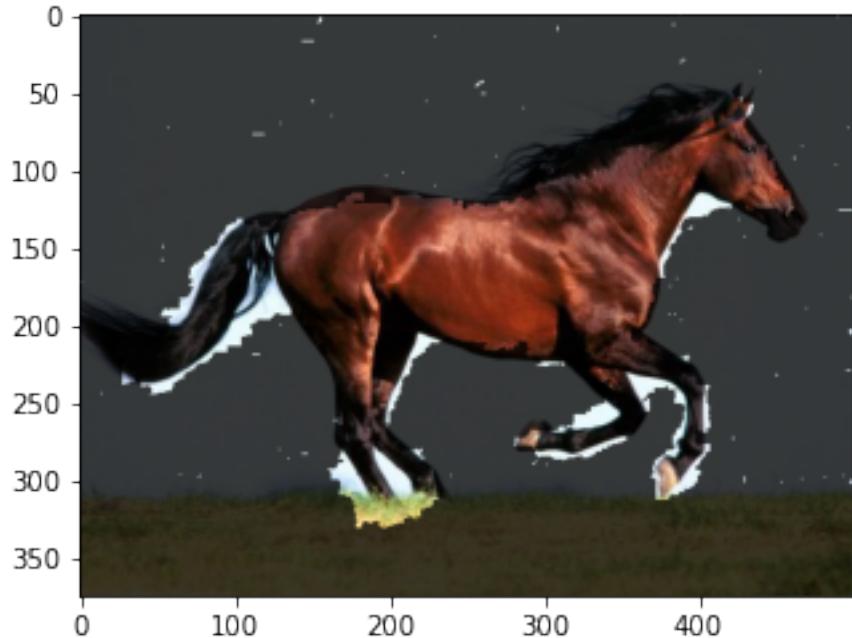
# Lo muestro junto con la imagen para ver el resultado
plt.figure()
wgs = (np.float_(np.logical_not(sgm)) + 0.3) / 1.3

# Replico los pesos para cada canal y ordeno los indices
wgs = np.rollaxis(np.tile(wgs, (3, 1, 1)), 0, 3)

rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
plt.imshow(np.uint8(np.multiply(rgb, wgs)))
plt.show()

```





Discusión de resultados Los resultados que obtenemos son muy malos pues la forma en que asignamos costes no es muy buena. A las semillas si que les asignamos unos costes que fijan unos píxeles como fondo o como objeto, pero luego la forma en que eso se intenta “extender” por el grafo no es muy buena. Básicamente esa información se va transfiriendo entre vecinos, pero por ejemplo no tenemos capacidad de marcar píxeles similares a las semillas directamente como tal (como haremos en el ejercicio 7 aprovechando los histogramas de las semillas). En este caso debemos acercarnos mucho a los bordes de cada bloque si queremos mejorar la segmentación. También es notable el “ruido” que se genera en la segmentación, por esto último que hemos comentado. Creemos que los píxeles sueltos etiquetados como objeto pueden ser consecuencia del algoritmo de grafos y no de las semillas etiquetadas como objeto.

Ejercicio 7. Mejora el algoritmo anterior. Sugerencia: * Refina la segmentación iterativamente, * Mejora la función de afinidad entre píxeles, * Mejora los términos unitarios, ...

Aplica el algoritmo a las imágenes utilizadas en el ejercicio anterior. Muestra y discute los resultados. Jusfifica en qué y por qué mejoran los resultados y plantea posibles soluciones para los fallos.

```
[20]: import cv2
import pygame
import numpy as np
import maxflow
import matplotlib.pyplot as plt
from scipy import signal

def grad_map(grad):
```

```

grad_abs = np.abs(grad)
m = np.mean(grad_abs)
s = np.std(grad_abs) / 2
return 1 - 1 / (1 + np.exp(-(grad_abs - m) / s))

def segment(img, markedImg, K = 3):

    # Create the graph.
    g = maxflow.Graph[float]()

    # Add the nodes. nodeids has the identifiers of the nodes in the grid.
    nodeids = g.add_grid_nodes(img.shape[:2])

    # Obtenemos los pixeles etiquetados
    fg_idx = np.all(markedImg == [0, 0, 255], 2)
    bg_idx = np.all(markedImg == [0, 255, 0], 2)
    semillas = [img[fg_idx], img[bg_idx]]

    # Calculamos los histogramas
    hists = [np.histogram(semillas[k], 256, (0, 255))[0] for k in range(2)]

    # Suavizamos los histogramas
    hists_smooth = [signal.filtfilt(0.2, [1, -0.8], hists[k]) for k in range(2)]

    # Normalizar
    pmfs_smooth = [hists_smooth[k] / np.sum(hists_smooth[k]) for k in range(2)]

    # Calcular posteriores
    posterior = pmfs_smooth[0] / (pmfs_smooth[0] + pmfs_smooth[1])
    datacost = posterior[:, :, 0].astype(np.uint8)

    # Fijamos los costes de las semillas
    bgcost = 1 - datacost
    bgcost[fg_idx] = 0
    bgcost[bg_idx] = K
    datacost[fg_idx] = K
    datacost[bg_idx] = 0

    g.add_grid_tedges(nodeids, datacost, bgcost)

    # Convertir a yuv
    img_yuv = cv2.cvtColor(img.astype(np.uint8), cv2.COLOR_BGR2YUV)
    img_y, _, _ = cv2.split(img_yuv)

    # Obtenemos gradiente horizontal y vertical
    sobelh = cv2.Sobel(img_y, cv2.CV_64F, 1, 0)

```

```

sobelv = cv2.Sobel(img_y, cv2.CV_64F, 0, 1)

smoothcostx = grad_map(sobelh)
smoothcosty = grad_map(sobelv)

# Añadimos costes horizontales
structure = np.array([[0, 0, 0],
                      [0, 0, 1],
                      [0, 0, 0]])
g.add_grid_edges(nodeids, 128. * smoothcostx, structure, symmetric = True)

# Añadimos costes verticales
structure = np.array([[0, 0, 0],
                      [0, 0, 0],
                      [0, 1, 0]])
g.add_grid_edges(nodeids, 128. * smoothcosty, structure, symmetric = True)

# Find the maximum flow.
g.maxflow()

return g.get_grid_segments(nodeids)

def roundline(srf, color, start, end, radius = 1):
    dx = end[0] - start[0]
    dy = end[1] - start[1]
    distance = max(abs(dx), abs(dy))
    for i in range(distance):
        x = int(start[0] + float(i) / distance * dx)
        y = int(start[1] + float(i) / distance * dy)
        pygame.draw.circle(srf, color, (x, y), radius)

def interactive_graph_cut(img, K = 3, radio = 2):
    imgCopy = img.copy()
    markedImg = img.copy()
    sgm = None

    screen = pygame.display.set_mode((img.shape[1] * 2, img.shape[0]))

    paintedImg = pygame.image.frombuffer(imgCopy, imgCopy.shape[-2::-1], 'BGR')
    paintedImgR = paintedImg.get_rect()

    segmentedImg = pygame.image.frombuffer(markedImg, markedImg.shape[-2:: -1], 'BGR')

```

```

segmentedImgR = segmentedImg.get_rect()
segmentedImgR.right = 2 * img.shape[1]
segmentedImgR.left = img.shape[1]

# Top left corner
screen.blit(paintedImg, paintedImgR)
# Top right corner
screen.blit(segmentedImg, segmentedImgR)

pygame.display.flip() # update the display

draw_on = False
last_pos = (0, 0)
color_red = (255, 0, 0)
color_green = (0, 255, 0)

user_has_drawn_bg = False
user_has_drawn_fg = False

while True:
    e = pygame.event.wait()
    if e.type == pygame.QUIT:
        break;
    if e.type == pygame.MOUSEBUTTONDOWN:
        if e.pos[0] < img.shape[1]:
            if pygame.mouse.get_pressed()[0]:
                color=color_red
                user_has_drawn_fg = True
            elif pygame.mouse.get_pressed()[2]:
                color=color_green
                user_has_drawn_bg = True
            pygame.draw.circle(screen, color, e.pos, radio)
            draw_on = True
        if e.type == pygame.MOUSEBUTTONUP:
            draw_on = False
            if user_has_drawn_fg and user_has_drawn_bg:
                markedImg = np.ndarray(shape = (img.shape[0], 2 * img.shape[1]), ↵
                ↵+ (4,), dtype = 'u1', buffer = screen.get_buffer().raw)[:, :img.shape[1], :3]
                sgm = segment(img, markedImg, K)
                wgs = (np.float_(np.logical_not(sgm)) + 0.3) / 1.3
                wgs = np.rollaxis(np.tile(wgs, (3, 1, 1)), 0, 3)
                segmentedImg = np.uint8(np.multiply(img, wgs))
                pixl_arr = pygame.pixelcopy.make_surface(np. ↵
                ↵swapaxes(segmentedImg, 0, 1))
                screen.blit(pixl_arr, segmentedImgR)
            if e.type == pygame.MOUSEMOTION:
                if e.pos[0] < img.shape[1]:

```

```

        if draw_on:
            pygame.draw.circle(screen, color, e.pos, radio)
            roundline(screen, color, e.pos, last_pos, radio)
            last_pos = e.pos
        pygame.display.flip()

pygame.quit()

if sgm is not None:
    # Muestro el resultado de la segmentacion
    plt.figure()
    plt.imshow(np.uint8(np.logical_not(sgm)), cmap = 'gray')
    plt.show()

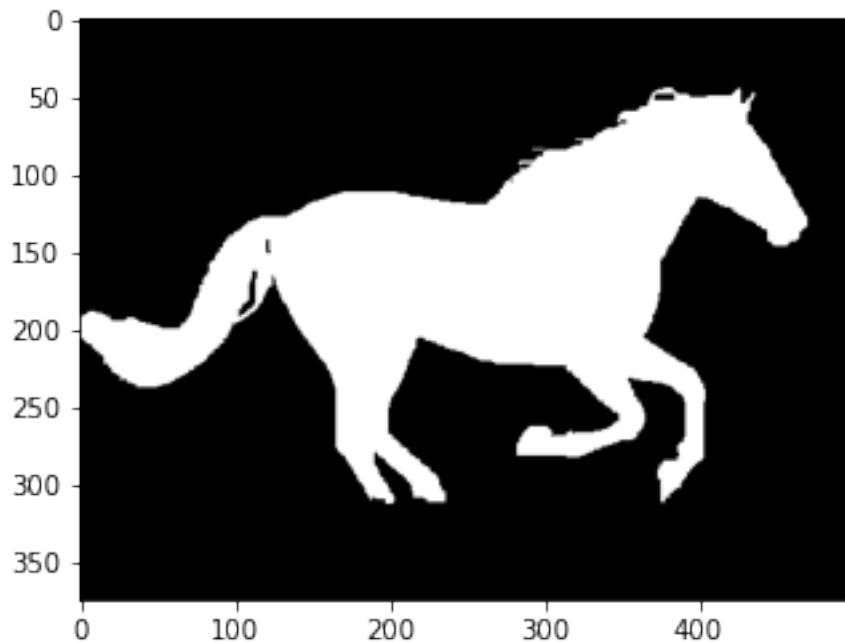
return sgm, markedImg

imgName='./imagenes/horse.jpg'

img = cv2.imread(imgName)

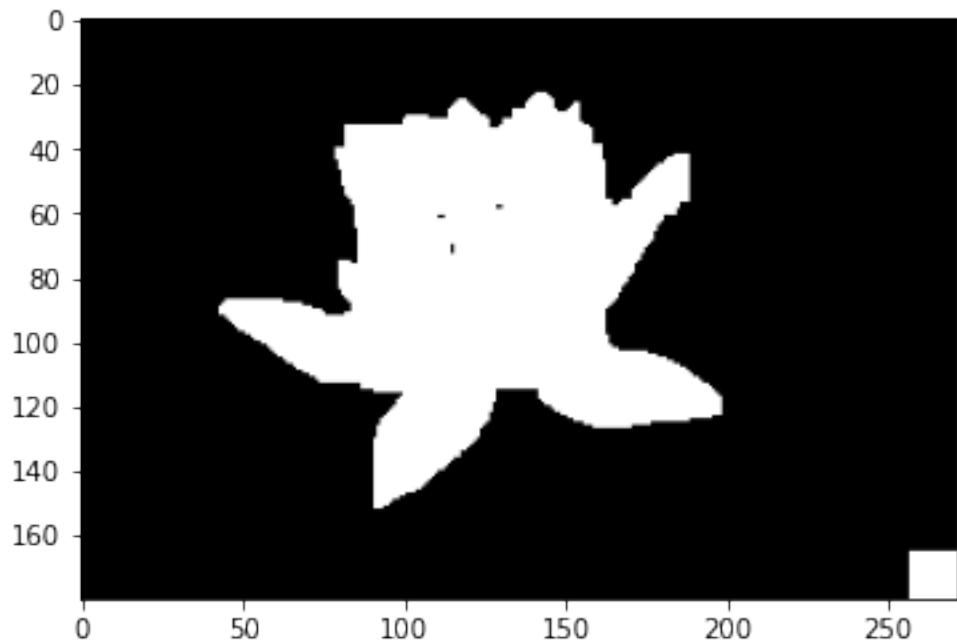
sgm = interactive_graph_cut(img)

```

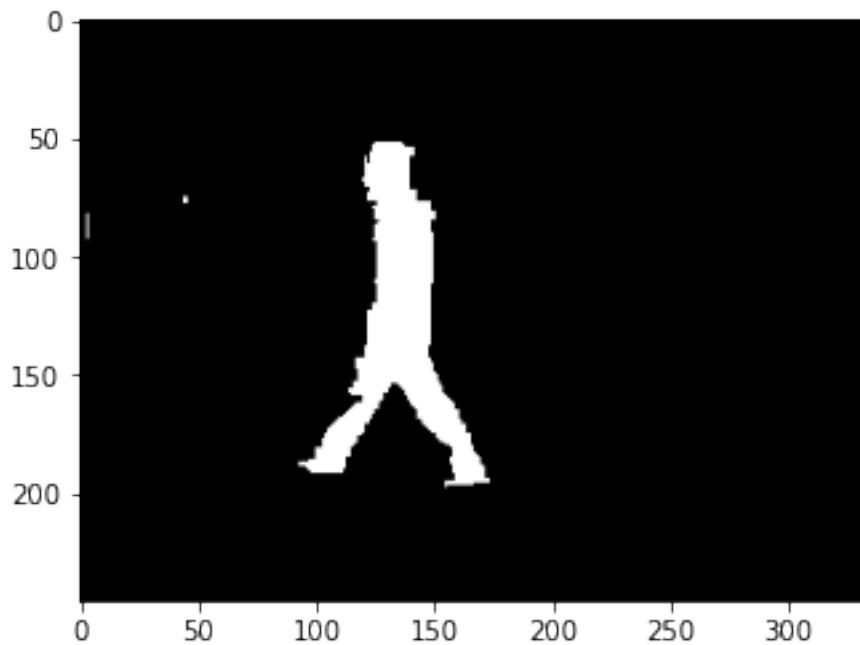


```
[21]: imgName='./imagenes/flower.png'
img = cv2.imread(imgName)
```

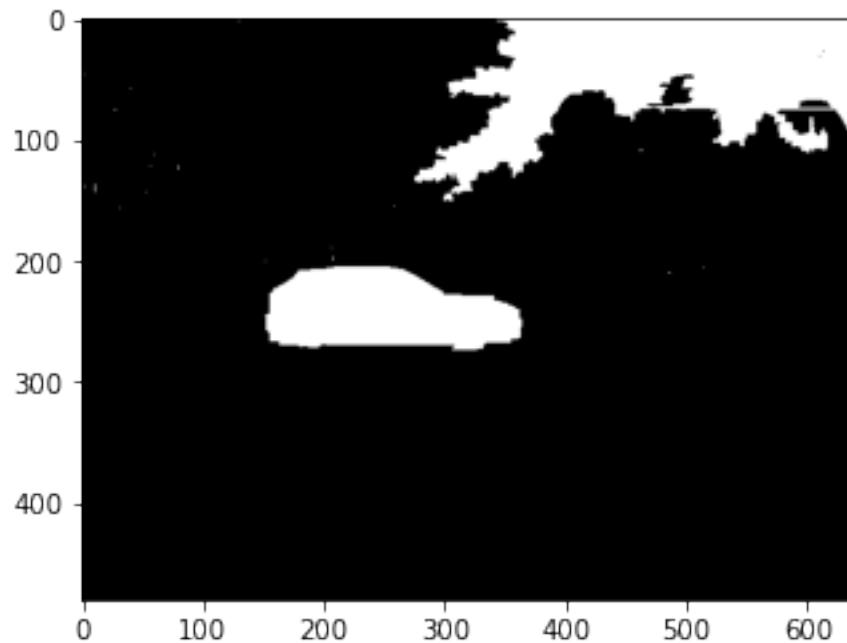
```
sgm = interactive_graph_cut(img)
```



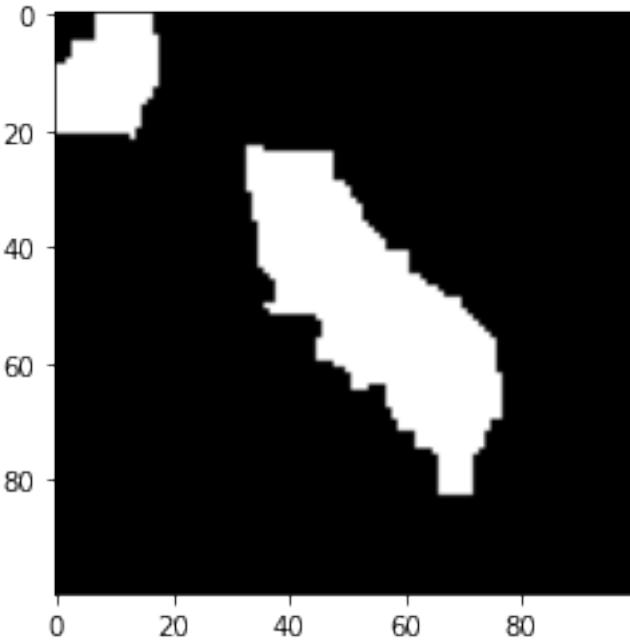
```
[22]: imgName='./imagenes/persona.png'  
img = cv2.imread(imgName)  
sgm = interactive_graph_cut(img)
```



```
[23]: imgName='./imagenes/car.jpg'  
img = cv2.imread(imgName)  
sgm = interactive_graph_cut(img)
```



```
[24]: imgName='./imagenes/mit2.png'  
img = cv2.imread(imgName)  
sgm = interactive_graph_cut(img, 8)
```



Discusión de resultados Este ejercicio mejora en mucho la implementación del ejercicio anterior. Al dibujar dos pequeñas líneas ya prácticamente queda todo correctamente etiquetado. En primer lugar, el programa permite iterar con el etiquetado, para corregir los resultados del algoritmo hasta que quedemos contentos. En segundo lugar, mejoramos los costes asociados a los nodos del grafo y si que usamos los histogramas de las semillas, por lo que en la propia construcción del grafo, esta información ya llega a todos los píxeles. A partir de ahí, unos mejores costes entre vecinos que además se obtienen utilizando un filtro de sobel, por lo que también se recoge información sobre los posibles bordes detectados. La última imagen segmentada nos dio especiales problemas, aún intentando forzar un trozo como objeto pintándolo nos daba problemas, por lo que añadimos un parámetro K que nos permite directamente pasar el coste de los t-links de las semillas, para forzar en mayor o menor medida las etiquetas de algunos píxeles.

Posibles mejoras podrían ser probar otras funciones para asociar costes e incluso que se pueda alternar entre una u otra. Si por ejemplo la imagen tiene bordes muy marcados entre fondo y objeto la versión ya implementada podría ser la elegida, pero si no estuviesen tan marcados, podría ser conveniente trabajar con otras funciones de coste.

También se podrían añadir otros parámetros similares a nuestra K , que diesen mas flexibilidad, de manera que podamos darle más importancia a los histogramas de las semillas, a la información que aportan los vecinos, etc.

Lo que observamos en los ejemplos es que cada imagen es un mundo, por lo que más que buscar un algoritmo único, parece más razonable buscar la versatilidad y algoritmos que permitan ser ajustados en base a unos parámetros para trabajar con la imagen que queramos segmentar.

[]: