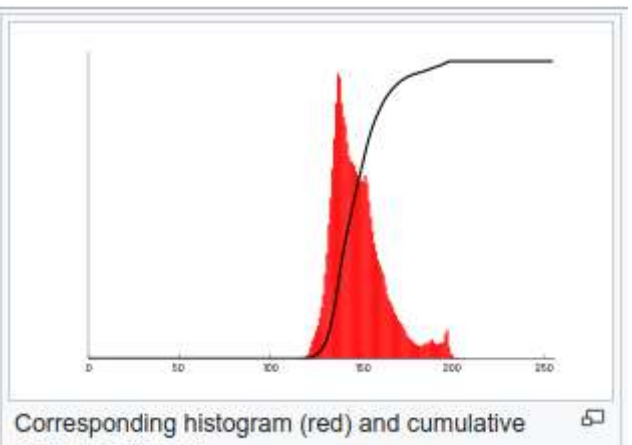
Digital Images Processing with Computer Vision

Work to do

Implement point transformation routines for image processing and analyze the results. Additionally, study concepts of image filtering, program routines for smoothing and edge extraction. Lastly, implement an image segmentation algorithm along with a line extraction algorithm based on the Hough transform.

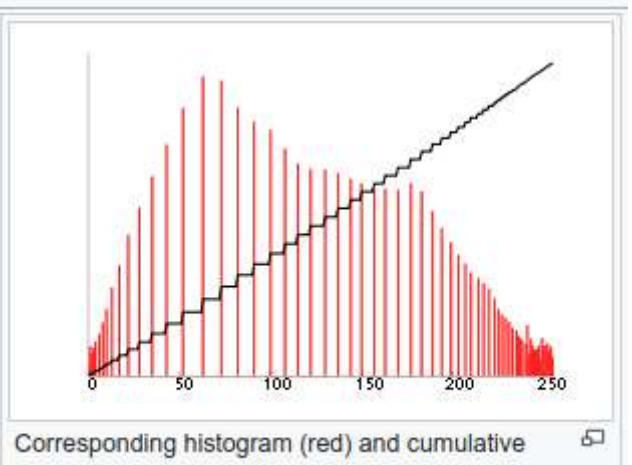
1. Puntual transformations

Before Histogram Equalization



Corresponding histogram (red) and cumulative histogram (black)

After Histogram Equalization



Corresponding histogram (red) and cumulative histogram (black)

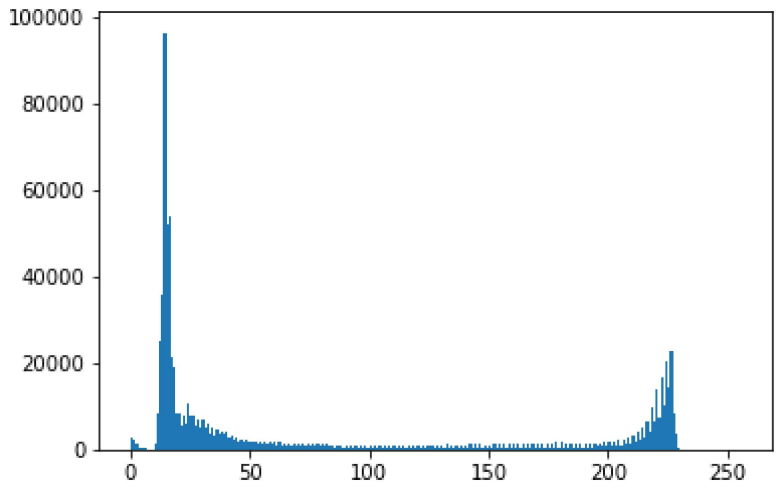
The idea here is to normalize the histogram for better distribution of the pixels values.

$$h(v) = round\left(\frac{cdf(v) - cdf_{min}}{(M \times N) - cdf_{min}} \times (L-1)\right) \quad (1)$$

where cdfmin is the minimum non-zero value of the cumulative distribution function, M × N gives the image's number of pixels (for the example above 64, where M is width and N the height) and L is the number of grey levels used (in most cases, like this one, 256).

Exercise 1. Load the image `escilum.tif`. Calculate and show the histogram, for example, using the function `hist()` from `matplotlib.pyplot`. Having the histogram, discuss what problem the image has in the lower left region.

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

%matplotlib inline

image = cv2.imread("/input/computer-vision-course/imagenes/escilum.tif")
# plt.imshow(image)

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



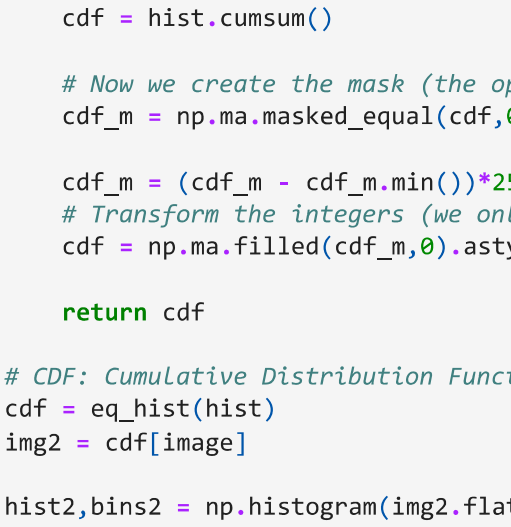
Results

The lower left region of the image is very dark and this is reflected in the histogram by the peak on the left side (near 0). The low contrast between the pixels makes it difficult to analyze what objects are in the region.

Exercise 2. Write a function `eq_hist(histogram)` that calculates the point transformation function that equalizes the histogram. Applies the transformation function to the above image. Calculates and redisplays the resulting histogram and image, as well as the transformation function.

Discuss the results obtained. What would be the result if we were to re-equalize the resulting image?

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

%matplotlib inline

image = cv2.imread("/input/computer-vision-course/imagenes/escilum.tif")
# plt.imshow(image)

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

def eq_hist(hist):
    # We calculate the distribution function
    cdf = hist.cumsum()

    # Now we create the mask (the operations will be applied to non marked numbers (those distinct to 0))
    cdf_m = np.ma.masked_equal(cdf,0)

    cdf_m = (cdf_m - cdf_m.min())*255/(cdf_m.max()-cdf_m.min())
    # Transform the integers (we only want the int part)
    cdf = np.ma.filled(cdf_m,0).astype('uint8')

    return cdf

# CDF: Cumulative Distribution Function
cdf = eq_hist(hist)
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()
```



Results

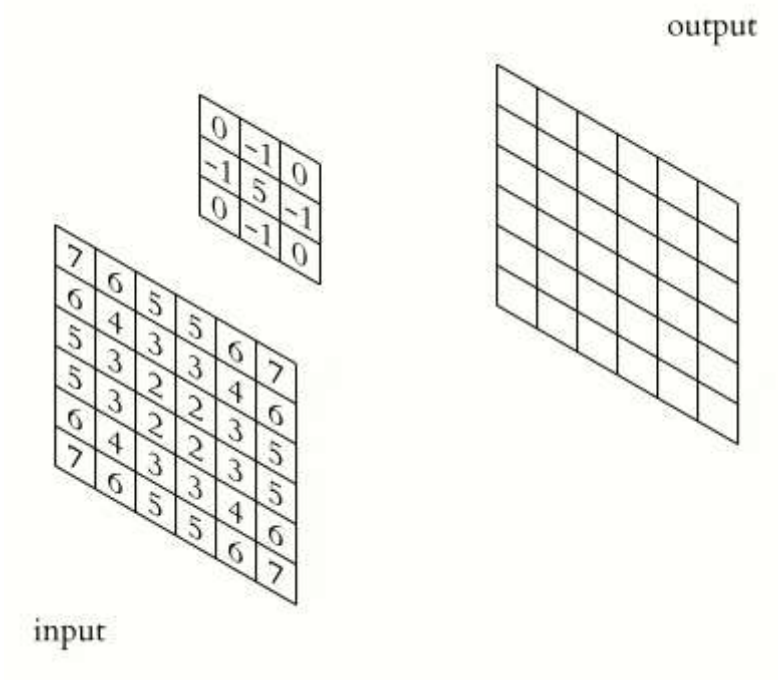
The result of equalizing the image more than once will always be the same, i.e. we will get the same histogram and the same image. In addition, we can see how in the result the grays are better distributed in the histogram and the image can be seen more clearly.

2. Filtering

Convolution involving one-dimensional signals is referred to as 1D convolution or just convolution. Otherwise, if the convolution is performed between two signals spanning along two mutually perpendicular dimensions (i.e., if signals are two-dimensional in nature), then it will be referred to as 2D convolution. This concept can be extended to involve multi-dimensional signals due to which we can have multi-dimensional convolution.

In the digital domain, convolution is performed by multiplying and accumulating the instantaneous values of the overlapping samples corresponding to two input signals, one of which is flipped. This definition of 1D convolution is applicable even for 2D convolution except that, in the latter case, one of the inputs is flipped twice.

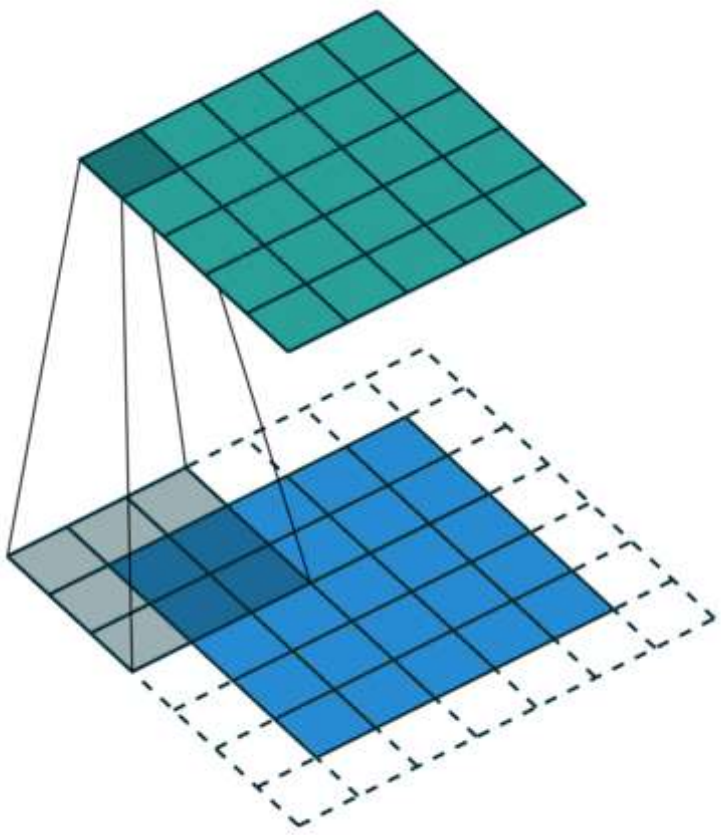
This kind of operation is extensively used in the field of digital image processing wherein the 2D matrix representing the image will be convolved with a comparatively smaller matrix called 2D kernel.



Depending on the parameteres of the `convolve()` function, you can either reduce, mantaine or increase resulting matrix size.

Stride is the size of the step the convolution filter moves each time. A stride size is usually 1, meaning the filter slides pixel by pixel. By increasing the stride size, your filter is sliding over the input with a larger interval and thus has less overlap between the cells.

The animation below shows stride size 1 in action:



To do the convolutions use the functions `convolve` or `convolve1d` from `scipy.ndimage.filters`.

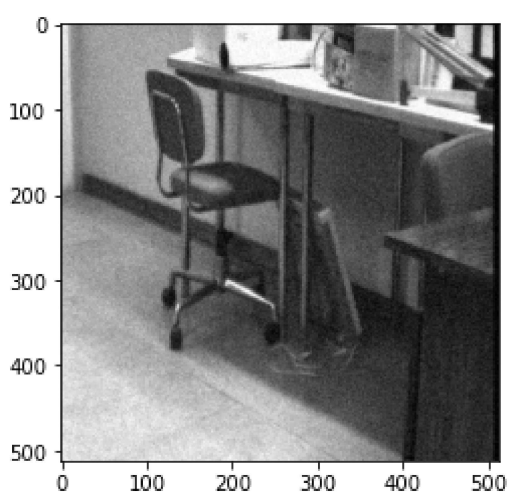
Load the images `escgaus.bmp` and `escimp5.bmp` which are contaminated respectively with Gaussian and impulsional noise. You can also use any other images that you find interesting.

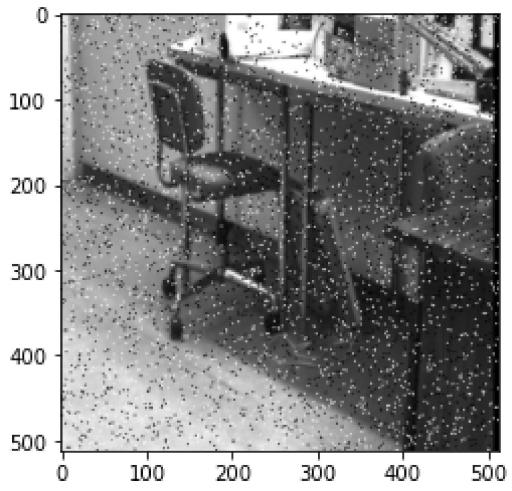
In [4]:
```
from scipy.ndimage.filters import convolve, convolve1d, gaussian_filter
import math
from skimage import io, viewer
from scipy import fftpack

img1 = cv2.imread('/input/computer-vision-course/imagenes/escgaus.bmp', 0)
img2 = cv2.imread('/input/computer-vision-course/imagenes/escimp5.bmp', 0)
```

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

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

```
/opt/conda/lib/python3.7/site-packages/skimage/viewer/utils/__init__.py:1: UserWarning: Recommended matplotlib backend
is `Agg` for full skimage.viewer functionality.
  from .core import *
```

Out[4]: <matplotlib.image.AxesImage at 0x7d8c674b7190>





Exercise 3. Write a function `masc_gaus(sigma, n)` that builds a 1-dimension mask for a gaussian filter with n shape and σ variance. Then filter the images in the previous exercise using bi-dimensional filters and diferent shapes n and variances σ.

In this exercise you have to implement the function that builds the mask. You cannot use functions that build the mask or perform the filtering automatically.

Show and discuss the results. Paint some of the constructed masks.

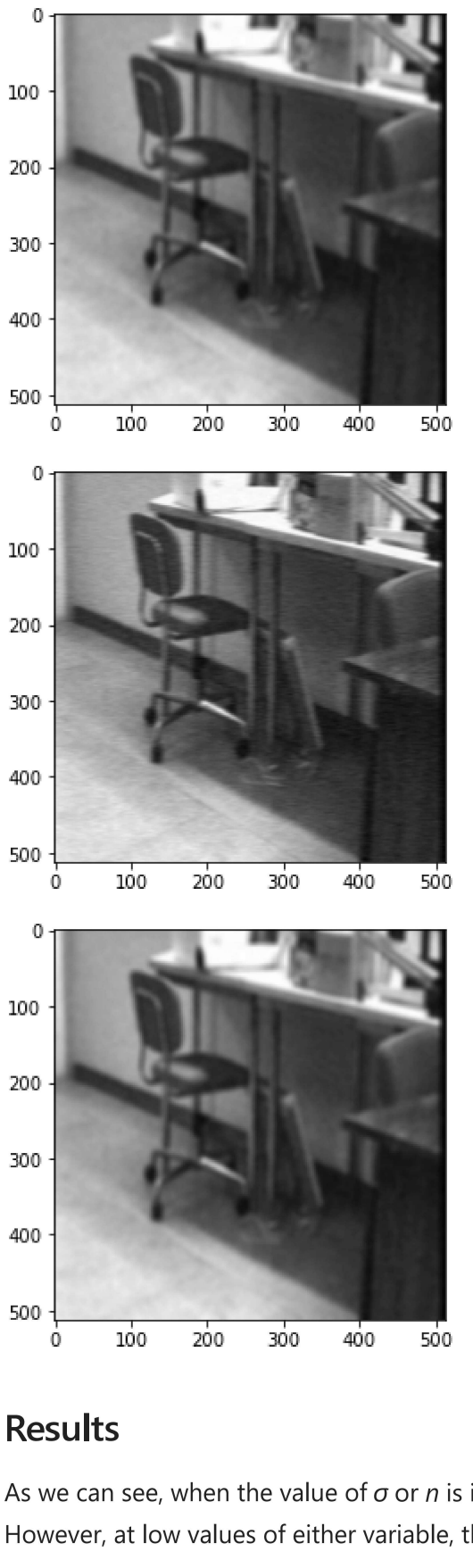
In [5]:
```
def masc_gaus_1d(sigma, n):
    width = n//2
    dx = 1
    x = np.arange(-width, width)
    kernel_1d = np.exp(-(x ** 2) / (2 * sigma ** 2))
    kernel_1d = kernel_1d / (math.sqrt(2 * np.pi) * sigma)

    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 [6]:
```
sigma = 5
n = 11

# I have use to compare:
# 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[6]: <matplotlib.image.AxesImage at 0x7d8c65b7ae90>



Results

As we can see, when the value of σ or n is increased, the image becomes more and more distorted until it becomes completely black. However, at low values of either variable, the image remains practically the same as in the original.

Exercise 4. Write a function `masc_deriv_gaus(sigma, n)` that constructs a one-dimensional mask of a filter derived from the Gaussian of size n and variance σ. Filter the image `corridor.jpg` with two-dimensional filters derived from the Gaussian to extract the edges of the image. Test it with different values for n and/or σ.

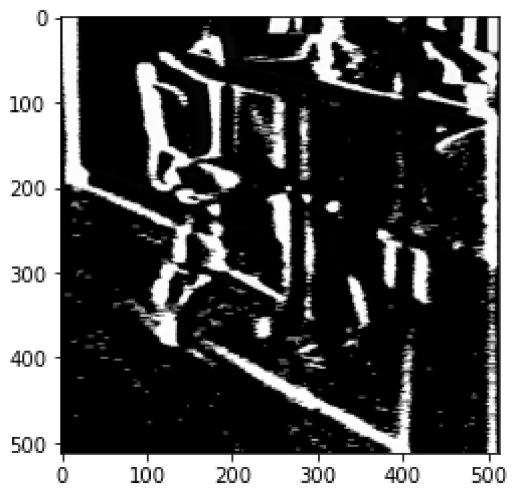
Show the results and discuss about them.

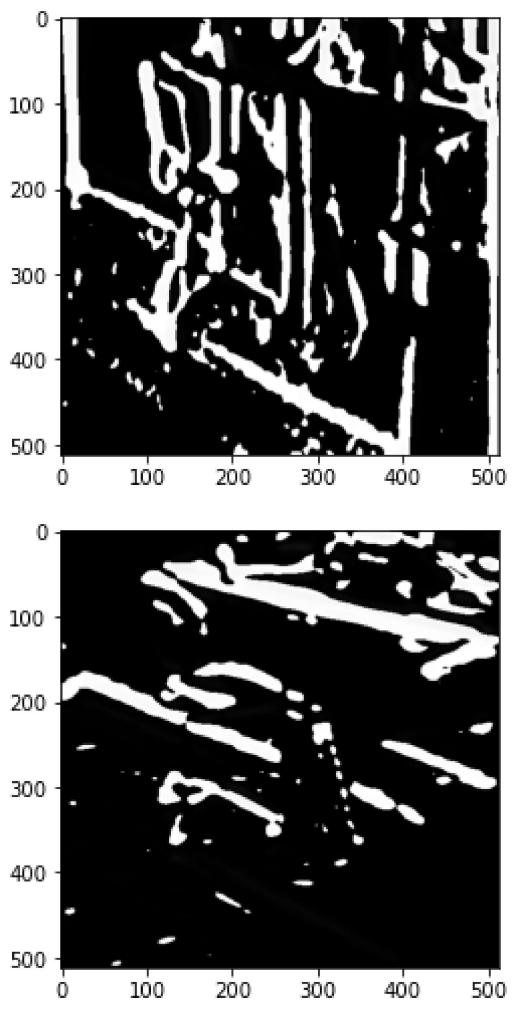
In [7]:
```
def masc_deriv_gaus_1d(sigma, n, order):
    width = n//2
    dx = 1
    x = np.arange(-width, width)
    kernel_1d = x * np.exp(-(x ** 2) / (2 * sigma ** 2))
    kernel_1d = -kernel_1d / (math.sqrt(2 * np.pi) * sigma ** 3)

    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 [8]:
```
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

# X-derivate
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)

# Y-derivate
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[8]: <matplotlib.image.AxesImage at 0x7d8c65a40dd0>





Results

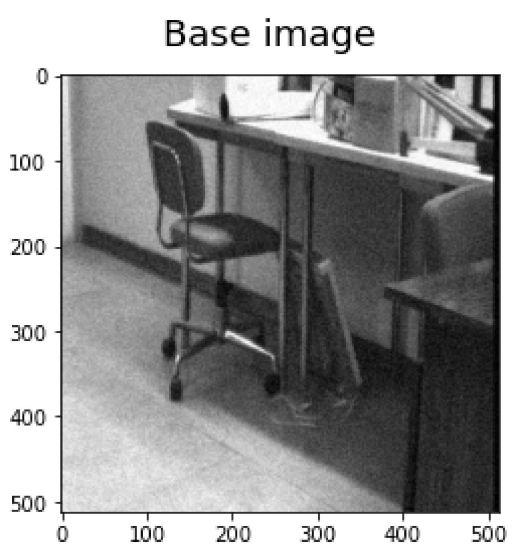
As we can see, the first order derivative gives us the edges of the image. The *x*-derivative gives us the vertical edges and the *y*-derivative gives us the horizontal edges.

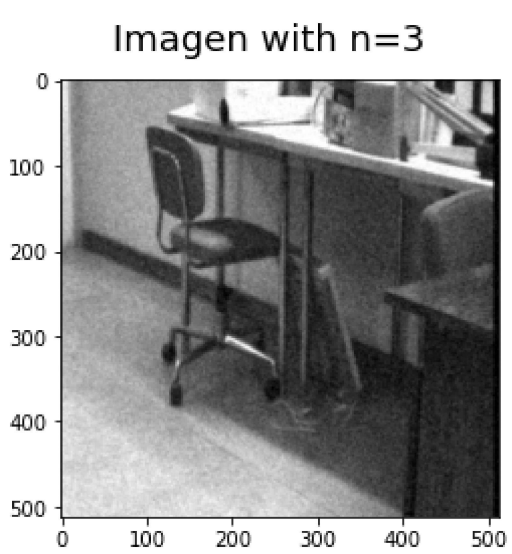
Exercise 5. Apply the median filter to the images `escgaus.bmp` and `escimp5.bmp` with different window sizes. Show and discusses the results

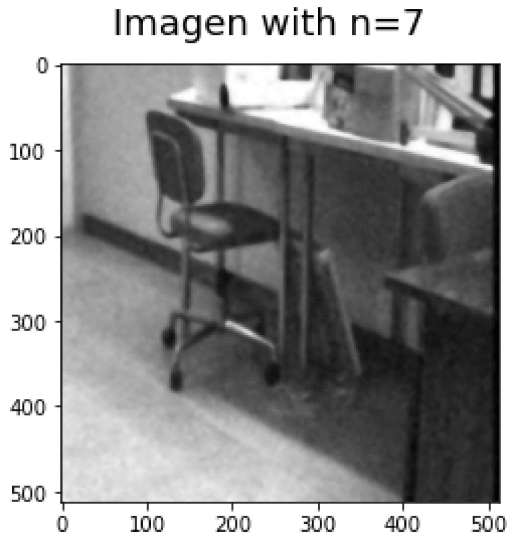
To do this exercise you can use `cv2.medianBlur()` from OpenCV, `scipy.ndimage.median_filter()` from SciPy or build your own function. To do it, create a function called `mediana(img, n)` and apply it to the to the image with the function: `scipy.ndimage.filters()`.

In [10]:
```
image = cv2.imread("/input/computer-vision-course/imagenes/escgaus.bmp")
fig = plt.figure()
fig.suptitle('Base image', fontsize=18)
plt.imshow(image)

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

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

image = cv2.imread("/input/computer-vision-course/imagenes/escimp5.bmp")

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

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

Out[10]:
```
<matplotlib.image.AxesImage at 0x7d8c65981050>
```

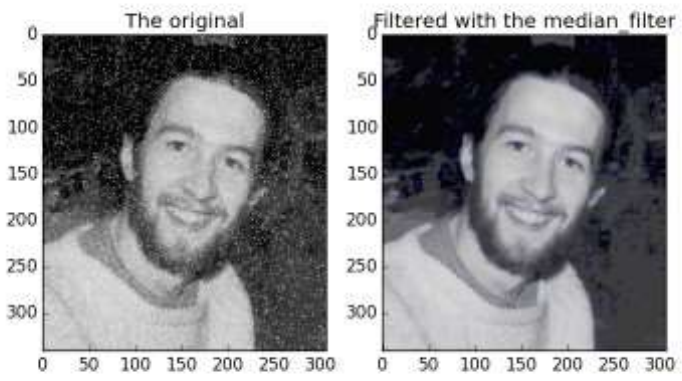






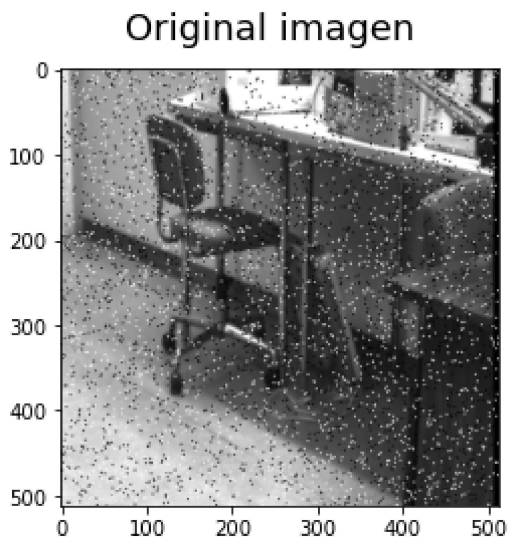
Median filter

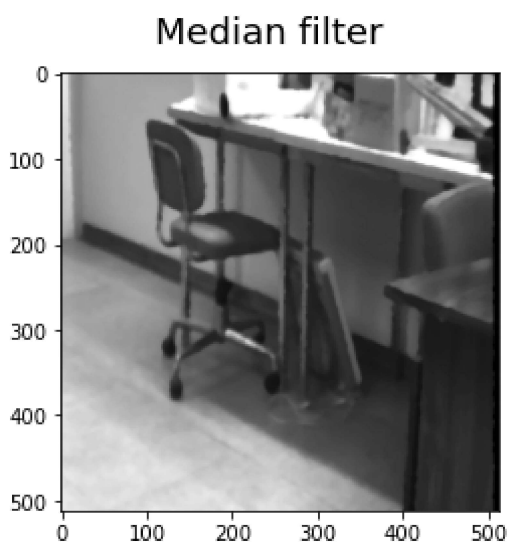
The median filter is a non-linear digital filtering technique, often used to remove noise from an image or signal. Such noise reduction is a typical pre-processing step to improve the results of later processing (for example, edge detection on an image). Median filtering is very widely used in digital image processing because, under certain conditions, it preserves edges while removing noise, also having applications in signal processing.



In [11]:
```
image = cv2.imread("/input/computer-vision-course/imagenes/escimp5.bmp", 0)
fig = plt.figure()
fig.suptitle('Original imagen', 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('Gaussian filter', fontsize=18)
plt.imshow(img_convolved,plt.cm.gray)

fig = plt.figure()
fig.suptitle('Median filter', fontsize=18)
plt.imshow(cv2.medianBlur(image,7),plt.cm.gray)
```

Out[11]:
```
<matplotlib.image.AxesImage at 0x7d8c67479910>
```





Results

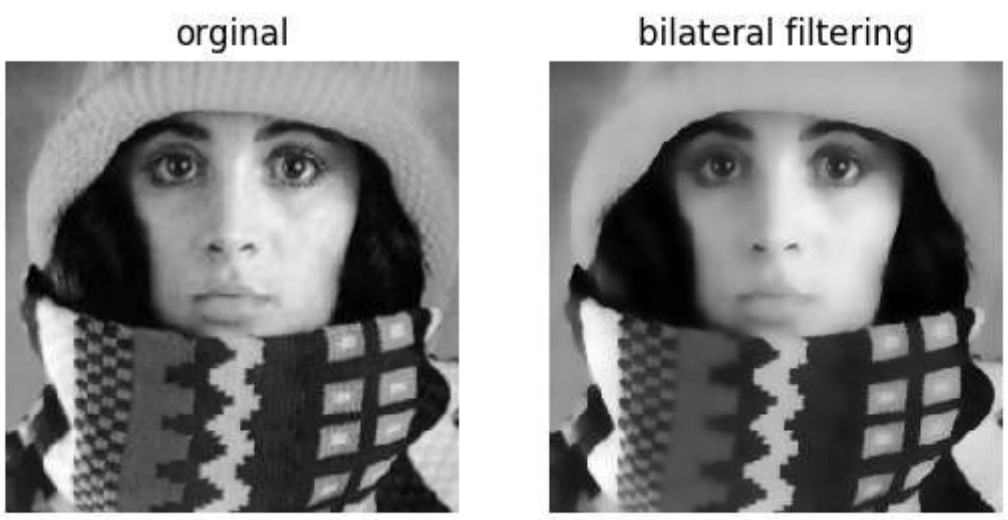
With images like `escgaus.bmp` the median filter gives results similar to a Gaussian filter. However, when further smoothing is attempted, the image becomes increasingly smeared.

In the case of images with `Salt & Pepper` noise, we can see that the median filter achives very good results, completely eliminating noise from the image.

As can be seen, the median filter has given better results than the Gaussian filter when filtering impulsive noises or the so-called `Salt & Pepper` noises.

Bilateral filter

A bilateral filter is a non-linear, edge-preserving, and noise-reducing smoothing filter for images. It replaces the intensity of each pixel with a weighted average of intensity values from nearby pixels. This weight can be based on a Gaussian distribution. Crucially, the weights depend not only on Euclidean distance of pixels, but also on the radiometric differences (e.g., range differences, such as color intensity, depth distance, etc.). This preserves sharp edges.



Exercise 6. Use the function `cv2.bilateralFilter()` from OpenCV to perform a bilateral filter on the image. Select the appropriate parameters and apply them to the images `escgaus.bmp` and `escimp5.bmp`.

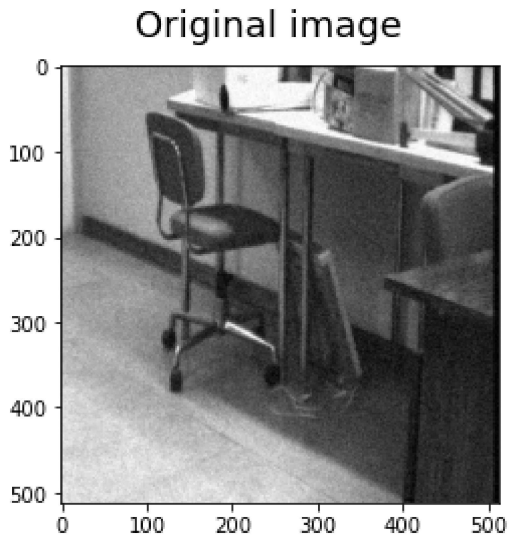
If we call σ_r to the variance of the Gaussian that controls the weighting due to the difference between pixel values and σ_s to the variance of the Gaussian that controls the weighting due to pixel position, answer thr following questions:

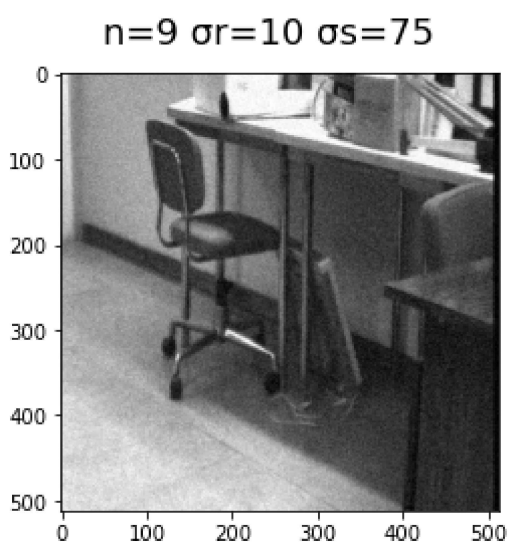
- How does the bilateral filter behave when variance σ_r is too high? In this case, what happens when σ_s is high or low?
- How does it behave when σ_r is too low? In this case, how does th filter behaves when σ_s is high and low?

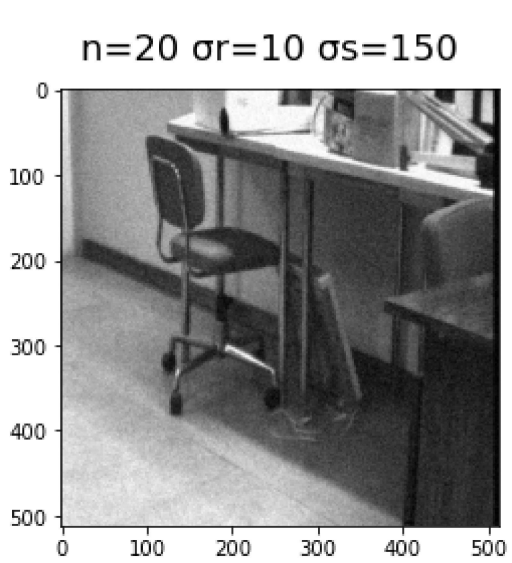
Show and discuss about the results with different parameter values.

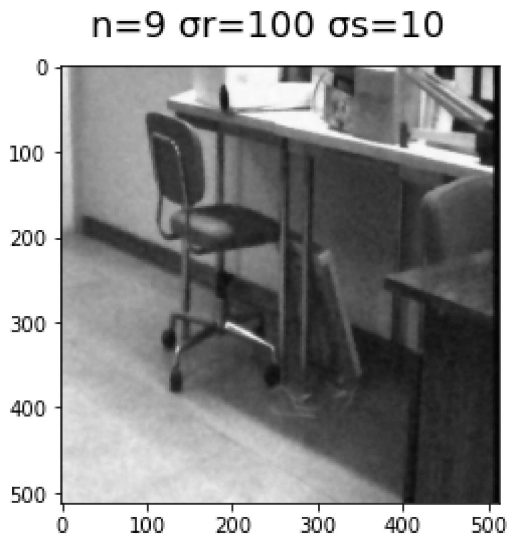
In [12]:
```
image = cv2.imread("/input/computer-vision-course/imagenes/escgaus.bmp")

fig = plt.figure()
fig.suptitle('Original image', fontsize=18)
plt.imshow(image)

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

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

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

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

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

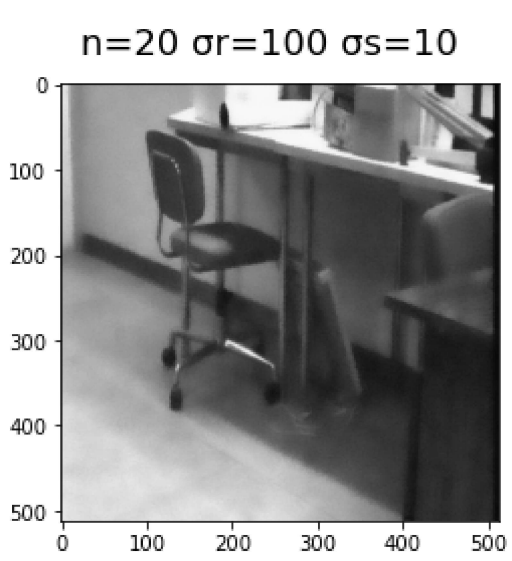
Out[12]:
```
<matplotlib.image.AxesImage at 0x7d8c656991d0>
```









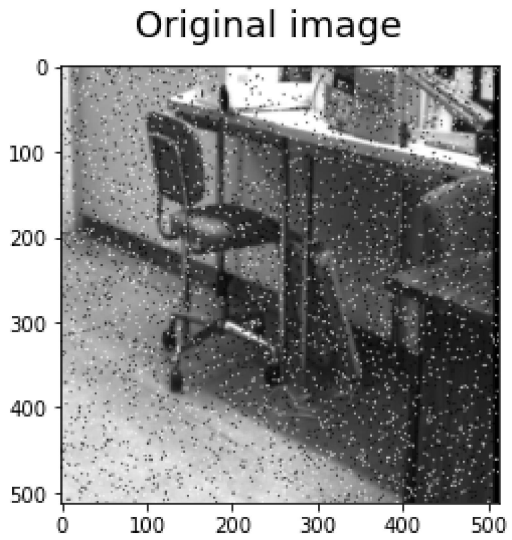


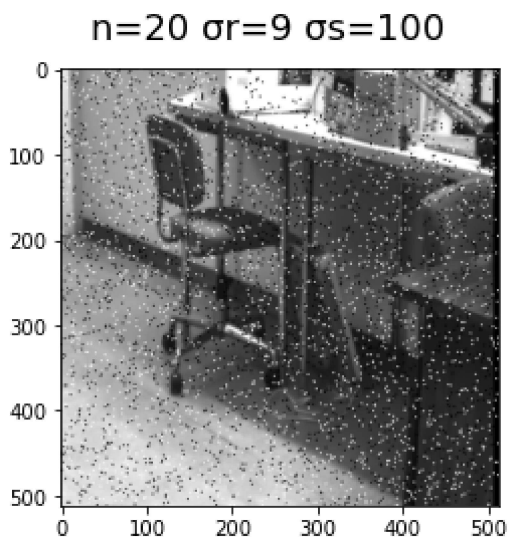
Result

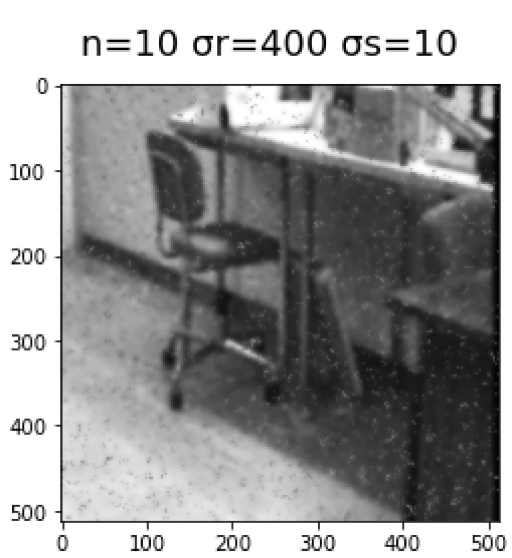
With `escgaus.bmp` and the bilateral filter we obtain better results than those seen with the Gaussian or the median filter. The σ_s configuration allows us to control the edges of the small structures we want to protect while sigma σ_r allows us to control the level of smoothing we want in the image.

In [13]:
```
image = cv2.imread("/input/computer-vision-course/imagenes/escimp5.bmp")
fig = plt.figure()
fig.suptitle('Original image', fontsize=18)
plt.imshow(image)

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

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

fig = plt.figure()
fig.suptitle('n=20 σr=200 σs=200', fontsize=18)
plt.imshow(cv2.bilateralFilter(image,20,200,200))

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

Out[13]:
```
<matplotlib.image.AxesImage at 0x7d8c654cee90>
```





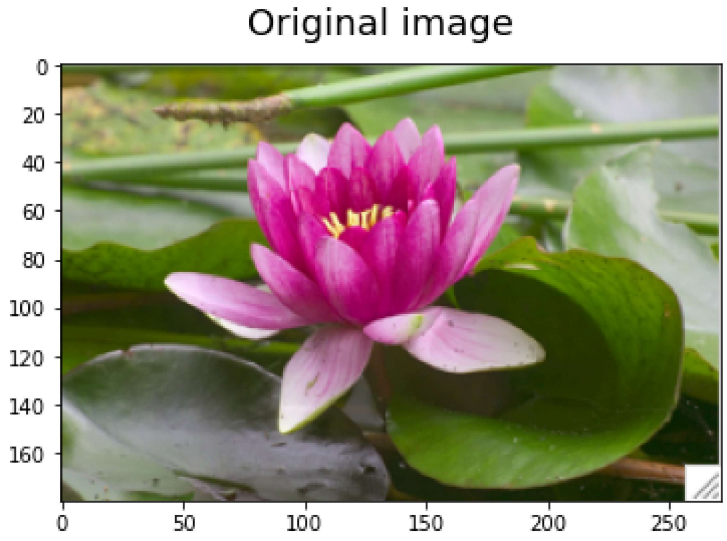


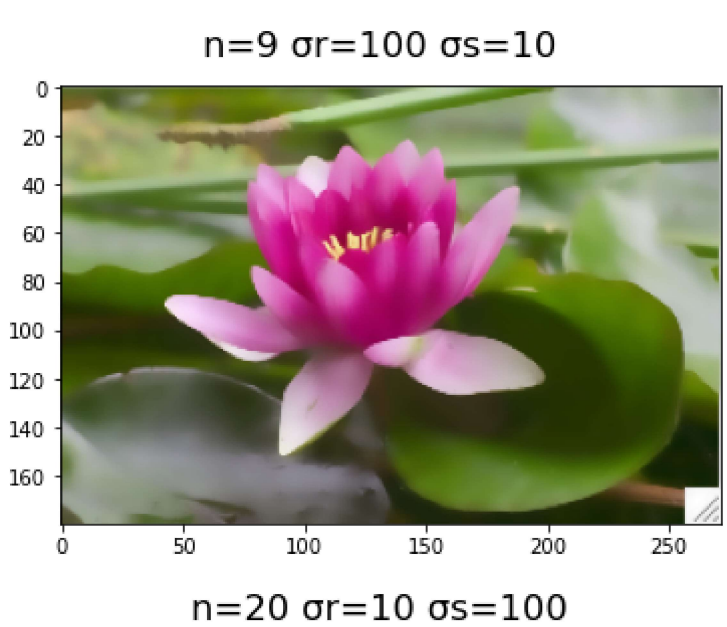
Result

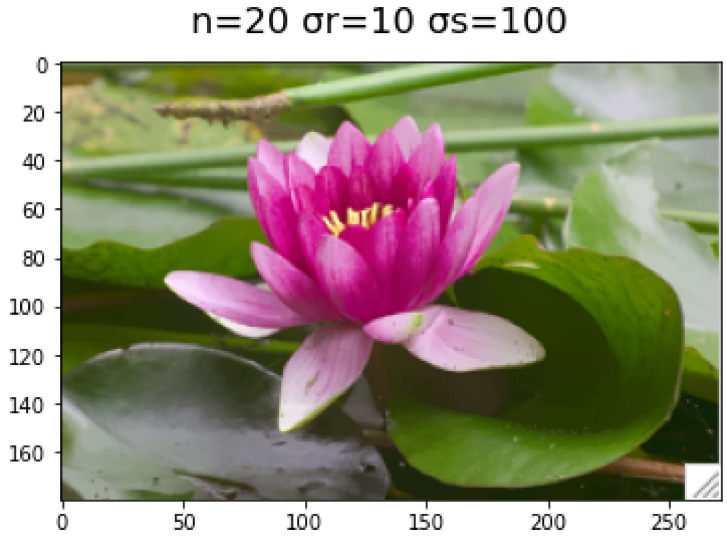
Unlike in the previous section, given an image with "Salt & Pepper" noise such as `escimp5.bmp` the bilateral filter does not give good results, being a better option to use the median filter to remove the "Salt & Pepper" noise.

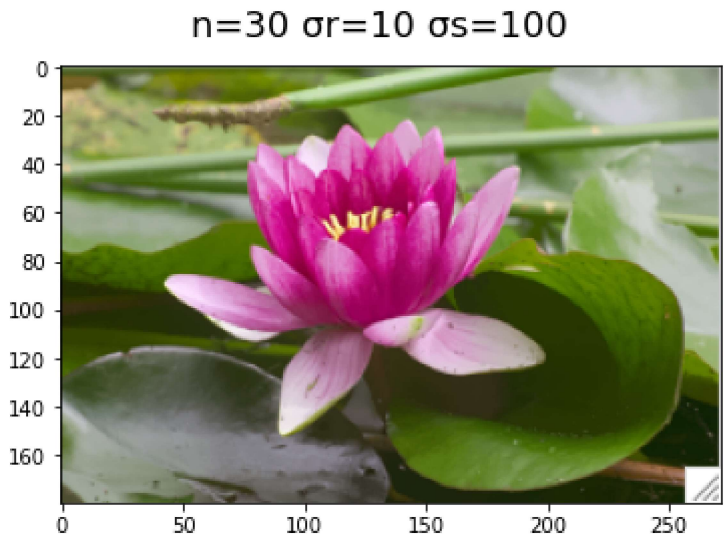
In [14]:
```
image = cv2.imread("/input/computer-vision-course/imagenes/flower.png")
fig = plt.figure()
fig.suptitle('Original image', fontsize=18)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))

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

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

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

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

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

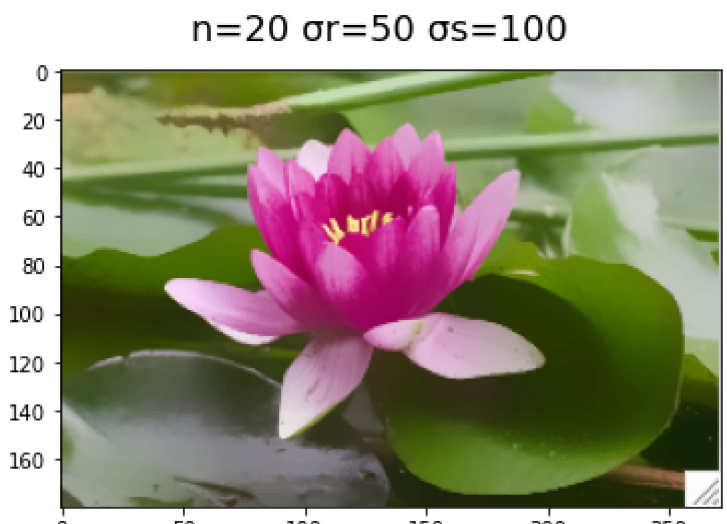
Out[14]:
```
<matplotlib.image.AxesImage at 0x7d8c6522cc10>
```

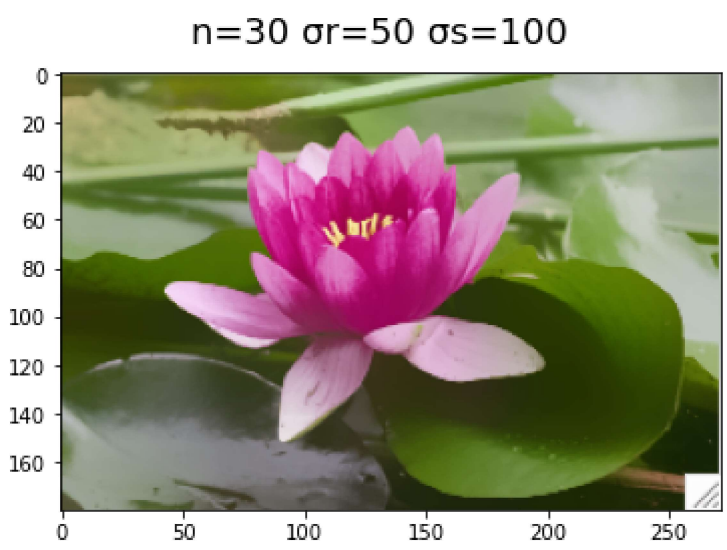












Answers

Answering the questions:

- How does the bilateral filter behave when variance σ_r is too high? In this case, what happens when σ_s is high or low?
- How does it behave when σ_r is too low? In this case, how does th filter behaves when σ_s is high and low?

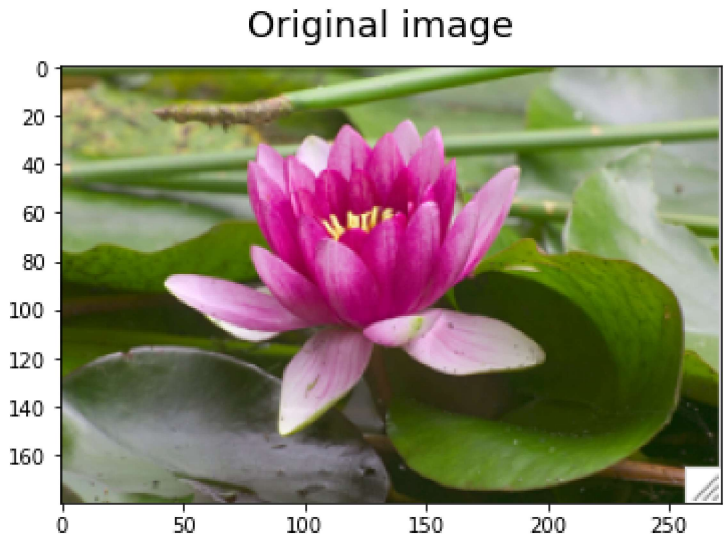
To answer this question, firstly we need to understand the parameters of the bilateral filter:

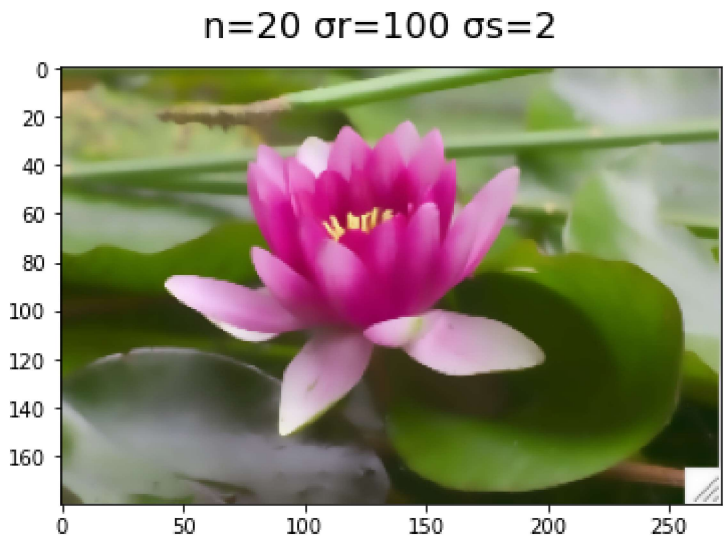
- As sigma sub-r (σ_r) becomes larger the bilateral filter behaves more and more like a Gaussian filter.
- As sigma sub-s (σ_s) gets larger the bilateral filter smooths smaller and smaller structures. With a small sigma sub-s (σ_s) these small structures and edges are protected in the smoothing.

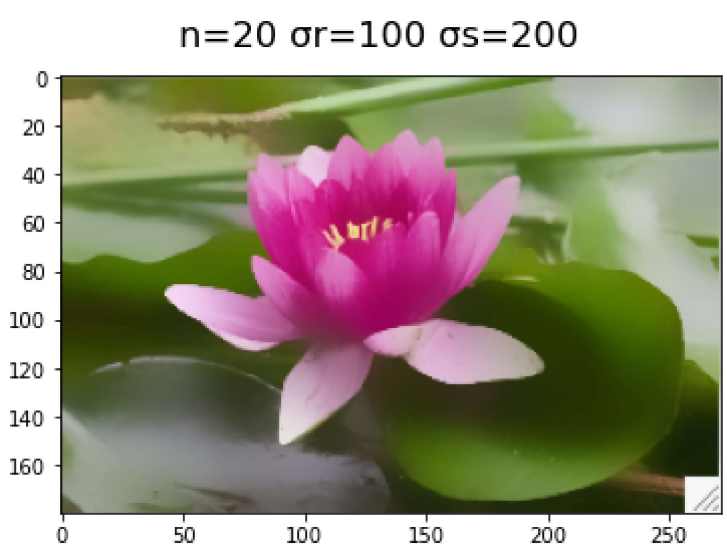
Lets test it with some examples:

In [15]:
```
image = cv2.imread("/input/computer-vision-course/imagenes/flower.png")
fig = plt.figure()
fig.suptitle('Original image', fontsize=18)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))

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

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

Out[15]:
```
<matplotlib.image.AxesImage at 0x7d8c65165050>
```







Result

To understand the filter, let's compare the second and third images by looking at the green leaf in the lower right corner. As you can see:

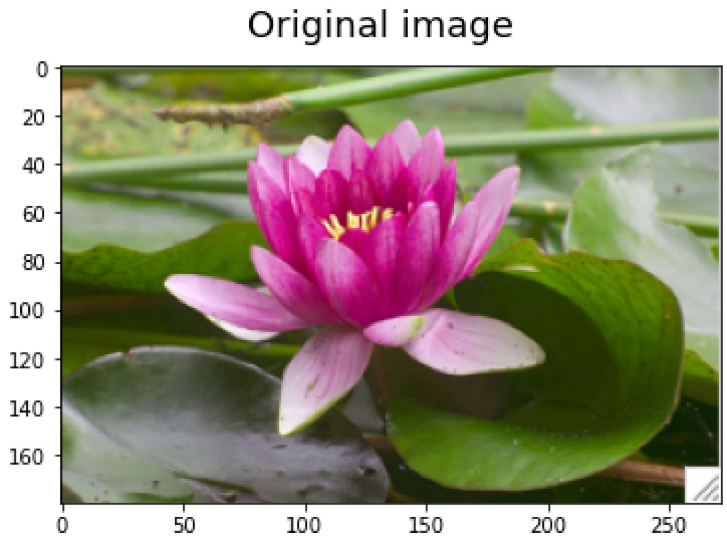
- By having a high sigma sub-s (σ_s = 200, third image) the edges of the smaller structures are ignored and disappears, as can be seen in the green leaf.
- Having a low sigma sub-s (σ_s = 2, second image) protects the edges of the smaller structures when smoothing the image. This can be seen by looking at the green leaf in the second image and comparing it with the one in the third image..

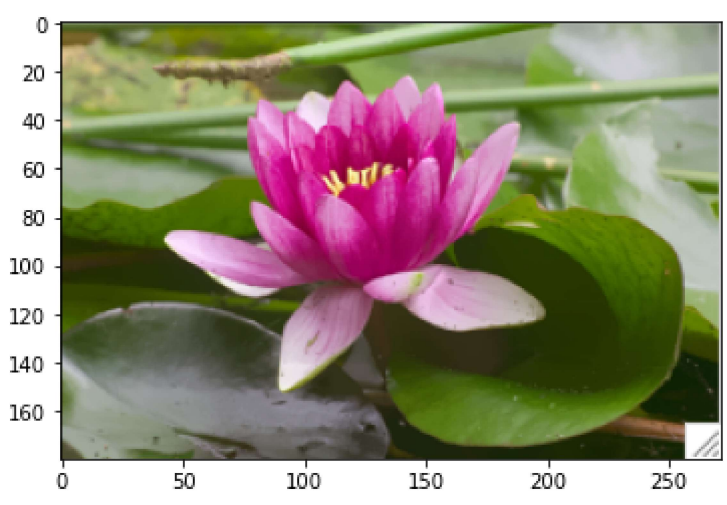
Comparing the second image with the third we can see that the edges have been protected much better with a low σ_s.

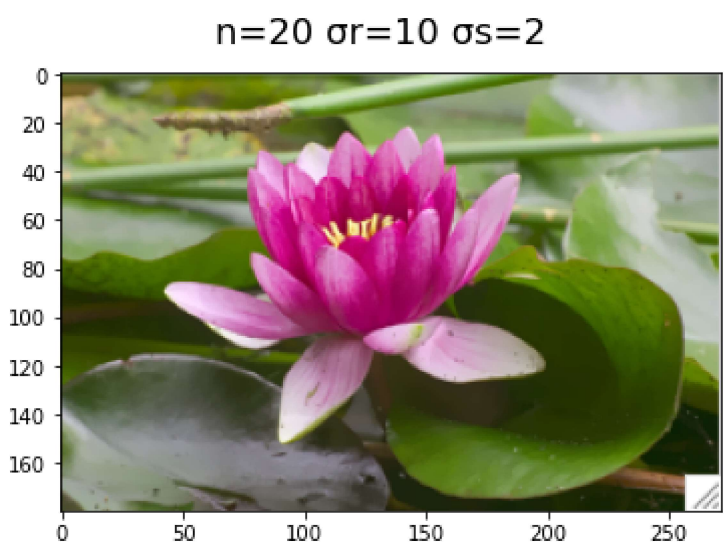
Having a high σ_r affects the level of image smoothing. This can be seen in the following code fragment:

In [16]:
```
image = cv2.imread("/input/computer-vision-course/imagenes/flower.png")
fig = plt.figure()
fig.suptitle('Original image', fontsize=18)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))

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

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

Out[16]:
```
<matplotlib.image.AxesImage at 0x7d8c655bf910>
```







Result

Comparing the results of this code fragment with the results of the previous code fragment we can see that if σ_r acquires a high value the image is smoothed more intensely. When σ_r acquires a very high value it starts to behave like a Gaussian mask, but when it acquires a low value the filter does not soften the image at all.

On the other hand, in this example with a low σ_r we can see that even if σ_s acquires very high or very low values the result is still the same; since the image is almost not smoothed, the edges of the image are not altered and the resulting image is very similar to the initial image. We can see that with a $\sigma_r = 10$ the shadow of the lower right leaf has been smoothed, but as it has been a very soft smoothing the edges have not been altered and both in the image with high σ_s and in the one with low σ_s the edges are still maintained.

3. Hough transform

Exercise 7. Use the Hough transform to find linear segments in the image `corridor.jpg`. To extract the edges of the image, use the functions written in exercises 3 and 4. Use the function `cv2.HoughLinesP()` from OpenCV.

Tune the parameteres and use different functions to extract the edges. Then discuss about the results.

In [17]:
```python
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)
```

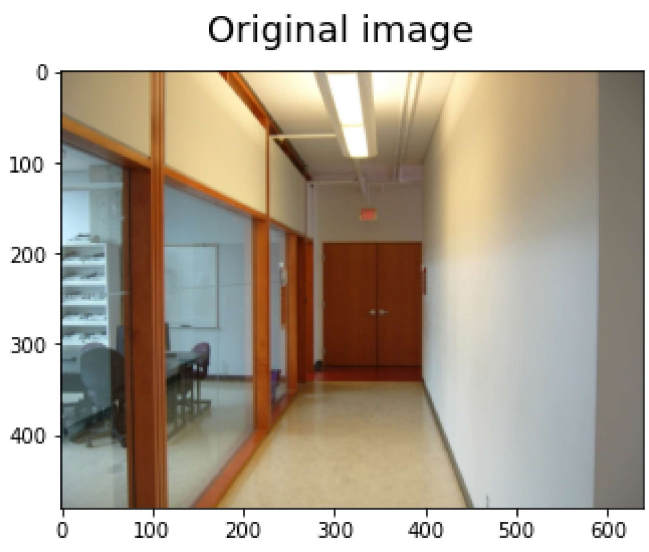
```
    kernel_2d = np.exp(-(x2d ** 2 + y2d ** 2) / (2 * sigma ** 4)) / (2 * np.pi * sigma ** 4)

    # 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
```

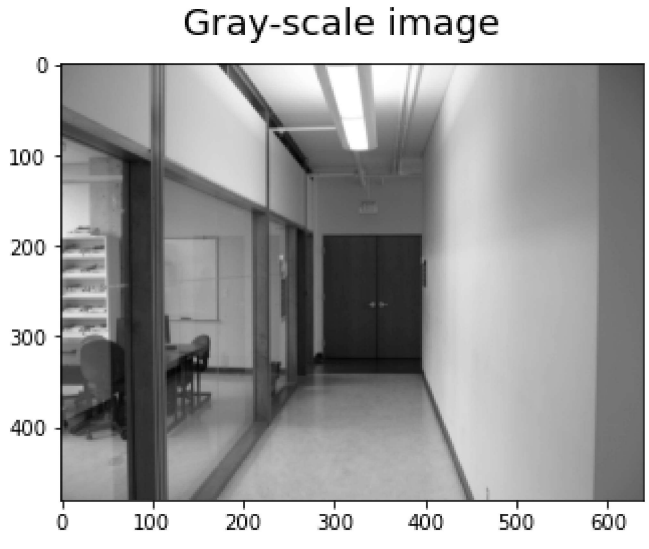
In [18]:

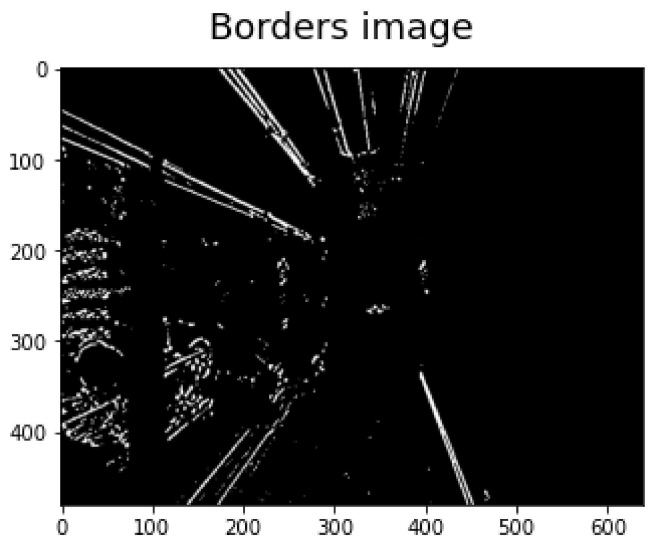
```
from scipy.ndimage.filters import convolve, convolve1d, gaussian_filter

sigma = 1.08
n = 35

image = cv2.cvtColor(cv2.imread('/input/computer-vision-course/imagenes/corridor.jpg'), cv2.COLOR_BGR2RGB)
fig = plt.figure()
fig.suptitle('Original image', fontsize=18)
plt.imshow(image)

gray = cv2.cvtColor(image,cv2.COLOR_RGB2GRAY)
fig = plt.figure()
fig.suptitle('Gray-scale image', 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('Borders image', fontsize=18)
plt.imshow(edges, cmap = plt.cm.gray)

lines = cv2.HoughLinesP(edges,rho = 1,theta = 1*np.pi/180,threshold = 100,minLineLength = 100,maxLineGap = 50)
for line in lines:
    for x1,y1,x2,y2 in line:
        cv2.line(image,(x1,y1),(x2,y2),(0,255,0),2)

cv2.imwrite('/input/computer-vision-course/imagenes/corridor_hough.jpg',image)
fig = plt.figure()
fig.suptitle('Hough transform image', fontsize=18)
plt.imshow(cv2.imread('/input/computer-vision-course/imagenes/corridor_hough.jpg'))
```

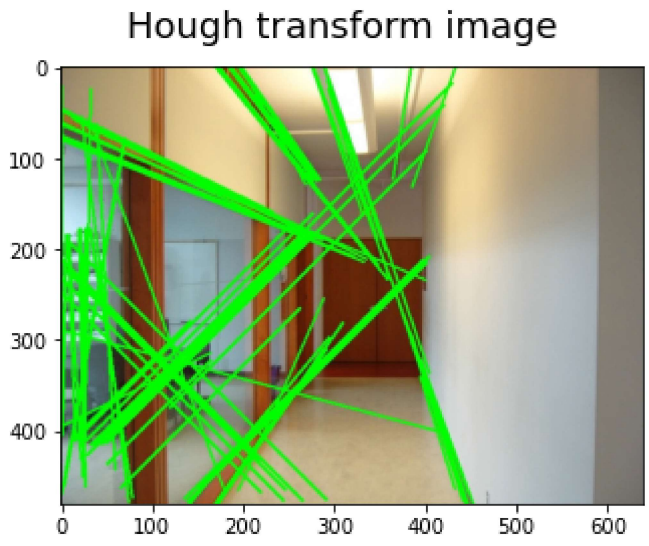
Out[18]:

```
<matplotlib.image.AxesImage at 0x7d8c67456890>
```



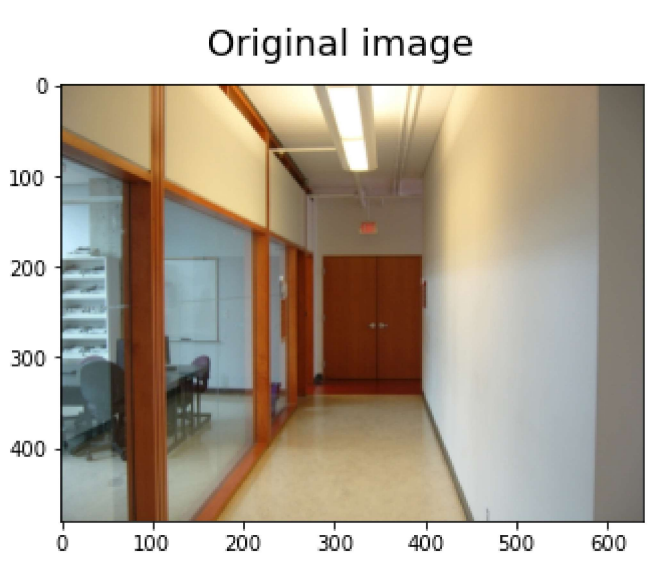


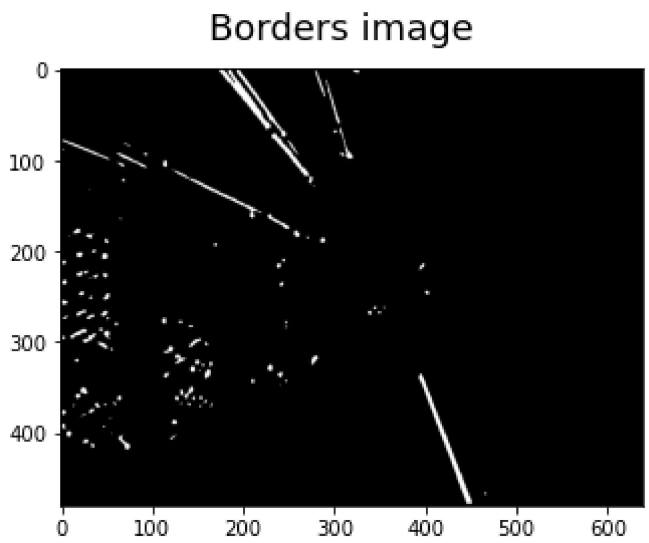


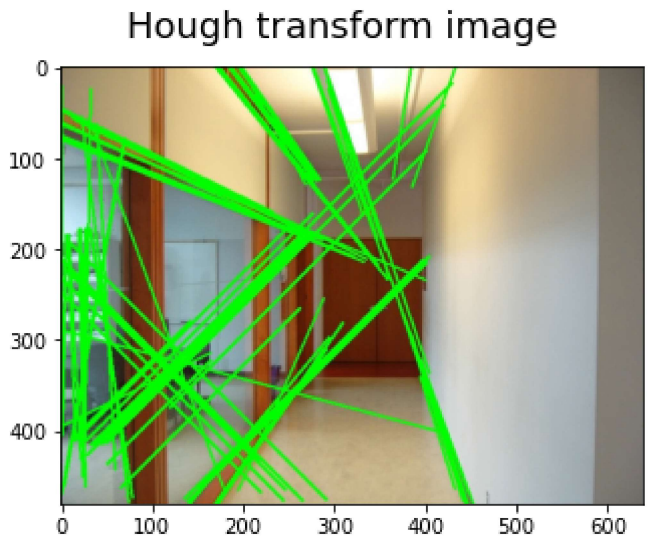


In [19]:
```
from scipy.ndimage.filters import convolve, convolve1d, gaussian_filter

sigma = 2.08
n = 35

image = cv2.cvtColor(cv2.imread('/input/computer-vision-course/imagenes/corridor.jpg'), cv2.COLOR_BGR2RGB)
fig = plt.figure()
fig.suptitle('Original image', 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('Borders image', fontsize=18)
plt.imshow(edges, cmap = plt.cm.gray)

lines = cv2.HoughLinesP(edges,rho = 1,theta = 1*np.pi/180,threshold = 100,minLineLength = 100,maxLineGap = 50)
for line in lines:
    for x1,y1,x2,y2 in line:
        cv2.line(image,(x1,y1),(x2,y2),(0,255,0),2)

cv2.imwrite('/input/computer-vision-course/imagenes/corridor_hough.jpg',image)
fig = plt.figure()
fig.suptitle('Hough transform image', fontsize=18)
plt.imshow(cv2.imread('/input/computer-vision-course/imagenes/corridor_hough.jpg'))
```

Out[19]: <matplotlib.image.AxesImage at 0x7d8c6596cd90>

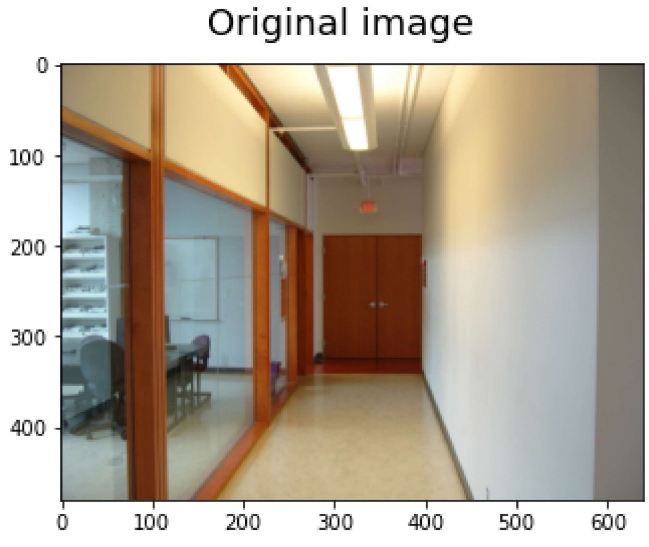


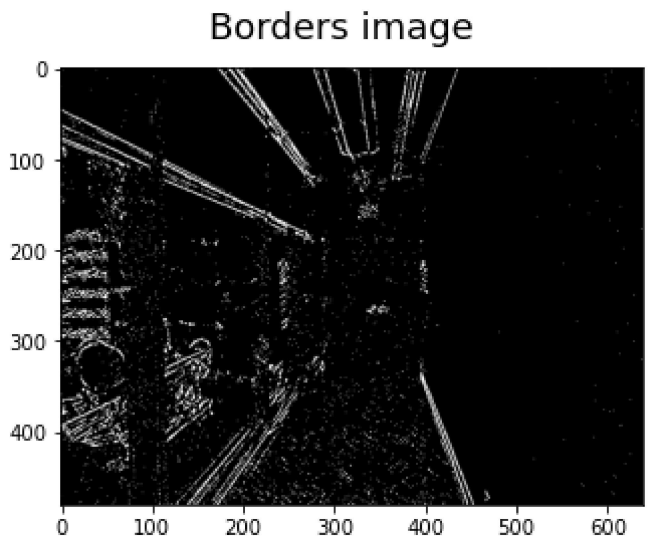


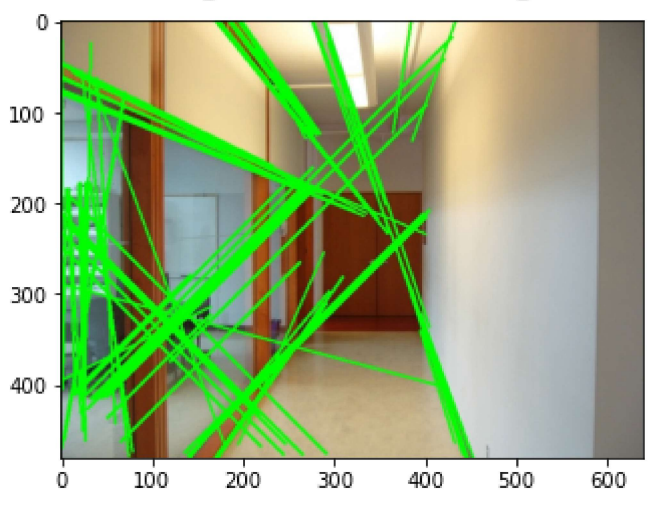


In [20]:
```
from scipy.ndimage.filters import convolve, convolve1d, gaussian_filter

sigma = 0.5
n = 10

image = cv2.cvtColor(cv2.imread('/input/computer-vision-course/imagenes/corridor.jpg'), cv2.COLOR_BGR2RGB)
fig = plt.figure()
fig.suptitle('Original image', 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('Borders image', fontsize=18)
plt.imshow(edges, cmap = plt.cm.gray)

lines = cv2.HoughLinesP(edges,rho = 1,theta = 1*np.pi/180,threshold = 100,minLineLength = 100,maxLineGap = 50)
for line in lines:
    for x1,y1,x2,y2 in line:
        cv2.line(image,(x1,y1),(x2,y2),(0,255,0),2)

cv2.imwrite('/input/computer-vision-course/imagenes/corridor_hough.jpg',image)
fig = plt.figure()
fig.suptitle('Hough transform image', fontsize=18)
plt.imshow(cv2.imread('/input/computer-vision-course/imagenes/corridor_hough.jpg'))
```

Out[20]: <matplotlib.image.AxesImage at 0x7d8c652f0d50>







Results

Playing with the different parameters we get some effects or others; with a low sigma we get more edges, while with a high sigma we will find more selected and clearer edges.

Optional

Now lets try the same with `cv.Canny()` to find the edges in the sudoku picture.

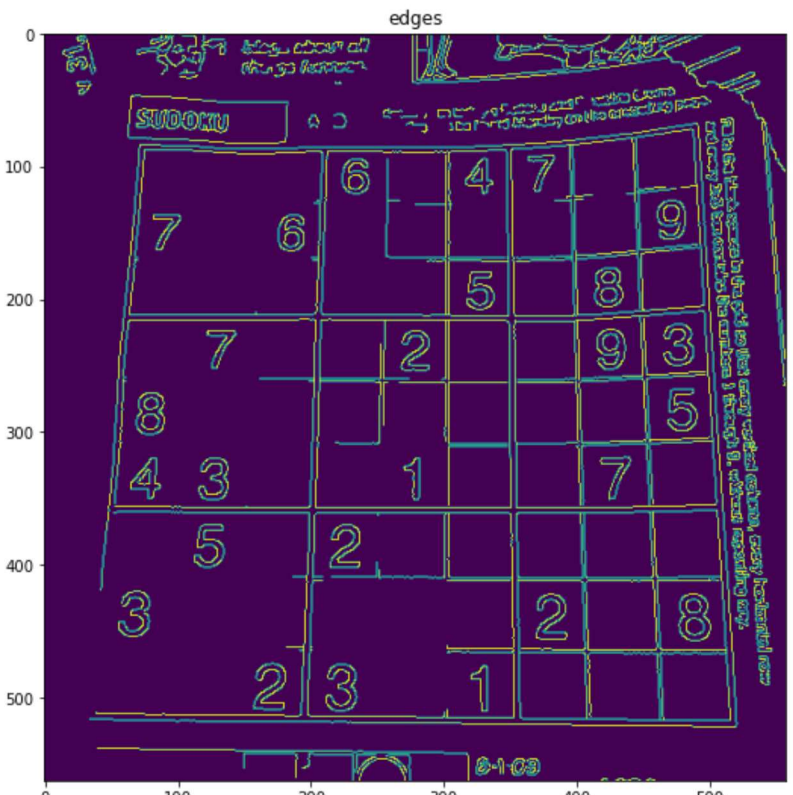
- `cv.line()` : https://docs.opencv.org/master/d6/d6e/group__imgproc__draw.html#ga7078a9fae8c7e7d13d24dac2520ae4a2
- `cv.HoughLines()` : https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_houghlines/py_houghlines.html

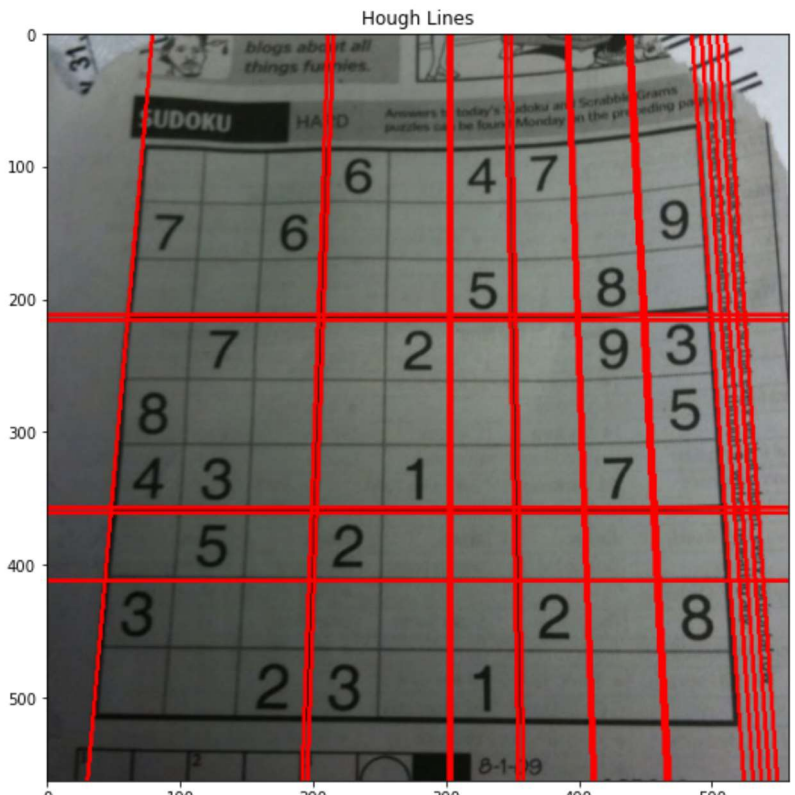
> Everything explained above is encapsulated in the OpenCV function, `cv.HoughLines()`. It simply returns an array of (ρ, θ) values. ρ is measured in pixels and θ is measured in radians. **First parameter**, Input image should be a binary image, so apply threshold or use `cv.Canny()` edge detection before finding applying hough transform. **Second and third parameters** are ρ and θ accuracies respectively. **Fourth argument** is the threshold, which means minimum vote it should get for it to be considered as a line. Remember, number of votes depend upon number of points on the line. So it represents the minimum length of line that should be detected.

In [21]:
```
image = cv2.imread('/input/opencv-samples-images/data/sudoku.png')
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
plt.figure(figsize=(20, 20))

# Grayscale and Canny Edges extracted
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
edges = cv2.Canny(gray, 100, 170, apertureSize = 3)

plt.subplot(2, 2, 1)
plt.title("edges")
plt.imshow(edges)

# Run HoughLines using a rho accuracy of 1 pixel
# Theta accuracy of np.pi / 180
# Our line threshold is set to 200 (number of points on line)
lines = cv2.HoughLines(edges, 1, np.pi/180, 200)

# We iterate through each line and convert it to the format
# required by cv.lines
for line in lines:
    rho, theta = line[0]

    a = np.cos(theta)
    b = np.sin(theta)

    x0 = a * rho
    y0 = b * rho
    x1 = int(x0 + 1000 * (-b))
    y1 = int(y0 + 1000 * (a))
    x2 = int(x0 - 1000 * (-b))
    y2 = int(y0 - 1000 * (a))
```

```
    cv2.line(image, (x1, y1), (x2, y2), (255, 0, 0), 2)


plt.subplot(2, 2, 2)
plt.title("Hough Lines")
plt.imshow(image)
```

Out[21]: <matplotlib.figure.AxesImage at 0x7d8c65782710>

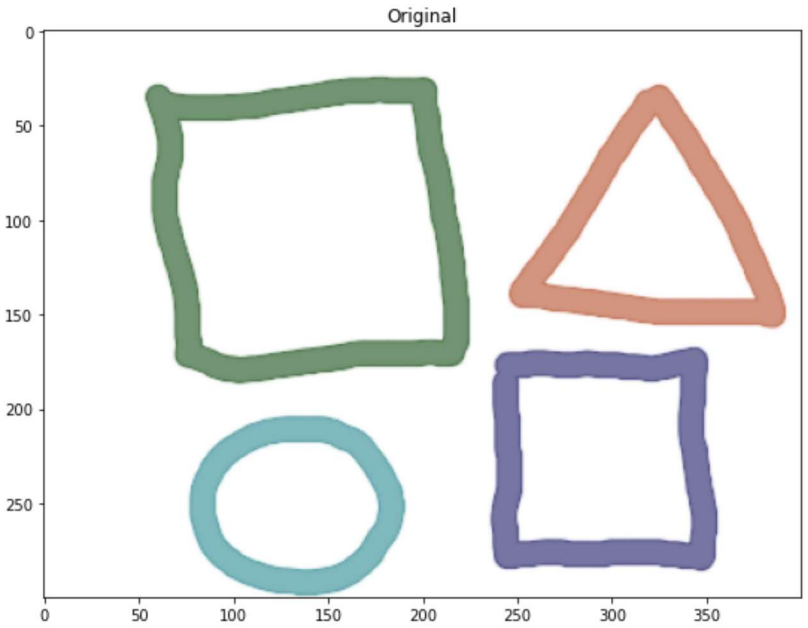


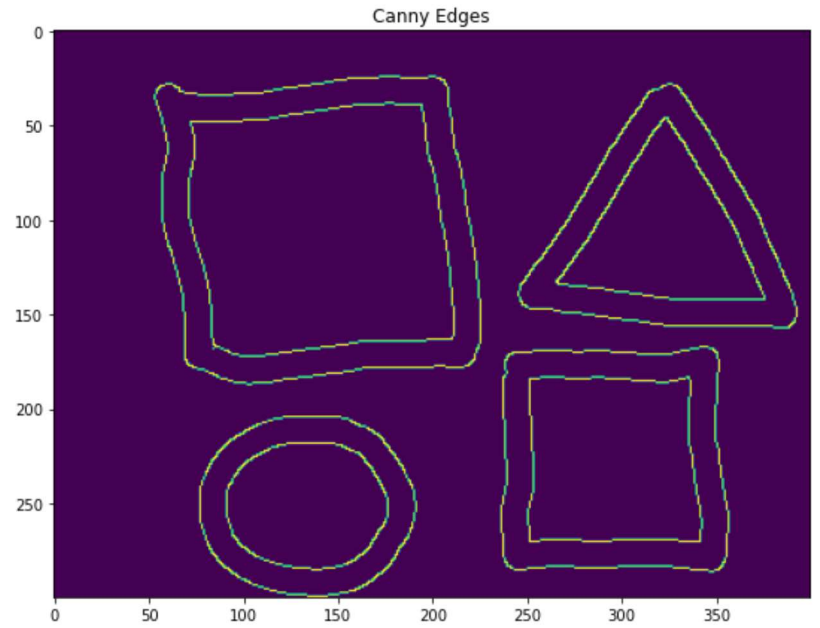


Exercise 8. In this exercise we will apply the knowledge from the last exercise to find the contours of some figures in the image `pic3.png`. To extract the edges of the image and the contours, use the functions `cv2.Canny()` and `cv2.findContours()` from OpenCV.

In [22]:
```
# Let's load an image with some figures
image = cv2.imread('/kaggle/input/opencv-samples-images/data/pic3.png')
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

plt.figure(figsize=(20, 20))

plt.subplot(2, 2, 1)
plt.title("Original")
plt.imshow(image)


# Grayscale
gray = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)

# Find Canny edges
edged = cv2.Canny(gray, 30, 200)

plt.subplot(2, 2, 2)
plt.title("Canny Edges")
plt.imshow(edged)


# Finding Contours
# Use a copy of your image e.g. edged.copy(), since findContours alters the image
contours, hierarchy = cv2.findContours(edged, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)

plt.subplot(2, 2, 3)
plt.title("Canny Edges After Contouring")
plt.imshow(edged)

print("Number of figures (contours) found = " + str(len(contours)))

# Draw all contours
# Use '-1' as the 3rd parameter to draw all
cv2.drawContours(image, contours, -1, (0,255,0), 3)

plt.subplot(2, 2, 4)
plt.title("Contours")
plt.imshow(image)
```

Out[22]:
```
Number of figures (contours) found = 4
<matplotlib.figure.AxesImage at 0x7d8c657ee5d0>
```

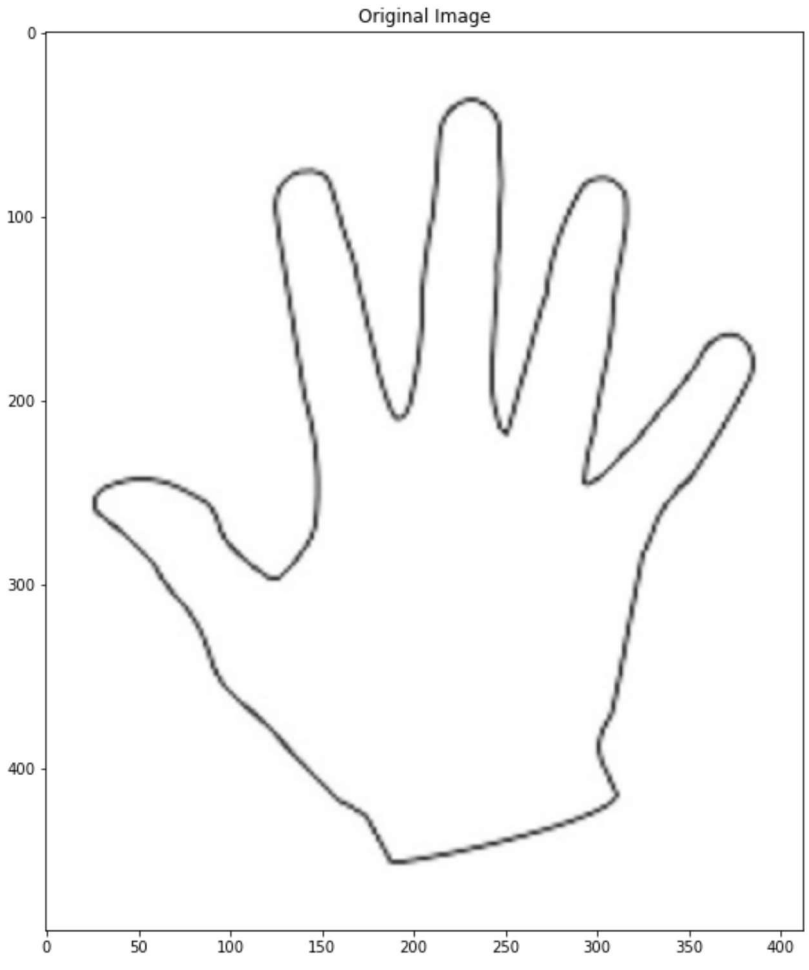


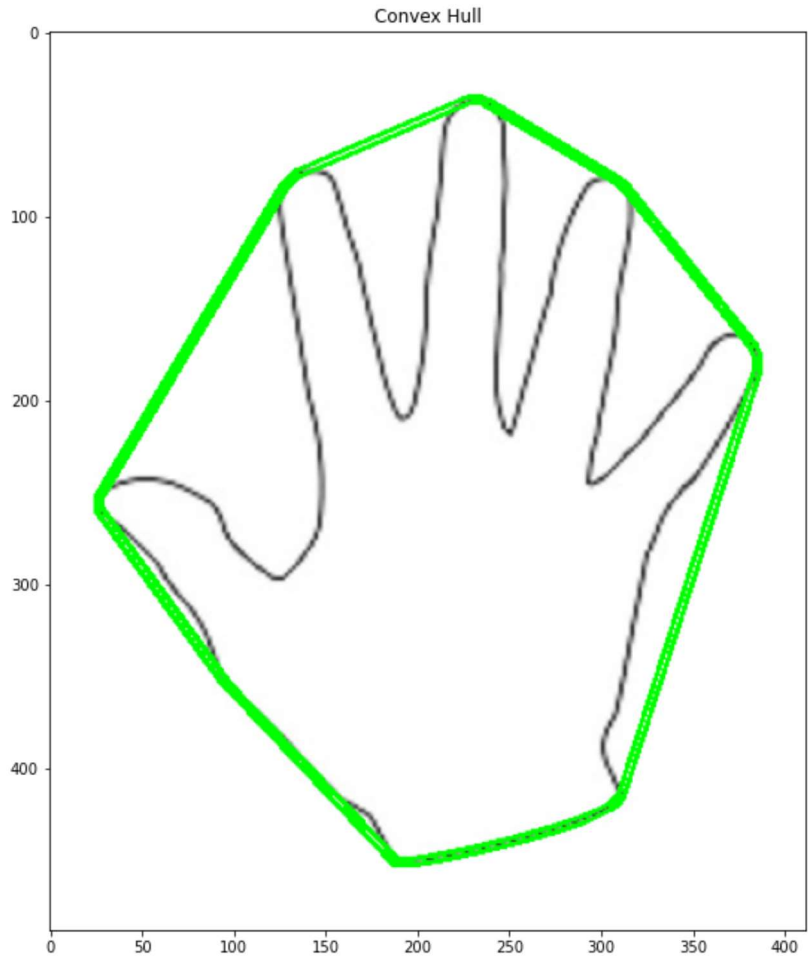


Exercise 9. In this exercise we will apply the knowledge from the last exercise to find the contours of some figures in the image `hand.jpg`. After that, we will try to approximate the contours using Convex Hull with the function `cv2.convexHull`.

In [23]:
```
image = cv2.imread('/input/opencv-samples-images/hand.jpg')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

plt.figure(figsize=(20, 20))

plt.subplot(1, 2, 1)
plt.title("Original Image")
plt.imshow(image)

# Threshold the image
ret, thresh = cv2.threshold(gray, 176, 255, 0)

# Find contours
contours, hierarchy = cv2.findContours(thresh.copy(), cv2.RETR_LIST, cv2.CHAIN_APPROX_NONE)

# Sort Contors by area and then remove the largest frame contour
n = len(contours) - 1
contours = sorted(contours, key=cv2.contourArea, reverse=False)[:n]

# Iterate through contours and draw the convex hull
for c in contours:
    hull = cv2.convexHull(c)
    cv2.drawContours(image, [hull], 0, (0, 255, 0), 2)

    plt.subplot(1, 2, 2)
    plt.title("Convex Hull")
    plt.imshow(image)
```

```
/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:25: MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently reuses the earlier instance.  In a future version, a new instance will always be created and returned.  Meanwhile, this warning can be suppressed, and the future behavior ensured, by passing a unique label to each axes instance.
```


Original Image
0
100
200
300
400
0
50
100
150
200
250
300
350
400


Convex Hull
0
100
200
300
400
0
50
100
150
200
250
300
350
400