PracticaPI2018

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
```

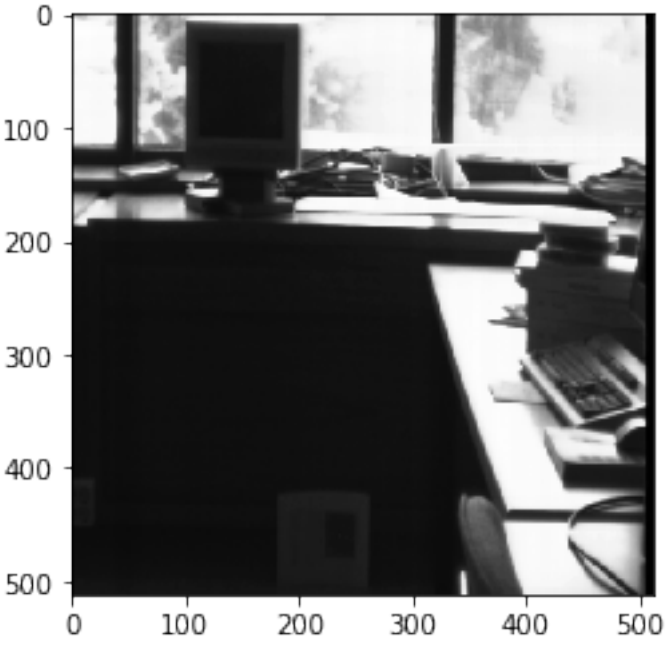
1.5 Transformaciones puntuales

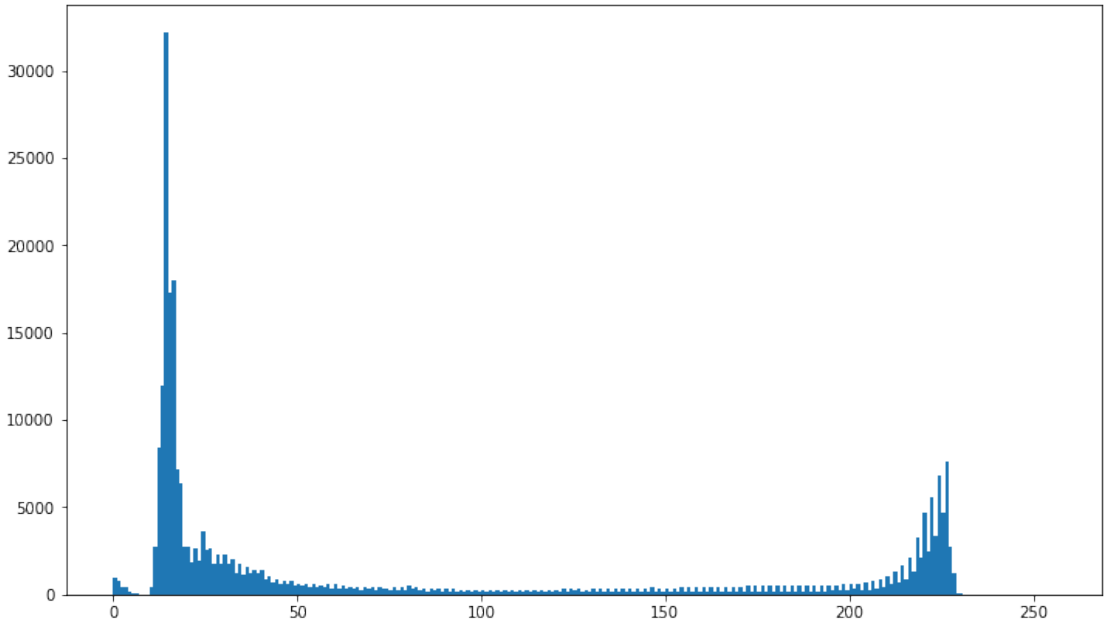
En este apartado te recomiendo que uses al menos la imagen indicada, que puedes encontrar en el directorio de imágenes del aula virtual. También puedes probar con otras que te parezcan interesantes.

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 [10]: from scipy.ndimage.filters import convolve, convolve1d

In [1]: import numpy as np
        import matplotlib.pyplot as plt

        %matplotlib inline

In [2]: imgName='images/escilum.tif'

        img = plt.imread(imgName)

        plt.imshow(img, cmap='gray')

        bins = list(range(0,256+1))
```

```
plt.figure(figsize=(12,7))

histogram = plt.hist(img.flatten(), bins=bins)

plt.show()
```





El problema es que la imagen tiene una distribución de los niveles de gris muy mala. Los niveles de gris de los píxeles de la imagen están concentrados principalmente en dos rangos muy pequeños: un rango oscuro con niveles de gris entre 10 y 30 y un rango claro con niveles de gris entre 210 y 230. En concreto la esquina inferior es una zona oscura, y aunque hay hay partes diferenciables con distintos niveles de gris, al estar esos niveles en un rango tan pequeño no hay contraste suficiente para distinguirlas a simple vista y sin esfuerzo.

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.

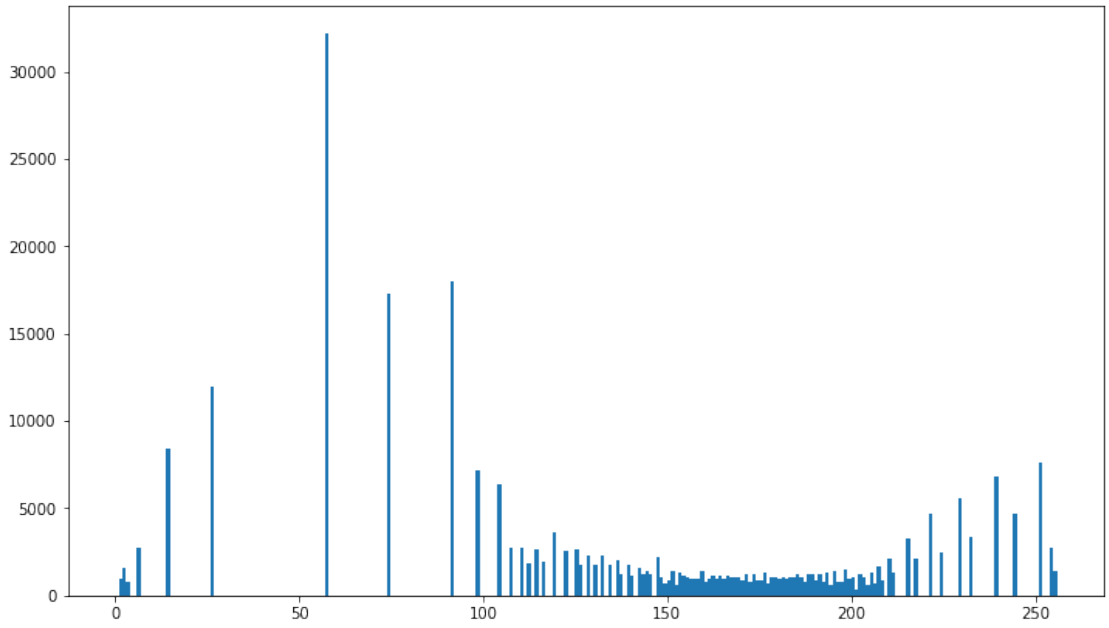
Discute los resultados obtenidos. £Cuál sería el resultado si volviésemos a ecualizar la imagen resultante?

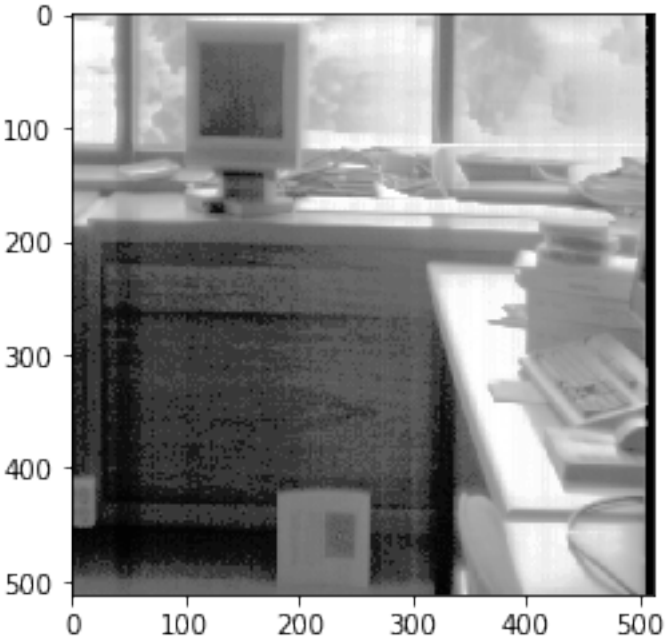
En este ejercicio tienes que implementar la función que ecualiza el histograma. No puedes usar funciones que lo hagan por ti.

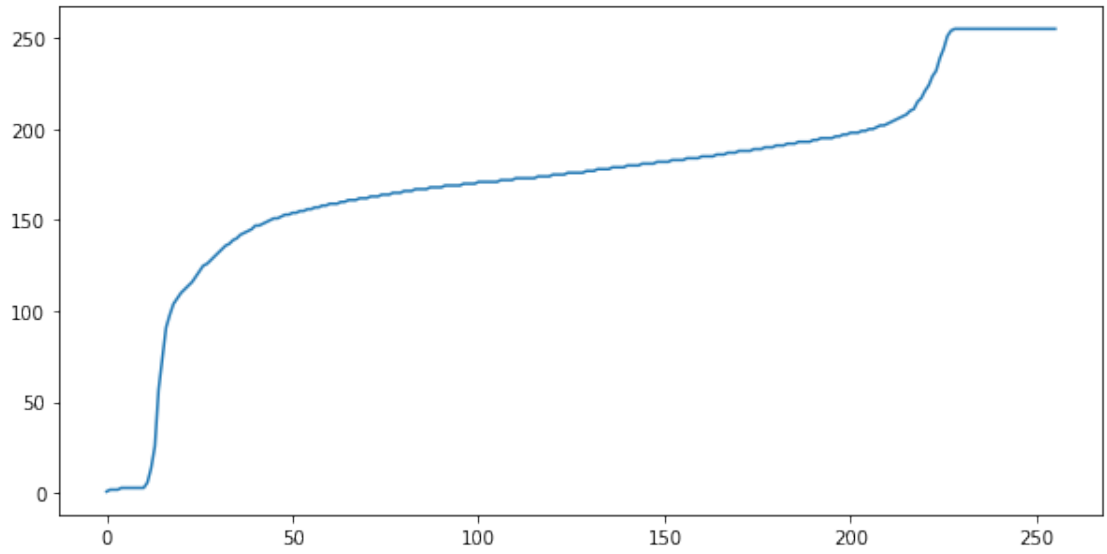
```
In [3]: def eq_hist(histogram):

            values, _, _ = histogram

            cs = np.cumsum(values)

            f = [int(round(255*cs[i]/cs[255])) for i in range(256)]

            return f
```

```
In [4]: def eq_img(img, histogram):

            func = eq_hist(histogram)

            equalize_img = np.vectorize(lambda i: func[int(i)])

            imgeq = equalize_img(img)

            plt.figure(figsize=(12,7))

            new_histogram = plt.hist(imgeq.flatten(), bins=bins)

            plt.show()

            plt.imshow(imgeq, cmap='gray')

            plt.figure(figsize=(10,5))

            plt.plot(func)

            return imgeq, new_histogram, func
```

```
In [5]: img2,hist2,func2 = eq_img(img, histogram)
```


30000
25000
20000
15000
10000
5000
0
0
50
100
150
200
250


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



Al ecualizar la imagen, la nueva distribución de los niveles de gris es más uniforme (lo más uniforme que puede ser manteniendo la monotonía de los niveles de gris y asegurándose que todos los píxeles con un mismo nivel de gris en la imagen original siguen teniendo un nivel de gris igual en la imagen ecualizada). Como resultado, ahora hay más contraste entre las partes de la imagen que antes eran complicadas de discernir por tener niveles de gris muy parecidos, y ahora se aprecian mejor a simple vista.

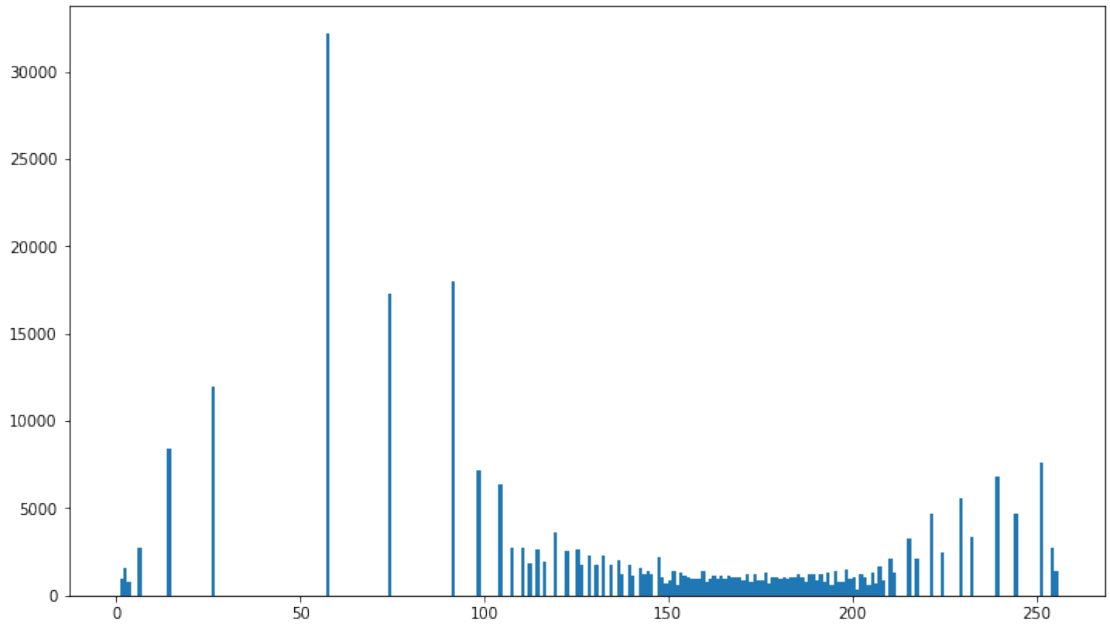
Esto sin embargo tiene dos efectos adversos. Por un lado se aprecia más el ruido de la imagen en esas zonas donde antes los niveles de gris se concentraban en un rango muy pequeño, como en la parte inferior izquierda o en la pantalla del ordenador. Y por otro, el rango de niveles de gris que había en la imagen original entre esos dos rangos muy oscuro y muy claro, se ha concentrado en un rango más pequeño, de forma que algunas partes de la imagen con niveles de gris intermedios que antes se apreciaban muy bien ahora se ven con menos claridad o contraste. Un ejemplo de estos son las sombras y reflejos en la mesa.

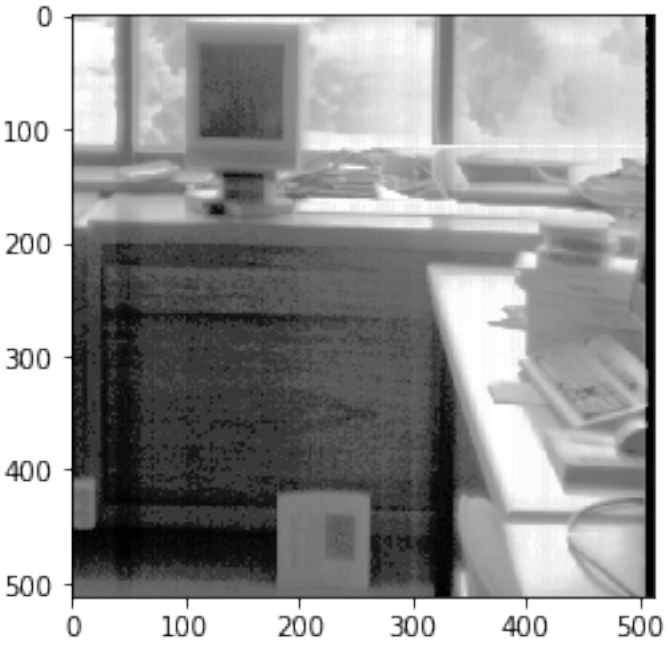
Si se vuelve a equalizar la imagen resultante, el resultado es el mismo. En el caso continuo está claro: la distribución resultante ya es uniforme, y además es fácil comprobar analíticamente que la función de ecualización es idempotente. En el caso discreto, podría no estar tan claro que el resultado es necesariamente el mismo, y podría pensarse que debido a las aproximaciones necesarias para discretizar podría ser ligeramente distinto (aunque siempre muy similar, y estabilizándose tras un par de ecualizaciones consecutivas). Sin embargo es fácil convencerse de que no, de que también es el mismo necesariamente.

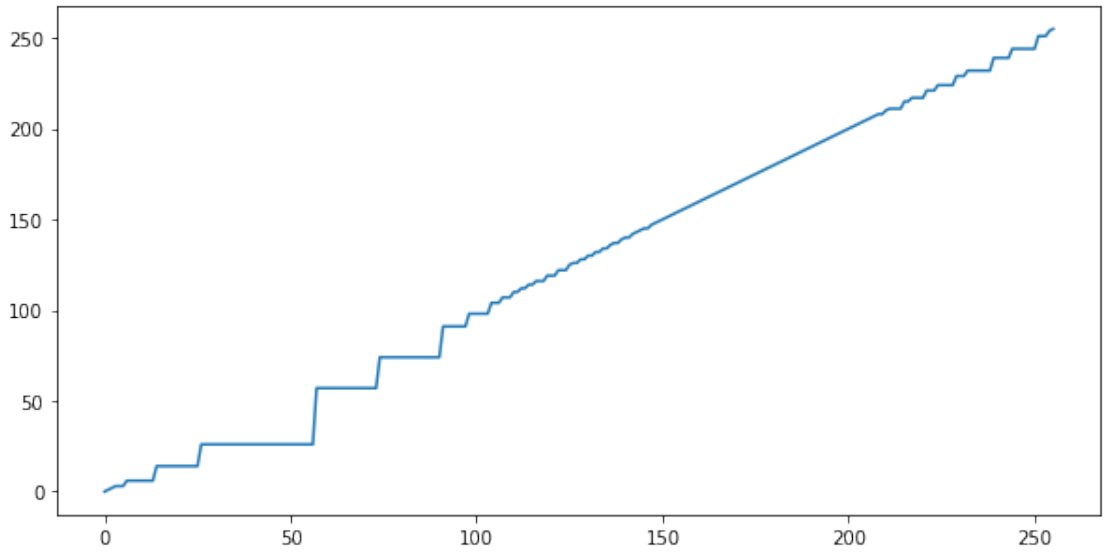
Consideremos por ejemplo un valor de gris k, que en el la imagen ecualizada le corresponden n_k píxeles. Habrá un máximo valor de gris j al que la función de transformación le asigna el valor k. Denotemos h_1 a la función que asigna a cada nivel de gris el número de píxeles con ese valor en la imagen original, h_2 a la función análoga en la imagen equalizada, f a la función de transformación entre las dos imágenes, y g a la función de transformación que ecualiza la nueva imagen, que queremos ver que es la identidad. Recordamos que f se define como $f(x) = 255 * \frac{\sum_{i=0}^{x} h_1(i)}{\sum_{i=0}^{255} h_1(i)}$, y g igual pero con h_2. Entonces está claro que f es monótona, que $\sum_{i=0}^{j} h_1(i) = \sum_{i=0}^{k} h_2(i)$, y que $\sum_{i=0}^{255} h_1(i) = \sum_{i=0}^{255} h_2(i)$. Pero entonces $g(k) = f(j) = k$, y vemos que g es en efecto la identidad.

Veámoslo en la imagen en cuestión:

```
In [6]: img3,hist3,func3 = eq_img(img2, hist2)
```





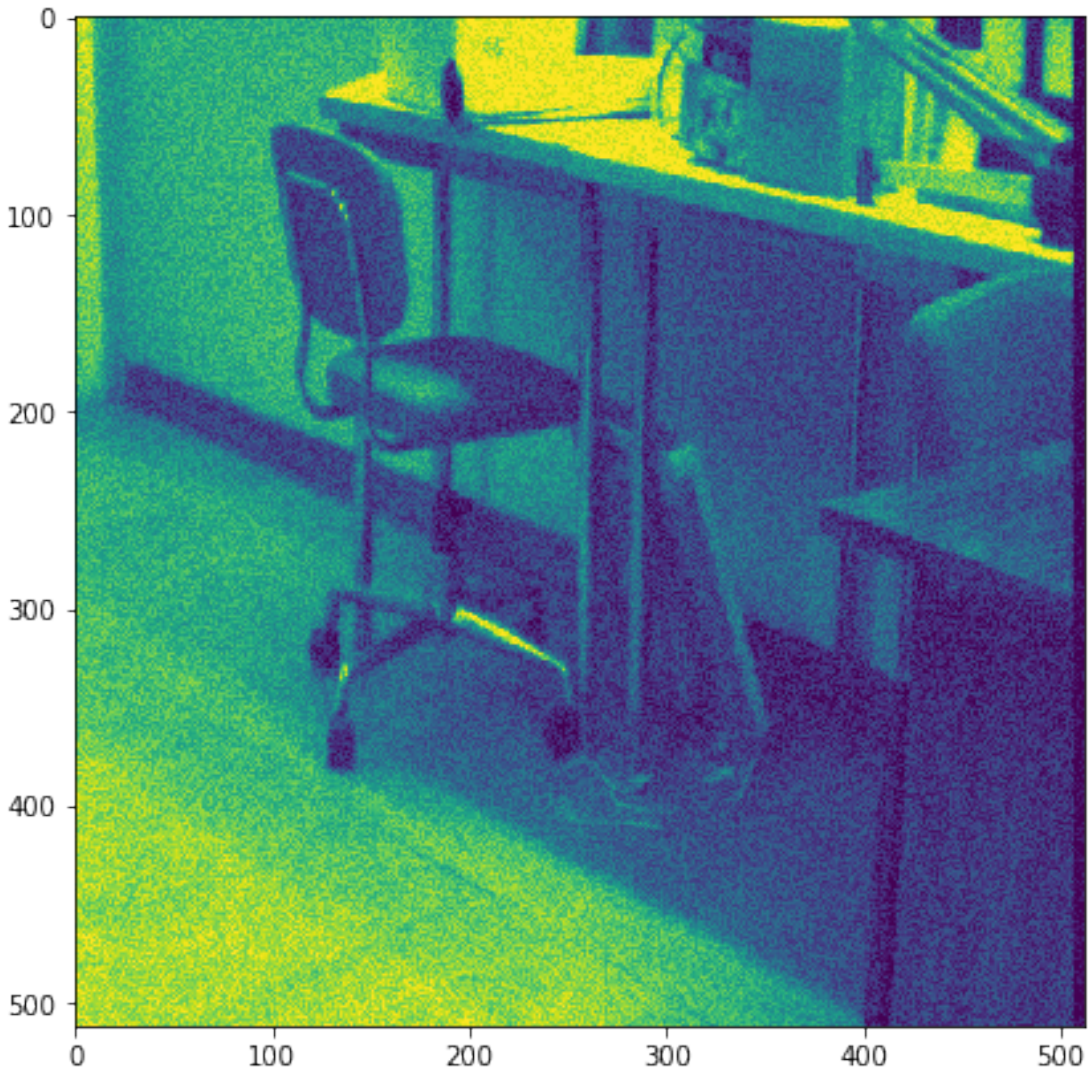


1.6 Filtrado

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

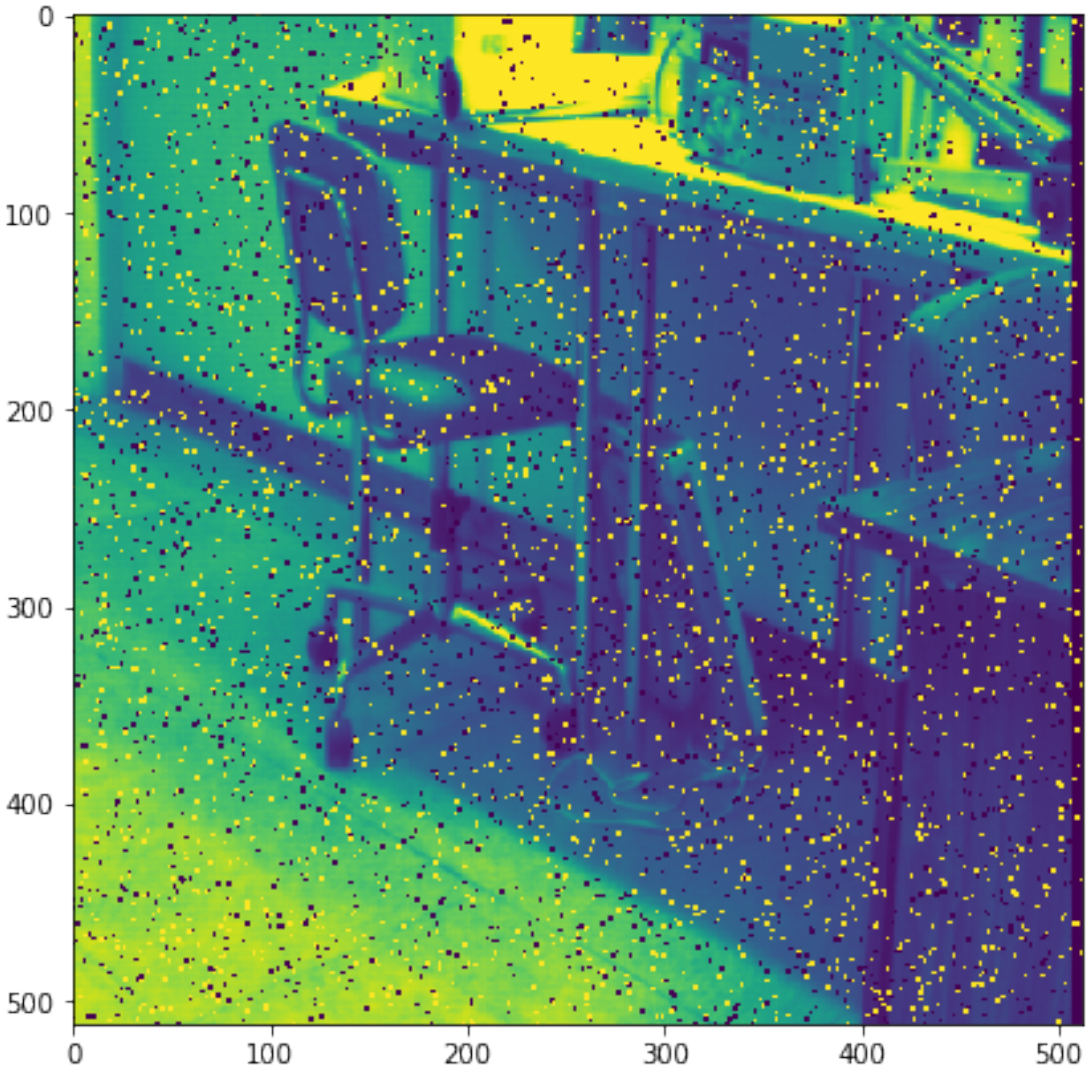
Carga y muestra las imágenes `escgaus.bmp` y `escimp5.bmp` que están contaminadas respectivamente con ruido de tipo gaussiano e impulsional. En los siguientes ejercicios también puedes utilizar otras imágenes que te parezcan interesantes.

```
In [7]: plt.figure(figsize=(12,7))
        escgaus = plt.imread('images/escgaus.bmp')
        plt.imshow(escgaus)

Out[7]: <matplotlib.image.AxesImage at 0x11bd496d8>
```




```
In [8]: plt.figure(figsize=(12,7))
        imp = plt.imread('images/escimp5.bmp')
        plt.imshow(imp)
```

```
Out[8]: <matplotlib.image.AxesImage at 0x11b0a9e80>
```



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.

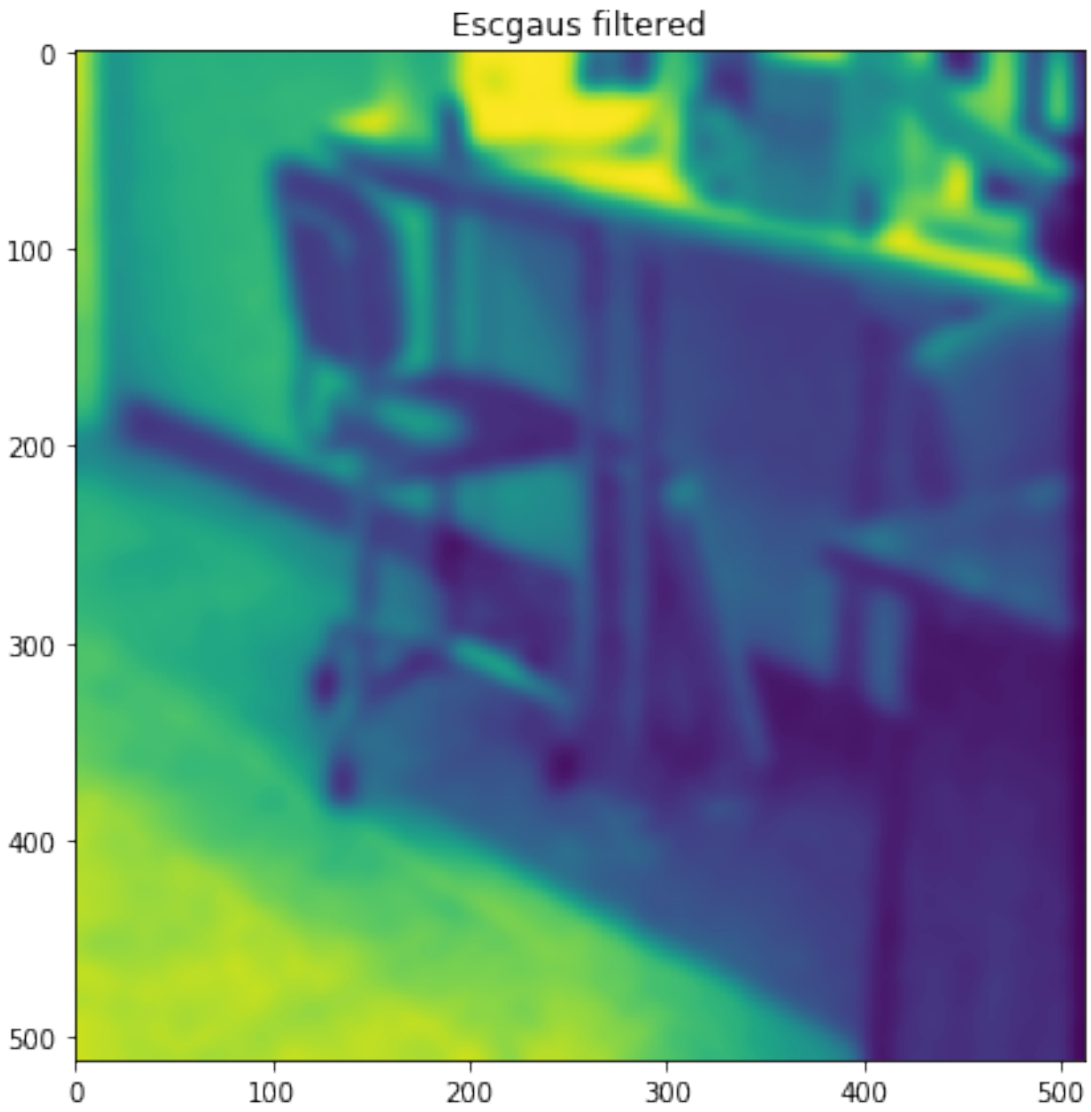
Muestra y discute los resultados. Pinta alguna de las máscaras construidas.

```
In [9]: def masc_gaus(sigma, n):

            if n%2 == 0 or n < 0:
                print("n must be odd and positive")

            if n < 5*sigma:
                print("n must be lower or equal than 5*alpha")
```

```
    center = (n-1)/2

    mask = np.arange(n) - center

    mask = mask*mask/(sigma*sigma)

    mask = np.exp(-mask/2)

    sum_all = np.sum(mask)

    mask = mask/sum_all

    return mask
```

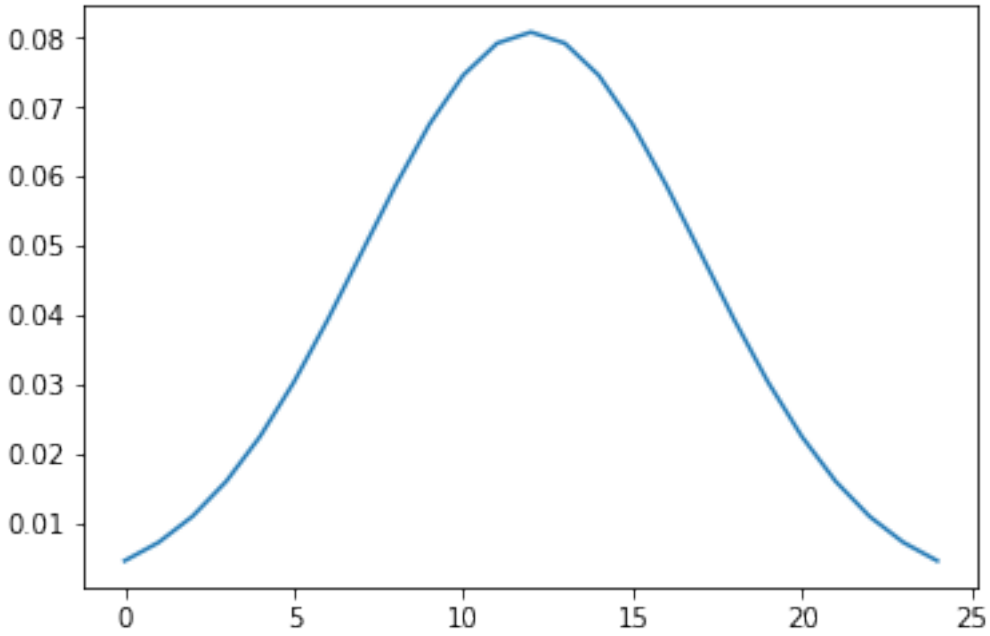
In [11]:
```
plt.figure(figsize=(12,7))
mask = masc_gaus(5,25)
aux = convolve1d(escgaus, mask, axis = 0)
filtered_escgaus = convolve1d(aux, mask, axis=1)
plt.title('Escgaus filtered')
plt.imshow(filtered_escgaus)
```

Out[11]:
```
<matplotlib.image.AxesImage at 0x11e0c4390>
```



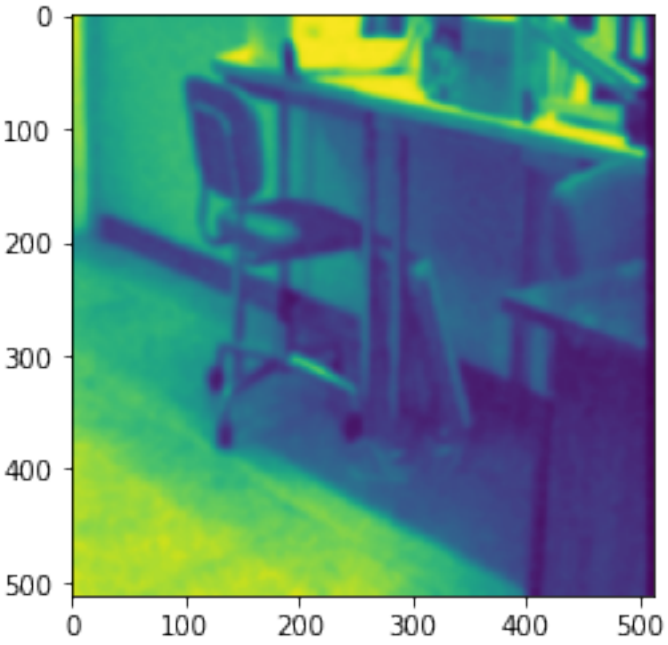
Mascara 1

```
In [12]: plt.plot(mask)

Out[12]: [<matplotlib.lines.Line2D at 0x11e32e278>]
```




```
In [13]: mask2 = masc_gaus(3,15)
         aux = convolve1d(escgaus, mask2, axis = 0)
         filtered_escgaus_2 = convolve1d(aux, mask2, axis=1)
         plt.imshow(filtered_escgaus_2)
```

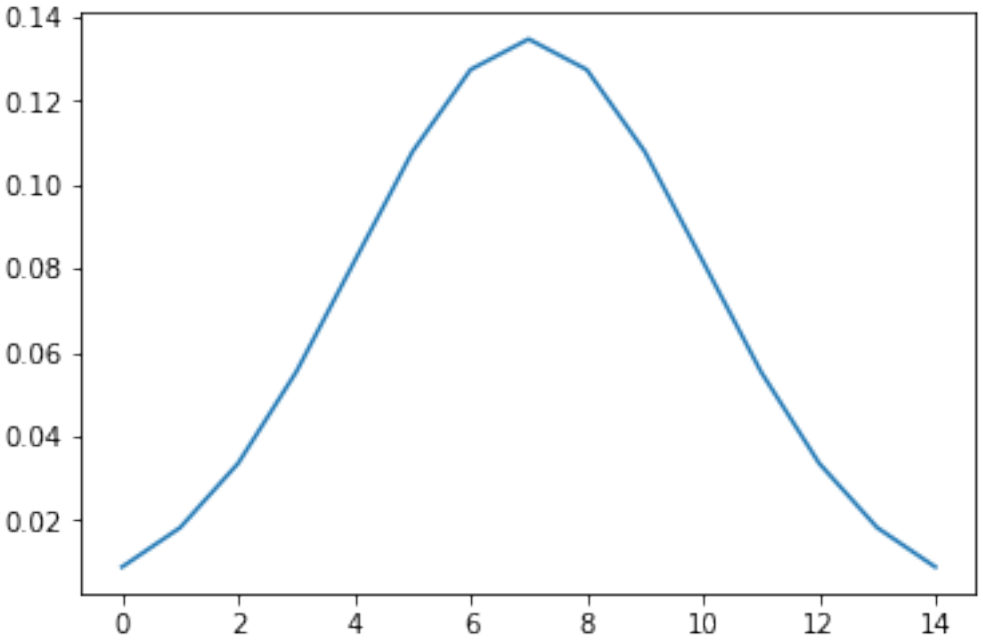
```
Out[13]: <matplotlib.image.AxesImage at 0x11e38bdd8>
```



Mascara 2

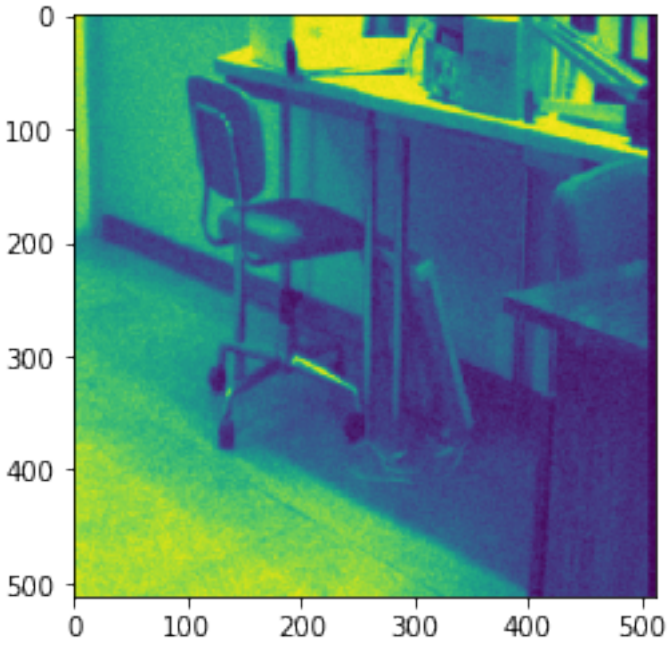
```
In [14]: plt.plot(mask2)
```

```
Out[14]: [<matplotlib.lines.Line2D at 0x11e4a5668>]
```




```
In [17]: mask3 = masc_gaus(1,5)
         aux = convolve1d(escgaus, mask3, axis = 0)
         filtered_escgaus_3 = convolve1d(aux, mask3, axis=1)
         plt.imshow(filtered_escgaus_3)
```

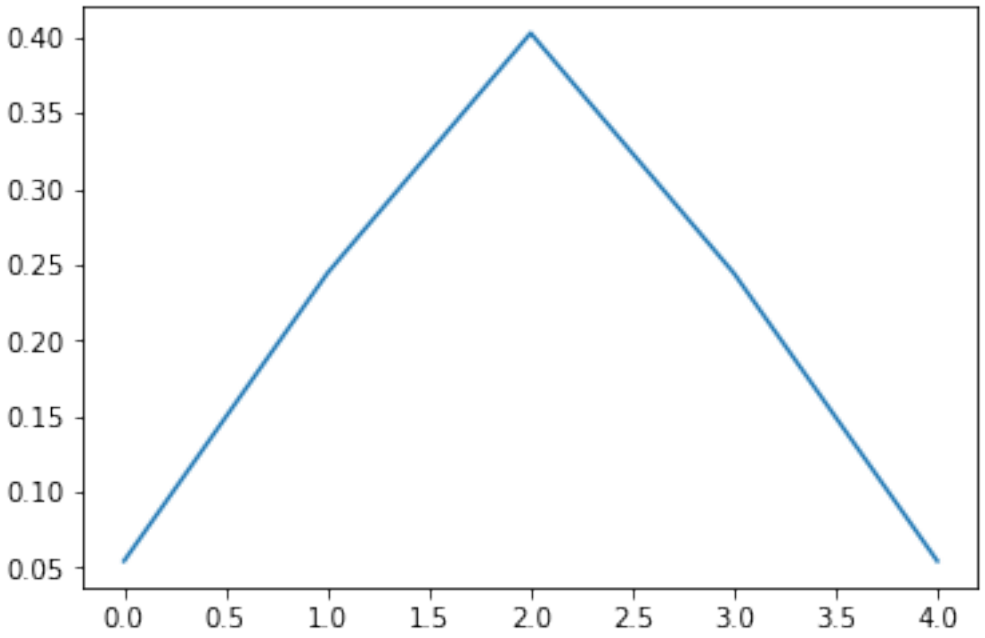
```
Out[17]: <matplotlib.image.AxesImage at 0x11e6c9e48>
```



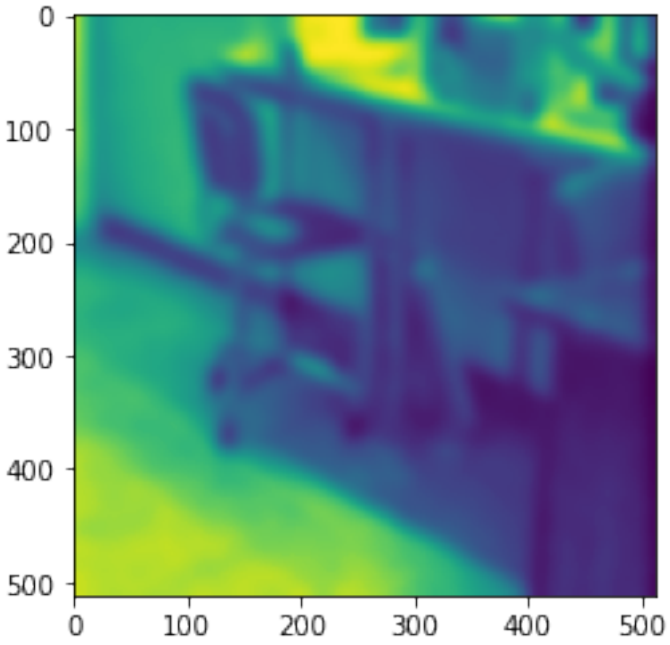
Mascara 3

```
In [18]: plt.plot(mask3)
```

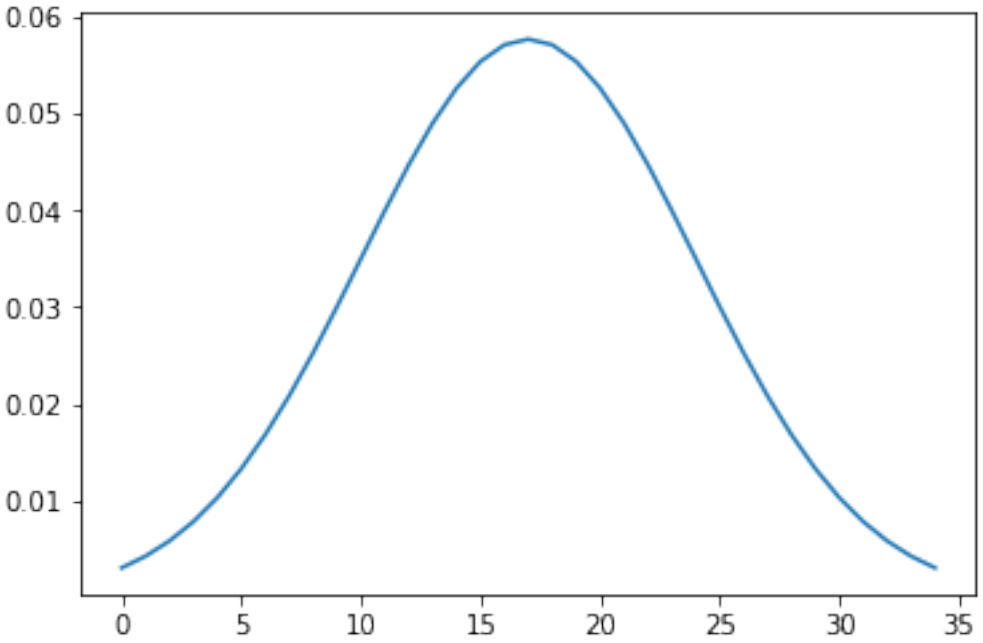
```
Out[18]: [<matplotlib.lines.Line2D at 0x11bd9ecf8>]
```



```
In [19]: mask4 = masc_gaus(7,35)
         aux = convolve1d(escgaus, mask4, axis = 0)
         filtered_escgaus4 = convolve1d(aux, mask4, axis=1)
         plt.imshow(filtered_escgaus4)

Out[19]: <matplotlib.image.AxesImage at 0x11c4e3f60>
```




```
In [20]: plt.plot(mask4)

Out[20]: [<matplotlib.lines.Line2D at 0x11e8ce208>]
```



En este caso los resultados son muy buenos, las imagenes se ven un poco mas borrosas pero no en exceso y se consigue quitar el ruido casi por completo

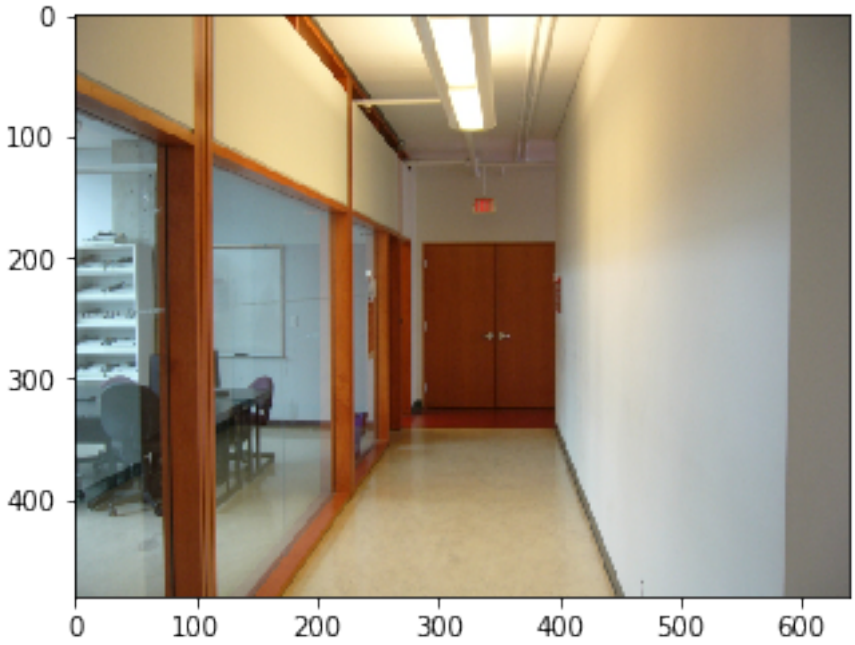
Ejercicio 4. Escribe una función `masc_deriv_gaus(sigma, n)` que construya una máscara de una dimensión de un filtro derivada del gaussiano de tamaño n y varianza . Filtra la imagen `corridor.jpg` con filtros bidimensionales de derivada del gaussiano para extraer los bordes de la imagen. Prueba con diferentes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

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

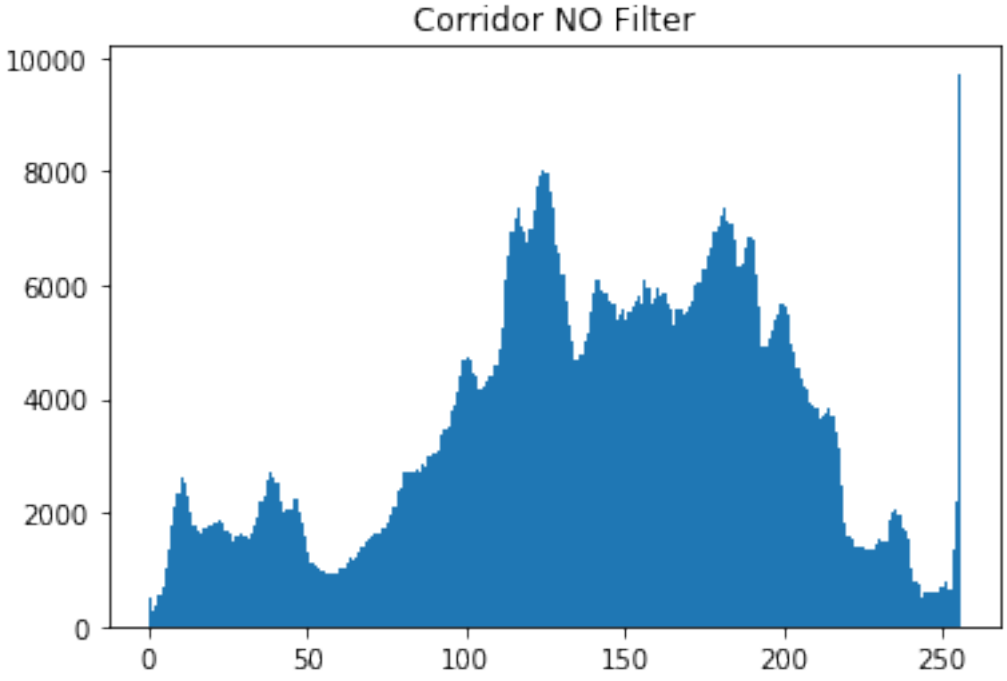
```
In [21]: def masc_deriv_gaus(sigma, n):

             if n%2 == 0 or n < 0:
                 print("n must be odd and positive")

             if n < 5*sigma:
                 print("n must be lower or equal than 5*alpha")

             return -(np.arange(n)/sigma**2)*np.exp(-((np.arange(n)/sigma)**2)/2)
```

```
In [22]: corridor = plt.imread('./images/corridor.jpg')
         plt.imshow(corridor)
```

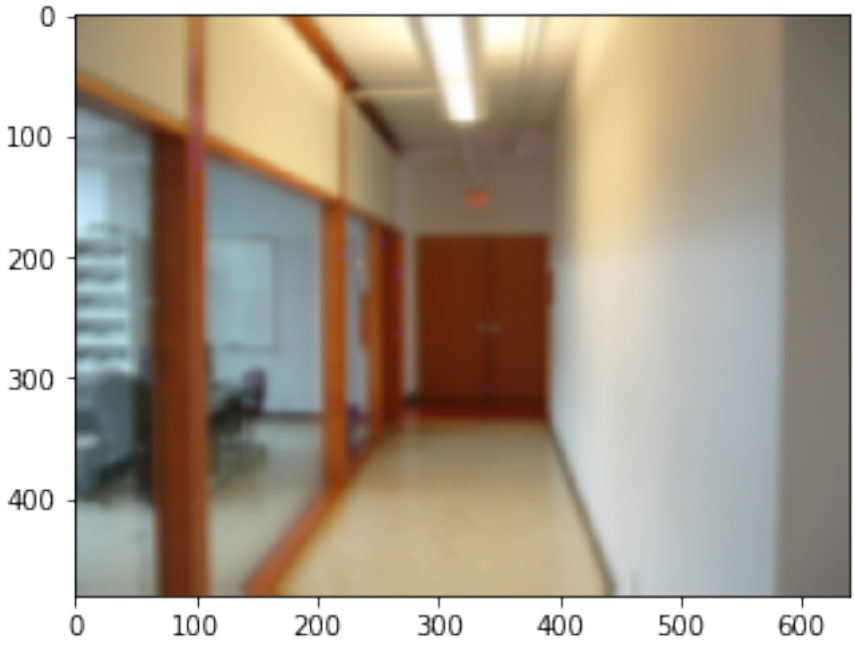
```
Out[22]: <matplotlib.image.AxesImage at 0x11e92aa58>
```



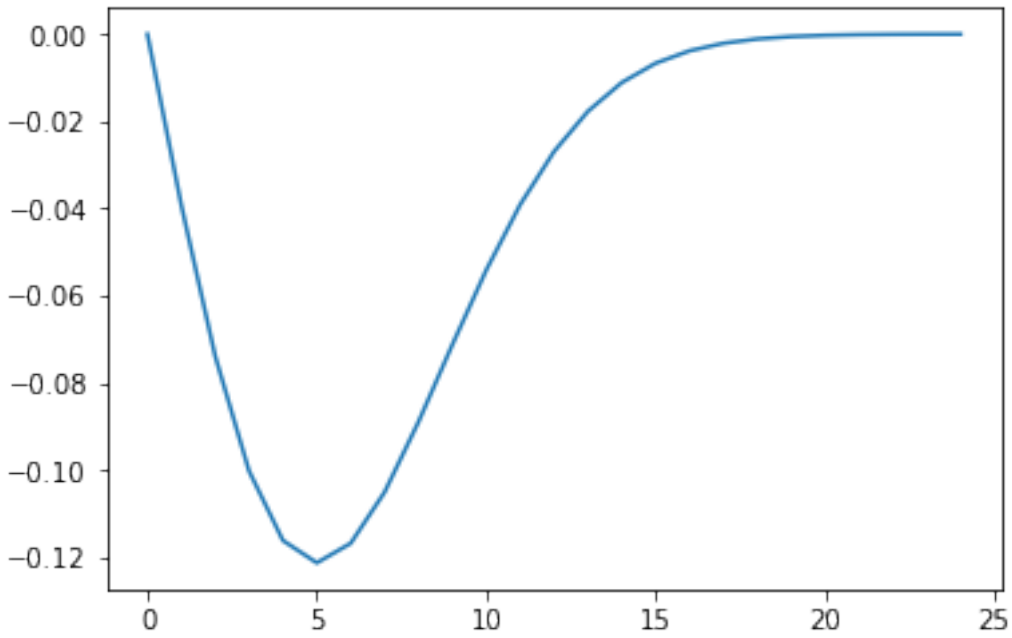
In [23]:
```
img_to_plot = corridor.flatten()
plt.hist(img_to_plot, bins=bins)
plt.title('Corridor NO Filter')
plt.show()
```



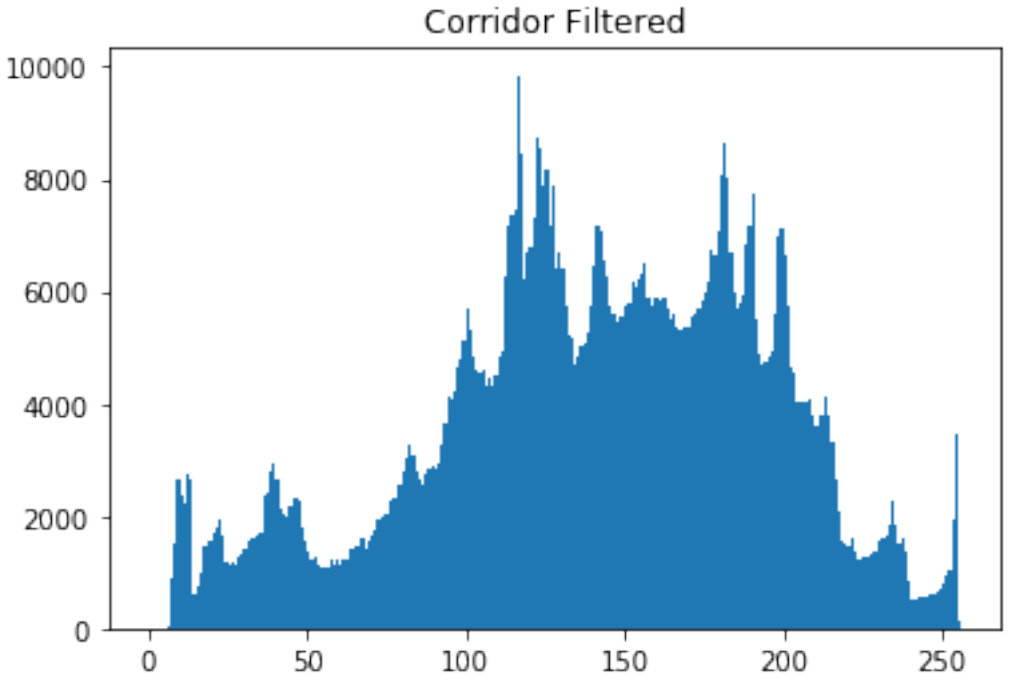
```
In [39]: mask_der = masc_deriv_gaus(5,25)
         aux = convolve1d(corridor, mask_der, axis = 0)
         filtered_corridor = convolve1d(aux, mask_der, axis=1)
         plt.imshow(filtered_corridor)

Out[39]: <matplotlib.image.AxesImage at 0x11f7e8a58>
```




```
In [40]: plt.plot(mask_der)

Out[40]: [<matplotlib.lines.Line2D at 0x12087f390>]
```

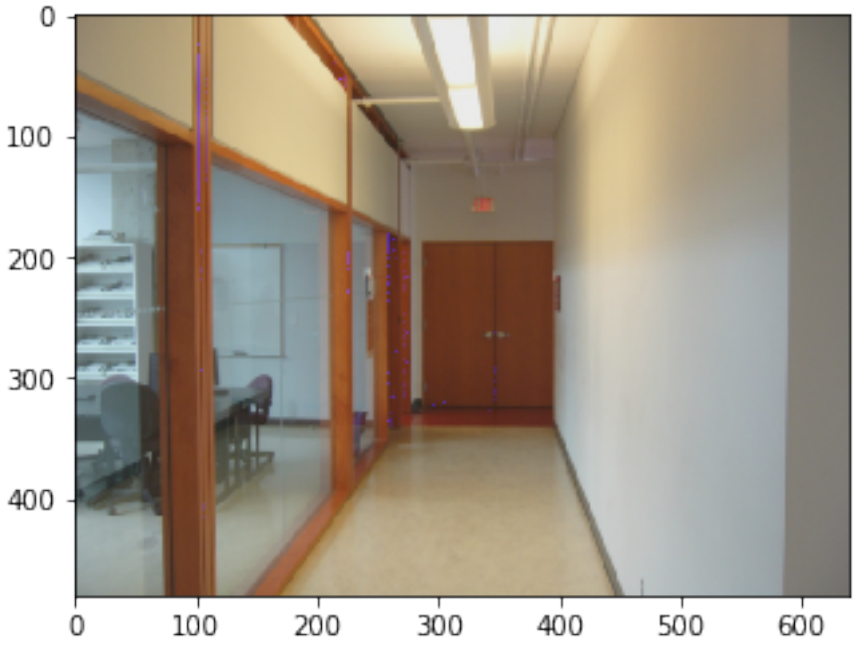


In [25]:
```
img_to_plot = filtered_corridor.flatten()
plt.hist(img_to_plot, bins=bins)
plt.title('Corridor Filtered')
plt.show()
```

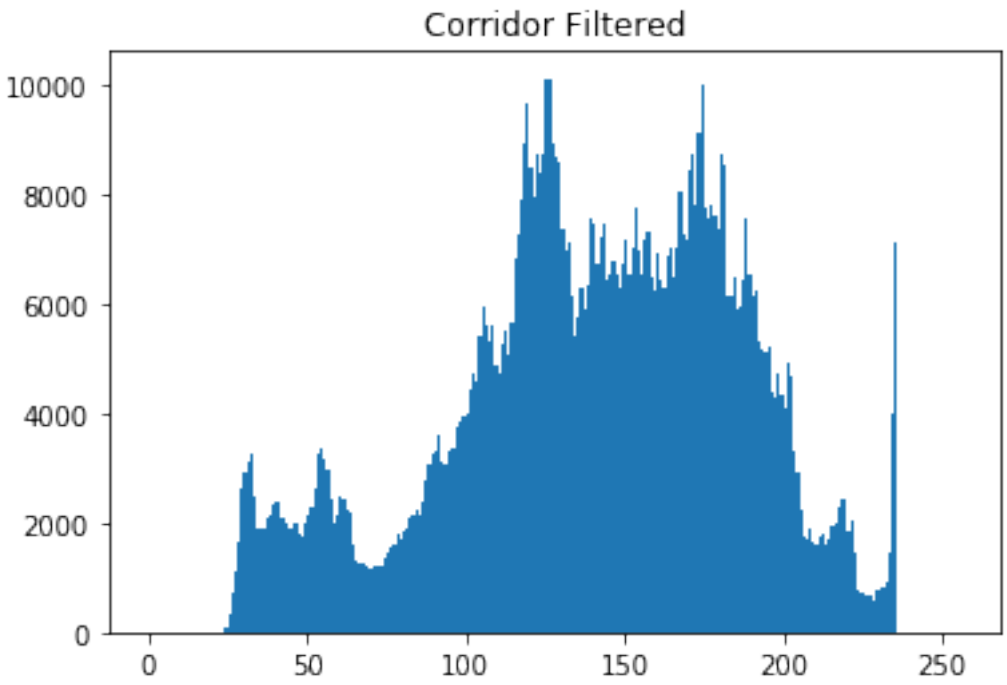


In [26]:
```
mask_der = masc_deriv_gaus(1,5)
aux = convolve1d(corridor, mask_der, axis = 0)
filtered_corridor = convolve1d(aux, mask_der, axis=1)
plt.imshow(filtered_corridor)
```

Out[26]: <matplotlib.image.AxesImage at 0x11f2240b8>

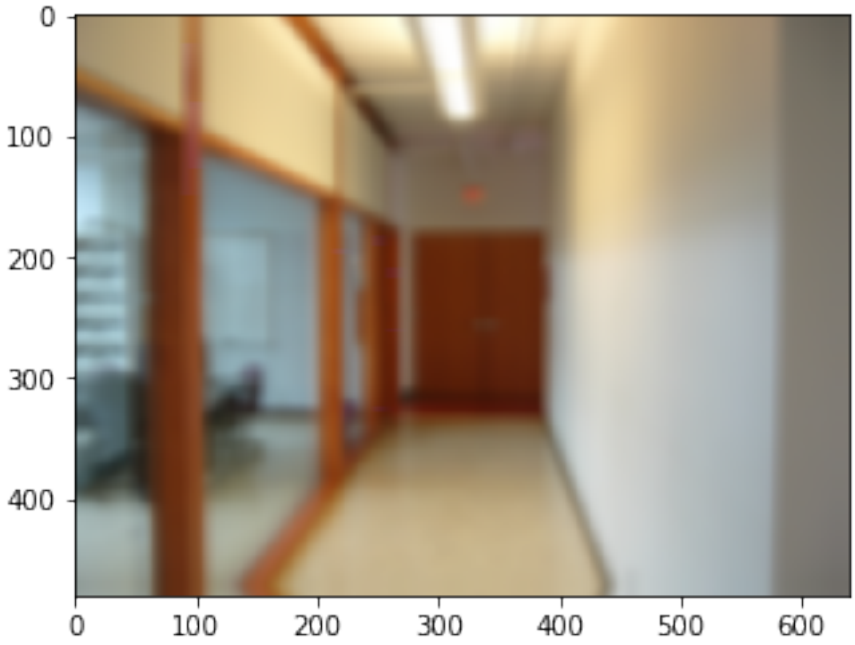


In [27]:
```
img_to_plot = filtered_corridor.flatten()
plt.hist(img_to_plot, bins=bins)
plt.title('Corridor Filtered')
plt.show()
```

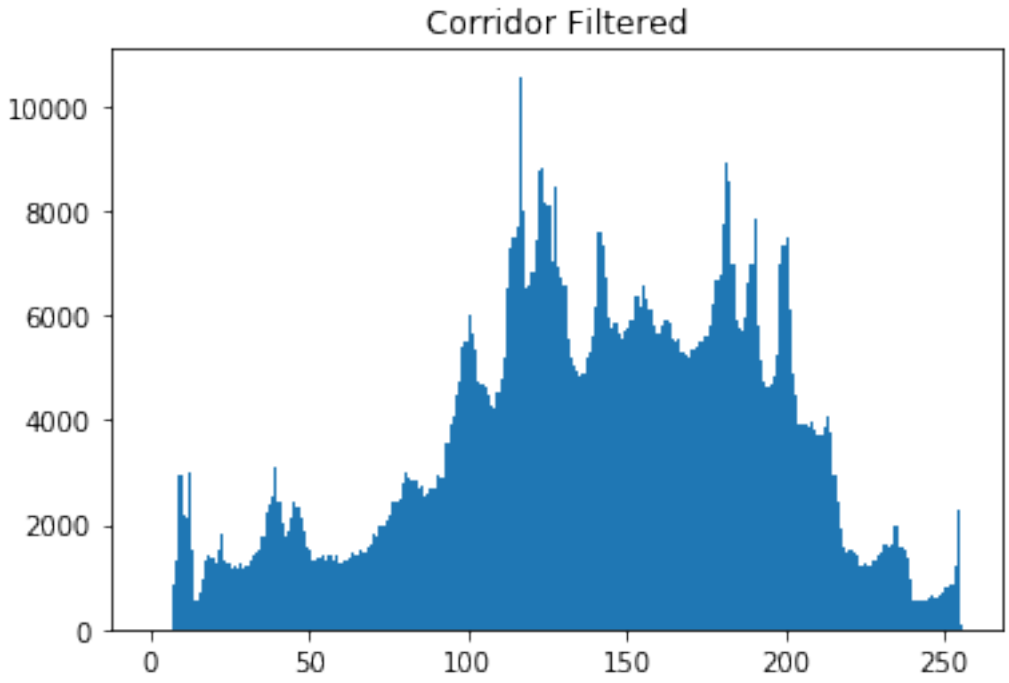



```
In [28]: mask_der = masc_deriv_gaus(7,35)
         aux = convolve1d(corridor, mask_der, axis = 0)
         filtered_corridor = convolve1d(aux, mask_der, axis=1)
         plt.imshow(filtered_corridor)
```

```
Out[28]: <matplotlib.image.AxesImage at 0x11eb98f60>
```



```
In [29]: img_to_plot = filtered_corridor.flatten()
         plt.hist(img_to_plot, bins=bins)
         plt.title('Corridor Filtered')
         plt.show()
```



En esta ocasión vemos que el filtro en las imagenes hace que la imagen este mas borrosa, pero consigue quitar el ruido

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 [30]: # Posible ejemplo de código
         import time

         start_time = time.clock()
         # ejecuta convoluciones ....
         aux = convolve1d(escgaus, mask, axis = 0)
         filtered_escgaus = convolve1d(aux, mask, axis=1)

         print(time.clock() - start_time, "seconds")
```

```
0.010680999999998164 seconds
```

```
In [31]: # Posible ejemplo de código
         import time

         td_mask = np.dot(np.array([[x] for x in mask]), np.array([mask]))

         start_time = time.clock()
         # ejecuta convoluciones ....
         convolve(escgaus, td_mask)

         print(time.clock() - start_time, "seconds")
```

```
0.271412999999999 seconds
```

Mask 2:

```
In [32]: # Posible ejemplo de código
         import time

         start_time = time.clock()
         # ejecuta convoluciones ....
         aux = convolve1d(escgaus, mask2, axis = 0)
         filtered_escgaus = convolve1d(aux, mask2, axis=1)

         print(time.clock() - start_time, "seconds")
```

```
0.008321999999999719 seconds
```

```
In [33]: # Posible ejemplo de código
         import time

         td_mask2 = np.dot(np.array([[x] for x in mask2]), np.array([mask2]))

         start_time = time.clock()
         # ejecuta convoluciones ....
         convolve(escgaus, td_mask2)

         print(time.clock() - start_time, "seconds")
```

```
0.09355899999999906 seconds
```

Mask 3

```
In [34]: # Posible ejemplo de código
         import time

         start_time = time.clock()
```

```
        # ejecuta convoluciones ....
        aux = convolve1d(escgaus, mask3, axis = 0)
        filtered_escgaus = convolve1d(aux, mask3, axis=1)

        print(time.clock() - start_time, "seconds")
```

```
0.005564000000003233 seconds
```

```
In [35]: # Posible ejemplo de código
         import time

         td_mask3 = np.dot(np.array([[x] for x in mask3]), np.array([mask3]))

         start_time = time.clock()
         # ejecuta convoluciones ....
         convolve(escgaus, td_mask3)

         print(time.clock() - start_time, "seconds")
```

```
0.013178999999997387 seconds
```

Mask 4

```
In [36]: # Posible ejemplo de código
         import time

         start_time = time.clock()
         # ejecuta convoluciones ....
         aux = convolve1d(escgaus, mask4, axis = 0)
         filtered_escgaus = convolve1d(aux, mask4, axis=1)

         print(time.clock() - start_time, "seconds")
```

```
0.015007000000000659 seconds
```

```
In [37]: # Posible ejemplo de código
         import time

         td_mask4 = np.dot(np.array([[x] for x in mask4]), np.array([mask4]))

         start_time = time.clock()
         # ejecuta convoluciones ....
         convolve(escgaus, td_mask4)

         print(time.clock() - start_time, "seconds")
```

```
0.49980500000000205 seconds
```

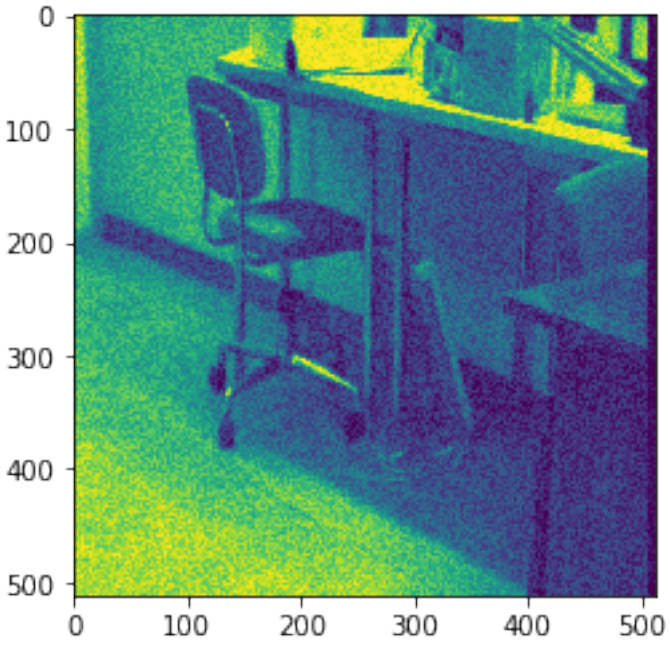
Vemos que el tiempo de cuando utilizamos convoluciones en dos dimensiones es mucho mayor que cuando hacemos dos veces convolucion de una dimension, es por esto que para este caso es mas recomendable hacer convoluciones en una dimension

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.

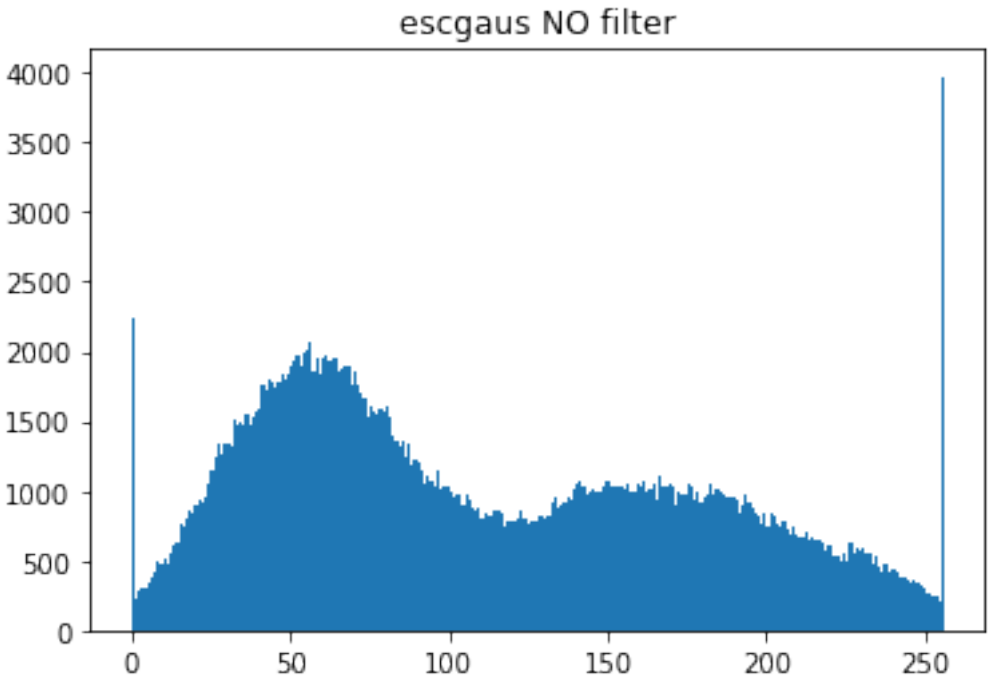
Para realizar este ejercicio puedes utilizar la función `cv2.medianBlur()` de OpenCV, `scipy.ndimage.median_filter()` de SciPy o hacer tu propia función. Para ello puedes escribir una función `mediana(img, n)` y aplicarla a la imagen con la función `scipy.ndimage.filters()`.

```
In [41]: plt.imshow(escgaus)
```

```
Out[41]: <matplotlib.image.AxesImage at 0x1208da390>
```

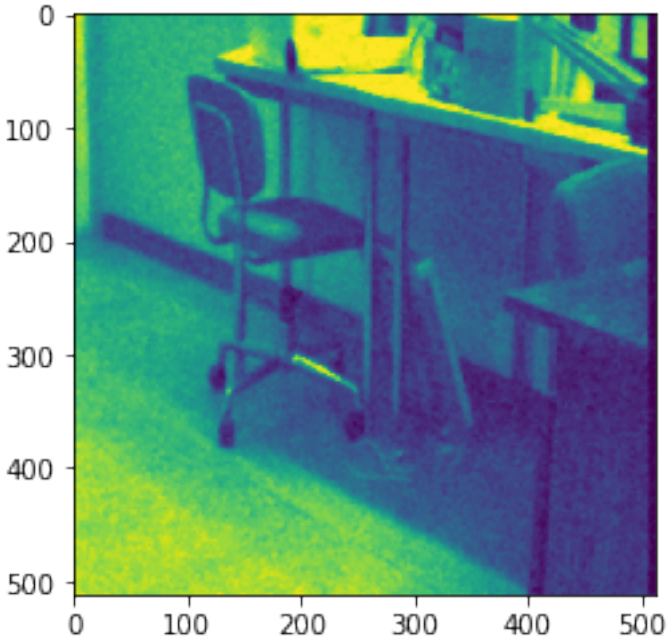



```
In [42]: img_to_plot = escgaus.flatten()
         plt.hist(img_to_plot, bins=bins)
         plt.title('escgaus NO filter')
         plt.show()
```

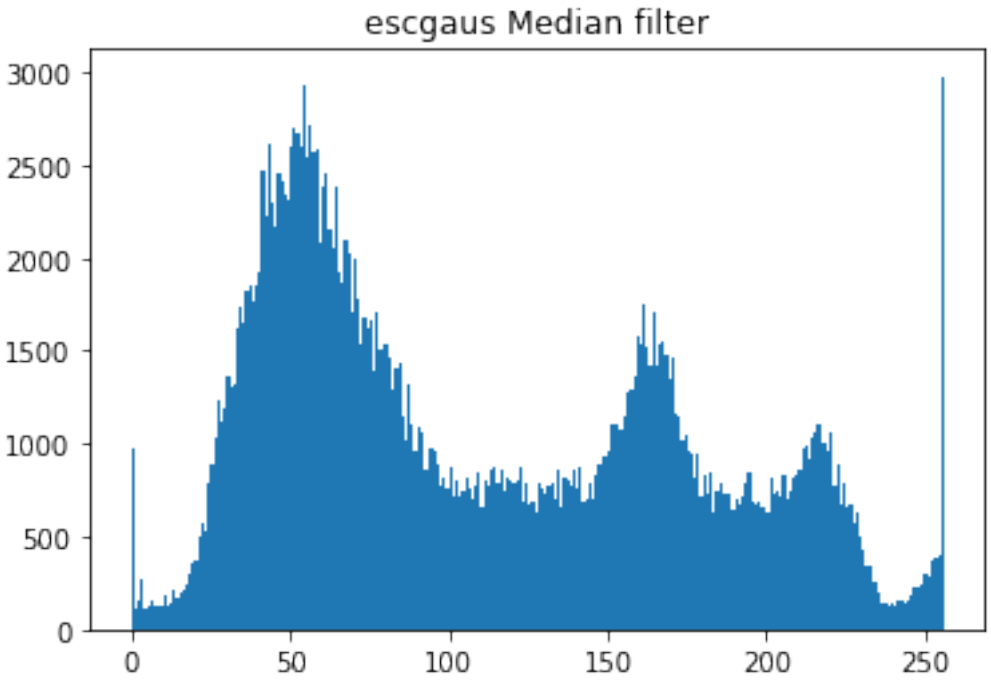


In [43]:
```
import cv2
plt.imshow(cv2.medianBlur(escgaus, 5))
```

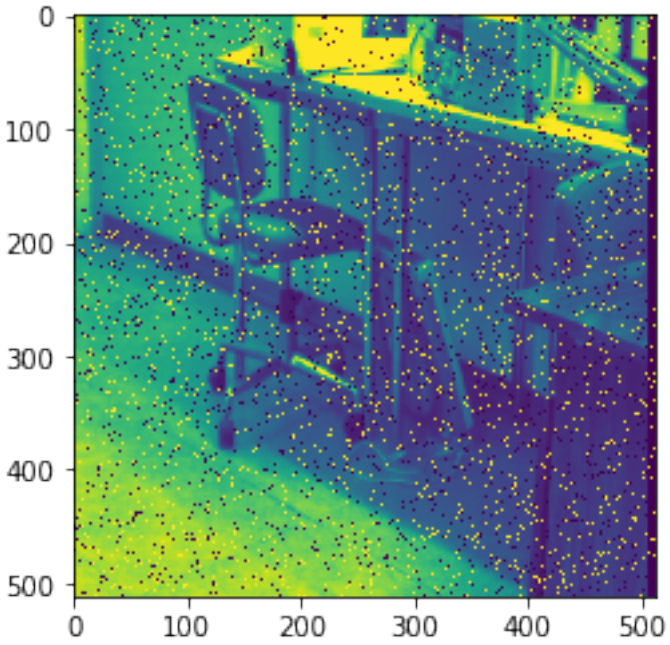
Out[43]: <matplotlib.image.AxesImage at 0x11f438390>



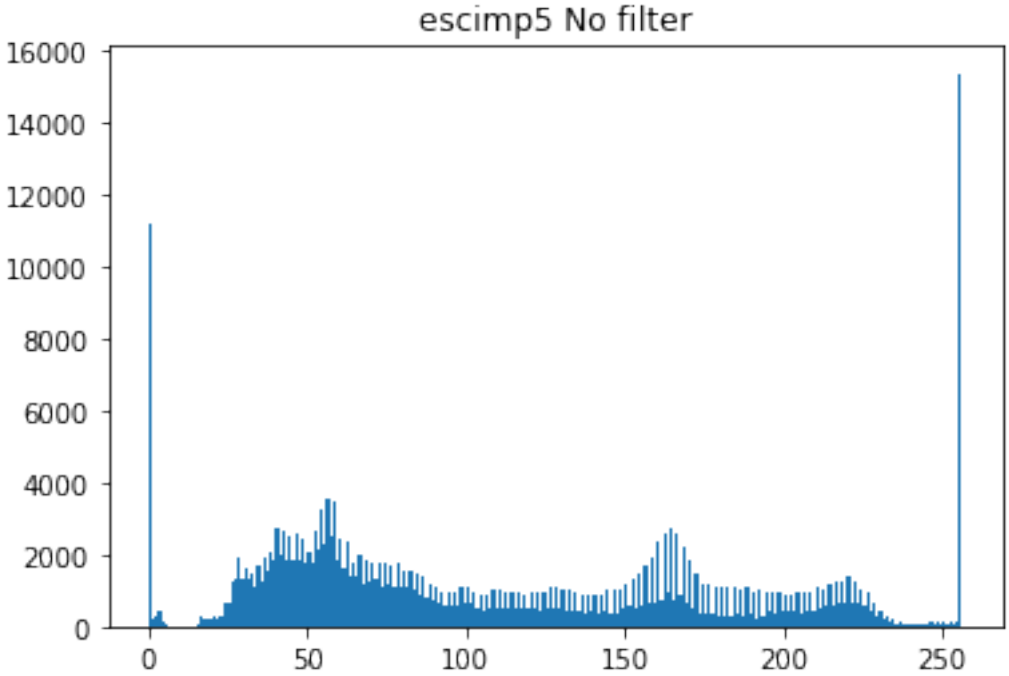
```
In [44]: img_to_plot = cv2.medianBlur(escgaus, 5).flatten()
         plt.hist(img_to_plot, bins=bins)
         plt.title('escgaus Median filter')
         plt.show()
```




```
In [46]: escimp5 = plt.imread('images/escimp5.bmp')
         plt.imshow(escimp5)

Out[46]: <matplotlib.image.AxesImage at 0x1269be908>
```

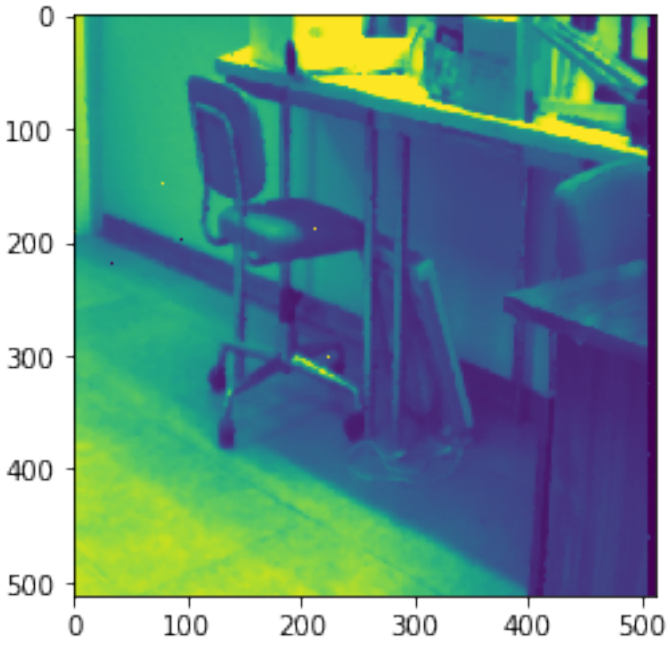



```
In [47]: img_to_plot = escimp5.flatten()
         plt.hist(img_to_plot, bins=bins)
         plt.title('escimp5 No filter')
         plt.show()
```

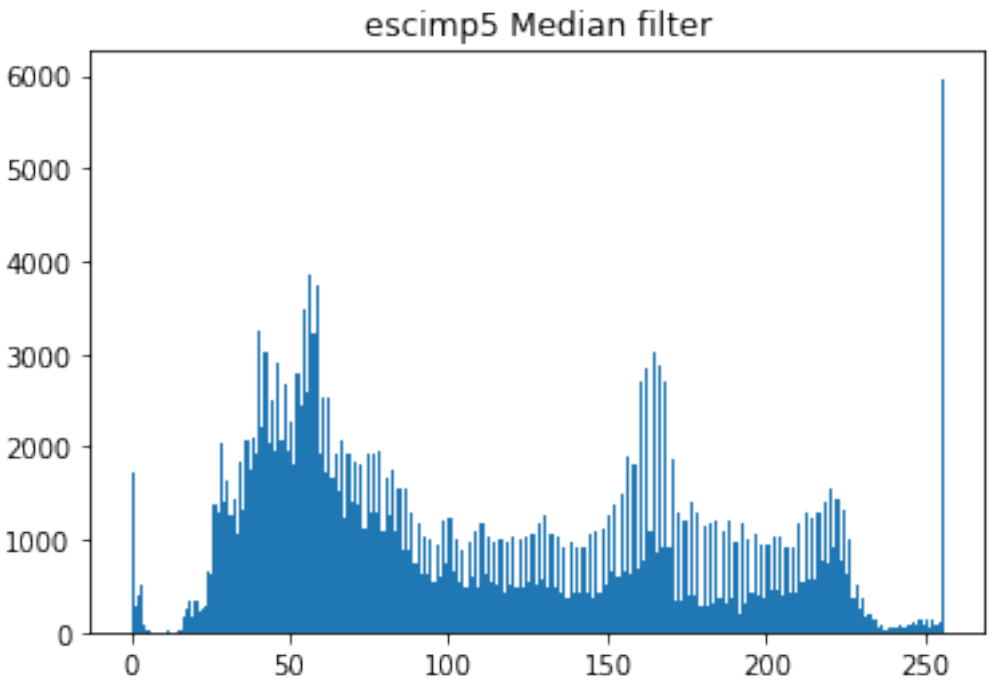


```
In [48]: plt.imshow(cv2.medianBlur(escimp5, 5))
```

```
Out[48]: <matplotlib.image.AxesImage at 0x126d2eac8>
```




```
In [49]: img_to_plot = cv2.medianBlur(escimp5, 5).flatten()
         plt.hist(img_to_plot, bins=bins)
         plt.title('escimp5 Median filter')
         plt.show()
```



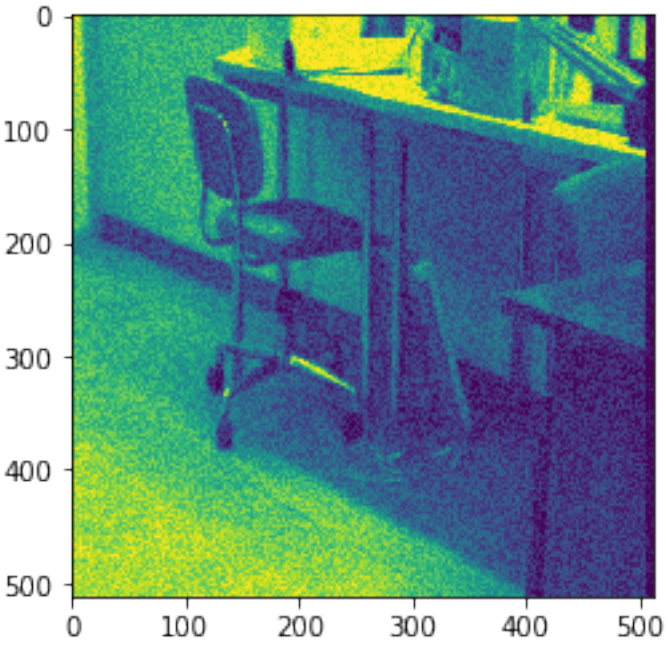
Ejercicio 7. Utiliza la funcion `cv2.bilateralFilter()` de OpenCV para realizar el filtrado bilateral de una imagen. Selecciona los parámetros adecuados y aplícalo a las imágenes `escgaus.bmp` y `escimp5.bmp` y otras que elijas tú.

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.

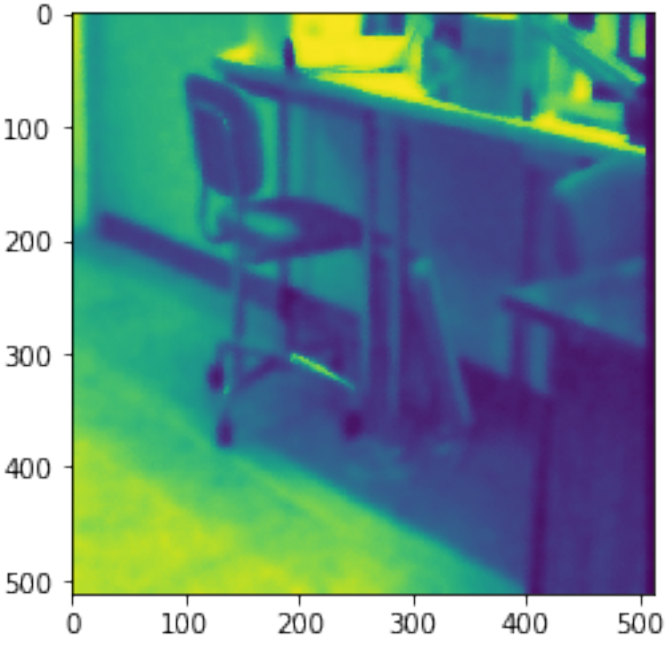
```
In [50]: plt.imshow(escgaus)
```

```
Out[50]: <matplotlib.image.AxesImage at 0x11f4a0a90>
```

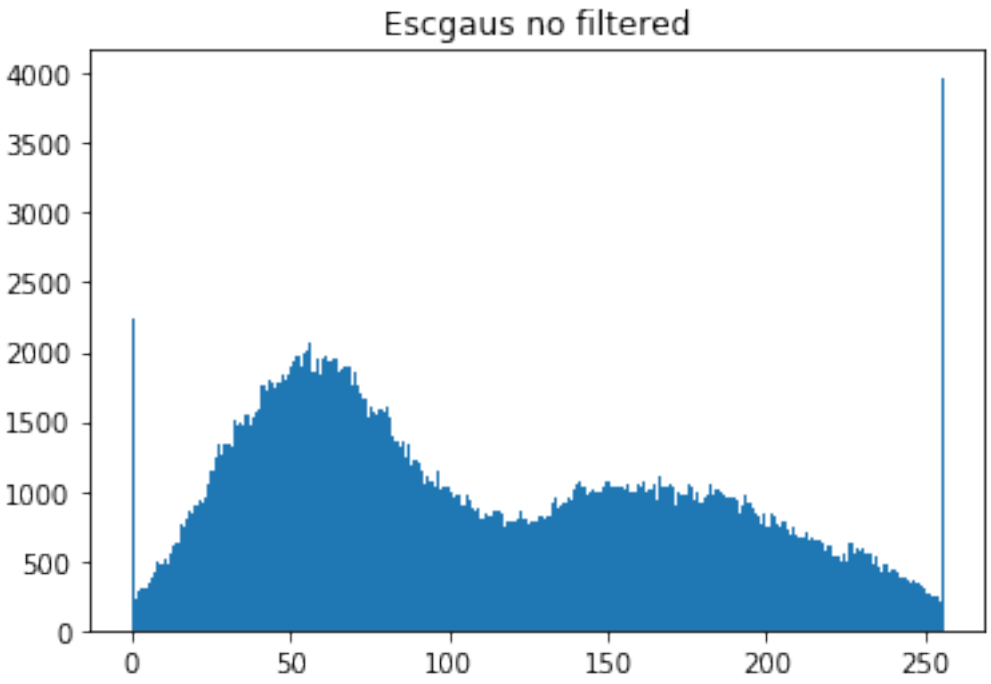



```
In [51]: plt.imshow(cv2.bilateralFilter(escgaus, 15, 80, 80))
```

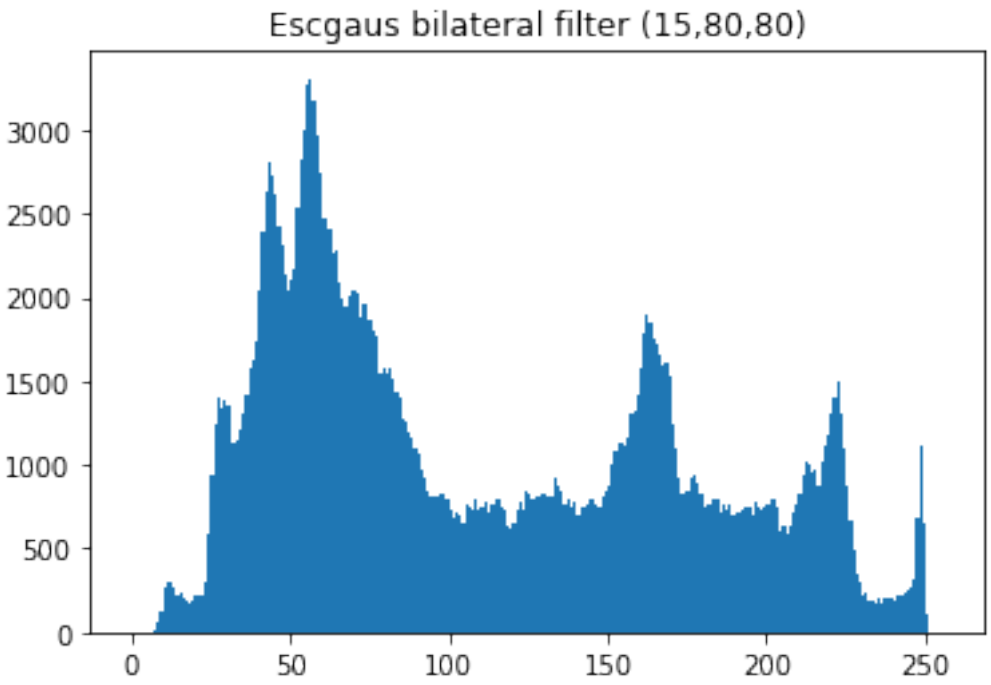
```
Out[51]: <matplotlib.image.AxesImage at 0x11f2399b0>
```



In [52]:
```
img_to_plot = escgaus.flatten()
plt.hist(img_to_plot, bins=bins)
plt.title('Escgaus no filtered')
plt.show()
```

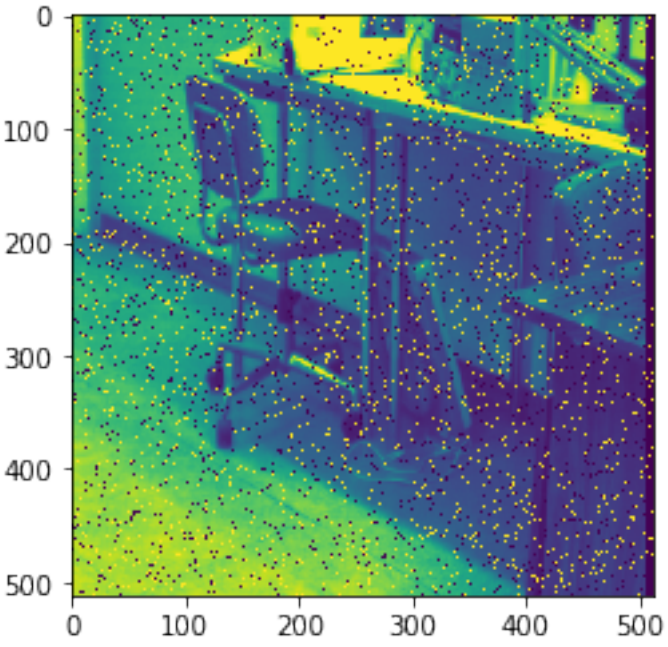


In [53]:
```
img_to_plot = cv2.bilateralFilter(escgaus, 15, 80, 80).flatten()
plt.hist(img_to_plot, bins=bins)
plt.title('Escgaus bilateral filter (15,80,80)')
plt.show()
```



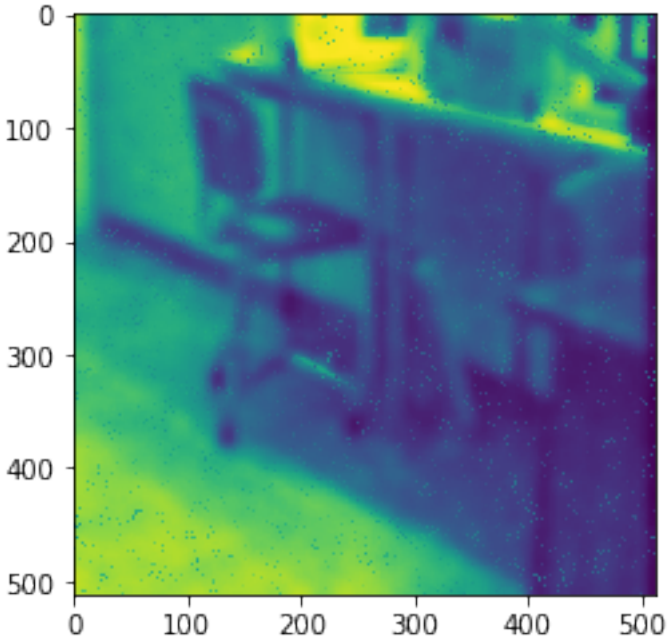
In [54]: plt.imshow(escimp5)

Out[54]: <matplotlib.image.AxesImage at 0x120ae8cc0>

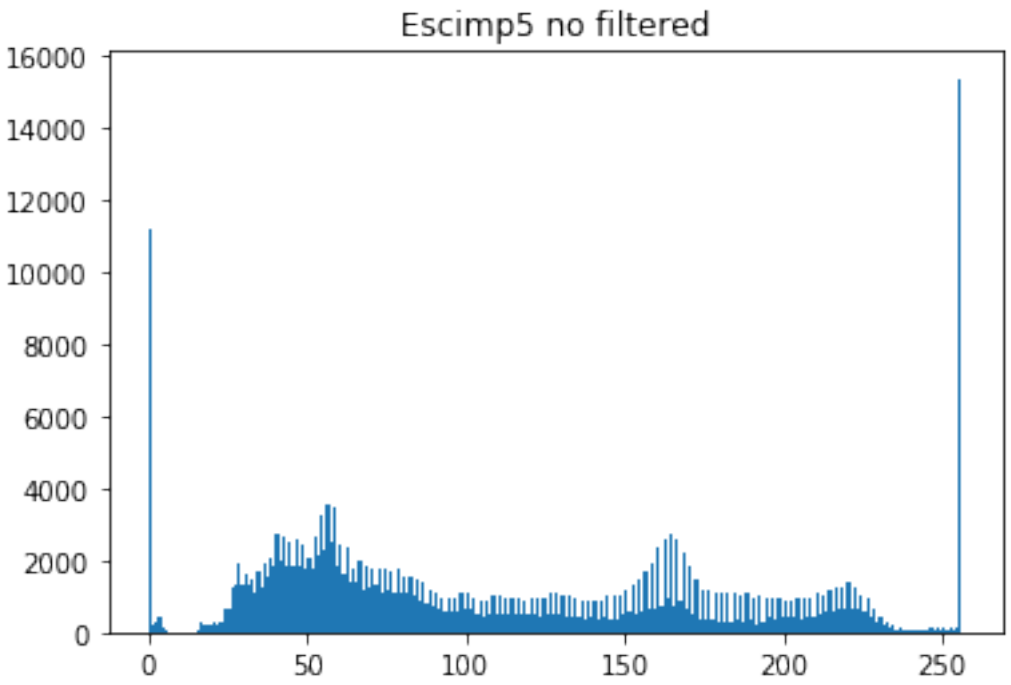


```
In [55]: plt.imshow(cv2.bilateralFilter(escimp5, 20, 140, 140))
```

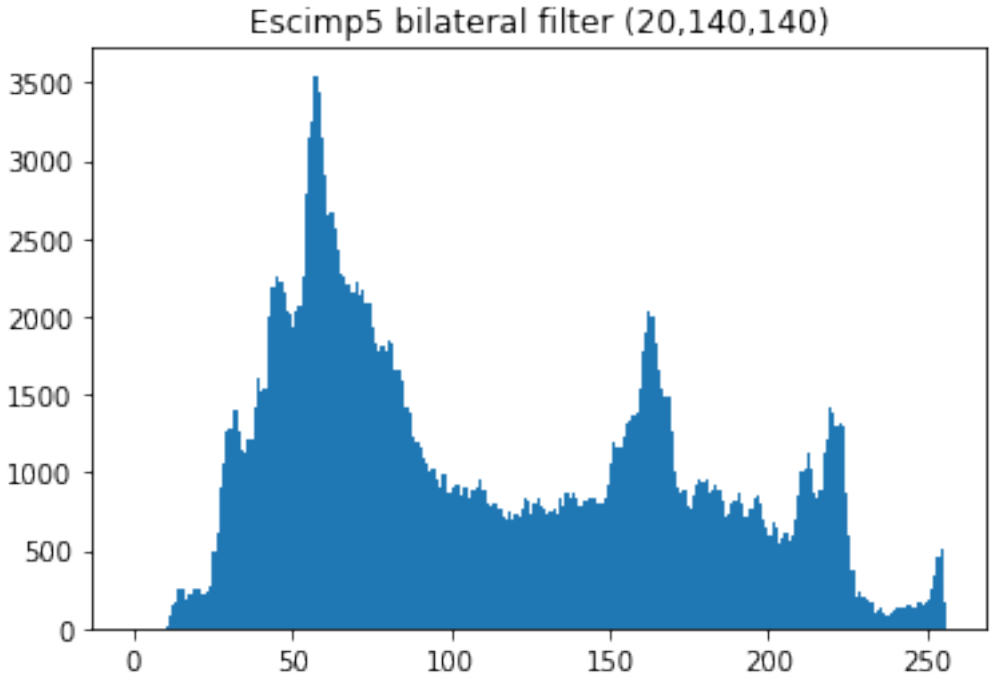
```
Out[55]: <matplotlib.image.AxesImage at 0x120ac4ef0>
```




```
In [56]: img_to_plot = escimp5.flatten()
         plt.hist(img_to_plot, bins=bins)
         plt.title('Escimp5 no filtered')
         plt.show()
```




```
In [57]: img_to_plot = cv2.bilateralFilter(escimp5, 20, 140, 140).flatten()
         plt.hist(img_to_plot, bins=bins)
         plt.title('Escimp5 bilateral filter (20,140,140)')
         plt.show()
```



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).

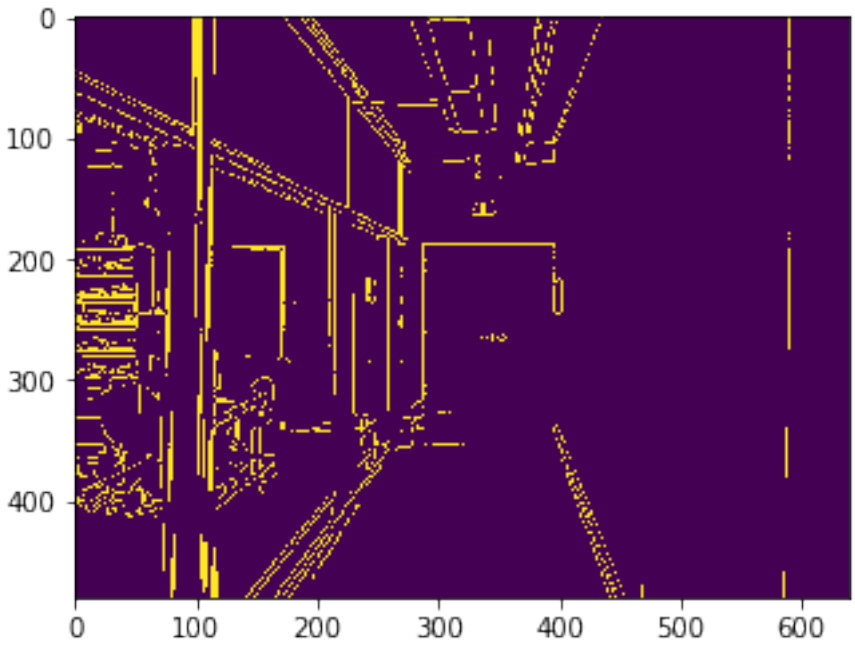
```
In [3]: import matplotlib.pyplot as plt
        import numpy as np
        import cv2 as cv

        img = plt.imread('images/corridor.jpg')
        plt.imshow(img)
```

```
Out[3]: <matplotlib.image.AxesImage at 0x7fc6d1098390>
```


```
In [4]: def get_borders(img): # should be implemented using exercises 3 and 4
            aux = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
            return cv.Canny(img, 50, 150, apertureSize = 3)
```

```
In [6]: borders = get_borders(img)

        plt.imshow(borders)
```

```
Out[6]: <matplotlib.image.AxesImage at 0x7fc67514ce48>
```




```
In [8]: minLineLength = 10
        maxLineGap = 1

        lines = cv.HoughLinesP(borders, rho=1, theta=np.pi/180, threshold=100, minLineLength=m

        for x1,y1,x2,y2 in lines[0]:
            cv.line(img,(x1,y1),(x2,y2),(0,255,0),2)

        plt.imshow(img)
```

```
Out[8]: <matplotlib.image.AxesImage at 0x7fc6750901d0>
```

No nos ha dado tiempo a experimentar con distintos parámetros de HoughLinesP por falta de tiempo. Igualmente, tampoco nos ha dado tiempo a experimentar con distintos algortimos de extracción de bordes y por tanto tampoco a integrar los filtros implementados en los ejercicios 3 y 4.

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 [1]: ###################################
        # Segmentacion de imagen a la "Grab Cut" simplificado
        # por Luis Baumela. UPM. 15-10-2015
        # Vision por Computador. Master en Inteligencia Artificial
        ###################################

        # Comentarios en español del fichero original
        # Comments in english are new


        import numpy as np
        import maxflow
        import matplotlib.pyplot as plt
```

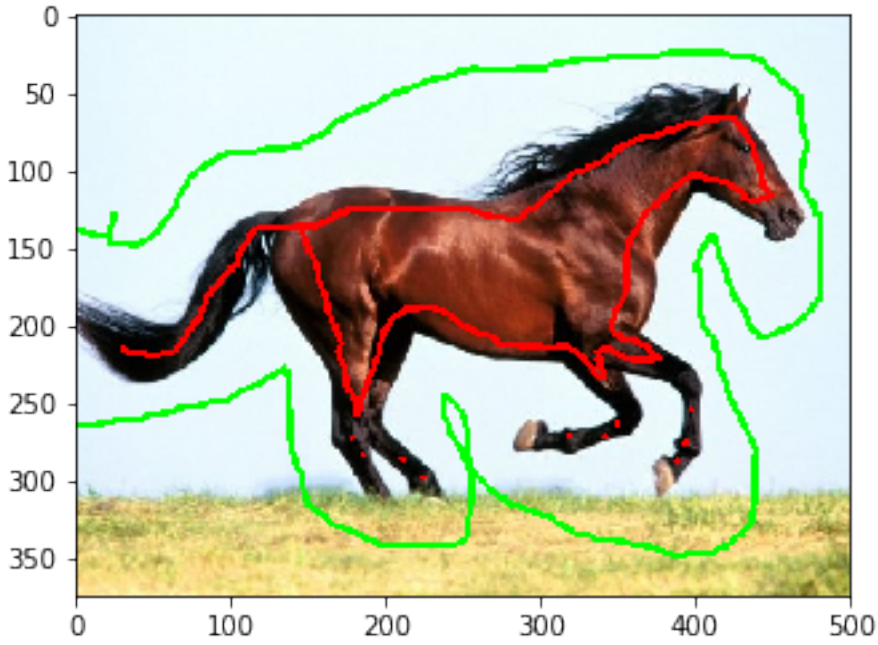
```
import select_pixels as sel
from math import sqrt

def segmentation(imgName):

    img = plt.imread(imgName)
    plt.imshow(img)

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

    v,h = img.shape[0], img.shape[1]

    # 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

    # version 1

    # Estos son los costes de los vecinos horizontales
    zeros_v = np.zeros(3*v).reshape(v,1,3)
    img_shift_h = np.concatenate((zeros_v,img[:,0:-1,:]),axis=1)
    aux_h = img-img_shift_h
    #exp_aff_h = 255 - 0.2989*aux_h[:,:,0] - 0.5870*aux_h[:,:,1] - 0.1140*aux_h[:,:,2]
    exp_aff_h = 255 - np.linalg.norm(aux_h,axis=2)/sqrt(3) # Is the norm the best for


    # Estos son los costes de los vecinos verticales
    zeros_h = np.zeros(3*h).reshape(1,h,3)
    img_shift_v = np.concatenate((zeros_h,img[0:-1,:,:]),axis=0)
    aux_v = np.absolute(img-img_shift_v)
    #exp_aff_v = 255 - 0.2989*aux_v[:,:,0] - 0.5870*aux_v[:,:,1] - 0.1140*aux_v[:,:,2]
    exp_aff_v = 255 - np.linalg.norm(aux_v,axis=2)/sqrt(3)


    # version 2 (debería ser igual que la 1, pero no se por qué no es así)

    exp_aff_h = np.zeros((img.shape[0], img.shape[1]))

    for i in range(1, img.shape[0]):
        for j in range(0, img.shape[1]):
            exp_aff_h[i][j] = 255 - np.linalg.norm(img[i][j]-img[i-1][j])/sqrt(3)
    # no hace falta calcular aparte la primera fila
```

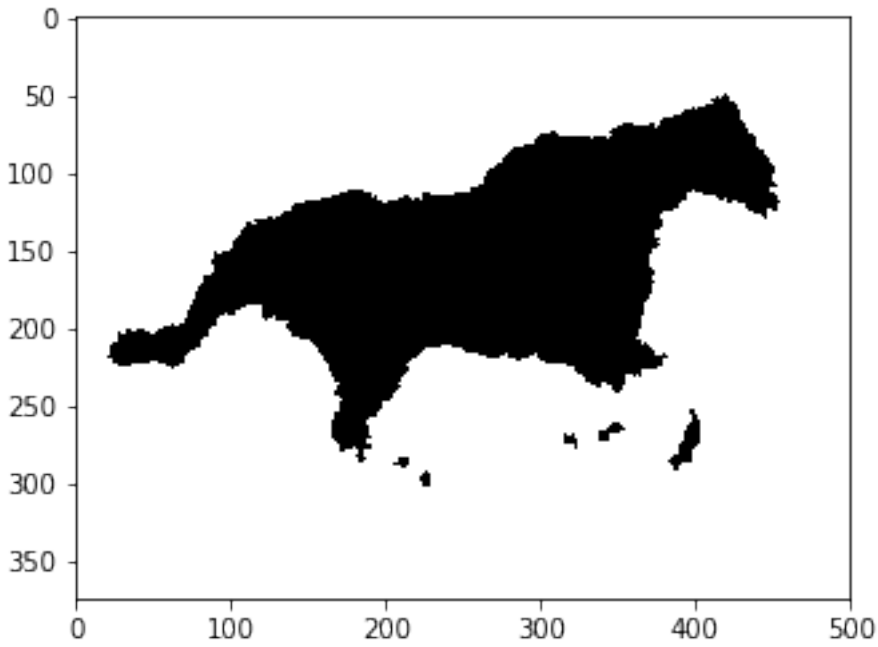
```
# Estos son los costes de los vecinos verticales
exp_aff_v =np.zeros((img.shape[0], img.shape[1]))

for i in range(0, img.shape[0]):
    for j in range(1, img.shape[1]):
        exp_aff_v[i][j] = 255 - np.linalg.norm(img[i][j]- img[i][j-1])/sqrt(3)
# no hace falta calcular aparte la primera columna


# What is the pythonic way to do what we do above?

# 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 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)))

mask_fg = np.all(np.equal(markedImg,(255,0,0)),2)
mask_bg = np.all(np.equal(markedImg,(0,255,0)),2)

exp_source = np.zeros((v,h))
exp_source[mask_fg]=np.inf

exp_sink = np.zeros((v,h))
exp_sink[mask_bg] = np.inf

# Incluyo las conexiones a los nodos terminales
# Pesos de los nodos terminales
g.add_grid_tedges(nodeids, exp_sink, exp_source)

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

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

In [2]:
```
imgName='images/horse.jpg'

segmentation(imgName)
```





In [4]:
```
persona='images/persona.png'

segmentation(persona)
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-4-a8f2d5c0e232> in <module>()
  1 persona='images/persona.png'
  2
----> 3 segmentation(persona)


<ipython-input-1-c7c21c66f012> in segmentation(imgName)
 21
 22     # Marco algunos pixeles que pertenecen el objeto y el fondo
---> 23     markedImg = sel.select_fg_bg(img)
 24     plt.imshow(markedImg)
 25


/home/ignacio/Desktop/cv/pr1/select_pixels.py in select_fg_bg(img, radio)
 32
 33 #    imgpyg=pygame.image.load(imgName)
---> 34     imgpyg=pygame.image.frombuffer(img,img.shape[-2::-1],'RGB')
 35     screen.blit(imgpyg,(0,0))
 36     pygame.display.flip() # update the display


ValueError: Buffer length does not equal format and resolution size
```

Los resultados no son buenos, principalmente debido a que la función de afinidad no es buena. Como no tenemos en cuenta el coste de clasificar dos pixeles adyacentes en la misma clase, lo único que importa para determinar el fondo y la figura principal es minimizar el coste de clasificar dos pixeles en diferentes clases a lo largo del borde. Pero en esa minimización, el perímetro de ese borde juega un papel importante (cuanto mayor, mayor coste) y al final se tiende a minimizar ese perímetro en lugar de minimizar el coste en cada punto del borde. Eso explica la tendencia a que haya bordes rectilíneos, y que si se se usan puntos en la segmentación manual en lugar de lineas o curvas cerradas, esos puntos se queden aislados en la clasificación

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.

Una primera mejora podría ser introducir un coste al clasificar dos pixeles adyacentes en la misma clase. Pero entonces se tiene que cumplir la desigualdad Vij(0,0)+Vij(1,1) <= Vij(0,1)+Vij(1,0) para poder plantear el problema de optimización como un problema de flujo máximo, y no podemos usar costes altos cuando cuando Vij(0,1) y Vij(1,0) son bajos, que es pricisamente lo que interesaría. Además, adaptar el grafo de flujo para que refleje esos costes no es sencillo. Si Vij(0,0)=Vij(1,1)=Kij es más fácil, ya que se pueden añadir aristas de xi a xj y de xj a xi con capacidad -Kij, de forma que si el corte no separa los dos nodos se deja de restar esa cantidad, lo cual es equvalente a que ése sea el coste de no separarlos. Por supuesto, las aristas no pueden tener capacidad negativa, así que lo que en realidad lo que haríamos sería restar esas capacidades a las capacidades originales y asegurarnos que el resultado es positivo. Eso impondría la restricción Kij <= Vij(1,0), Kij <= Vij(0,1), algo más fuerte que la anterior. En el caso general habría que modificar las capacidades de las aristas que conectan los nodos con la fuente y el sumidero, usando expresiones más complejas Por ambas razones no intentaremos avanzar en esta dirección.

Otra opción podría ser añadir un coste para clasificar en la misma clase a píxeles adyacentes

diagonalmente, o incluso a píxeles a menos de una distancia dada, ponderando el coste por un factor que decrezca a medida que esa distancia aumenta. Tampoco exploraremos esta idea.

Otra opción seria mejorar los términos unitarios, como se sugiere en el enunciado. Si sabemos por ejemplo que los píxeles del fondo van a ser más claros y los de la figura principal más oscuros, podemos modificar los costes unitarios consecuentemente. El problema es que no sabemos eso de antemano, y queremos que nuestro algoritmo sea genérico y no dependa de la imagen a procesar. Aún así, se pueden plantear ideas en esta dirección. Por ejemplo, se podría definir el coste unitario de clasificar un píxel como fondo a la distancia mínima de ese píxel a los píxeles clasificados inicialmente por el usuario como fondo, e igual para el foreground. Tampoco intentaremos implementar esta idea, aunque parece prometedora.

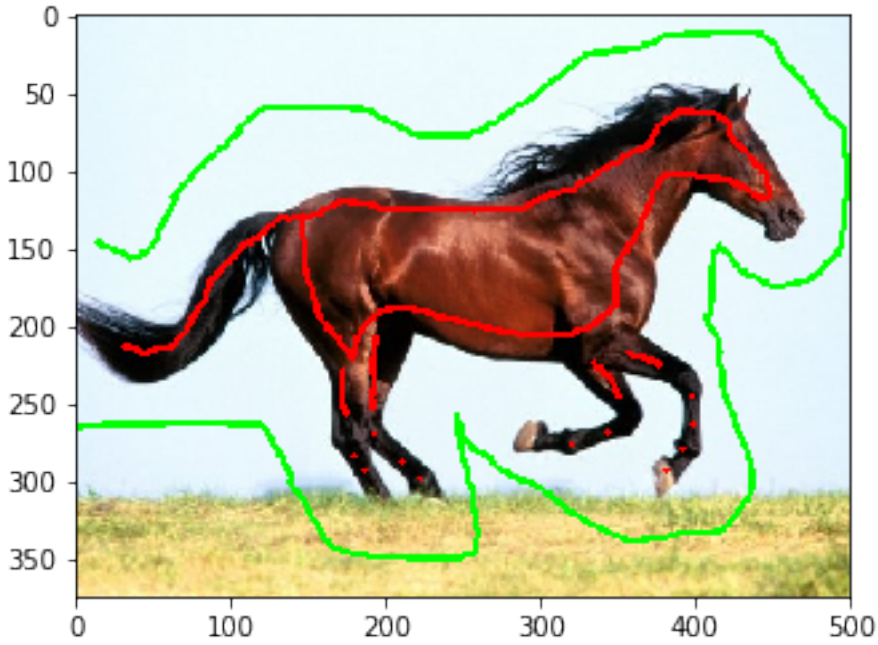
La optimización que plantearemos entonces será la segunda planteada por el enunciado: mejorar la función de afinidad entre píxeles. Si se analiza la conclusión que hemos sacado de la versión anterior, se puede deducir que el problema que tenía la función de afinidad era en parte que era lineal (más o menos). Daba costos altos y bajos cuando tenía que darlos, pero ese coste disminuía linealmente (más o menos) con la distancia entre los nivles de color entre los píxeles. Esto hacía que el coste de clasificar en la misma clase una región con píxeles muy parecidos pero estrecha, como la pata del caballo, coste que dependía como dijimos del perímetro de esa pata, no fuese tan pequeño como el coste de "amputar" la pata, el cual vendia de pagar un coste mayor píxel a píxel, pero a lo largo de muchos menos píxeles. Todo esto se arreglaría sí el coste de separar dos píxeles con colores muy parecidos se disparase exponencialmente, y el coste de separar dos píxeles no muy parecidos tendiese a cero exponencialmente. Y eso es precisamente lo que hacen funciones de afinidad exponencial como las vistas en el capítulo dedicado a "graph cuts" que hay en la bibliografía de la asignatura. Esa será la optimización que implementaremos, usar la siguiente función de afinidad exponencial: $v(1,0) = e^{-\frac{||x-y||}{\sigma}}$, donde x e y representan los colores de los píxeles adyacentes cuya afinidad estamos calculando. Sin embargo, lo que no haremos será experimentar con ese parametro σ para encontrar la solución óptima.

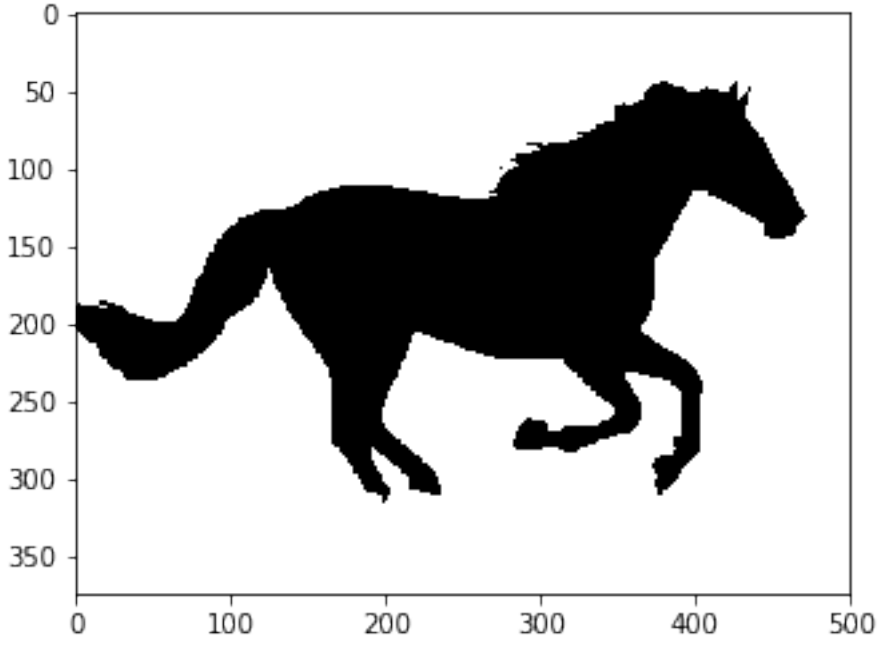
```
In [6]: ###################################
        # Segmentacion de imagen a la "Grab Cut" simplificado
        # por Luis Baumela. UPM. 15-10-2015
        # Vision por Computador. Master en Inteligencia Artificial
        ###################################

        # Comentarios en español del fichero original
        # Comments in english are new


        import numpy as np
        import maxflow
        import matplotlib.pyplot as plt
        import select_pixels as sel
        from math import sqrt

        def segmentation_optim(imgName):

            img = plt.imread(imgName)
            plt.imshow(img)
```

```
# Marco algunos pixeles que pertenecen el objeto y el fondo
markedImg = sel.select_fg_bg(img)
plt.imshow(markedImg)

v,h = img.shape[0], img.shape[1]

# 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

# Estos son los costes de los vecinos horizontales
zeros_v = np.zeros(3*v).reshape(v,1,3)
img_shift_h = np.concatenate((zeros_v,img[:,0:-1,:]),axis=1)
aux_h = img-img_shift_h
#exp_aff_h = 255 - 0.2989*aux_h[:,:,0] - 0.5870*aux_h[:,:,1] - 0.1140*aux_h[:,:,2]
exp_aff_h = np.exp(-np.linalg.norm(aux_h,axis=2)/1) # Is the norm the best for thi
#exp_aff_h = np.linalg.norm(aux_h,axis=2)# Is the norm the best for this


# Estos son los costes de los vecinos verticales
zeros_h = np.zeros(3*h).reshape(1,h,3)
img_shift_v = np.concatenate((zeros_h,img[0:-1,:,:]),axis=0)
aux_v = np.absolute(img-img_shift_v)
#exp_aff_v = 255 - 0.2989*aux_v[:,:,0] - 0.5870*aux_v[:,:,1] - 0.1140*aux_v[:,:,2]
exp_aff_v = np.exp(-np.linalg.norm(aux_v,axis=2)/1)
#exp_aff_v = np.linalg.norm(aux_v,axis=2)


# What is the pythonic way to do what we do above?

# 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 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)))
```

```
            mask_fg = np.all(np.equal(markedImg,(255,0,0)),2)
            mask_bg = np.all(np.equal(markedImg,(0,255,0)),2)

            exp_source = np.zeros((v,h))
            exp_source[mask_fg]=np.inf

            exp_sink = np.zeros((v,h))
            exp_sink[mask_bg] = np.inf

            # Incluyo las conexiones a los nodos terminales
            # Pesos de los nodos terminales
            g.add_grid_tedges(nodeids, exp_sink, exp_source)

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

In [8]:
```
caballo = 'images/horse.jpg'

segmentation_optim(caballo)
```


0
50
100
150
200
250
300
350
0
100
200
300
400
500


0
50
100
150
200
250
300
350
0
100
200
300
400
500


```
In [9]: persona = 'images/persona.png'

        segmentation_optim(persona)

        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-9-4200663cf6ab> in <module>()
          1 persona = 'images/persona.png'
          2
    ----> 3 segmentation_optim(persona)


        <ipython-input-6-3e9de1828d17> in segmentation_optim(imgName)
         21
         22     # Marco algunos pixeles que pertenecen el objeto y el fondo
    ---> 23     markedImg = sel.select_fg_bg(img)
         24     plt.imshow(markedImg)
         25


        /home/ignacio/Desktop/cv/pr1/select_pixels.py in select_fg_bg(img, radio)
         32
         33 #    imgpyg=pygame.image.load(imgName)
```

```
---> 34     imgpyg=pygame.image.frombuffer(img,img.shape[-2::-1],'RGB')
     35     screen.blit(imgpyg,(0,0))
     36     pygame.display.flip() # update the display
```

```
ValueError: Buffer length does not equal format and resolution size
```