PracticaPI2018_2

October 23, 2018

1 Procesamiento de Imágenes Digitales

Visión Computacional 2018-19 Practica 1. 3 de octubre de 2018

Este enunciado está en el archivo "PracticaPI2018.ipynb" o su versión "pdf" que puedes encontrar en el Aula Virtual.

1.1 Objetivos

Los objetivos de esta práctica son: * Programar algunas de las rutinas de transformaciones puntuales de procesamiento de imágenes y analizar el resultado de su aplicación. * 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 * Jupyter http://jupyter.org/. * Los paquetes "pip" y "PyMaxFlow" * Las librerías científicas de Python: NumPy, SciPy, y Matplotlib. * El paquete PyGame. * La librería 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 martes 23 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 los resultados. Las respuestas a los ejercicios debes introducirlas en tantas celdas de código o texto como creas necesarias, insertadas inmediatamente después de un enuciado y antes del siguiente.
2. Un documento "pdf" generado a partir del fuente de Jupyter, por ejemplo usando el comando `jupyter nbconvert --execute --to pdf notebook.ipynb`, o simplemente imprimiendo el "notebook" desde el navegador en la opción del menú "File->Print preview". Asegúrate de que el documento "pdf" contiene todos los resultados correctamente ejecutados.

- Esta práctica puede realizarse en parejas.

1.4 Instala el entorno de trabajo

En la distribución Linux Ubuntu 18.04, éstos son los comandos necesarios para instalar el entorno:
1. Instala los paquetes Python y Jupyter

```
``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 "pytho
```

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"

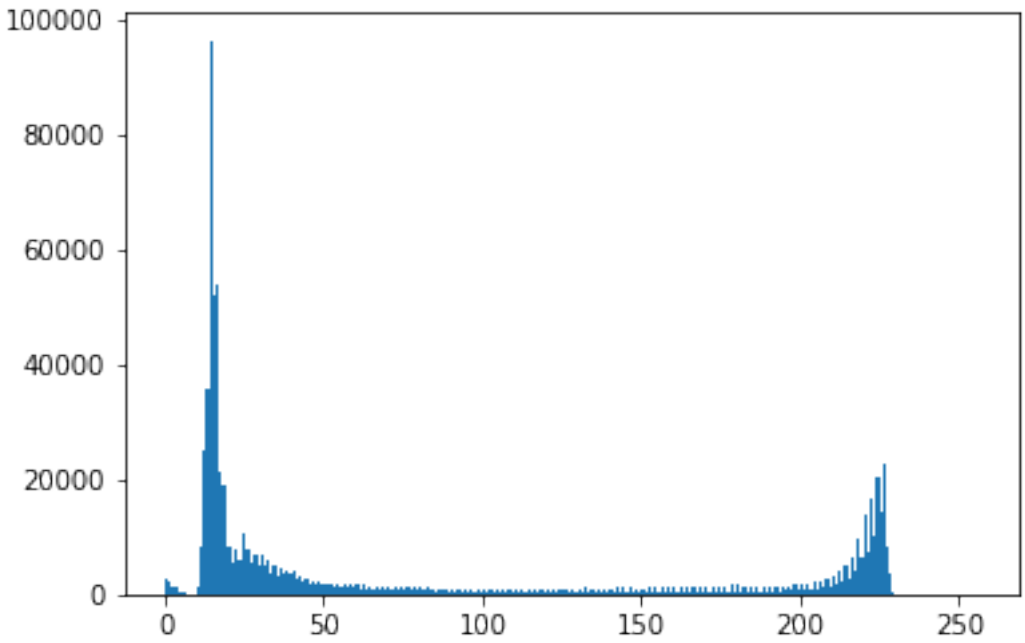
```
``apt install python-pygame``

Si deseas trabajar en Python 3.X, la versión 18.04 de Ubuntu no tiene el paquete "python3-pygar
```

1.5 Transformaciones puntuales

En este apartado te recomiendo que uses al menos la imagen indicada, que puedes encontrar en el directorio de imágenes del aula virtual. También puedes probar con otras que te parezcan interesantes.

Ejercicio 1. Carga la imagen `escilum.tif`. Calcula y muestra su histograma, por ejemplo, con la función `hist()` de `matplotlib.pyplot`. A la vista del histograma, discute qué problema tiene la imagen para analizar visualmente la región inferior izquierda.

```
In [2]: import os
        import matplotlib.pyplot as plt
        import cv2
        import numpy as np

        %matplotlib inline

        image = cv2.imread("imagenes/escilum.tif")
        # plt.imshow(image)

        plt.hist(image.ravel(),256,[0,256])
        plt.show()
```



1.5.1 Resultado

La regíon inferior izquierda de la imagen está muy oscura y esto queda reflejado en el histograma viendo el pico de la parte izquierda (cercana al 0). El poco contraste entre los pixeles dificulta analizar que objetos hay en la región.

Ejercicio 2. Escribe una función `eq_hist(histograma)` que calcule la función de transformación puntual que ecualiza el histograma. Aplica la función de transformación a la imagen anterior. Calcula y muestra nuevamente el histograma y la imagen resultantes, así como la función de transformación.

Discute los resultados obtenidos. £Cuál sería el resultado si volviésemos a ecualizar la imagen resultante?

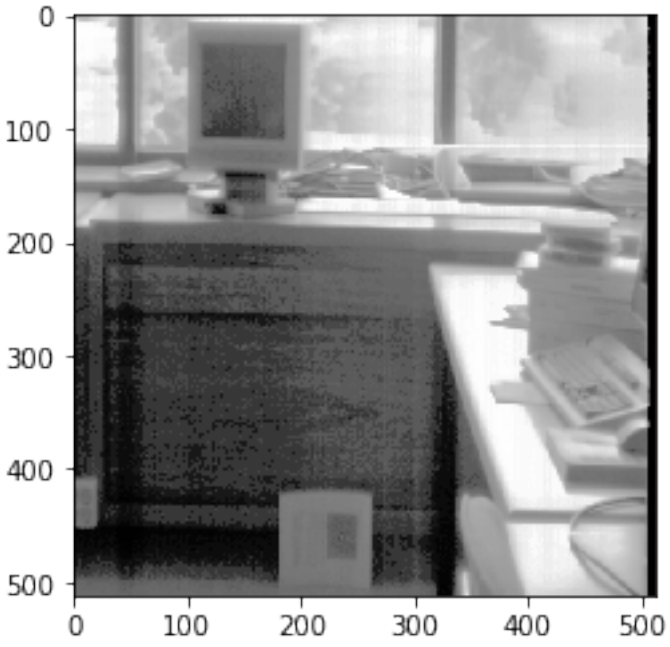
En este ejercicio tienes que implementar la función que ecualiza el histograma. No puedes usar funciones que lo hagan por ti.

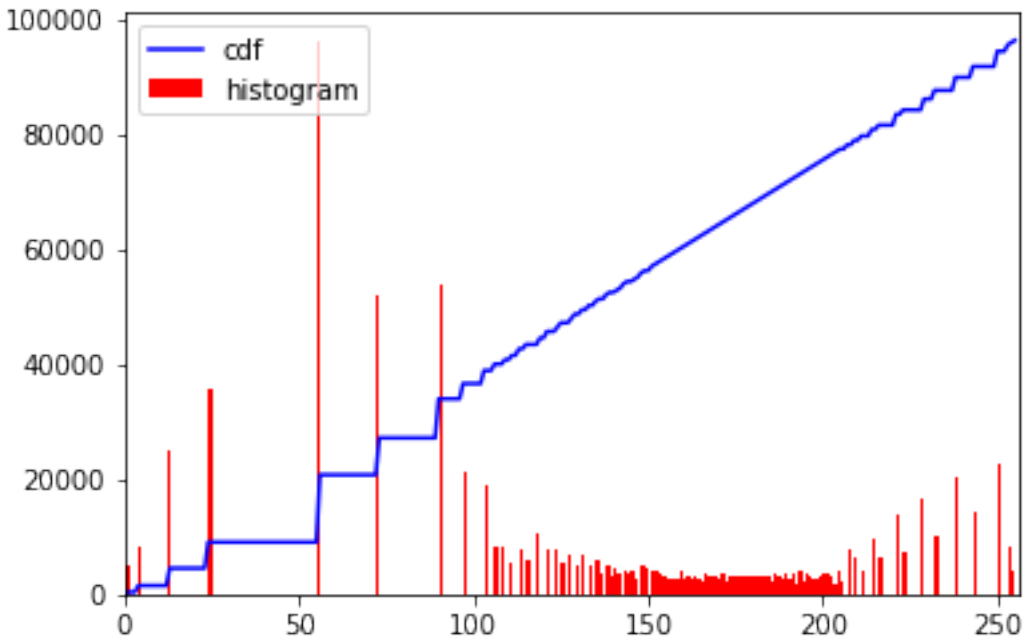
```
In [3]: import os
        import matplotlib.pyplot as plt
        import cv2
        from pprint import pprint as pp

        %matplotlib inline

        image = cv2.imread("imagenes/escilum.tif")
        # plt.imshow(image)

        hist,bins = np.histogram(image.flatten(),256,[0,256])
```

```
        def eq_hist(hist):
            # Obtenemos la función de distribución
            cdf = hist.cumsum()

            # Creamos la máscara (no se aplica filtro a ningún valor puesto que ninguno vale 0
            cdf_m = np.ma.masked_equal(cdf,0)

            cdf_m = (cdf_m - cdf_m.min())*255/(cdf_m.max()-cdf_m.min())
            # Transformamos en enteros (solo cogemos la parte entera)
            cdf = np.ma.filled(cdf_m,0).astype('uint8')

            return cdf

        # CDF: Cumulative Distribution Function
        cdf = eq_hist(hist)
        pp(cdf)
        pp(type(cdf))
        img2 = cdf[image]

        hist2,bins2 = np.histogram(img2.flatten(),256,[0,256])

        cdf2 = eq_hist(hist2)
        img3 = cdf2[img2]
        cdf_normalized = cdf2 * hist2.max()/ cdf2.max()

        plt.figure()
        plt.imshow(img3)

        plt.figure()
        plt.plot(cdf_normalized, color = 'b')
        plt.hist(img3.flatten(),256,[0,256], color = 'r')
        plt.xlim([0,256])
        plt.legend(('cdf','histogram'), loc = 'upper left')
        plt.show()
```

```
array([  0,   0,   1,   1,   1,   1,   1,   1,   1,   1,   2,   4,  13,
        24,  56,  73,  90,  97, 103, 106, 109, 111, 113, 115, 119, 121,
       124, 125, 128, 129, 131, 133, 135, 136, 138, 139, 141, 142, 143,
       144, 146, 147, 148, 148, 149, 150, 151, 151, 152, 152, 153, 153,
       154, 154, 155, 155, 156, 156, 157, 157, 158, 158, 159, 159, 159,
       160, 160, 160, 161, 161, 161, 162, 162, 162, 163, 163, 163, 164,
       164, 164, 165, 165, 166, 166, 166, 166, 167, 167, 167, 167, 168,
       168, 168, 168, 169, 169, 169, 169, 169, 170, 170, 170, 170, 170,
       171, 171, 171, 171, 171, 171, 172, 172, 172, 172, 173, 173, 173,
       173, 173, 174, 174, 174, 174, 174, 175, 175, 175, 175, 176, 176,
       176, 176, 177, 177, 177, 177, 178, 178, 178, 178, 179, 179, 179,
       179, 180, 180, 180, 180, 181, 181, 181, 181, 182, 182, 182, 182,
       183, 183, 183, 183, 184, 184, 184, 185, 185, 185, 186, 186, 186,
```

```
       186, 187, 187, 187, 188, 188, 188, 189, 189, 189, 190, 190, 190,
       191, 191, 191, 191, 192, 192, 193, 193, 193, 193, 194, 194, 195,
       195, 195, 196, 196, 196, 197, 197, 198, 198, 199, 199, 200, 200,
       201, 201, 202, 203, 204, 205, 207, 207, 209, 211, 214, 216, 221,
       223, 229, 232, 238, 243, 250, 253, 254, 254, 255, 255, 255, 255,
       255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
       255, 255, 255, 255, 255, 255, 255, 255, 255], dtype=uint8)
<class 'numpy.ndarray'>
```





1.5.2 Resultado

El resultado de equalizar la imagen más de una vez será siempre el mismo, es decir, obtendremos el mismo histograma y la misma imagen. Además, podemos ver como en el resultado los grises estan mejor distribuidos en el hisograma y la imagen se puede ver de manera más clara.

1.5.3 Fuentes

Información obtenida de: https://en.wikipedia.org/wiki/Histogram_equalization

1.6 Filtrado

Para realizar las convoluciones utiliza la función `convolve` o `convolve1d` de `scipy.ndimage.filters`, segun corresponda.

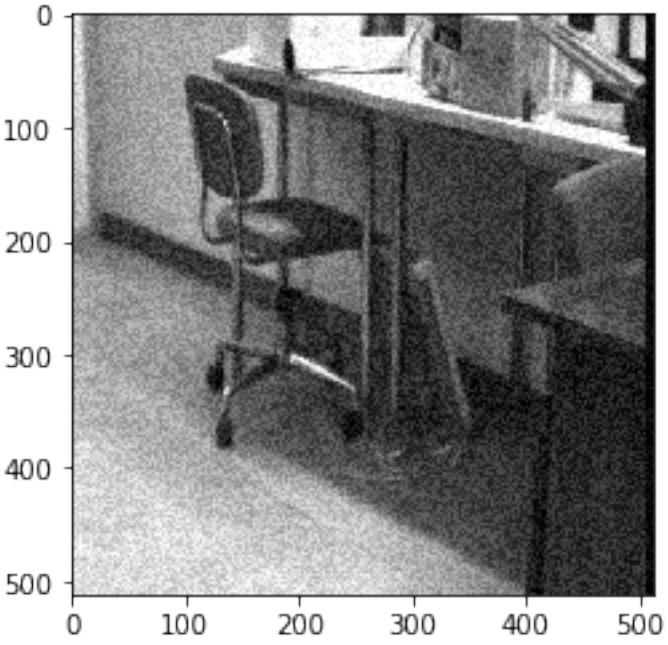
Carga y muestra las imágenes `escgaus.bmp` y `escimp5.bmp` que están contaminadas respectivamente con ruido de tipo gaussiano e impulsional. En los siguientes ejercicios también puedes utilizar otras imágenes que te parezcan interesantes.

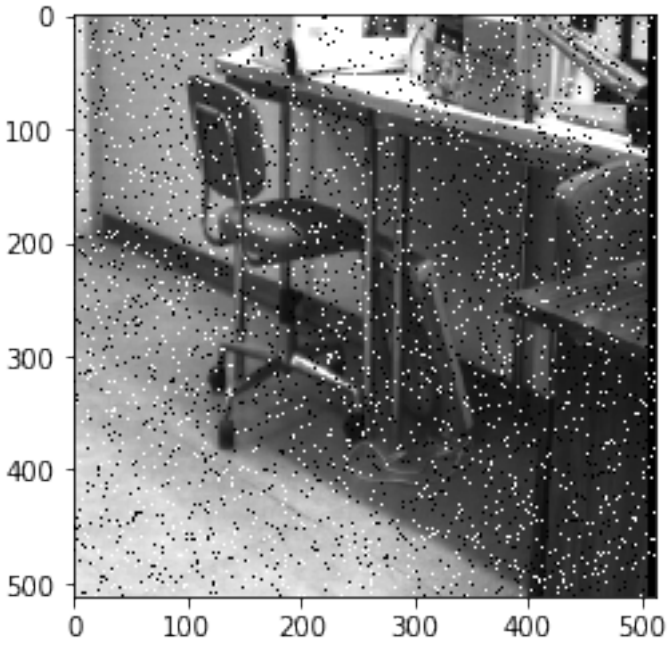
```
In [4]: from scipy.ndimage.filters import convolve, convolve1d, gaussian_filter
        import math
        from skimage import io, viewer
        from scipy import fftpack

        img1 = cv2.imread('imagenes/escgaus.bmp', 0)
        img2 = cv2.imread('imagenes/escimp5.bmp', 0)
```

```
        plt.figure()
        plt.imshow(img1, cmap = plt.cm.gray)

        plt.figure()
        plt.imshow(img2, cmap = plt.cm.gray)
```

```
/home/vision/.local/lib/python3.6/site-packages/skimage/viewer/__init__.py:6: UserWarning: Viev
  warn('Viewer requires Qt')
```

Out[4]: <matplotlib.image.AxesImage at 0x7fcd7a2e5cf8>





Ejercicio 3. Escribe una función `masc_gaus(sigma, n)` que construya una máscara de una dimensión de un filtro gaussiano de tamaño n y varianza . Filtra las imágenes anteriores con filtros bidimensionales de diferentes tamaños de n , y/o .

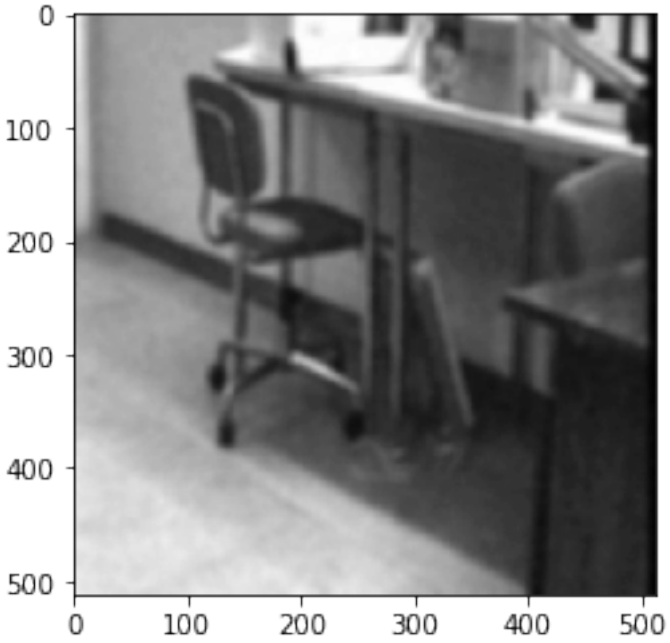
En este ejercicio tienes que implementar la función que construye la máscara. No puedes usar funciones que construyan la máscara o realicen el filtrado automáticamente.

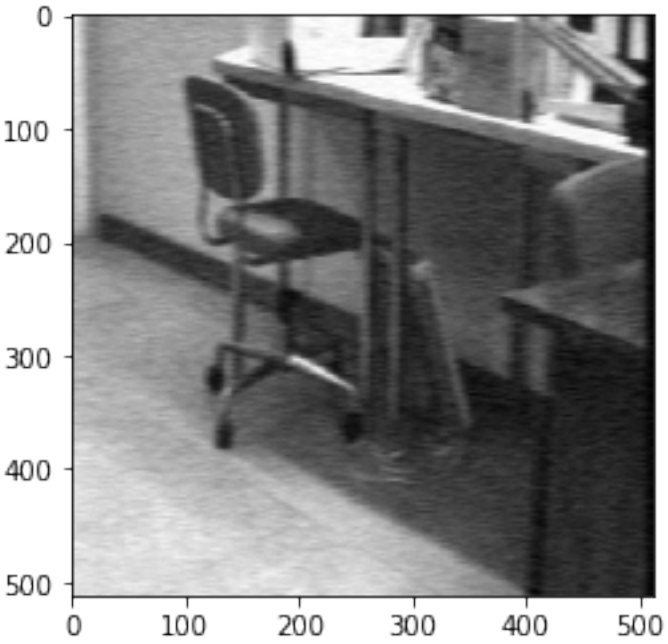
Muestra y discute los resultados. Pinta alguna de las máscaras construidas.

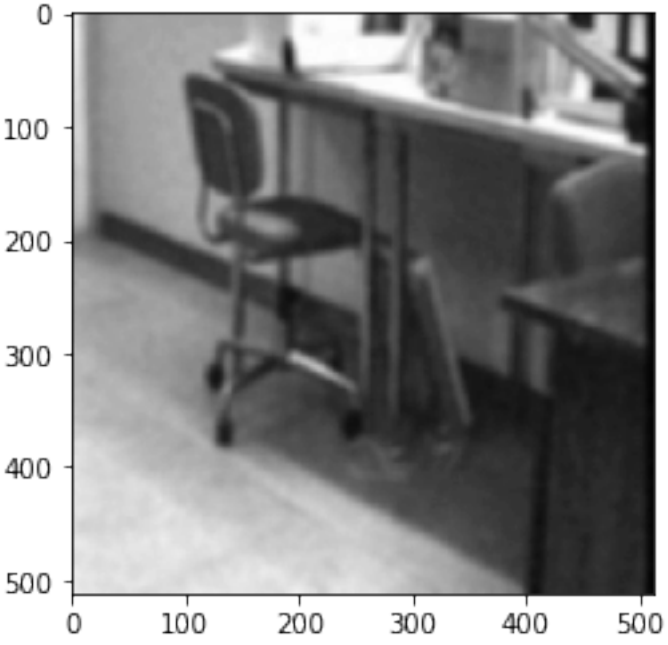
```
In [5]: def masc_gaus_1d(sigma, n):
            width = n//2
            dx = 1
            x = np.arange(-width, width)
            kernel_1d = np.exp(-(x ** 2) / (2 * sigma ** 2))
            kernel_1d = kernel_1d / (math.sqrt(2 * np.pi) * sigma)

            return kernel_1d

        def masc_gaus_2d(sigma, n):
            width = n//2
            dx = 1
            dy = 1
            x = np.arange(-width, width)
            y = np.arange(-width, width)
            x2d, y2d = np.meshgrid(x, y)
            kernel_2d = np.exp(-(x2d ** 2 + y2d ** 2) / (2 * sigma ** 2))
            kernel_2d = kernel_2d / (2 * np.pi * sigma ** 2)

            return kernel_2d
```

In [29]:
```
sigma = 5
n = 11

# Se ha usado para comparar
# filter_op = gaussian_filter(img1, 7)

kernel1D = masc_gaus_1d(sigma, n)

t2_kernel1D = kernel1D[:, None]
t_kernel1D = t2_kernel1D.T
k_kernel2D = t2_kernel1D * t_kernel1D

img_convolved_1d1 = convolve(img1, k_kernel2D)
plt.figure()
plt.imshow(img_convolved_1d1, cmap = plt.cm.gray)

img_convolved_1d = convolve1d(img1, kernel1D)
plt.figure()
plt.imshow(img_convolved_1d, cmap = plt.cm.gray)

kernel = masc_gaus_2d(sigma, n)

img_convolved = convolve(img1, kernel)
plt.figure()
plt.imshow(img_convolved, cmap = plt.cm.gray)
```

Out[29]: <matplotlib.image.AxesImage at 0x7fcd6cfe97f0>




0
100
200
300
400
500
0
100
200
300
400
500


0
100
200
300
400
500
0
100
200
300
400
500

1.6.1 Resultados

Tal y como podemos ver, cuando se aumenta el valor de sigma o de n la imagen se va distorsinando cada vez más hasta hacer la imagen completamente negra. Sin embargo, a valores bajos de cualquiera de las dos variables, la imagen queda prácticamente igual que en la original.

1.6.2 Fuentes

Datos obtenidos siguiendo la fórmula de: https://en.wikipedia.org/wiki/Gaussian_blur

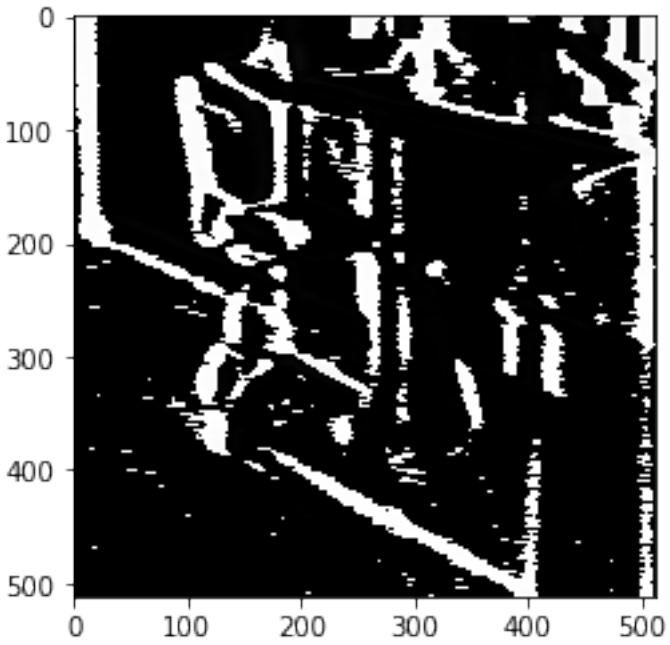
Ejercicio 4. Escribe una función `masc_deriv_gaus(sigma, n)` que construya una máscara de una dimensión de un filtro derivada del gaussiano de tamaño n y varianza . Filtra la imagen `corridor.jpg` con filtros bidimensionales de derivada del gaussiano para extraer los bordes de la imagen. Prueba con diferentes valores de n y/o .

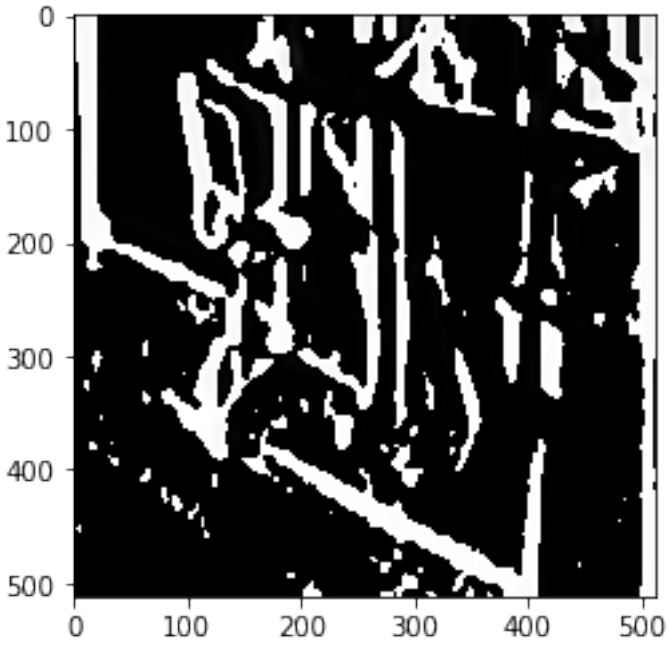
Muestra y discute los resultados. Pinta alguna de las máscaras construidas.

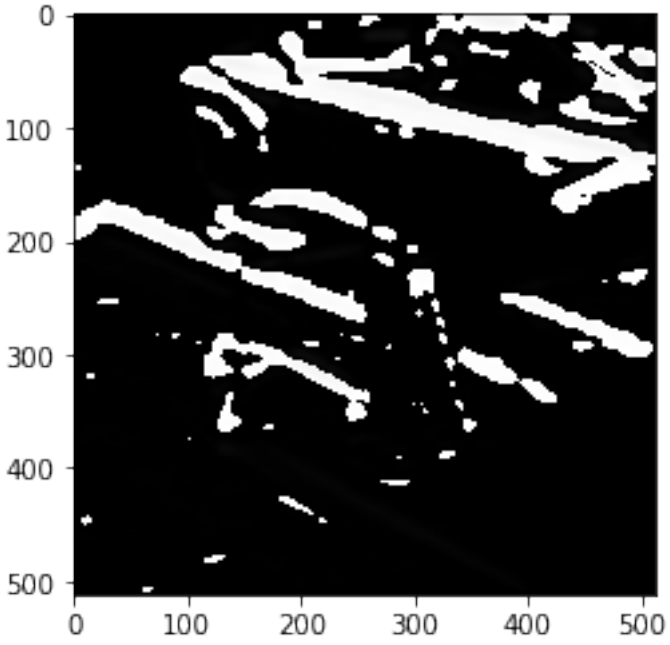
```
In [7]: def masc_deriv_gaus_1d(sigma, n, order):
            width = n//2
            dx = 1
            x = np.arange(-width, width)
            kernel_1d = x * np.exp(-(x ** 2) / (2 * sigma ** 2))
            kernel_1d = -kernel_1d / (math.sqrt(2 * np.pi) * sigma ** 3)

            return kernel_1d

        def masc_deriv_gaus_2d(sigma, n, orientation):
            width = n//2
            dx = 1
            dy = 1
            x = np.arange(-width, width)
            y = np.arange(-width, width)
            x2d, y2d = np.meshgrid(x, y)
            if orientation == 1:
                kernel_2d = x2d * np.exp(-(x2d ** 2 + y2d ** 2) / (2 * sigma ** 2))
            else:
                kernel_2d = y2d * np.exp(-(x2d ** 2 + y2d ** 2) / (2 * sigma ** 2))
            kernel_2d = -kernel_2d / (2 * np.pi * sigma ** 4)

            return kernel_2d

In [30]: sigma = 7
         n = 31
         order = 1

         # Orientation = 1 means x dervate. Orientation = 2 means y dervate.
         orientation = 1

         kernel1D_d = masc_deriv_gaus_1d(sigma, n, order)
         img_convolved_1d = convolve1d(img1, kernel1D_d)
         plt.figure()
```

```
plt.imshow(img_convolved_1d, cmap = plt.cm.gray)
# pp(kernel1D_d)

sigma = 4
n = 41

# Derivada en X
kernel2D_x = masc_deriv_gaus_2d(sigma, n, orientation)
img_convolved_d2_x = convolve(img1, kernel2D_x)
plt.figure()
plt.imshow(img_convolved_d2_x, cmap = plt.cm.gray)

# Derivada en Y
orientation = 2
kernel2D_y = masc_deriv_gaus_2d(sigma, n, orientation)
img_convolved_d2_y = convolve(img1, kernel2D_y)
plt.figure()
plt.imshow(img_convolved_d2_y, cmap = plt.cm.gray)
```

Out[30]: <matplotlib.image.AxesImage at 0x7fcd6d121f98>







1.6.3 Resultados

Como podemos observar, la derivada de primer orden nos da los bordes de la imagen. La derivadas en x nos da los bordes verticales y la derivada en y nos da los bordes horizontales

1.6.4 Fuentes

Hemos utilizado: http://campar.in.tum.de/Chair/HaukeHeibelGaussianDerivatives

Ejercicio 5. Compara los tiempos de ejecución de las convoluciones anteriores cuando se realizan con `convolve1d` en vez de con `convolve`. Analiza los tiempos para diferentes valores de n y justifica los resultados.

```
In [9]: # Posible ejemplo de código
        import time

        # ejecuta convoluciones ....
        print("Caso base\n")
        start_time = time.clock()

        sigma = 7
        n = 31

        kernel1D = masc_gaus_1d(sigma, n)
        img_convolved_1d1 = convolve(img1, k_kernel2D)

        print("1 Dimensión, Sigma: {}, n: {}. Total time: {} seconds.\n".format(sigma, n, time
        start_time = time.clock()

        kernel = masc_gaus_2d(sigma, n)
        img_convolved = convolve(img1, kernel)

        print("2 Dimensión, Sigma: {}, n: {}. Total time: {} seconds.\n".format(sigma, n, time


        # ejecuta convoluciones ....
        print("Caso con la n aumentada\n")
        start_time = time.clock()

        sigma = 7
        n = 51

        kernel1D = masc_gaus_1d(sigma, n)
        img_convolved_1d1 = convolve(img1, k_kernel2D)

        print("1 Dimensión, Sigma: {}, n: {}. Total time: {} seconds.\n".format(sigma, n, time
        start_time = time.clock()

        kernel = masc_gaus_2d(sigma, n)
        img_convolved = convolve(img1, kernel)

        print("2 Dimensión, Sigma: {}, n: {}. Total time: {} seconds.\n".format(sigma, n, time
```

```
        # ejecuta convoluciones ....
        print("Caso con la sigma aumentada\n")
        start_time = time.clock()

        sigma = 11
        n = 31

        kernel1D = masc_gaus_1d(sigma, n)
        img_convolved_1d1 = convolve(img1, k_kernel2D)

        print("1 Dimensión, Sigma: {}, n: {}. Total time: {} seconds.\n".format(sigma, n, time
        start_time = time.clock()

        kernel = masc_gaus_2d(sigma, n)
        img_convolved = convolve(img1, kernel)

        print("2 Dimensión, Sigma: {}, n: {}. Total time: {} seconds.\n".format(sigma, n, time
```

```
Caso base

1 Dimensión, Sigma: 7, n: 31. Total time: 0.06399299999999997 seconds.

2 Dimensión, Sigma: 7, n: 31. Total time: 0.5859199999999998 seconds.

Caso con la n aumentada

1 Dimensión, Sigma: 7, n: 51. Total time: 0.06201300000000032 seconds.

2 Dimensión, Sigma: 7, n: 51. Total time: 1.8030860000000004 seconds.

Caso con la sigma aumentada

1 Dimensión, Sigma: 11, n: 31. Total time: 0.06226800000000132 seconds.

2 Dimensión, Sigma: 11, n: 31. Total time: 0.6144159999999985 seconds.
```

1.6.5 Resultado

Tal y como se puede observar, en el caso de una dimensión, al aumentar la n el tiempo aumenta muy poco y al aumentar sigma, tampoco crece mucho. Sin embargo en el caso de dos dimensiones, al aumentar la n, los tiempos aumentan drasticamente, y al aumentar sigma, la diferencia es muy pequeña. Esto se debe a que al aumentar el tamaño de la matriz, se multiplica el tiempo de computo.

Ejercicio 6. Aplica el filtro de la mediana a las imágenes `escgaus.bmp` y `escimp5.bmp` con diferentes valores de tamaño de la ventana. Muestra y discute los resultados. Compáralos con los obtenidos en el Ejercicio 3.

Para realizar este ejercicio puedes utilizar la función `cv2.medianBlur()` de OpenCV, `scipy.ndimage.median_filter()` de SciPy o hacer tu propia función. Para ello puedes escribir una función `mediana(img, n)` y aplicarla a la imagen con la función `scipy.ndimage.filters()`.

```
In [10]: image = cv2.imread("imagenes/escgaus.bmp")
         fig = plt.figure()
         fig.suptitle('Imagen original', fontsize=18)
         plt.imshow(image)

         fig = plt.figure()
         fig.suptitle('Imagen con n=3', fontsize=18)
         plt.imshow(cv2.medianBlur(image,3))

         fig = plt.figure()
         fig.suptitle('Imagen con n=7', fontsize=18)
         plt.imshow(cv2.medianBlur(image,7))

         image = cv2.imread("imagenes/escimp5.bmp")

         fig = plt.figure()
         fig.suptitle('"Salt & pepper"', fontsize=18)
         plt.imshow(image)

         fig = plt.figure()
         fig.suptitle('"Salt & pepper" con n=7', fontsize=18)
         plt.imshow(cv2.medianBlur(image,7))

Out[10]: <matplotlib.image.AxesImage at 0x7fcd78a100b8>
```

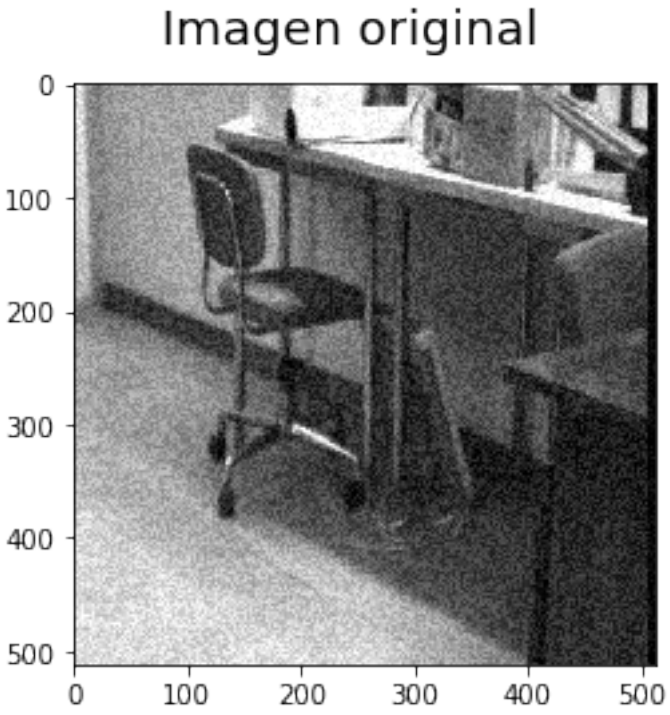


Imagen original
0
100
200
300
400
500
0
100
200
300
400
500

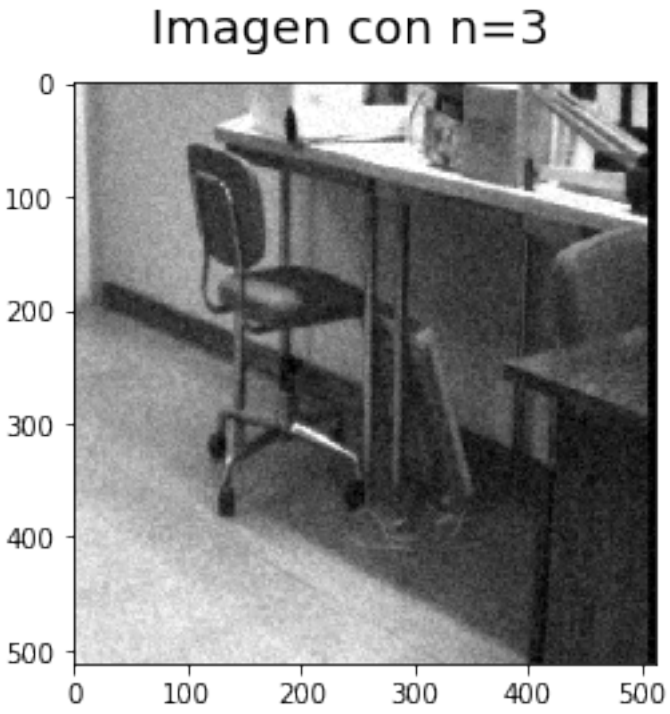


Imagen con n=3
0
100
200
300
400
500
0
100
200
300
400
500

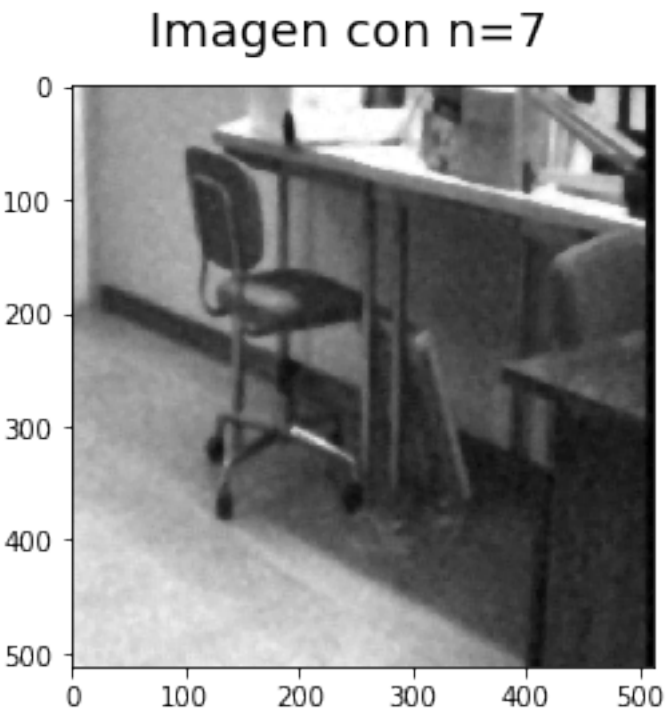
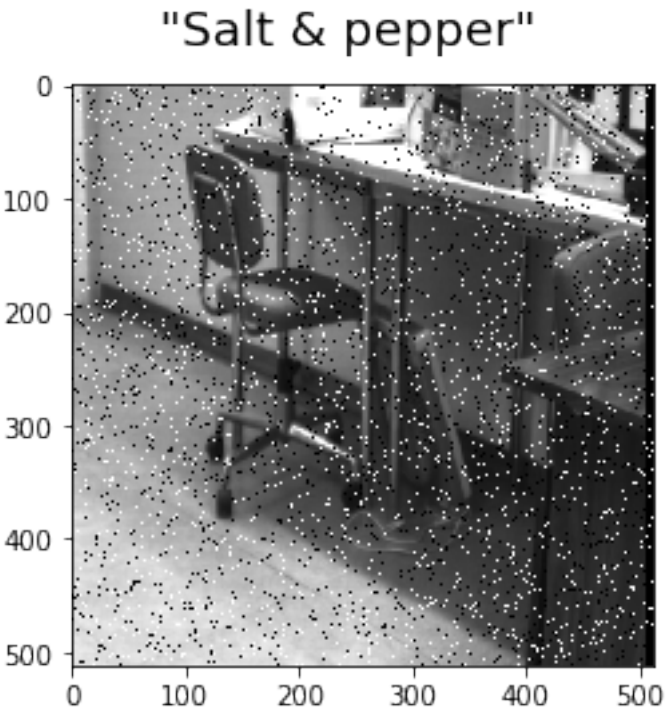
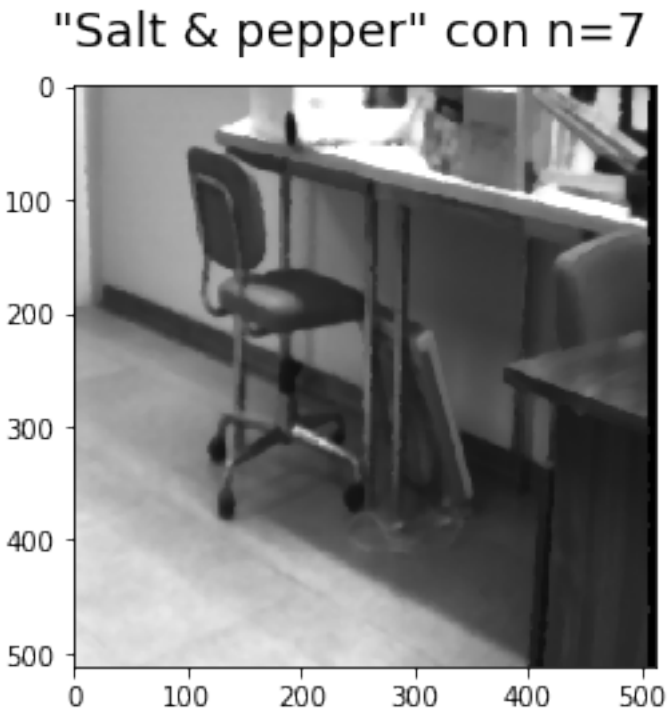


Imagen con n=7
0
100
200
300
400
500
0
100
200
300
400
500


"Salt & pepper"
0
100
200
300
400
500
0
100
200
300
400
500




```
In [11]: image = cv2.imread("imagenes/escimp5.bmp", 0)
         fig = plt.figure()
         fig.suptitle('Imagen original', fontsize=18)
         plt.imshow(image, plt.cm.gray)

         kernel = masc_gaus_2d(sigma = 7, n = 25)
         img_convolved = convolve(image, kernel)
         fig = plt.figure()
         fig.suptitle('Filtro gaussiano', fontsize=18)
         plt.imshow(img_convolved,plt.cm.gray)

         fig = plt.figure()
         fig.suptitle('Filtro de la mediana', fontsize=18)
         plt.imshow(cv2.medianBlur(image,7),plt.cm.gray)

Out[11]: <matplotlib.image.AxesImage at 0x7fcd787363c8>
```

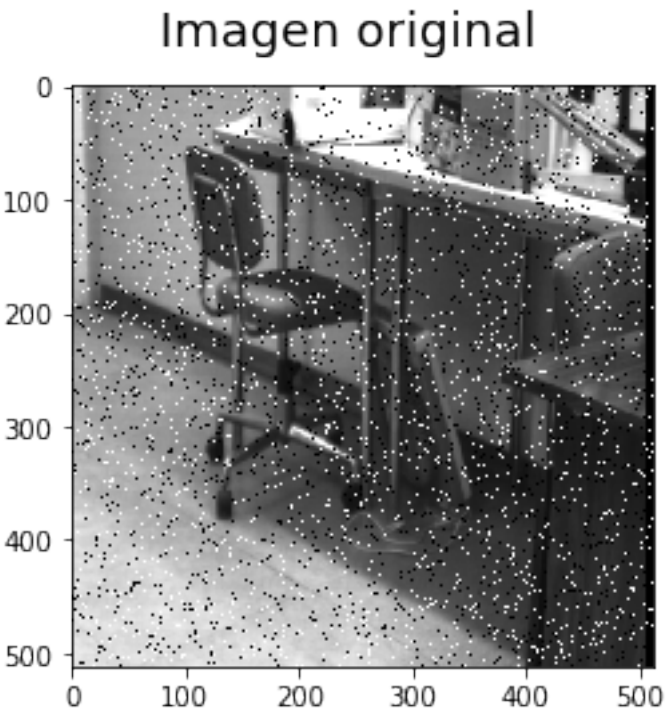
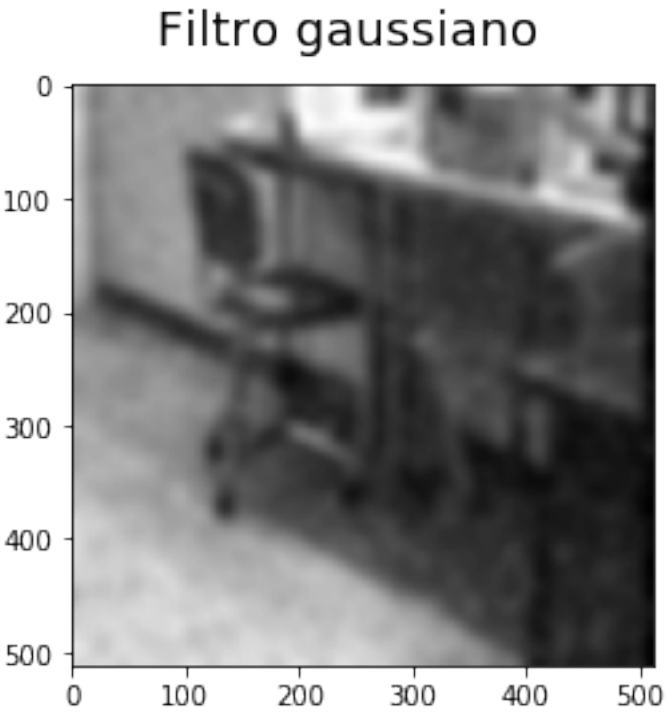
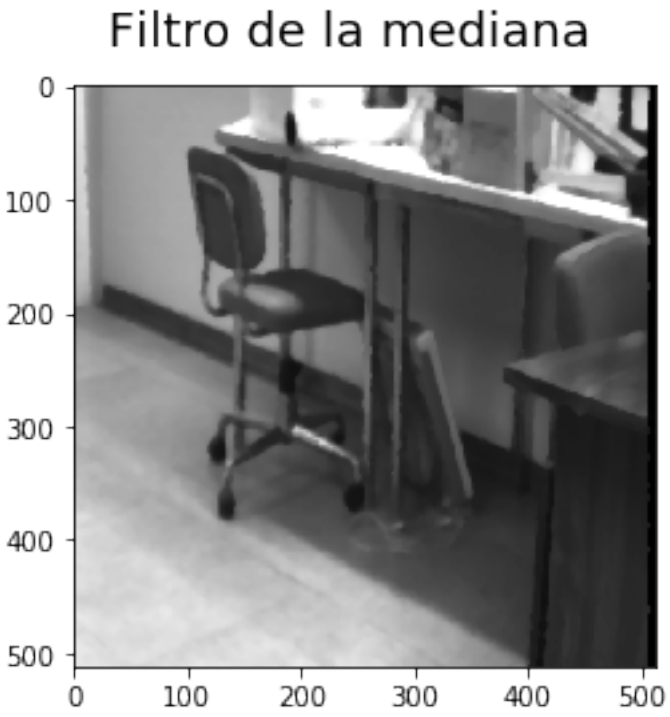


Imagen original
0
100
200
300
400
500
0
100
200
300
400
500


Filtro gaussiano
0
100
200
300
400
500
0
100
200
300
400
500



1.6.6 Resultado

Ante imagenes como la "escgaus.bmp" el filtro de la mediana da unos resultados semejantes a un filtro gaussiano. No obstante, cuando se intenta seguir suavizando la imagen cada vez se ve más emborronada.

Ante imagenes con el llamado ruido de "Salt & Pepper" podemos observar que el filtro de la mediana da muy buenos resultados, eliminando completamente el ruido de la imagen.

Como se puede observar, el filtro de la mediana ha dado mejores resultados que el filtro gaussiano a la hora de filtrar los ruidos impulsivos o los llamados ruidos de "Salt & Pepper"

Ejercicio 7. Utiliza la funcion `cv2.bilateralFilter()` de OpenCV para realizar el filtrado bilateral de una imagen. Selecciona los parámetros adecuados y aplícalo a las imágenes `escgaus.bmp` y `escimp5.bmp` y otras que elijas tú.

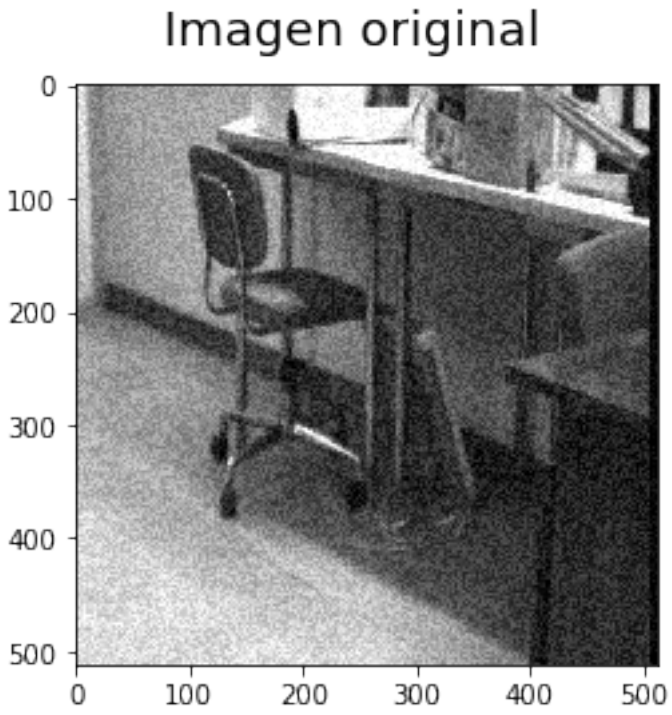
Si llamamos σ_r a la varianza de de la gaussiana que controla la ponderación debida a la diferencia entre los valores de los píxeles y σ_s a la varianza de la gaussiana que controla la ponderación debida a la posición de los píxeles. Responde a la siguientes preguntas: * £Cómo se comporta el filtro bilateral 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?

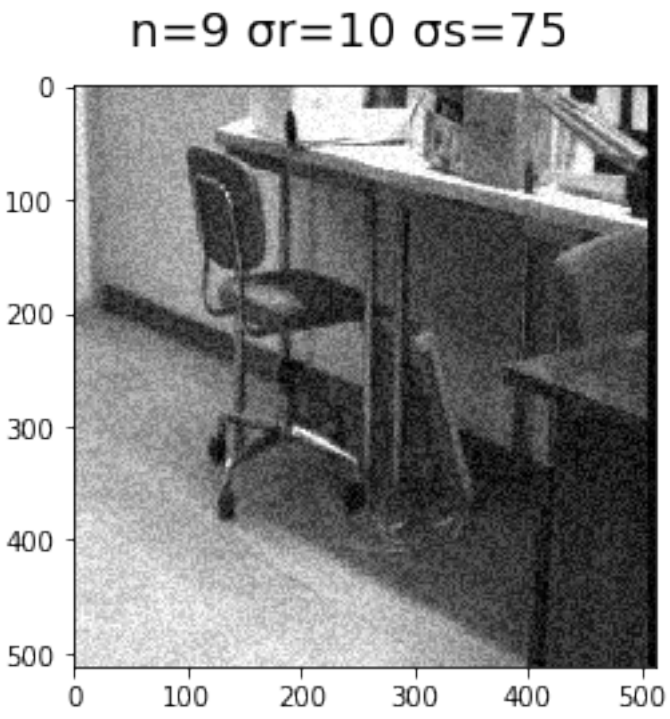
Muestra y discute los resultados para distintos valores de los parámetros y varias aplicaciones sucesivas del filtro. Compáralos con los obtenidos en los Ejercicios 3 y 6.

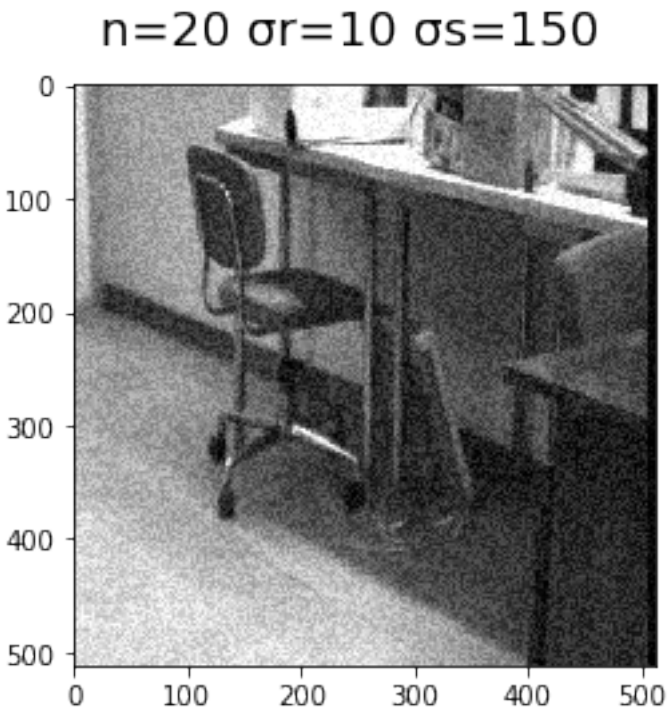
```
In [12]: image = cv2.imread("imagenes/escgaus.bmp")

         fig = plt.figure()
```

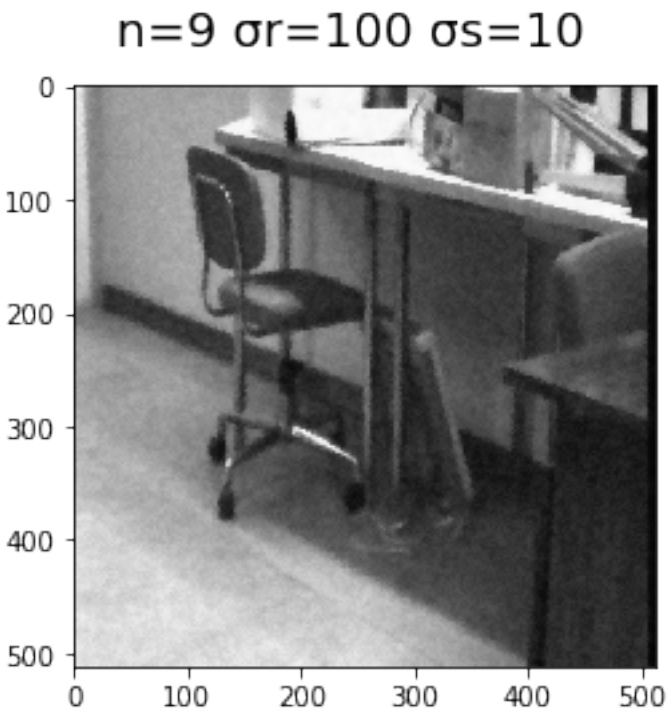
```
fig.suptitle('Imagen original', fontsize=18)
plt.imshow(image)

fig = plt.figure()
fig.suptitle('n=9 r=10 s=75', fontsize=18)
plt.imshow(cv2.bilateralFilter(image,9,10,75))

fig = plt.figure()
fig.suptitle('n=20 r=10 s=150', fontsize=18)
plt.imshow(cv2.bilateralFilter(image,20,10,150))

fig = plt.figure()
fig.suptitle('n=9 r=100 s=10', fontsize=18)
plt.imshow(cv2.bilateralFilter(image,9,100,10))

fig = plt.figure()
fig.suptitle('n=9 r=100 s=50', fontsize=18)
plt.imshow(cv2.bilateralFilter(image,9,100,50))

fig = plt.figure()
fig.suptitle('n=20 r=100 s=10', fontsize=18)
plt.imshow(cv2.bilateralFilter(image,20,100,10))
```

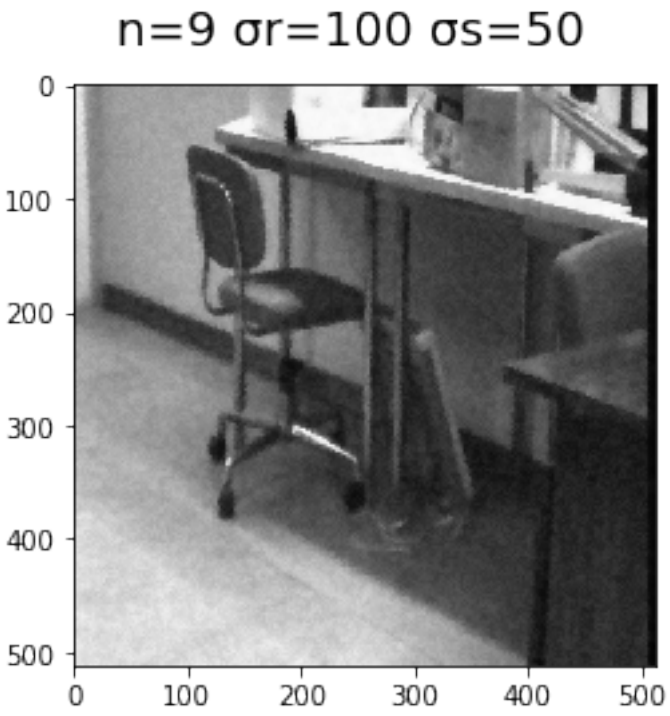
Out[12]: <matplotlib.image.AxesImage at 0x7fcd780d11d0>

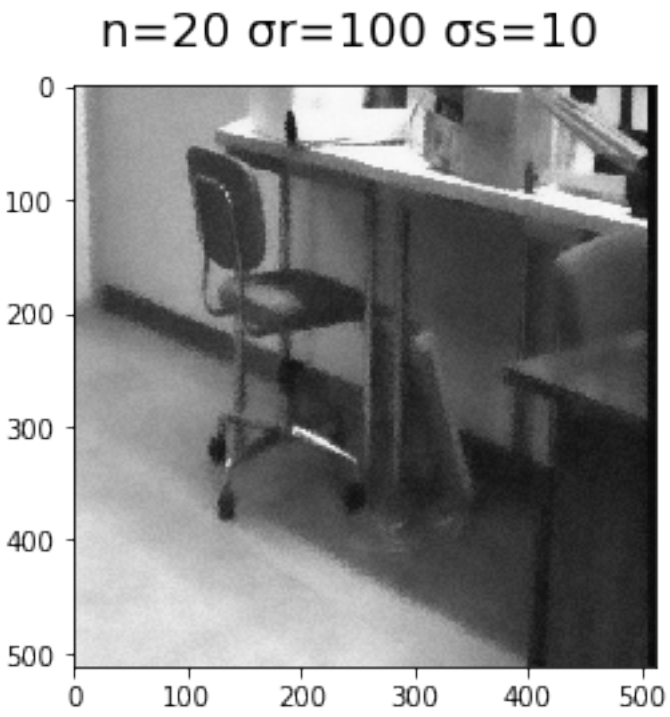



n=9 σr=10 σs=75
0
100
200
300
400
500
0
100
200
300
400
500


n=20 σr=10 σs=150
0
100
200
300
400
500
0
100
200
300
400
500


n=9 σr=100 σs=10
0
100
200
300
400
500
0
100
200
300
400
500


n=9 σr=100 σs=50
0
100
200
300
400
500
0
100
200
300
400
500



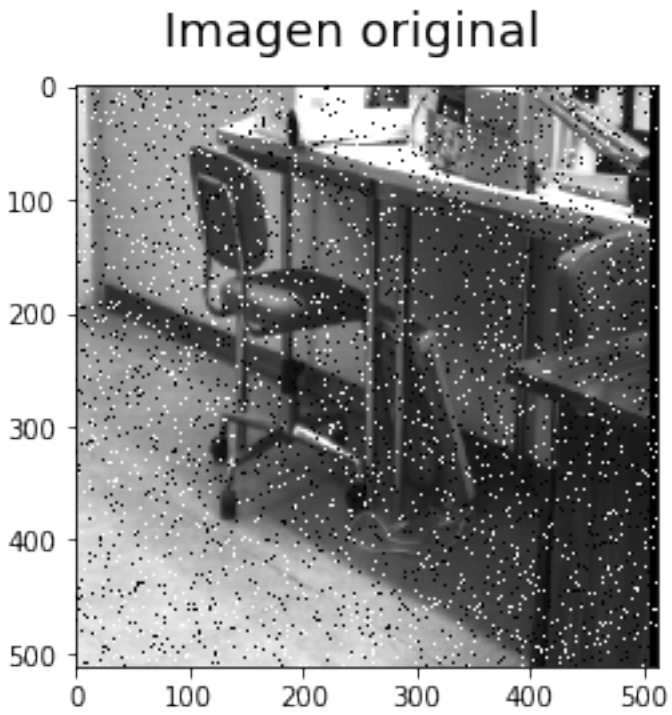
1.6.7 Resultado

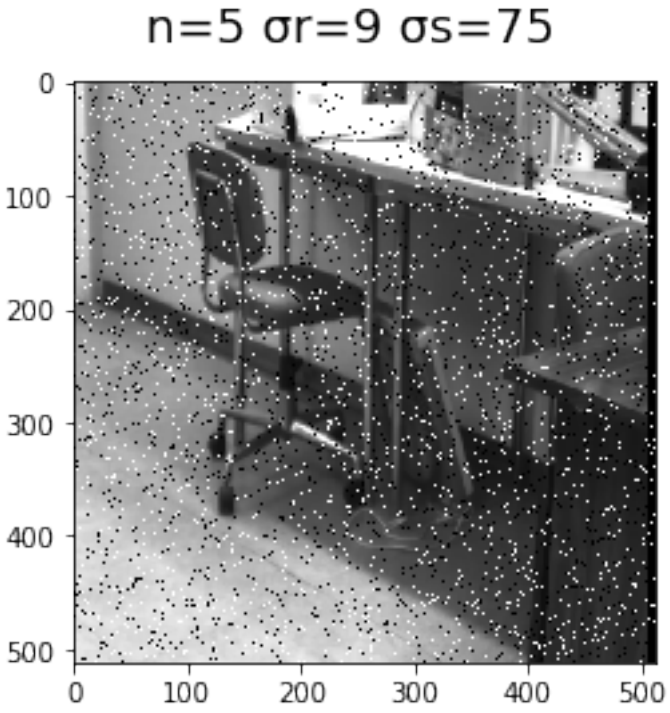
Con escgaus.bmp y el filtro bilateral obtenemos mejores resultados que los vistos con el filtro gaussiano o el filtro de la mediana. La configuración de sigma sub-s nos permite controlar los bordes de las estructuras pequeñas que deseamos proteger mientras sigma sub-r nos permite controlar el nivel de suavizado que deseamos en la imagen.

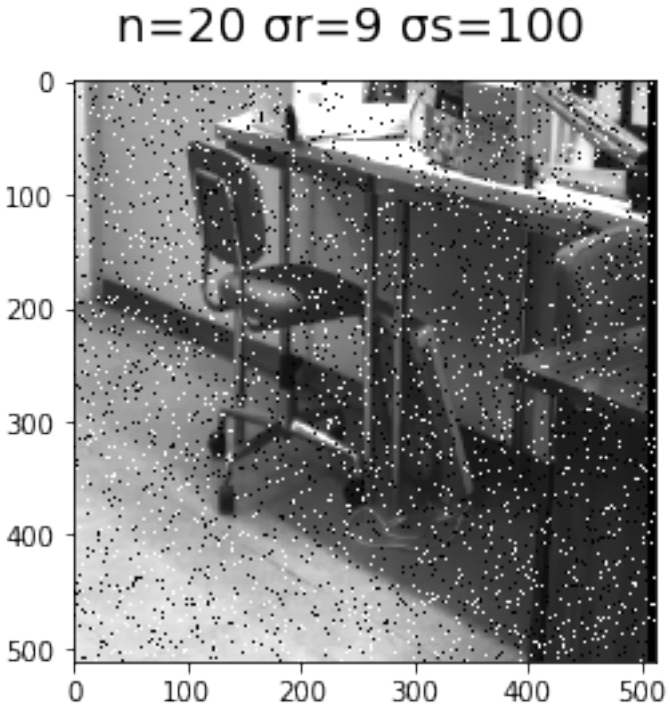
```
In [13]: image = cv2.imread("imagenes/escimp5.bmp")
         fig = plt.figure()
         fig.suptitle('Imagen original', fontsize=18)
         plt.imshow(image)

         fig = plt.figure()
         fig.suptitle('n=5 r=9 s=75', fontsize=18)
         plt.imshow(cv2.bilateralFilter(image,5,9,75))

         fig = plt.figure()
         fig.suptitle('n=20 r=9 s=100', fontsize=18)
         plt.imshow(cv2.bilateralFilter(image,20,9,100))

         fig = plt.figure()
         fig.suptitle('n=20 r=200 s=200', fontsize=18)
         plt.imshow(cv2.bilateralFilter(image,20,200,200))
```

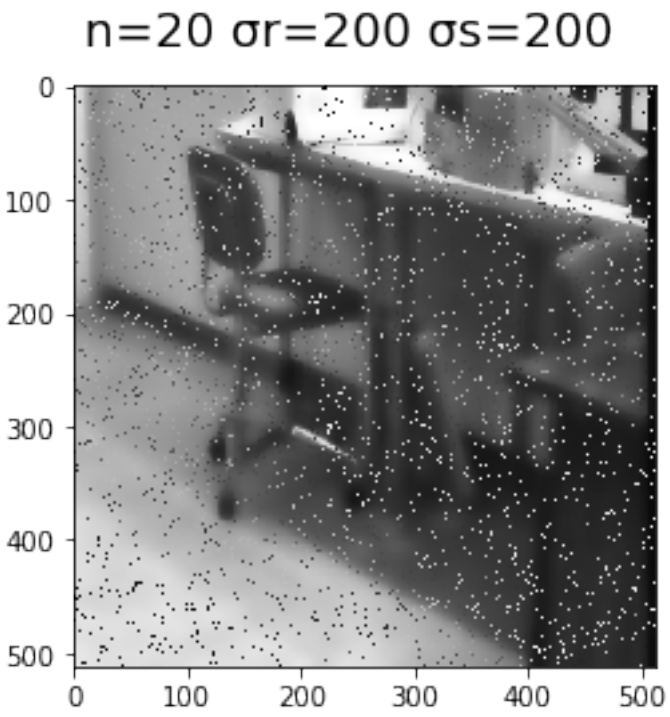
```
fig = plt.figure()
fig.suptitle('n=10 r=400 s=10', fontsize=18)
plt.imshow(cv2.bilateralFilter(image,10,400,10))
```

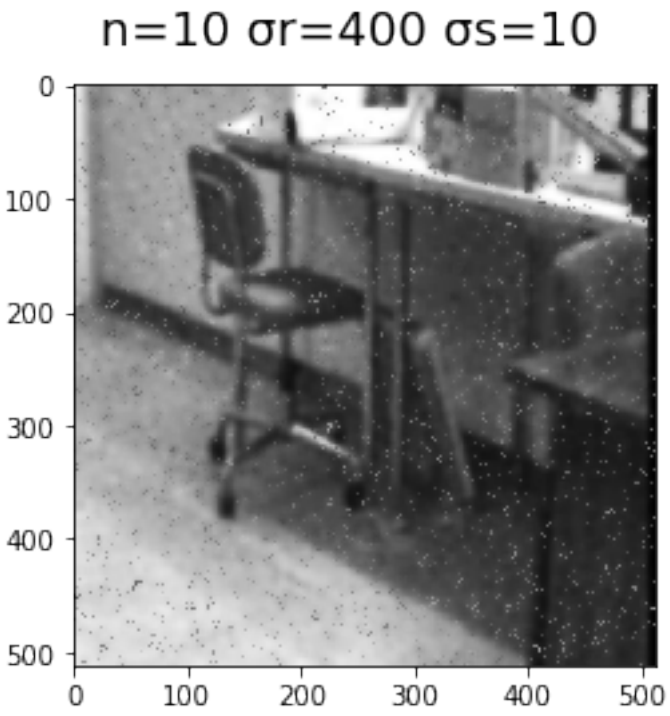
Out[13]: <matplotlib.image.AxesImage at 0x7fcd73e8edd8>




n=5 σr=9 σs=75
0
100
200
300
400
500
0
100
200
300
400
500


n=20 σr=9 σs=100
0
100
200
300
400
500
0
100
200
300
400
500


n=20 σr=200 σs=200
0
100
200
300
400
500
0
100
200
300
400
500


n=10 σr=400 σs=10
0
100
200
300
400
500
0
100
200
300
400
500

1.6.8 Resultado

A diferencia que en el apartado anterior, con una imagen con ruido de "Salt & Pepper" como puede ser escimp5.bmp el filtro bilateral no nos da buenos resultados, siendo una mejor opción usar el filtro de la mediana para quitar el ruido de "Sal & Pepper".

```
In [14]: image = cv2.imread("imagenes/flower.png")
         fig = plt.figure()
         fig.suptitle('Imagen original', fontsize=18)
         plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))

         fig = plt.figure()
         fig.suptitle('n=9 r=100 s=10', fontsize=18)
         plt.imshow(cv2.cvtColor(cv2.bilateralFilter(image,9,100,10), cv2.COLOR_BGR2RGB))

         fig = plt.figure()
         fig.suptitle('n=20 r=10 s=100', fontsize=18)
         plt.imshow(cv2.cvtColor(cv2.bilateralFilter(image,20,10,100), cv2.COLOR_BGR2RGB))

         fig = plt.figure()
         fig.suptitle('n=30 r=10 s=100', fontsize=18)
         plt.imshow(cv2.cvtColor(cv2.bilateralFilter(image,30,10,100), cv2.COLOR_BGR2RGB))

         fig = plt.figure()
         fig.suptitle('n=20 r=50 s=100', fontsize=18)
         plt.imshow(cv2.cvtColor(cv2.bilateralFilter(image,20,50,100), cv2.COLOR_BGR2RGB))

         fig = plt.figure()
         fig.suptitle('n=30 r=50 s=100', fontsize=18)
         plt.imshow(cv2.cvtColor(cv2.bilateralFilter(image,30,50,100), cv2.COLOR_BGR2RGB))

Out[14]: <matplotlib.image.AxesImage at 0x7fcd7806e358>
```

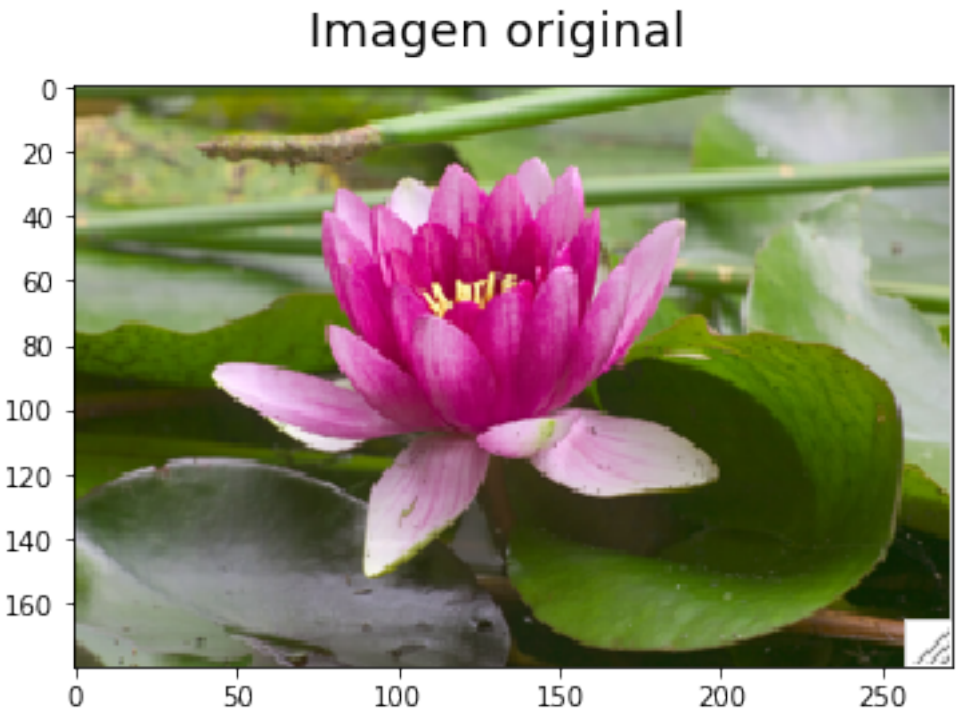
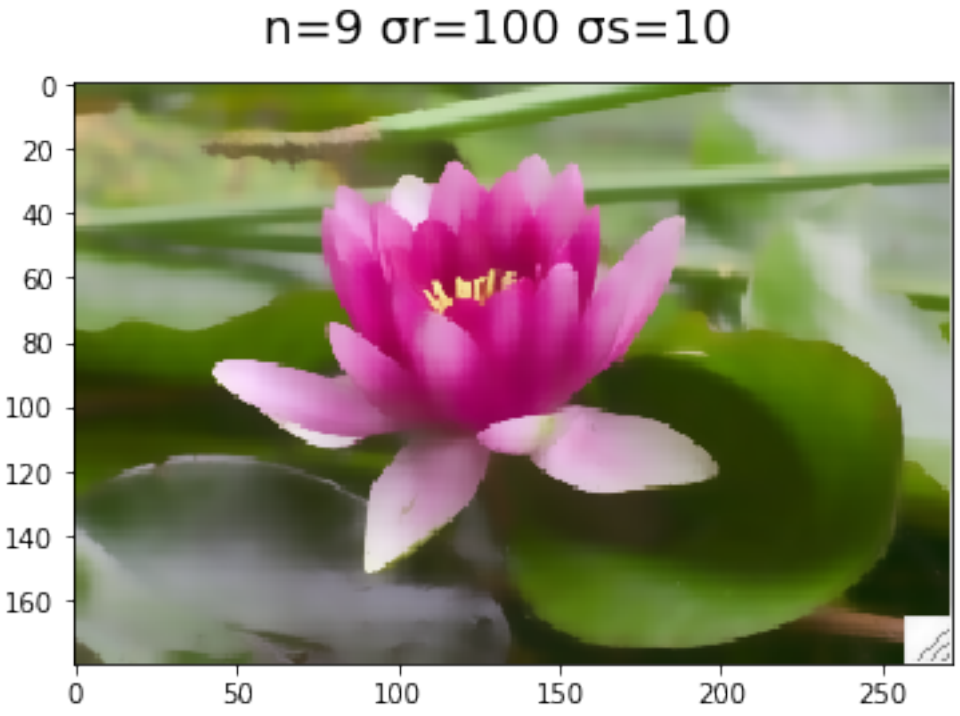
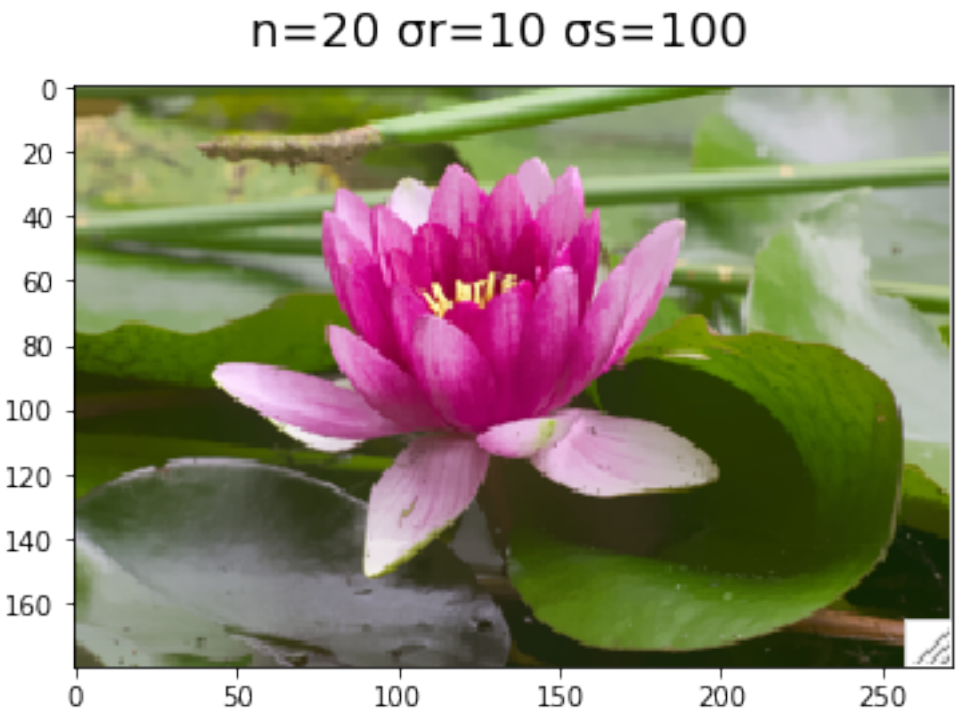
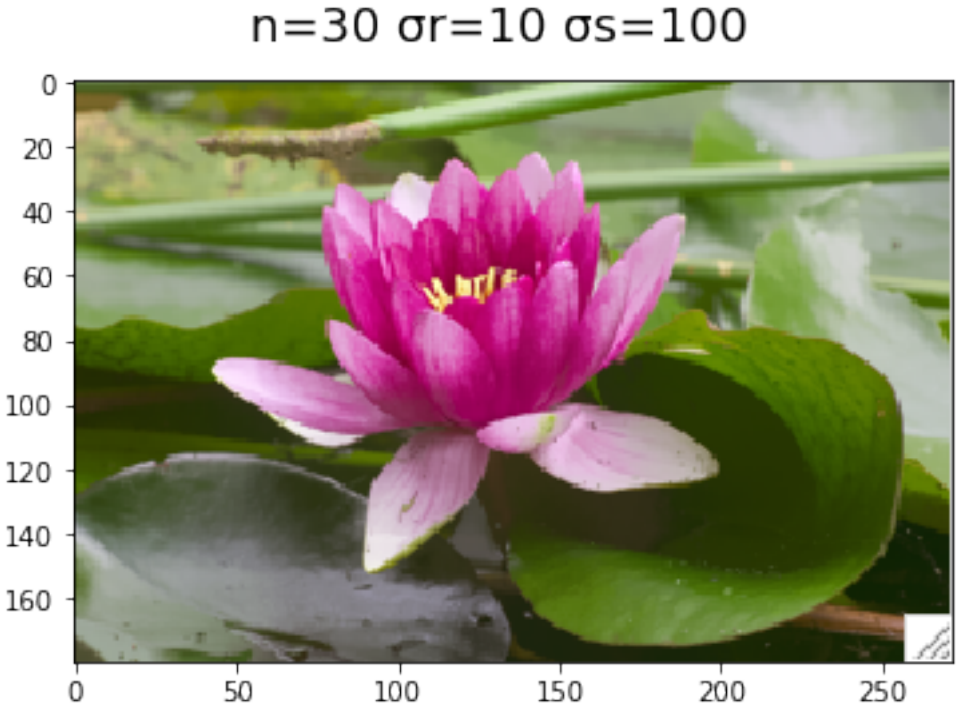
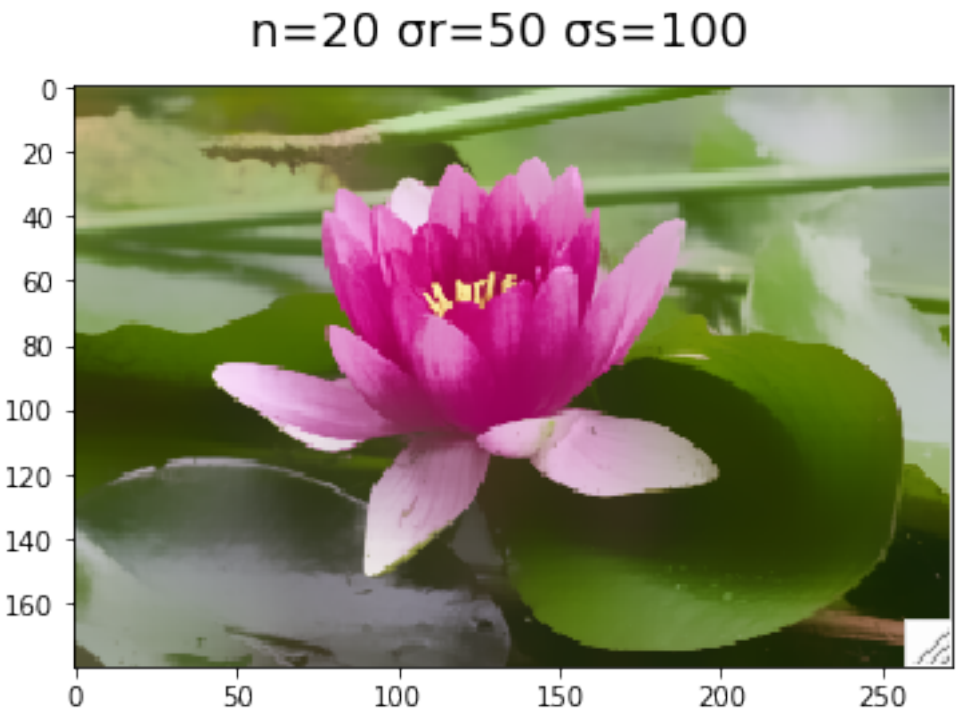


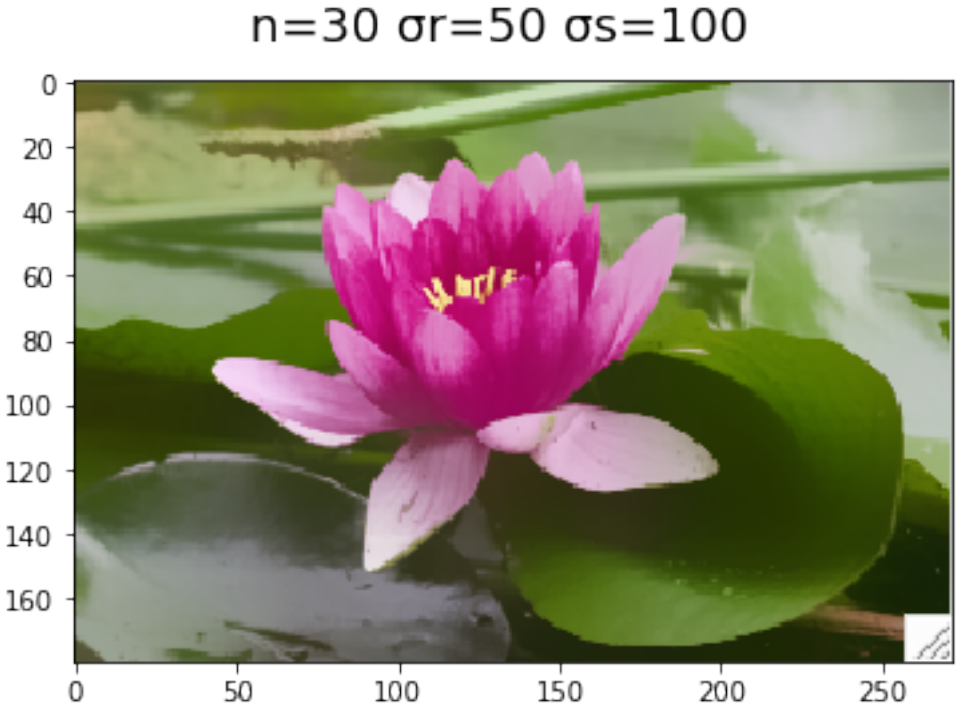
Imagen original
0
20
40
60
80
100
120
140
160
0
50
100
150
200
250


n=9 σr=100 σs=10
0
20
40
60
80
100
120
140
160
0
50
100
150
200
250


n=20 σr=10 σs=100
0
20
40
60
80
100
120
140
160
0
50
100
150
200
250


n=30 σr=10 σs=100
0
20
40
60
80
100
120
140
160
0
50
100
150
200
250


n=20 σr=50 σs=100
0
20
40
60
80
100
120
140
160
0
50
100
150
200
250


n=30 σr=50 σs=100
0
20
40
60
80
100
120
140
160
0
50
100
150
200
250

1.6.9 Cuestiones

Respondiendo a las preguntas que se presentaron más arriba: - £Cómo se comporta el filtro bilateral cuando la varianza es muy alta? £En este caso qué ocurre si es alta o baja?

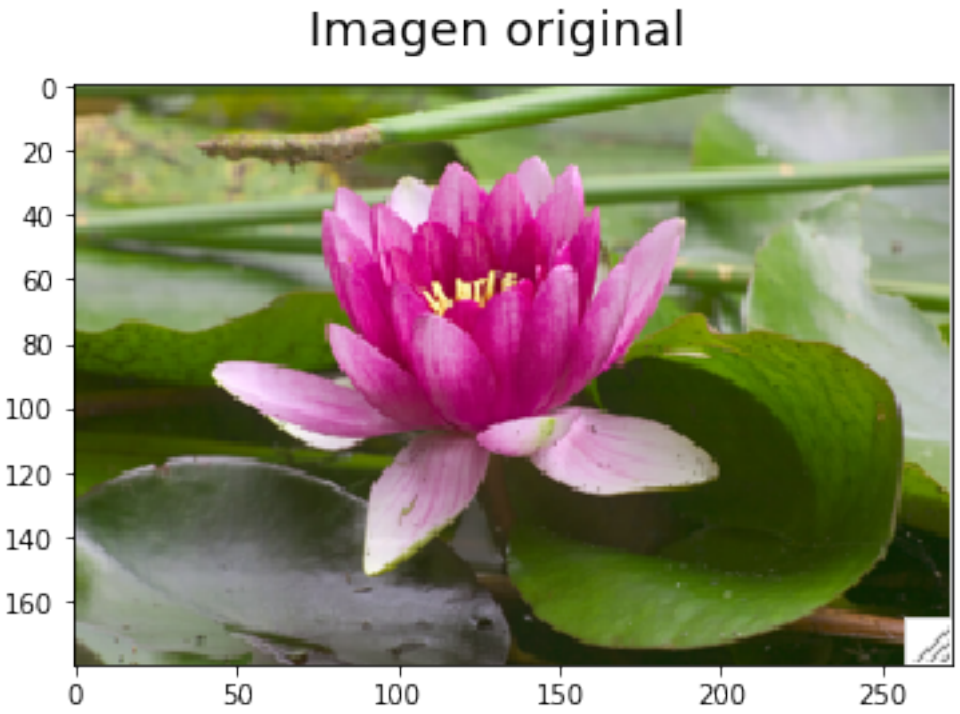
- £Cómo se comporta si es muy baja? £En este caso cómo se comporta el filtro dependiendo si es alta o baja?

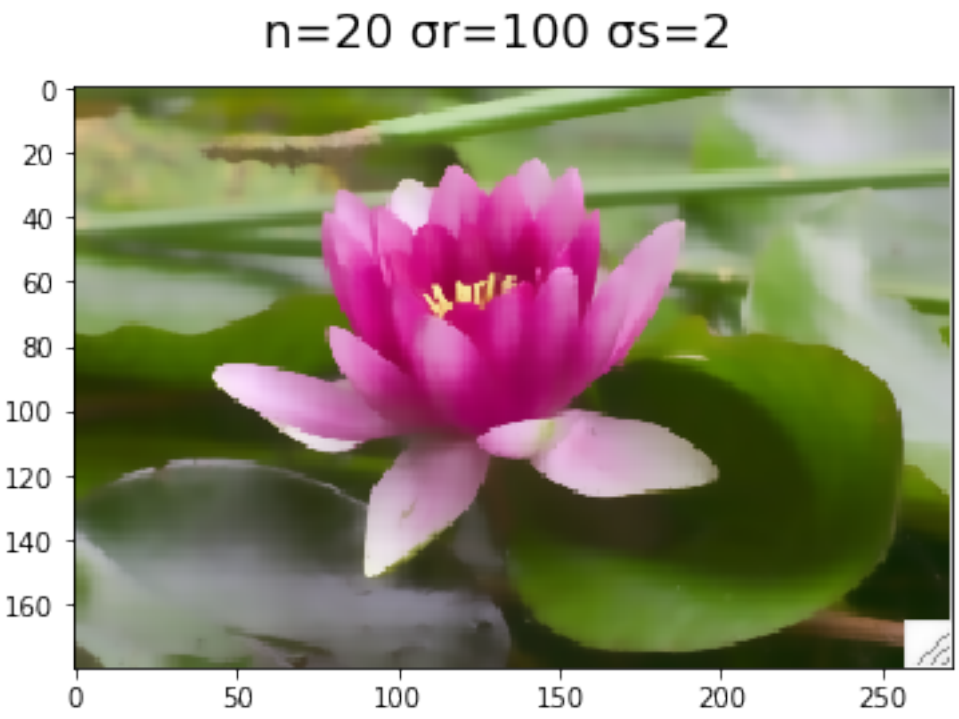
Para responder esta pregunta hay que entender los parámetros del filtro bilateral: - A medida que sigma sub-r se hace más grande el filtro bilateral se comporta cada vez más como un filtro gaussiano. - A medida que sigma sub-s se hace más grande el filtro bilateral suaviza estructuras cada vez más pequeñas. Con un sigma sub-s pequeño estas estructuras y bordes pequeños son protegidos en el suavizado.

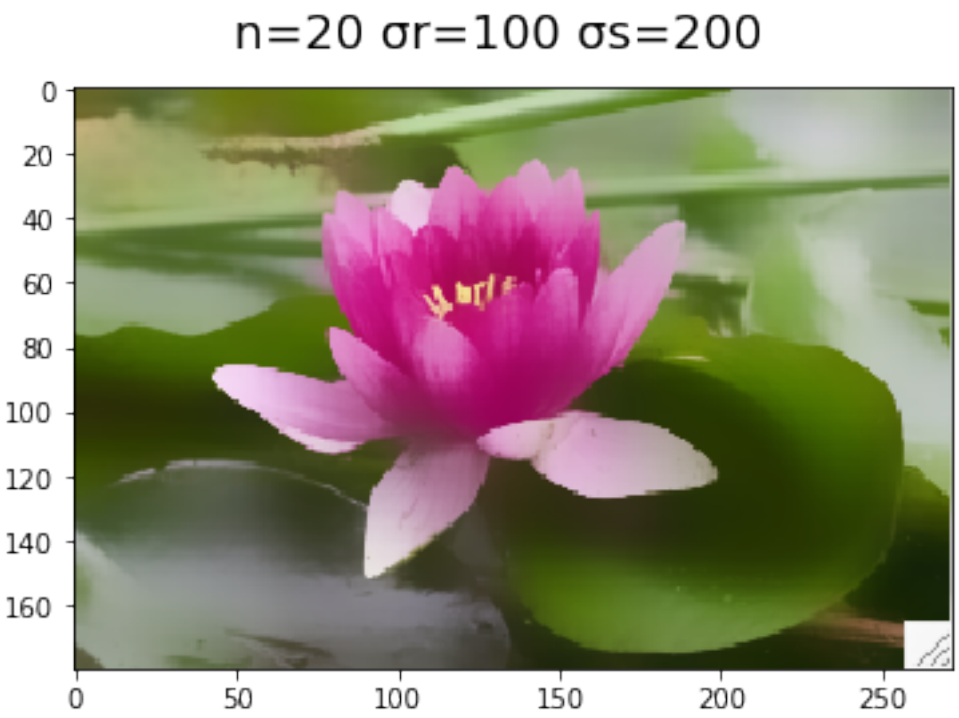
Probémoslo con unos ejemplos:

```
In [15]: image = cv2.imread("imagenes/flower.png")
         fig = plt.figure()
         fig.suptitle('Imagen original', fontsize=18)
         plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))

         fig = plt.figure()
         fig.suptitle('n=20 r=100 s=2', fontsize=18)
         plt.imshow(cv2.cvtColor(cv2.bilateralFilter(image,20,100,2), cv2.COLOR_BGR2RGB))

         fig = plt.figure()
         fig.suptitle('n=20 r=100 s=200', fontsize=18)
         plt.imshow(cv2.cvtColor(cv2.bilateralFilter(image,20,100,200), cv2.COLOR_BGR2RGB))
```

```
Out[15]: <matplotlib.image.AxesImage at 0x7fcd789954a8>
```




n=20 σr=100 σs=2
0
20
40
60
80
100
120
140
160
0
50
100
150
200
250


n=20 σr=100 σs=200
0
20
40
60
80
100
120
140
160
0
50
100
150
200
250

1.6.10 Resultado

Para entender el filtro comparemos la segunda y tercera imagen fijandonos sobretodo en la hoja verde de la esquina inferior derecha. Como se puede observar:

- Al tener un sigma sub-s alto (sigma sub-s = 200, tercera imagen) se ignoran los bordes de las estructuras más pequeñas, como se puede observar en la hoja verde en al tercera imagen.
- Al tener un sigma sub-s bajo (sigma sub-s = 2, segunda imagen) se protegen los bordes de las estructuras más pequeñas a la hora de suavizar la imagen. Esto se puede comprobar fijandose en la hoja verde en la segunda imagen y comparandola con la de la tercera.

Comparando la segunda imagen con la tercera podemos ver que se han protegido los bordes mucho mejor con un sigma sub-s bajo.

El tener un sigma sub-r alto afecta al nivel de suavizado de la imagen. Esto lo podremos ver en el siguiente fragmento de código:

```
In [16]: image = cv2.imread("imagenes/flower.png")
         fig = plt.figure()
         fig.suptitle('Imagen original', fontsize=18)
         plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))

         fig = plt.figure()
         fig.suptitle('n=20 r=10 s=200', fontsize=18)
         plt.imshow(cv2.cvtColor(cv2.bilateralFilter(image,20,10,200), cv2.COLOR_BGR2RGB))

         fig = plt.figure()
         fig.suptitle('n=20 r=10 s=2', fontsize=18)
         plt.imshow(cv2.cvtColor(cv2.bilateralFilter(image,20,10,2), cv2.COLOR_BGR2RGB))

Out[16]: <matplotlib.image.AxesImage at 0x7fcd73e98e10>
```

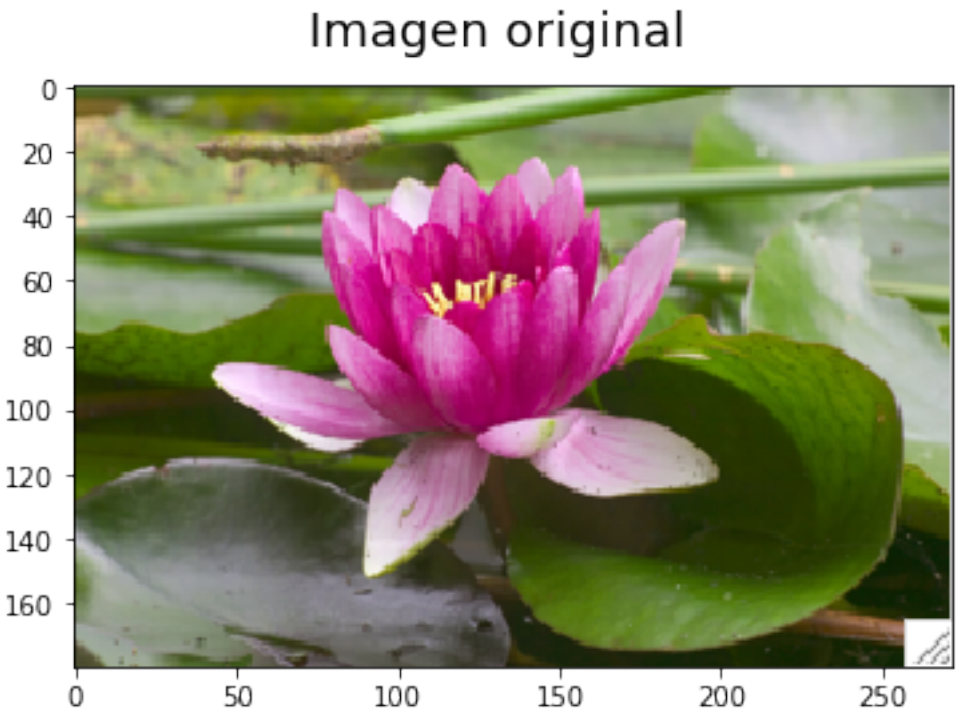
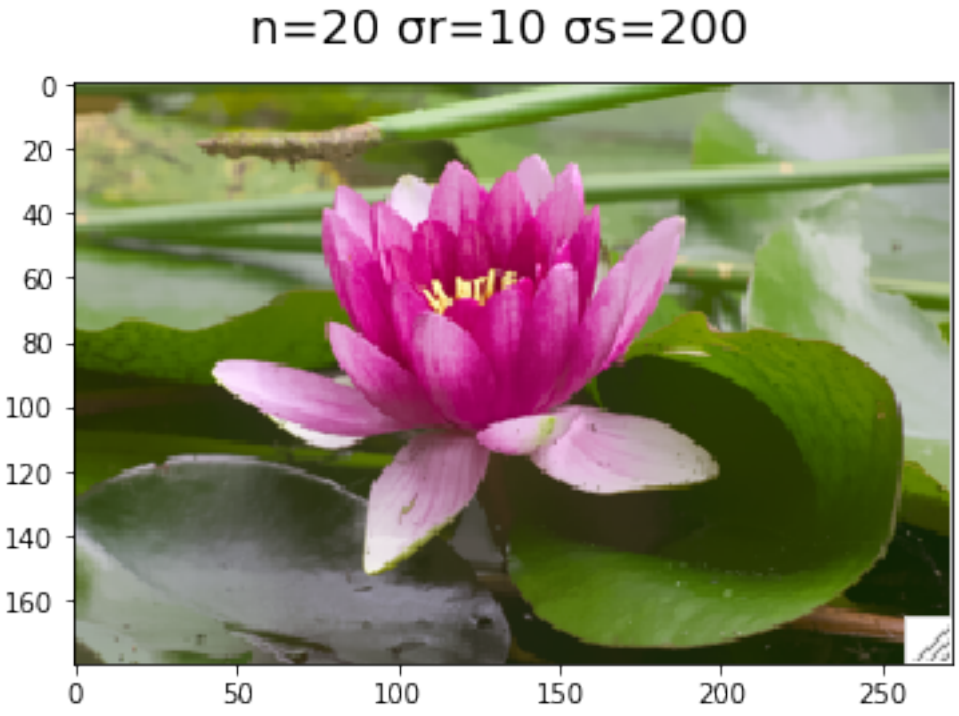
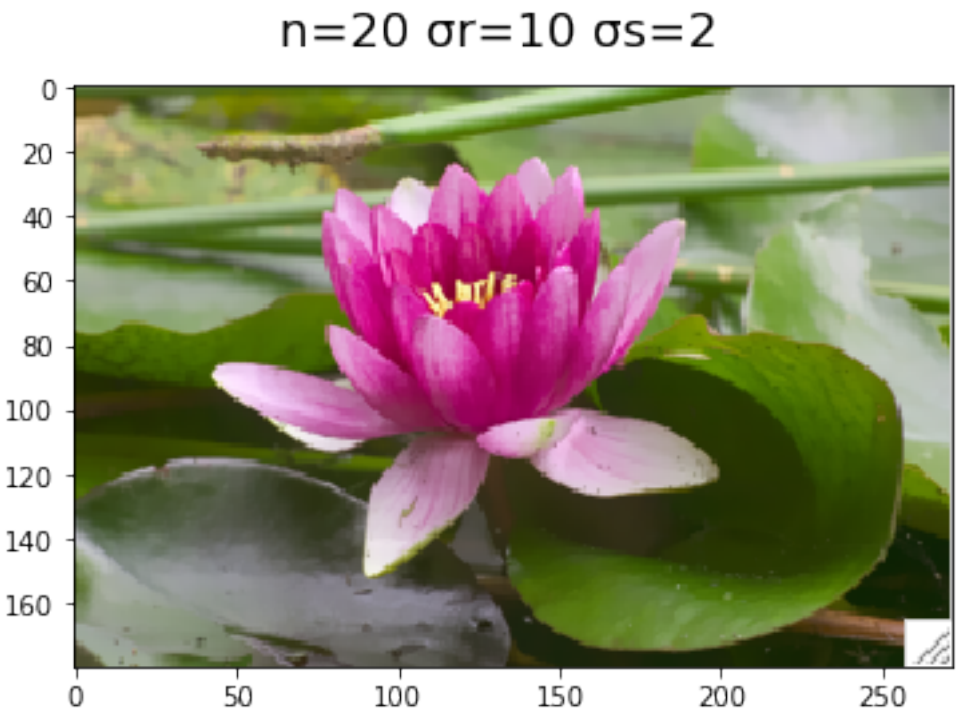


Imagen original
0
20
40
60
80
100
120
140
160
0
50
100
150
200
250


n=20 σr=10 σs=200
0
20
40
60
80
100
120
140
160
0
50
100
150
200
250



1.6.11 Resultado

Comparando los resultados de este fragmento de código con los resultados del anterior fragmento de código podemos ver que si sigma sub-r adquiere un valor alto la imagen se suaviza con más intensidad. Cuando sigma sub-r adquiere un valor muy alto esta empieza a comportarse como una mascara gaussiana, pero cuando adquiere un valor bajo el filtro no suaviza casi la imagen.

Por otro lado, en este ejemplo con un sigma sub-r bajo podemos ver que aunque sigma sub-s adquiera valores muy altos o muy bajos el resultado sigue siendo el mismo; al no suavizar casi la imagen, los bordes de la imagen no son alterados y la imagen resultado es muy similar a la imagen inicial. Podemos ver que con un sigma sub-r = 10 la sombra de la hoja inferior derecha ha sido suavizada, pero como ha sido un suavizado muy suave los bordes no han sido alterados y tanto en la imagen con sigma sub-s alta como en la de sigma sub-s baja los bordes siguen manteniendose.

1.7 Transformada Hough

Ejercicio 8. Emplea la transformada Hough para encontrar segmentos rectilíneos en la imagen `corridor.jpg`. Para extraer los bordes de la imagen utiliza las funciones escritas en los ejercicios 3 y 4. Utiliza la función `cv2.HoughLinesP()` de OpenCV.

Discute 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 (mira como ejemplo, https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_houghlines/py_houghlines.html).

In [17]:
```
def masc_2deriv_gaus_1d(sigma, n, order):
    width = n//2
    dx = 1
    x = np.arange(-width, width)
    kernel_1d = (sigma ** 2 - x ** 2) * np.exp(-(x ** 2) / (2 * sigma ** 2))
    kernel_1d = -kernel_1d / (math.sqrt(2 * np.pi) * sigma ** 5)

    return kernel_1d

def masc_2deriv_gaus_2d_x_or_y(sigma, n, orientation):
    width = n//2
    dx = 1
    dy = 1
    x = np.arange(-width, width)
    y = np.arange(-width, width)
    x2d, y2d = np.meshgrid(x, y)

    kernel_2d = np.exp(-(x2d ** 2 + y2d ** 2) / (2 * sigma ** 4)) / (2 * np.pi * sigma

    # Caso derivada en función de x
    if orientation == 1:
        kernel_2d = (-1 + (x2d ** 2 / sigma ** 2)) * kernel_2d
    # Caso derivada en función de y
    else:
        kernel_2d = (-1 + (y2d ** 2 / sigma ** 2)) * kernel_2d

    return kernel_2d

def masc_2deriv_gaus_2d_xy(sigma, n):
    width = n//2
    dx = 1
    dy = 1
    x = np.arange(-width, width)
    y = np.arange(-width, width)
    x2d, y2d = np.meshgrid(x, y)

    kernel_2d = np.exp(-(x2d ** 2 + y2d ** 2) / (2 * sigma ** 2))
    kernel_2d = (x2d * y2d / (2 * np.pi * sigma ** 6)) * kernel_2d

    return kernel_2d
```

In [18]:
```
from scipy.ndimage.filters import convolve, convolve1d, gaussian_filter

sigma = 1.08
n = 35

image = cv2.cvtColor(cv2.imread('imagenes/corridor.jpg'), cv2.COLOR_BGR2RGB)
fig = plt.figure()
```

```
fig.suptitle('Imagen original', fontsize=18)
plt.imshow(image)

gray = cv2.cvtColor(image,cv2.COLOR_RGB2GRAY)
fig = plt.figure()
fig.suptitle('Imagen escala de grises', fontsize=18)
plt.imshow(gray, cmap = plt.cm.gray)

kernel2D_x = masc_2deriv_gaus_2d_xy(sigma, n)
edges = convolve(gray, kernel2D_x)
fig = plt.figure()
fig.suptitle('Imagen bordes', fontsize=18)
plt.imshow(edges, cmap = plt.cm.gray)

lines = cv2.HoughLinesP(edges,rho = 1,theta = 1*np.pi/180,threshold = 100,minLineLeng
for line in lines:
    for x1,y1,x2,y2 in line:
        cv2.line(image,(x1,y1),(x2,y2),(0,255,0),2)

cv2.imwrite('imagenes/corridor_hough.jpg',image)
fig = plt.figure()
fig.suptitle('Transformada de Hough', fontsize=18)
plt.imshow(cv2.imread('imagenes/corridor_hough.jpg'))
```

Out[18]: <matplotlib.image.AxesImage at 0x7fcd789409b0>

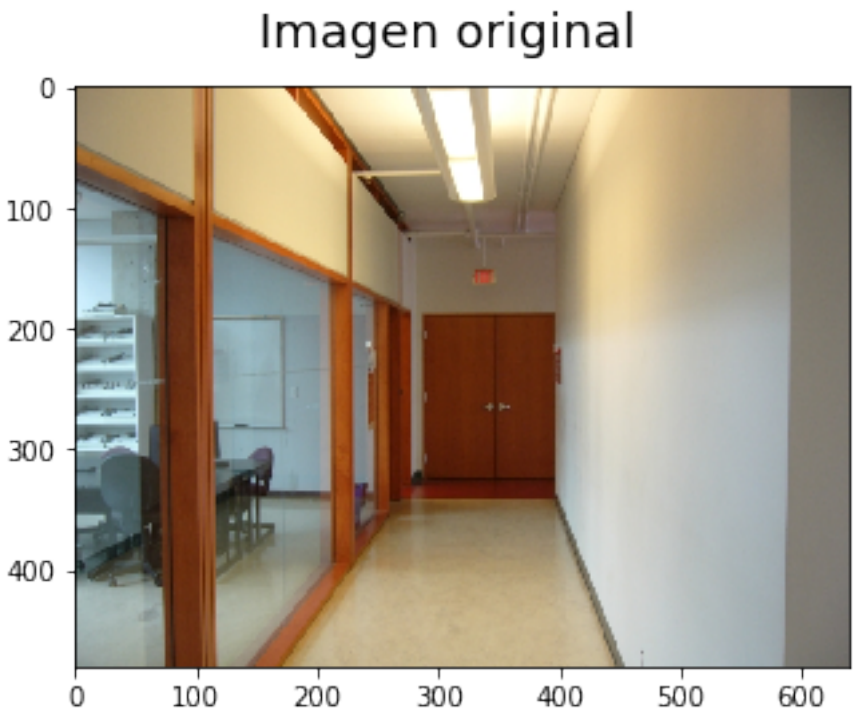



Imagen escala de grises
0
100
200
300
400
0
100
200
300
400
500
600

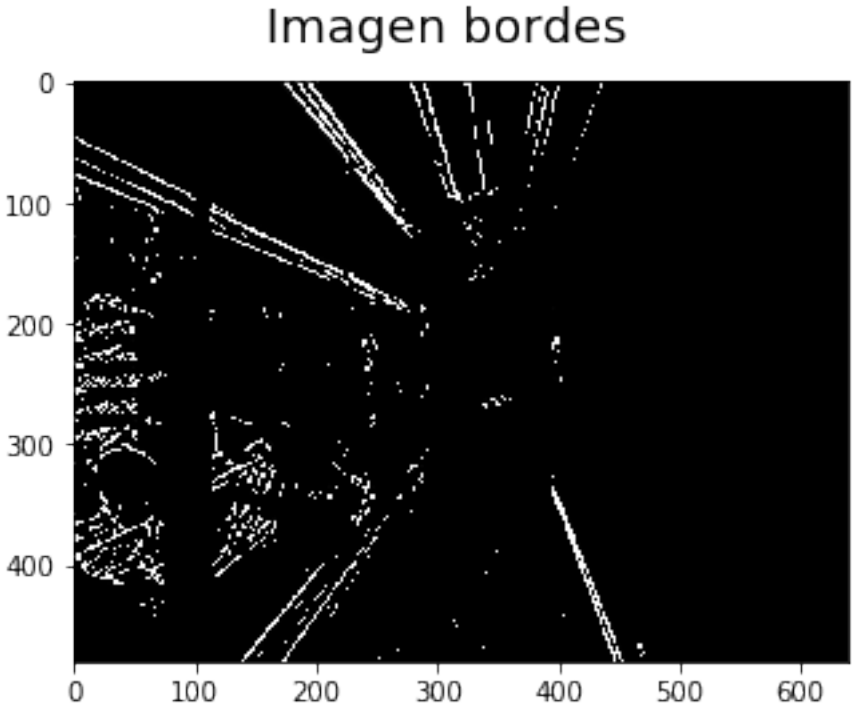
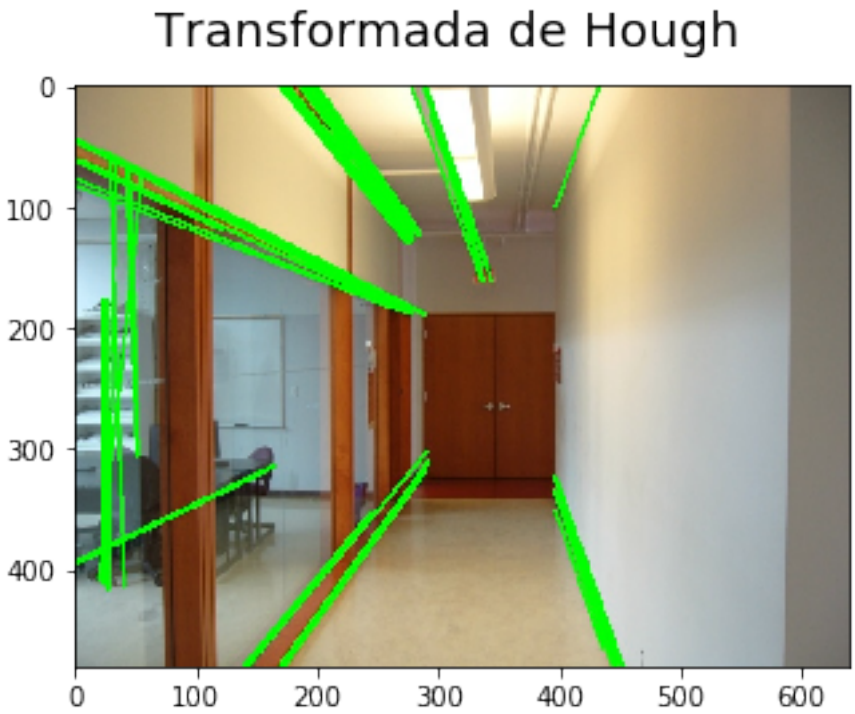


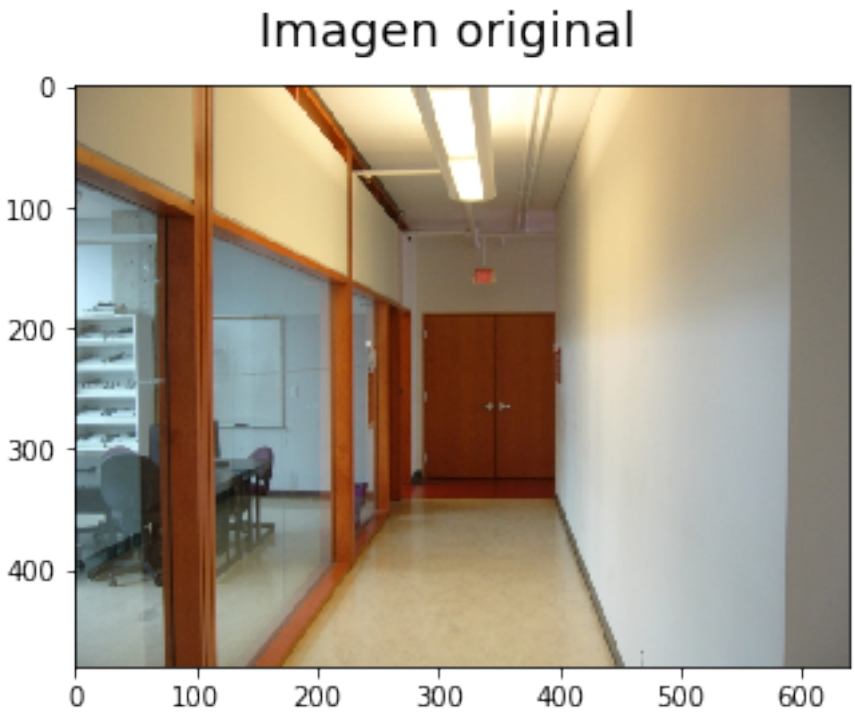
Imagen bordes
0
100
200
300
400
0
100
200
300
400
500
600



In [19]:
```
from scipy.ndimage.filters import convolve, convolve1d, gaussian_filter

sigma = 2.08
n = 35

image = cv2.cvtColor(cv2.imread('imagenes/corridor.jpg'), cv2.COLOR_BGR2RGB)
fig = plt.figure()
fig.suptitle('Imagen original', fontsize=18)
plt.imshow(image)

gray = cv2.cvtColor(image,cv2.COLOR_RGB2GRAY)

kernel2D_x = masc_2deriv_gaus_2d_xy(sigma, n)
edges = convolve(gray, kernel2D_x)
fig = plt.figure()
fig.suptitle('Imagen bordes', fontsize=18)
plt.imshow(edges, cmap = plt.cm.gray)

lines = cv2.HoughLinesP(edges,rho = 1,theta = 1*np.pi/180,threshold = 100,minLineLeng
for line in lines:
    for x1,y1,x2,y2 in line:
        cv2.line(image,(x1,y1),(x2,y2),(0,255,0),2)

cv2.imwrite('imagenes/corridor_hough.jpg',image)
```

```
fig = plt.figure()
fig.suptitle('Transformada de Hough', fontsize=18)
plt.imshow(cv2.imread('imagenes/corridor_hough.jpg'))
```

```
Out[19]: <matplotlib.image.AxesImage at 0x7fcd73d7fc18>
```



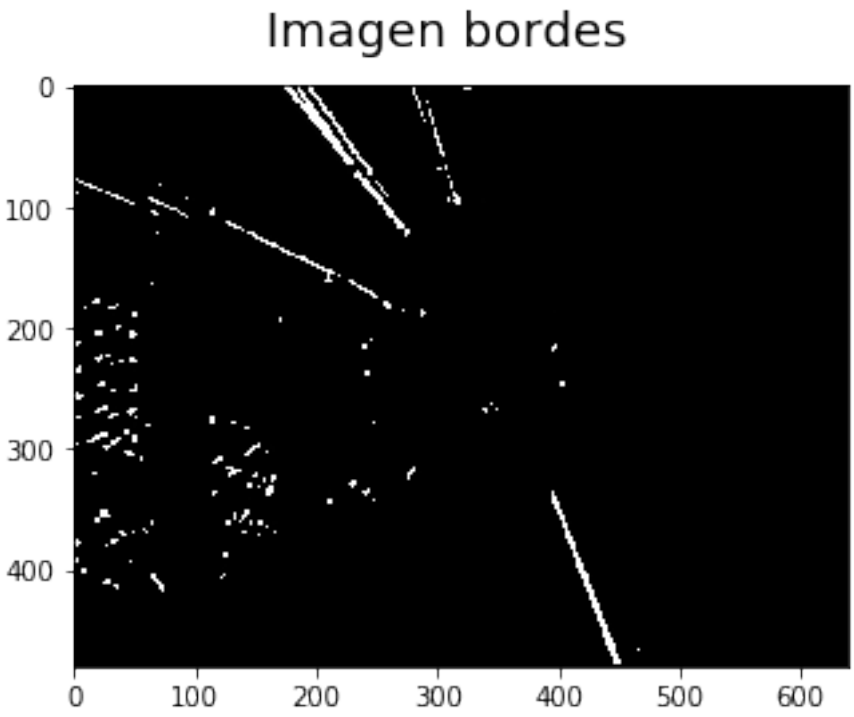
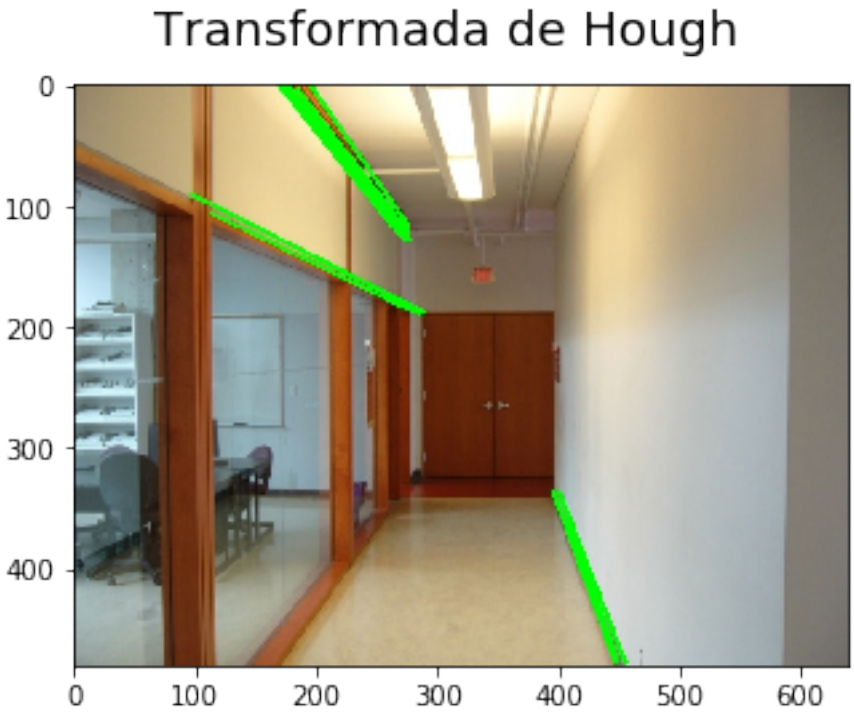


Imagen bordes
0
100
200
300
400
0
100
200
300
400
500
600


Transformada de Hough
0
100
200
300
400
0
100
200
300
400
500
600

In [20]:
```
from scipy.ndimage.filters import convolve, convolve1d, gaussian_filter

sigma = 0.5
n = 10

image = cv2.cvtColor(cv2.imread('imagenes/corridor.jpg'), cv2.COLOR_BGR2RGB)
fig = plt.figure()
fig.suptitle('Imagen original', fontsize=18)
plt.imshow(image)

gray = cv2.cvtColor(image,cv2.COLOR_RGB2GRAY)
kernel2D_x = masc_2deriv_gaus_2d_xy(sigma, n)
edges = convolve(gray, kernel2D_x)
fig = plt.figure()
fig.suptitle('Imagen bordes', fontsize=18)
plt.imshow(edges, cmap = plt.cm.gray)

lines = cv2.HoughLinesP(edges,rho = 1,theta = 1*np.pi/180,threshold = 100,minLineLengt
for line in lines:
    for x1,y1,x2,y2 in line:
        cv2.line(image,(x1,y1),(x2,y2),(0,255,0),2)

cv2.imwrite('imagenes/corridor_hough.jpg',image)
fig = plt.figure()
fig.suptitle('Transformada de Hough', fontsize=18)
plt.imshow(cv2.imread('imagenes/corridor_hough.jpg'))
```

Out[20]:
```
<matplotlib.image.AxesImage at 0x7fcd73d9aba8>
```

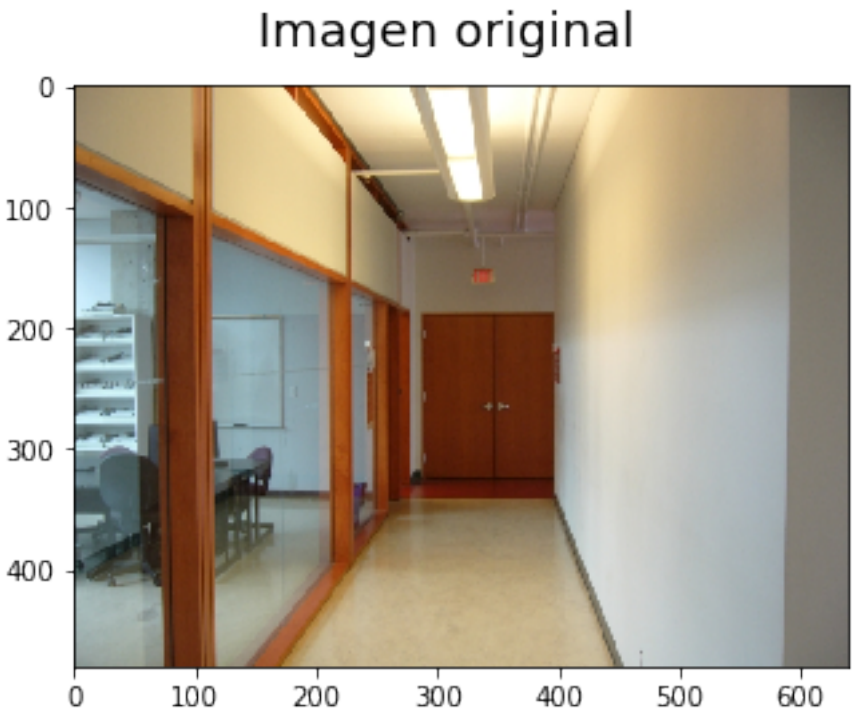


Imagen original
0
100
200
300
400
0
100
200
300
400
500
600

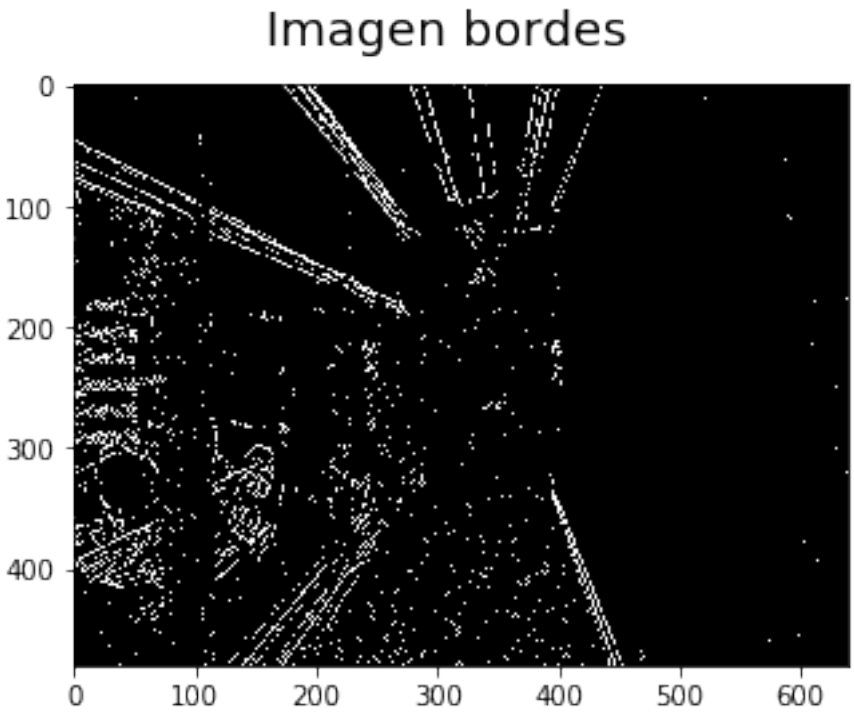
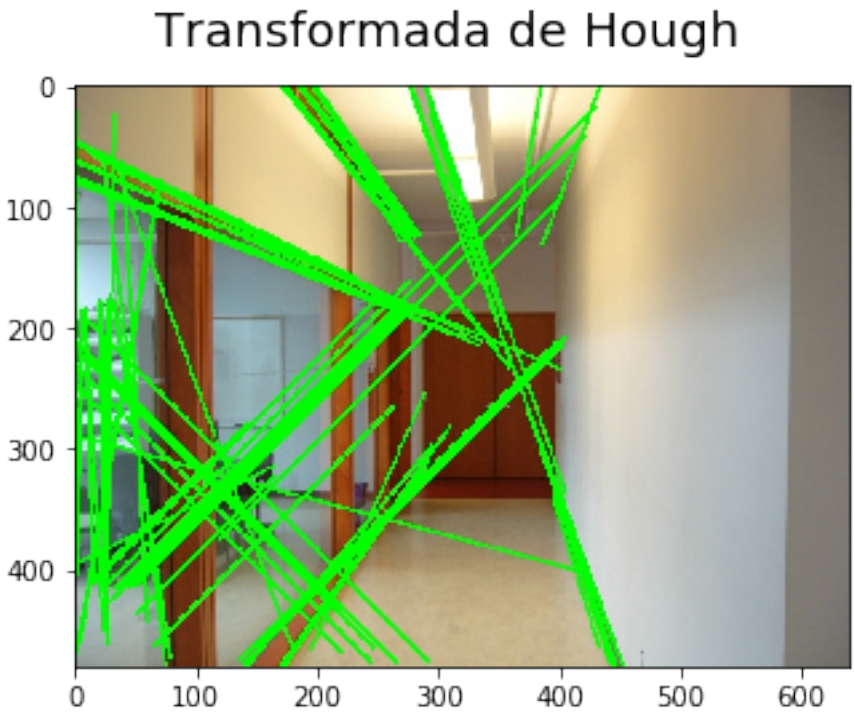


Imagen bordes
0
100
200
300
400
0
100
200
300
400
500
600



1.7.1 Resultado

Jugando con los distintos parámetros conseguimos tener unos efectos u otros; Con un sigma bajo conseguimos más bordes, mientras que con un sigma alto encontraremos bordes más escogidos y claros.

1.8 Segmentación

Ejercicio 9. Escribe una función que segmente el objeto central de una imagen a partir de una segmentación manual inicial realizada por el usuario. Puedes utilizar el código proporcionado en el archivo segm.py. En la optimización 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 las imágenes `persona.png` y `horse.jpg`. Muestra y discute los resultados.

```
In [21]: ########################################################################
         # 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 funcion here is "select_fg_bg" read documentation below
         ########################################################################
         import pygame
```

```
import numpy as np
import cv2
import sys

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().
            pygame.quit()

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

```
pygame 1.9.4
Hello from the pygame community. https://www.pygame.org/contribute.html
```

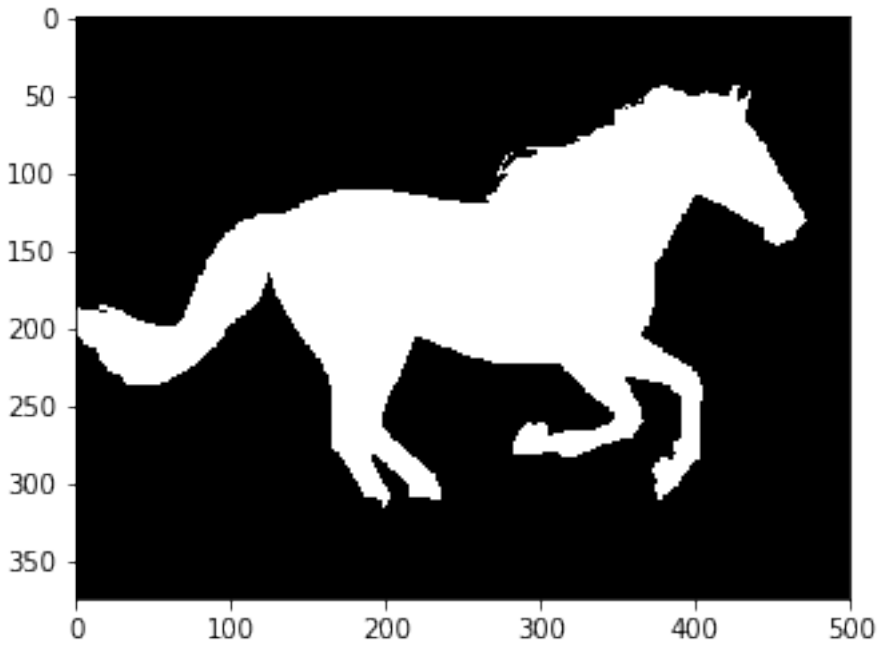
In [26]:
```
# ##################################
# Segmentacion de imagen a la "Grab Cut" simplificado
# por Luis Baumela. UPM. 15-10-2015
# Vision por Computador. Master en Inteligencia Artificial
# ##################################


import numpy as np
from scipy.misc import imread
import math
import maxflow
import matplotlib.pyplot as plt
from pprint import pprint as pp

imgName='imagenes/horse.jpg'
sigma = 0.7

img = 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])
pp(nodeids)

# Calcula los costes de los nodos no terminales del grafo
```

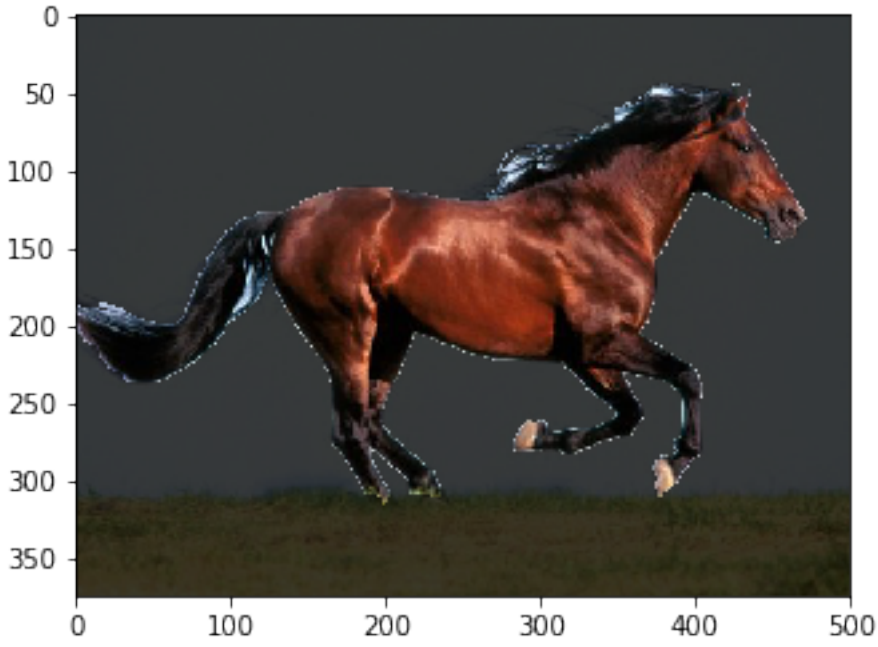
```
h, w =img.shape[:2]
def bpq(ip, iq, sigma):
    return math.exp(-((np.linalg.norm(ip, 2)-np.linalg.norm(iq, 2))**2)/2*sigma**2)

exp_aff_h = np.ones(img.shape[:2])
exp_aff_v = np.ones(img.shape[:2])

# Estos son los costes de los vecinos horizontales
for i in range(0, h):
    for j in range(1, w):
        exp_aff_h[i, j] = bpq(img[i][j], img[i][j-1], sigma)

# Estos son los costes de los vecinos verticales
for i in range(1, h):
    for j in range(0, w):
        exp_aff_v[i, j] = bpq(img[i][j], img[i-1][j], sigma)

# 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, structure=hor_struc,symmetric=True)
g.add_grid_edges(nodeids, exp_aff_v, structure=ver_struc,symmetric=True)

# Leo los pixeles 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
fg = np.ones(img.shape[:2])
bg = np.ones(img.shape[:2])
fg = fg * 55
bg = bg * 55

for fg_p in pts_fg:
    fg[fg_p[0]][fg_p[1]] = np.inf
    bg[fg_p[0]][fg_p[1]] = 0

for bg_p in pts_bg:
```

```
            fg[bg_p[0]][bg_p[1]] = 0
            bg[bg_p[0]][bg_p[1]] = np.inf

        # Pesos de los nodos terminales

        g.add_grid_tedges(nodeids, fg, bg)

        # 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 = plt.cm.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)
        plt.imshow(np.uint8(np.multiply(img,wgs)))
        plt.show()
```

```
/home/vision/.local/lib/python3.6/site-packages/ipykernel_launcher.py:18: DeprecationWarning:
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
```

```
array([[     0,      1,      2, ...,    497,    498,    499],
       [   500,    501,    502, ...,    997,    998,    999],
       [  1000,   1001,   1002, ...,   1497,   1498,   1499],
       ...,
       [186000, 186001, 186002, ..., 186497, 186498, 186499],
       [186500, 186501, 186502, ..., 186997, 186998, 186999],
       [187000, 187001, 187002, ..., 187497, 187498, 187499]])
```





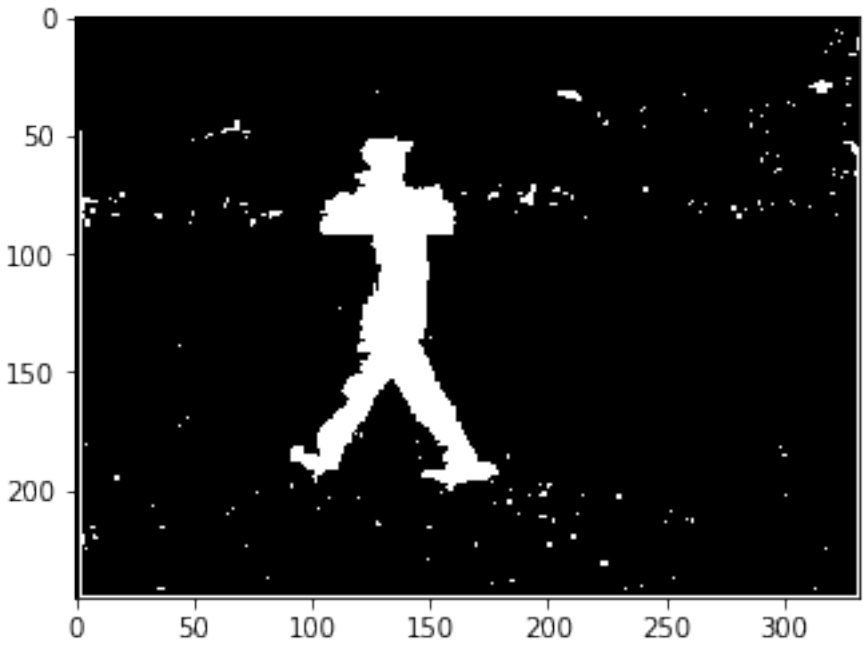

```
In [27]: imgName='imagenes/persona.png'
         sigma = 0.7
```

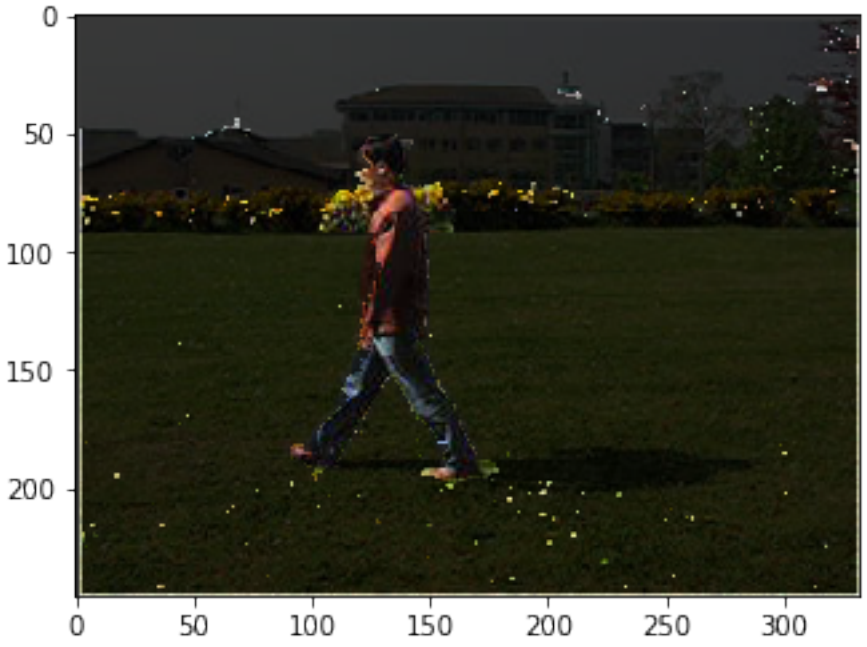
```
img = imread(imgName)

# 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])
pp(nodeids)

# Calcula los costes de los nodos no terminales del grafo
h, w =img.shape[:2]
def bpq(ip, iq, sigma):
    return math.exp(-((np.linalg.norm(ip, 2)-np.linalg.norm(iq, 2))**2)/2*sigma**2)

exp_aff_h = np.ones(img.shape[:2])
exp_aff_v = np.ones(img.shape[:2])

# Estos son los costes de los vecinos horizontales
for i in range(0, h):
    for j in range(1, w):
        exp_aff_h[i, j] = bpq(img[i][j], img[i][j-1], sigma)

# Estos son los costes de los vecinos verticales
for i in range(1, h):
    for j in range(0, w):
        exp_aff_v[i, j] = bpq(img[i][j], img[i-1][j], sigma)

# 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, structure=hor_struc,symmetric=True)
g.add_grid_edges(nodeids, exp_aff_v, structure=ver_struc,symmetric=True)

# Leo los pixeles 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
        fg = np.ones(img.shape[:2])
        bg = np.ones(img.shape[:2])
        fg = fg * 55
        bg = bg * 55

        for fg_p in pts_fg:
            fg[fg_p[0]][fg_p[1]] = np.inf
            bg[fg_p[0]][fg_p[1]] = 0

        for bg_p in pts_bg:
            fg[bg_p[0]][bg_p[1]] = 0
            bg[bg_p[0]][bg_p[1]] = np.inf

        # Pesos de los nodos terminales

        g.add_grid_tedges(nodeids, fg, bg)

        # 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 = plt.cm.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)
        plt.imshow(np.uint8(np.multiply(img,wgs)))
        plt.show()
```

```
/home/vision/.local/lib/python3.6/site-packages/ipykernel_launcher.py:4: DeprecationWarning: `i
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  after removing the cwd from sys.path.


array([[   0,    1,    2, ...,  329,  330,  331],
       [ 332,  333,  334, ...,  661,  662,  663],
       [ 664,  665,  666, ...,  993,  994,  995],
```

```
...,
[80676, 80677, 80678, ..., 81005, 81006, 81007],
[81008, 81009, 81010, ..., 81337, 81338, 81339],
[81340, 81341, 81342, ..., 81669, 81670, 81671]])
```





1.8.1 Resultado

Se puede ver que horse.jpg está perfectamente segmentado y su separación con respecto al fondo de la imagen se realiza correctamente.

En el caso de persona.png es más dificil segmentar la imagen debido a la grán cantidad de colores y contrastes en ella. No obstante hemos llegado a un buen resultado.

Fórmula para el cálculo del peso entre puntos: Lecturas/GraphCuts.pdf

Fórmula para el cálculo del valor de un punto: http://mathworld.wolfram.com/L2-Norm.html

https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.linalg.norm.html

Ejercicio 10. Mejora el algoritmo anterior. Puedes utilizar algunas de las que te sugiero a continuación u otras que creas más convenientes: * Refina la segmentación iterativamente. * Mejora la función de afinidad entre píxeles. * Mejora los términos unitarios

mejora los resultados de algunas de las imágenes anteriores. Muestra y discute los resultados.

1.8.2 Resultado

Para mejorar la segmentación hemos mejorado la función bpq

```
In [28]: def bpq(ip, iq, sigma):
             return math.exp(-((np.linalg.norm(ip, 2)-np.linalg.norm(iq, 2))**2)/2*sigma**2)
```

Esta función mejora el calculo del coste de energía para pasar de un nodo a su vecino. Esta mejora ya se ha incorporado en el ejercicio anterior.