2.1 Image processing tools

Image processing encompasses a number of techniques to improve the overall quality of an image (image smoothing, enhancement, etc.). Before going into those techniques in depth, it is useful to understand some basic concepts:

- Image histograms
- Brightness and contrast
- Binarization
- Look up tables

Next sections introduce those concepts in the context of a real problem.

Problem context - Number-plate recognition

Recently, the University of Málaga (UMA) is having trouble with private parking access. Someone has hacked their access system, so cars without previous authorization are parking there.

UMA asked computer vision students for help to implement some more secure methods that have to be included in a new security system.

They provided some images of unauthorized plates to ease the software development: `car_plate_1.jpg`, `car_plate_2.jpg` and `car_plate_3.jpg`.

In [1]:
```
import numpy as np
import cv2
import matplotlib.pyplot as plt
import matplotlib
from ipywidgets import interactive, fixed, widgets
matplotlib.rcParams['figure.figsize'] = (15.0, 15.0)

images_path = './images/'
```

2.1.1 Image histograms

And there we go! We are excited with the idea of developing a software to help UMA. For that, they provided us with a list of concepts and techniques that we have to master in order to design a successful plate recognition system.

The first one is such of **histogram**:

- A representation of the frequency of each color intensity appearing in the image.
- It is built by iterating over all the pixels in the image while counting the occurrence of each color. *Note that a RGB image has 3 histograms, one per channel.*
- It provides statistical information of the intensity distribution, like the image brightness or contrast.

The concepts of **brightness** and **contrast** are specially relevant for image processing:

- *brightness*: average intensity of image pixels, so dark images have a low brightness, while lighter ones have a high brightness.
- *contrast*: square distance of the intensities from the average, that is, a measure of the quality of the image given its usage of all the possible color intensities in the histogram.

Typically, a high quality image have a medium brightness and a high contrast.

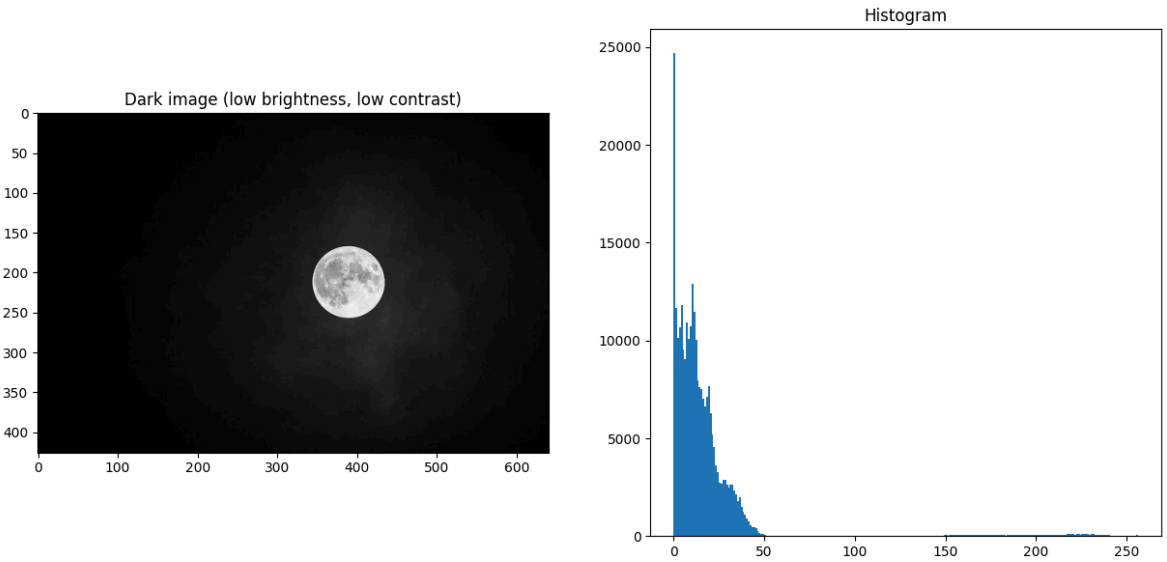
The following code illustrates these first concepts with a few examples!

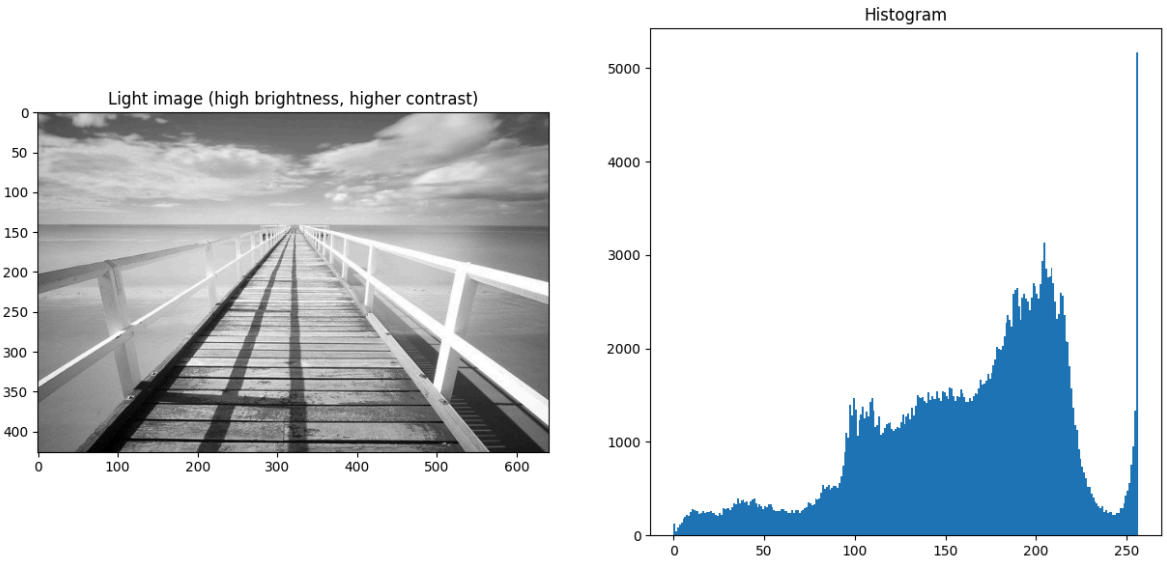
Interesting functions:

- `cv2.imread()` *function for reading images in OpenCV.*
- `plt.subplot()` *this method from matplotlib creates a grid of subfigures with a given number of rows and columns.*
- `plt.hist()` *function that computes and draws the histogram of an array.* `numpy.ravel()` *is a good helper here, since it converts a n-dimensional array into a flattened 1D array.*
- `plt.imshow()` *matplotlib function for displaying images. If the image is grayscale you should use the parameter* `cmap='gray'`.

In [2]:
```
# Read dark image and show it
image = cv2.imread(images_path + 'landscape_1.jpg',cv2.IMREAD_GRAYSCALE)
plt.subplot(2,2,1)
plt.title("Dark image (low brightness, low contrast)")
# plt.xticks([]), plt.yticks([])  # this option hides tick values on X and Y axi
plt.imshow(image, cmap='gray')

# Now, show its histogram
plt.subplot(2,2,2)
plt.title("Histogram")
plt.hist(image.ravel(),256,[0,256]) # ravel() returns a 1-D array, containing th

# Read light image and show it
image = cv2.imread(images_path + 'landscape_2.jpg',cv2.IMREAD_GRAYSCALE)
plt.subplot(2,2,3)
plt.title("Light image (high brightness, higher contrast)")
plt.imshow(image, cmap='gray')

# Now, its histogram
plt.subplot(2,2,4)
plt.title("Histogram")
```

```
plt.hist(image.ravel(),256,[0,256]) # ravel() returns a 1-D array, containing th
plt.show()
```





2.1.2 Binarization

One of the utilities of histograms is the fit of thresholds for **binarization**. Binarization consists of assigning the "0" or black value to the pixels having an intensity value under a given threshold (`th`), and "1" or white value to those having an intensity over it. Formally:

$$binarized(i,j) = \begin{cases} 0 & \text{if } intensity(i,j) < th \\ 1 & \text{otherwise} \end{cases} \quad \forall i \in [0 \dots n\_rows - 1], \forall j \in [0 \dots n\_$$

In our context, binarization can be a great tool for separating characters appearing on the plate (with a dark color) from the rest of the plate (with a lighter one). This will remove unnecessary information within the image. So let's implement it!

ASSIGNMENT 1: Cropping an image

Read the image `car_plate_1.jpg` and crop it to a rectangle (approximately) containing the plate.

Hint: to crop an image you can use numpy array slicing.

In [3]:
```
# ** ASSIGNMENT 1 **
# Load the image, crop it around the car plate and show it
# Write your code here!

image = cv2.imread(images_path + "car_plate_1.jpg", cv2.IMREAD_GRAYSCALE) # Loaa
plt.imshow(image, cmap='gray') # show original image
plt.show()

# Now that I know the coordinates of the car plate, I just have to crop it using
x_start_carplate, x_end_carplate = 200, 470
y_start_carplate, y_end_carplate = 300, 400

# Remember, in array slicing you first select rows (y coordinate) and then colum
image_cropped = image[y_start_carplate:y_end_carplate, x_start_carplate:x_end_ca
plt.imshow(image_cropped, cmap='gray') # show cropped image
plt.show()
```

Expected output:

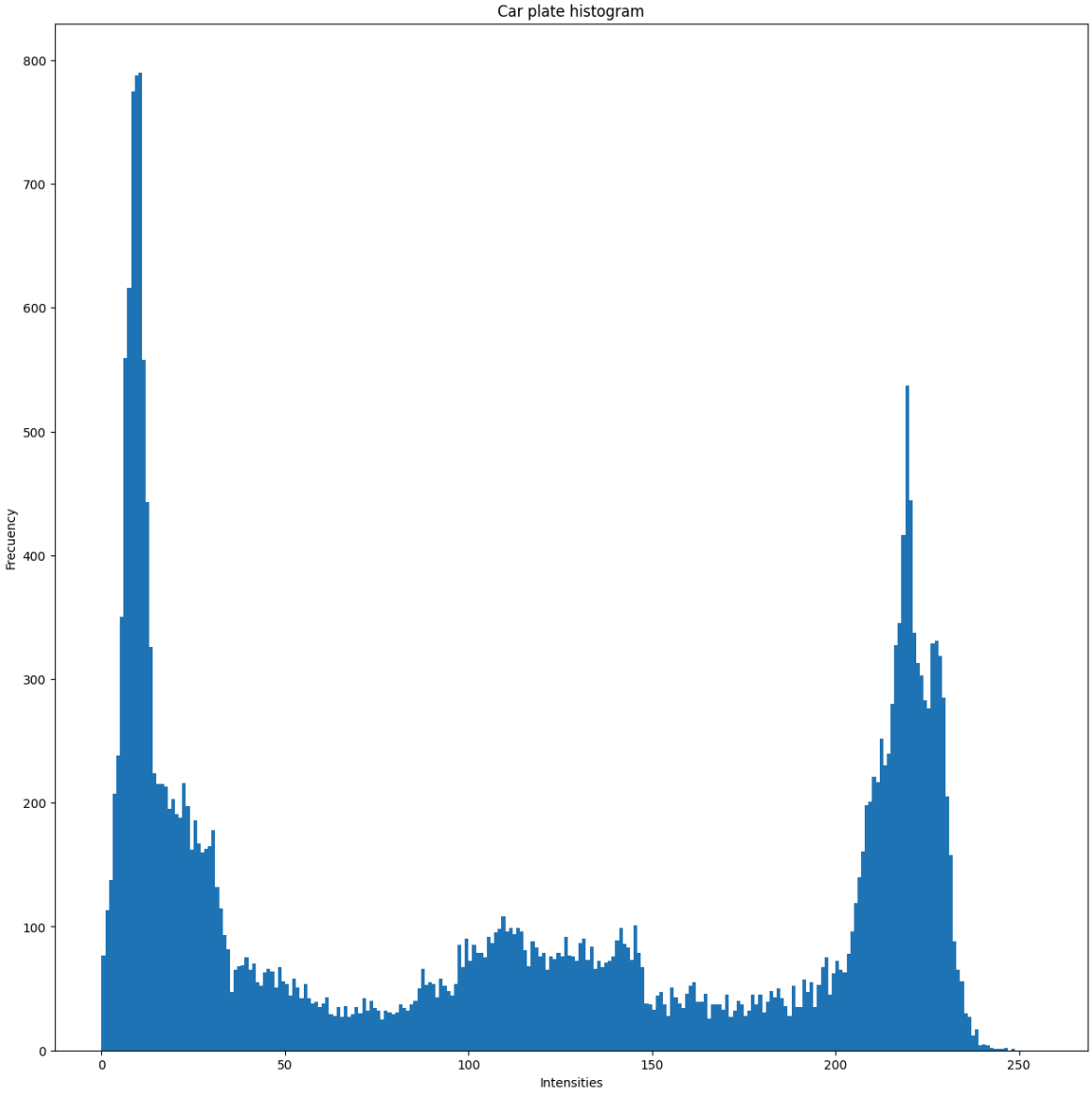
Now, we are going to use the first concept we learned, **histograms**, to see if the image is easily binarizable. To fulfill this condition, the intensity of the pixels in the image must be roughly grouped around two different values.

ASSIGNMENT 2: Showing a histogram

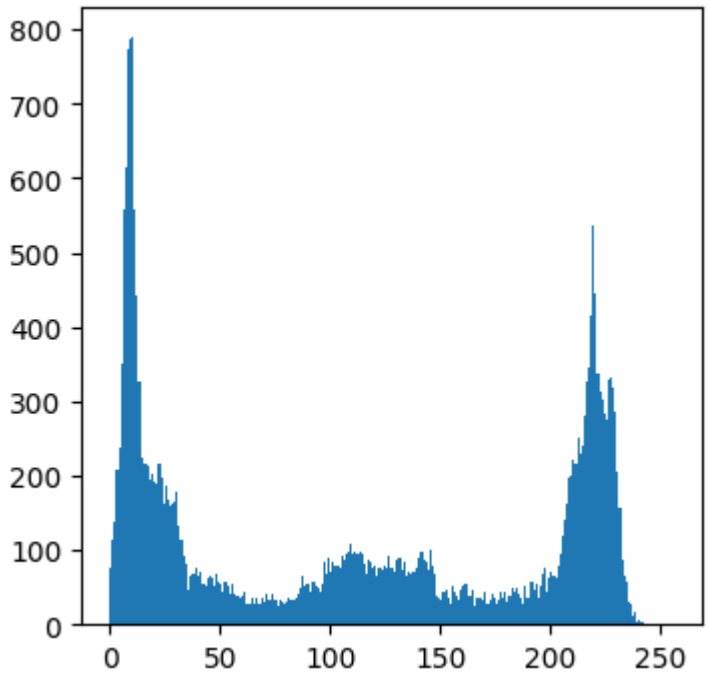
Show the histogram of `image_cropped`.

Tip: plot histogram using matplotlib, bins and range parameters are very important! Also recall the `ravel()` function

In [4]:
```
# ** ASSIGNMENT 2 **
# Compute the image histogram and show it
# Write your code here!

# Compute the histogram and show it
plt.title("Car plate histogram")
plt.hist(image_cropped.ravel(),bins=256,range=[0,256]) # ravel() returns a 1-D a
plt.ylabel("Frecuency")
plt.xlabel("Intensities")
plt.show()
```



Expected output:



Thinking about it (1)

Answer the following questions about the obtained results:

- According to your (growing) expertise, could we correctly binarize this image?

  It could be correctly binarized, as there are two clear divisions in the histogram (lower intensity peak vs higher intensity peak), so we could binarize as follows: black (lower intensity values) and white (higher intensity values). And car plates usually have two main colors, black (actual car plate number) and white (background), so binarization would be an optimal approach if all we want is to gather the number.

- Which threshold should we use?

  I'd try to find some middle ground between the two peaks, so something in the range of 100-120 (approximately the half between the two peaks).

ASSIGNMENT 3: Binarizing an image. Exiting, isn't it?

Now that we have some cues about how to binarize an image, let's take a look through OpenCV documentation to develop it.

Implement a function that:

- takes a gray image and a threshold as inputs,
- binarizes the image,
- and displays it!

*Hint: Notice that some OpenCV functions returns in first place a variable called **ret**, which content depends on the function itself.*

In [5]:
```
# ** ASSIGNMENT 3 **
# Implement a function that binarizes an image and displays it
def binarize(image,threshold):
    """ Binarizes an input image and returns it.

        Args:
            image: Input image to be binarized
            threshold: Pixels with intensity values under this parameters
                    are set to 0 (black), and those over it to 255 (white).

        Returns:
            image_binarized: Binarized image
    """                                 # grayscale image, thres, max, criteria
    ret,image_binarized = cv2.threshold(image,threshold,255,cv2.THRESH_BINARY)
    plt.figure(figsize=(6,6))
    plt.imshow(image_binarized, cmap='gray')
    plt.title('Binarized image')
    plt.show()

    return None
```

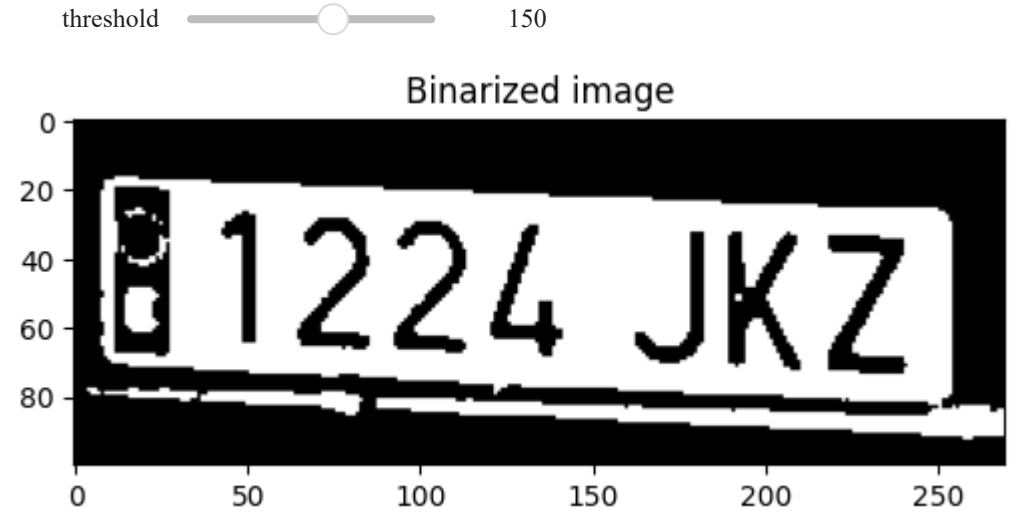
Extra! interacting with code

Jupyter has some interesting methods that allow interaction with our code, and we are going to leverage them throughout the course. Concretely, we will use the interaction function.

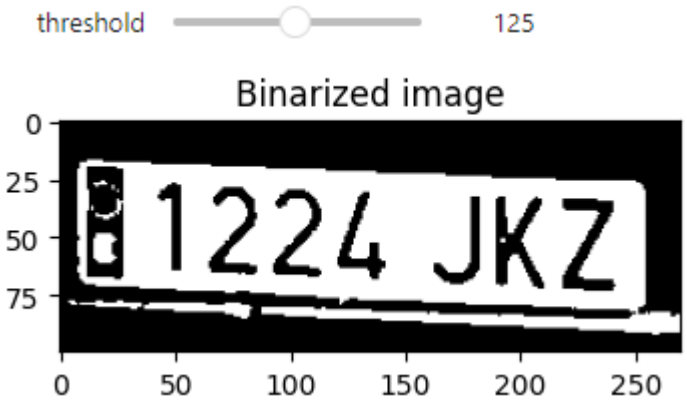
To play a bit with it, move the slider below to change the threshold value when calling the `binarize()` function.

In [6]:
```
# Interact with the threshold value
interactive(binarize, image=fixed(image_cropped), threshold=(0, 255, 25))
```

Out[6]:



Expected output:



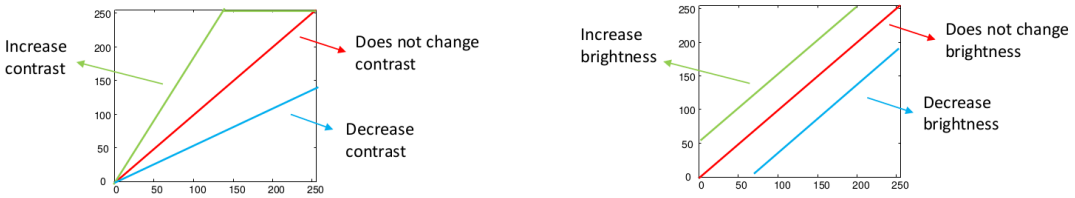
2.1.3 Look-up Tables (LUTs)

Another basic, widely used technique for image processing is such of **Look-up Tables (LUTs)**. A LUT is a table to look up the output intensity for each input one. That is, it defines a mapping between input intensity values and output ones.

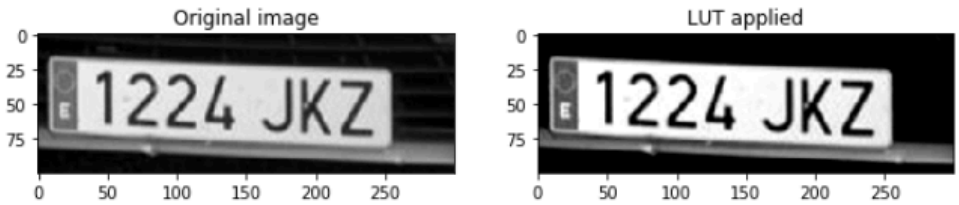
Note that if working with color (e.g. RGB) images, a LUT has to be defined for each color channel.

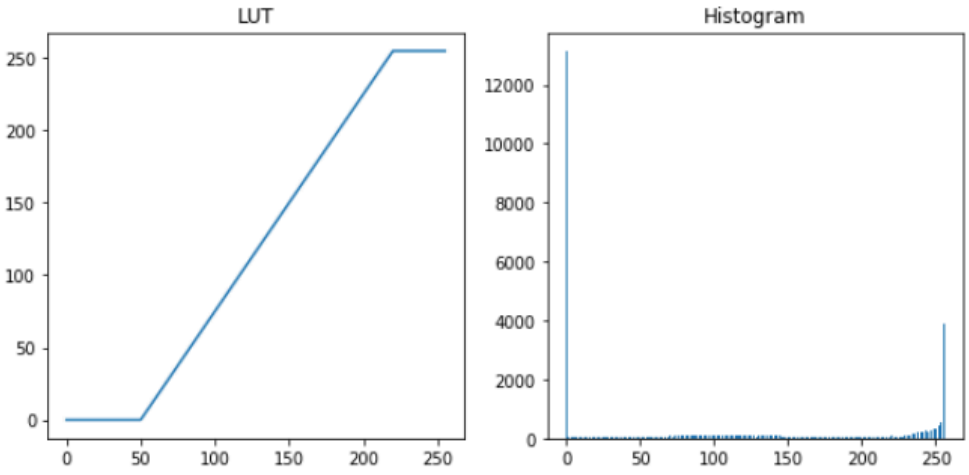
LUTs are extremely useful for modifying the brightness and contrast of images, that is, for adapting their histograms according to our needs. Some examples about the possibilities

that LUTs offer are shown below (the x-axes represent input intensity values, while y-axes represent output ones):



For example, the figure below shows the result of applying a LUT where the pixels with intensities from 0 to 50 are assigned to 0 (black), and those from 200 to 255 are assigned to 255 (white). Pixels with values in between are assigned to values from 1 to 254. The histogram of the resultant image is also shown:





Regarding our plate recognition problem, by doing this we remove noisy intensity values around black and white, obtaining a more *clean* image (see the resultant histogram). And, of course, we want to try it!

ASSIGNMENT 4a: Things get serious, implementing a LUT!

Implement the `lut_chart()` function to:

- take a gray image and a look-up table (256-length array),
- display a chart showing differences between the initial image and the resultant one after applying the LUT. *Tip: how to create subplots in Python*

Interesting functions:

- cv2.LUT(): *function that performs a look-up table transform of an array of arbitrary dimensions.*

In [7]:
```
# ** ASSIGNMENT 4a **
# Implement a function that takes a gray image and applies a LUT to it. This mus
# -- input image
# -- resultant image
# -- employed LUT
# -- histogram of the resultant image
def lut_chart(image, lut):
    """ Applies a LUT to an image and shows the result.

        Args:
            image: Input image to be modified.
            lut: a 256 elements array representing a LUT, where
                indices index input values, and their content the
                output ones.
    """
    # Apply LUT
    im_lut = cv2.LUT(image, lut)

    # Show the initial image
    plt.figure(1)
    plt.figure(figsize=(10,10))
    plt.subplot(2, 2, 1)
    plt.imshow(image, cmap='gray') # same as before
    plt.title('Original image')

    # Show the resultant one
    plt.subplot(2, 2, 2)
    plt.imshow(im_lut, cmap='gray')
    plt.title('LUT applied')

    # Plot the used LUT
    plt.subplot(2,2,3)
    plt.title('LUT')           # lut is the y axys
    plt.plot(np.arange(0,256), lut) # Hint: np.arange() can be useful as first a

    # And finally, the resultant histogram
    plt.subplot(2, 2, 4)
    plt.hist(im_lut.ravel(),bins=256,range=[0,256]) # same as before
    plt.title('Histogram')
    plt.show()
```

ASSIGNMENT 4b: Applying our amazing function

Finally, let's try our `lut_chart()` function with different look-up tables. Try to play with bright and contrast in order to get an enhanced image. For that:

1. Create a look-up table.
2. Call our function with it as second argument.

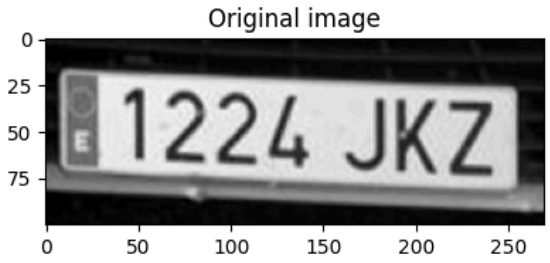
You can repeat the previous steps playing with different look-up tables and showing the results using *subplots*.

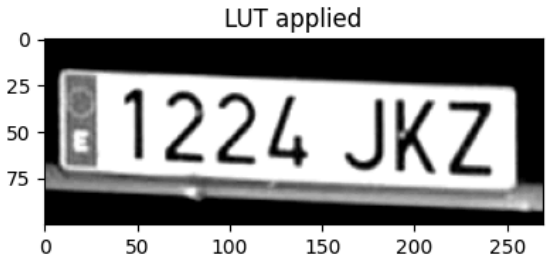
Hint: An easy way to create a LUT could be:

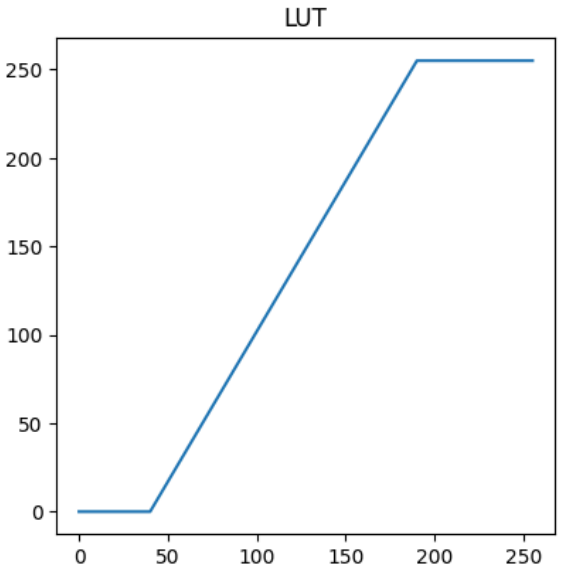
1. Create an *identity LUT* with numbers from 0 up to 255 with `numpy.arange()` (this function returns evenly spaced values within a given interval). If you use this directly as LUT, the pixels in the output will have the same intensity value as in the input. Try it!
2. Modify such LUT using `numpy.clip()`. For that you could call it as `lut = np.clip((lut-a)*b,0,255)`.

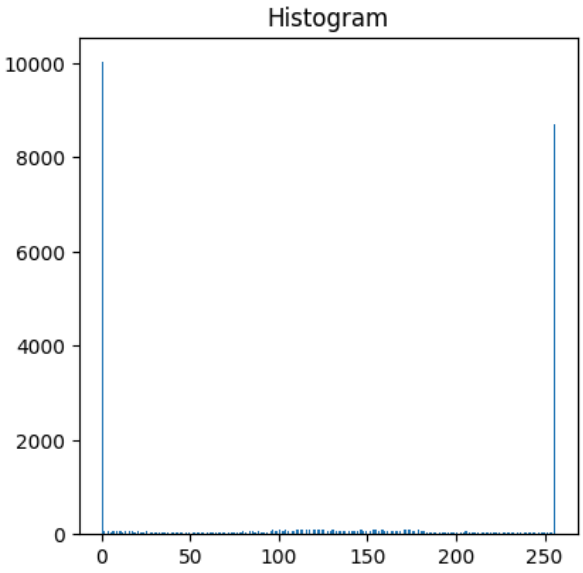
In [8]:
```
# ASSIGNMENT 4b
# Create the LUT array (have a look to numpy.arange and numpy.clip functions)
lut = np.arange(0,256)
a = 40    # initial valley (ceros iniciales)
b = 1.7   # slope (pendiente)
lut = np.clip((lut-a)*b, 0, 255)  # with clip, that which becomes negative gets


# Execute our function on the cropped car plate image
lut_chart(image_cropped,lut)
```

```
<Figure size 1500x1500 with 0 Axes>
```

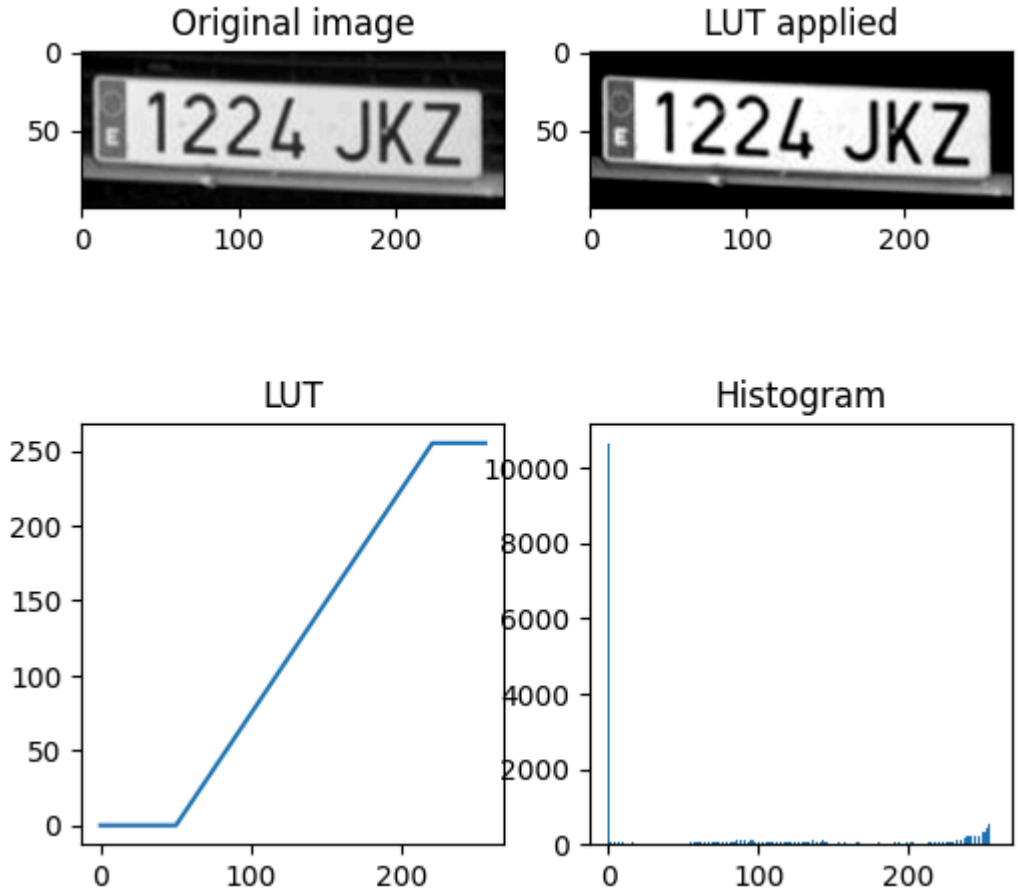








Expected output:



2.1.4 Convolutions

A 2D convolution, represented by the $\oplus$ symbol, is a fundamental tool in numerous image processing techniques (e.g. image smoothing, edge detection, etc.). Concretely, this mathematical operation is useful when implementing operators whose output pixel values are linear combinations of input ones.

There are two principal actors in a convolution: **the image** and **the kernel**. Both are 2D matrices, but usually the **kernel has a significant lower size** compared with the image. Let's define them as:

- **Image (I)**: The image in which some image processing technique is needed.

- **Kernel (K)**: A small 2D matrix that defines the linear operation that is going to be applied over the image.

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

Once we have defined the input image and the kernel, the convolution operation for a certain pixel with coordinates r and c results:

$$O(r,c) = \sum_{i=-w}^{w} \sum_{j=-w}^{w} I(r+i, c+j) * K(-i,-j)$$

where:

- O is the output image.
- w is the kernel aperture size (for example, the kernel shown above would have an aperture of $w = 1$).

But, what does this equation actually does?

> Convolution is the process of adding each element of the input image with its local neighbors, weighted by the kernel. For example, if we have two three-by-three matrices, one of them a kernel, and the other a piece of the image, convolution is the process of **flipping both the rows and columns of the kernel and then multiplying locationally similar entries and summing**

For example, the pixel value in the [2,2] position on the resulting image would be a weighted combination of all the entries of the image matrix, with weights given by the kernel.

$$\left(\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \otimes \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \right) [2,2] = (1*i) + (2*h) + (3*g) + (4*f) + (5*e) +$$

As said above, if we flip the kernel across both axes, the formula of the convolution turns into an element-wise matrix multiplication:

$$\left(\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \otimes \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \right) [2,2] = \left(\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} .* \begin{bmatrix} i & h & g \\ f & e & d \\ c & b & a \end{bmatrix} \right) = (1*i) + (2$$

When convolution is applied, it usually indexes out of bounds in the image, e.g.:

$$\left(\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \otimes \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \right) [1,1] = (? * i) + (? * h) + (? * g) + (? * f) + (1 * e) +$$

There are some **padding** options to deal with this problem:

- Fill out of bound values with a constant value (**usually 0** or **values in the border** of the image),
- reflecting image values (e.g. $I[0,0] = I[1,1]$)
- ...

Regarding our dear OpenCV library, it helps us by defining the 2D-convolution cv2.filter2D() method. Its main inputs are:

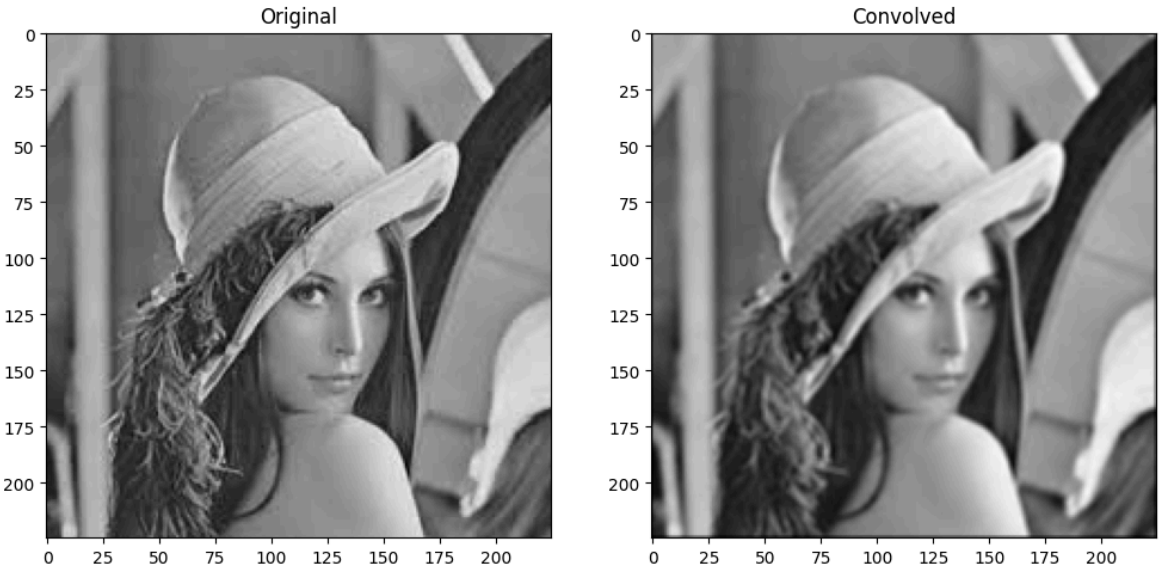
- **src**: source image
- **ddepth**: data type of output image (`cv2.CV_8U` for 8-bits unsigned integer, `cv2.CV_16S` for 16-bits signed integer, ...). **It depends on the kernel** you use. Note that there are kernels that can return negative values in the convolution output.
- **kernel**: convolution kernel.
- **borderType**: padding options (border types).

Let's try convolution in a example!

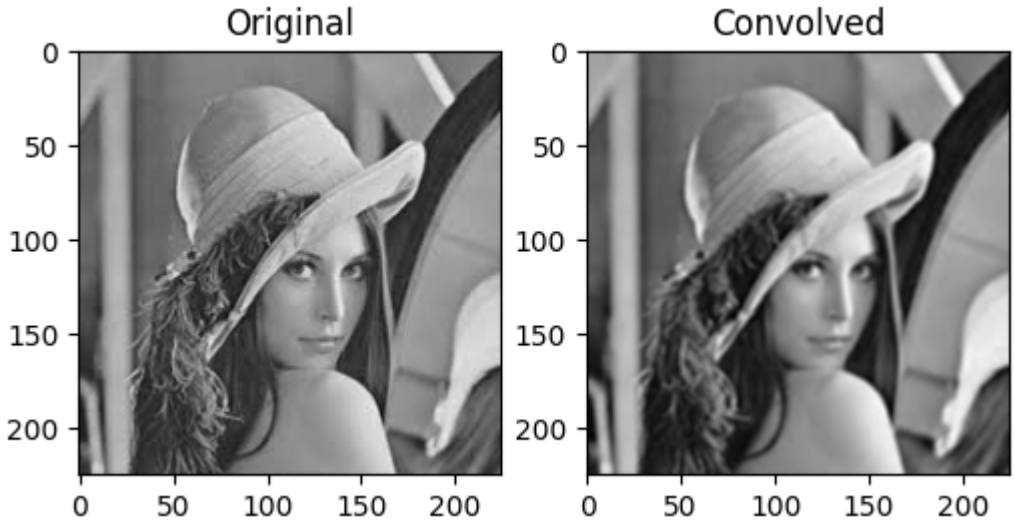
ASSIGNMENT 5

Apply a convolution to the grayscale image `lena.jpeg` using a 3×3 kernel with a constant value of $1/9$.

In [9]:
```
# ASSIGNMENT 5
# Read the image
image = cv2.imread(images_path + "lena.jpeg", cv2.IMREAD_GRAYSCALE)

# Define the kernel
kernel = np.ones((3,3))* (1/9)

# Apply convolution (note that convolution cannot return negative values using t
im_conv = cv2.filter2D(image, cv2.CV_8U, kernel, borderType=cv2.BORDER_CONSTANT)

# Show original image
plt.figure(figsize=(12,12))
plt.subplot(121)
plt.title('Original')
plt.imshow(image, cmap='gray')

# Show convolved image
plt.subplot(122)
plt.title('Convolved')
plt.imshow(im_conv, cmap='gray')
plt.show()
```



Expected output:



Thinking about it (2)

Answer the following questions about convolution:

- What is the difference between the original image and the convolved one?

  Second one has clearly more blur and less noise than the first one, as it was smoothed which is a technique that, besides lessening noise, creates blur.

- Can you guess which IP technique is such kernel implementing?

  Smoothing, as I sort of stated before (the kernel is an average filter, that averages every pixel intensity value with those of its neighbours and itself).

Additionally, you can use this demo to understand the convolution operator for image processing in a visual way. Anyway, **don't worry if you don't fully understand it**, convolution is a complex operation that have multiple applications and will be understood progressively while doing practical exercises. Exciting, isn't it?

Conclusion

Brilliant! With this notebook we have:

- Learned basic concepts within image processing like histograms, brightness, contrast, binarization and Look-up Tables.
- Played a bit with them in the context of a plate recognition system, observing their utility for improving the quality of an image according to our needs.
- Understood how convolution works.

2.2 Smoothing

Images can exhibit different levels of *noise*: a random variation of brightness or color information created by a random process, that is, artifacts that don't appear in the original scene and degrade its visual quality! this makes tasks like object detection or image recognition more challenging. It is mainly produced by factors like the sensor response (more in CMOS technology), environmental conditions, analog-to-digital conversion, *dead* sensor pixels, or bit errors in transmission, among others.

Although there are numerous types of noise, the two more common are:

- **Salt & pepper** noise (black and white pixels in random locations of the image) or **impulse** noise (only white pixels). Typical cause: faulty camera sensors, transmission errors, dead pixels.
- **Gaussian** noise (intensities are affected by an additive zero-mean Gaussian error). Typical cause: Poor illumination or high temperatures that affect the electronics.

In this section, we are going to learn about some smoothing techniques aiming to eliminate or reduce such noise, including:

- Convolution-based methods
  - Neighborhood averaging
  - Gaussian filter
- Median filter
- Image average

Problem context - Number-plate recognition

Returning to the parking access problem proposed by UMA, they were grateful with your previous work. However, after some testing of your code, there were some complaints about binarization because it is not working as well as they expected. It is suspected that the found difficulties are caused by image noise. The camera that is being used in the system is having some problems (e.g. challenging lighting conditions), so different types of noise are appearing in its captured images.

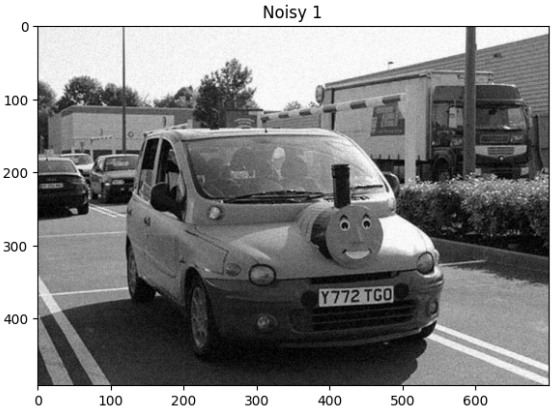
This way, UMA asked you again to provide some help with this problem!

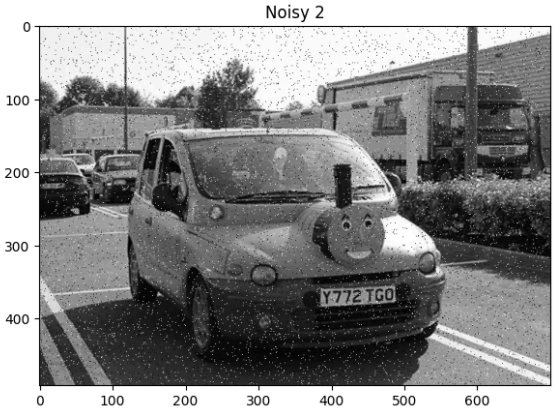
In [1]:
```
import numpy as np
from scipy import signal
import cv2
import matplotlib.pyplot as plt
import matplotlib
from ipywidgets import interactive, fixed, widgets
matplotlib.rcParams['figure.figsize'] = (15.0, 15.0)
import random

images_path = './images/'
```

ASSIGNMENT 1: Taking a look at images

First, **display the images** `noisy_1.jpg` and `noisy_2.jpg` and try to detect why binarization is in trouble when processing them.

In [2]:
```
# ASSIGNMENT 1
# Read 'noisy_1.jpg' and 'noisy_2.jpg' images and display them in a 1x2 plot
# Write your code here!

# Read images
noisy_1 = cv2.imread(images_path + "noisy_1.jpg", cv2.IMREAD_GRAYSCALE)
noisy_2 = cv2.imread(images_path + "noisy_2.jpg", cv2.IMREAD_GRAYSCALE)

# Display first one
plt.subplot(121)
plt.imshow(noisy_1, cmap="gray")
plt.title('Noisy 1')

# Display second one
plt.subplot(122)
plt.imshow(noisy_2, cmap="gray")
plt.title('Noisy 2')

plt.show()
```





Thinking about it (1)

Once you displayed both images, **answer the following questions:**

- What is the difference between them?

  The first one displays an example of gaussian noise (some pixel intensities diverge sligthly from the noiseless one), whereas the second one has salt and pepper noise (some pixel intensities are a straight black or white).

- Why can this happen (the noise)?

  There are different reasons, but mainly as a result of a physical sensor error in the camera that takes the photo (CMOS sensor response, dead pixels, transmission errors...). Other main ones, especially when talking about gaussian noise, are temperature, light and, in general, environmental conditions.

- What could we do to face this issue?

  Apply smoothing techniques such as neighbourhood averaging, gaussian noise filter (especially for first image) or median filter (especially for second image).

2.2.1 Convolution-based methods

There are some interesting smoothing techniques based on the convolution, a mathematical operation that can help you to alleviate problems caused by image noise. Two good examples are **neighborhood averaging** and **Gaussian filter**.

a) Neighborhood averaging

Convolving an image with a *small* kernel is similar to apply a function over all the image. For example, by using convolution it is possible to apply the first smoothing operator that you are going to try, **neighborhood averaging**. This operator averages the intensity values of pixels surrounding a given one, efficiently removing noise. Formally:

$$S(i,j) = \frac{1}{p} \sum_{(m,n)\in s} I(m,n)$$

with s being the set of p pixels in the neighborhood (mxn) of (i, j). Convolution permits us to implement it using a kernel, resulting in a linear operation! For example, a kernel for a 3x3 neighborhood would be:

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

You can think that the kernel is like a weight matrix for neighbor pixels, and convolution like a double `for` loop that applies the kernel pixel by pixel over the image. An important parameter when definig a kernel is its **aperture**, that is, how many row/columns it has in addition to the one in the middle in both sides. For example, the previous kernel has an aperture of 1, while a 5x5 kernel would have an aperture of 2, a 7x7 kernel of 3, and so on.

Not everything will be perfect, and the **main drawback** of neighborhood averaging is the blurring of the edges appearing in the image.

ASSIGNMENT 2: Applying average filtering

Complete the method `average_filter()` that convolves an input image using a kernel which values depend on its size (e.g. for a size 3x3 size its values are 1/9, for a 5x5 size 1/25 and so on). Then display the differences between the original image and the resultant one if `verbose` is `True`. It takes the image and kernel aperture size as input and returns the smoothed image.

Tip: OpenCV defines the 2D-convolution cv2.filter2D(src, ddepth, kernel) method, where:

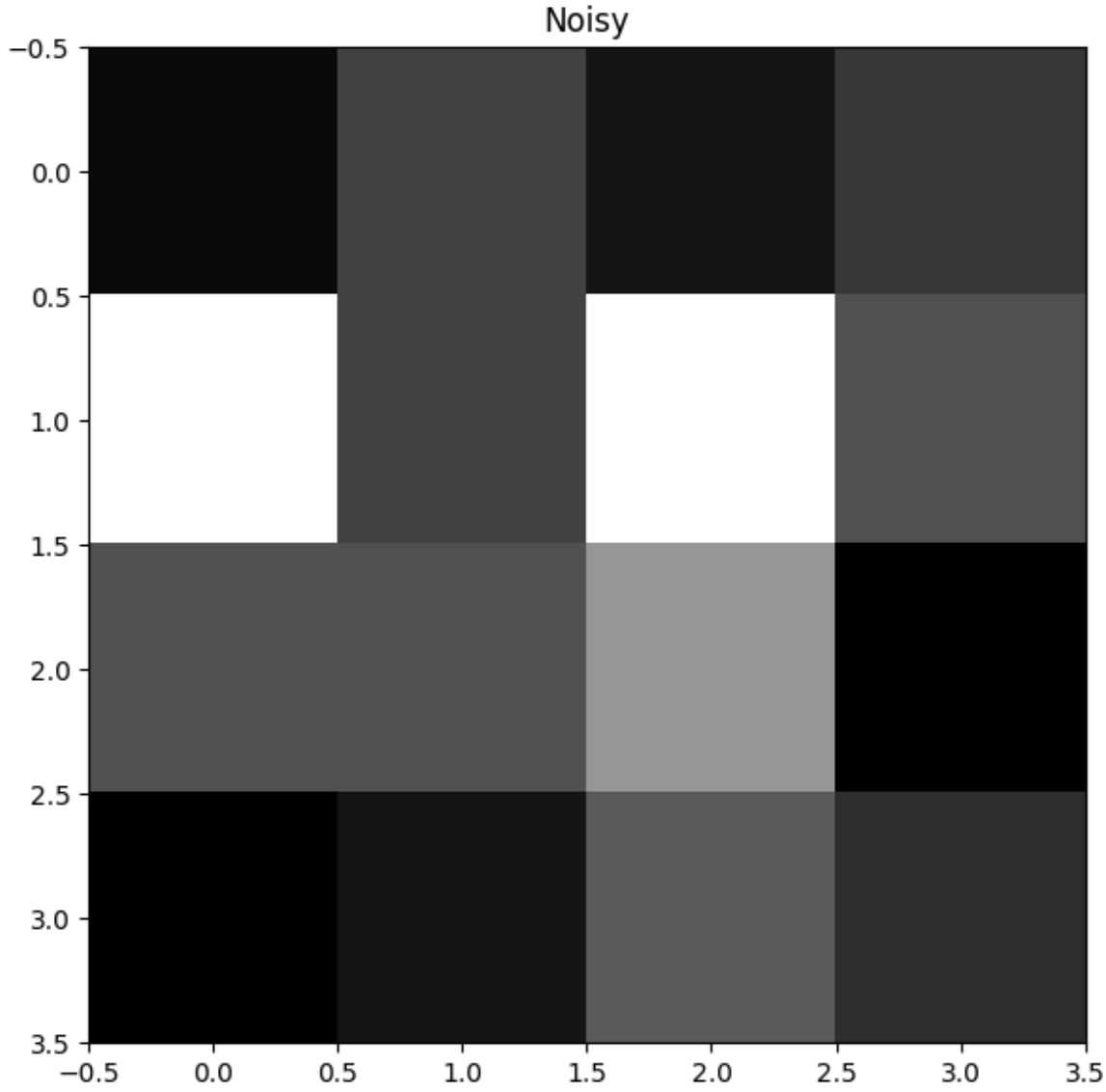
- *the `ddepth` parameter means desired depth of the destination image.*
  - *Input images (`src`) use to be 8-bit unsigned integer (`ddepth` =cv2.CV_8U).*
  - *However, output sometimes is required to be 16-bit signed (`ddepth` =cv2.CV_16S)*

In [3]:
```
# ASSIGNMENT 2
# Implement a function that applies an 'average filter' to an input image. The k
# Show the input image and the resulting one in a 1x2 plot.
def average_filter(image, w_kernel, verbose=False):
    """ Applies neighborhood averaging to an image and display the result.

        Args:
            image: Input image
            w_kernel: Kernel aperture size (1 for a 3x3 kernel, 2 for a 5x5, etc
            verbose: Only show images if this is True

        Returns:
            smoothed_img: smoothed image
    """
    # Write your code here!

    # Create the kernel
```

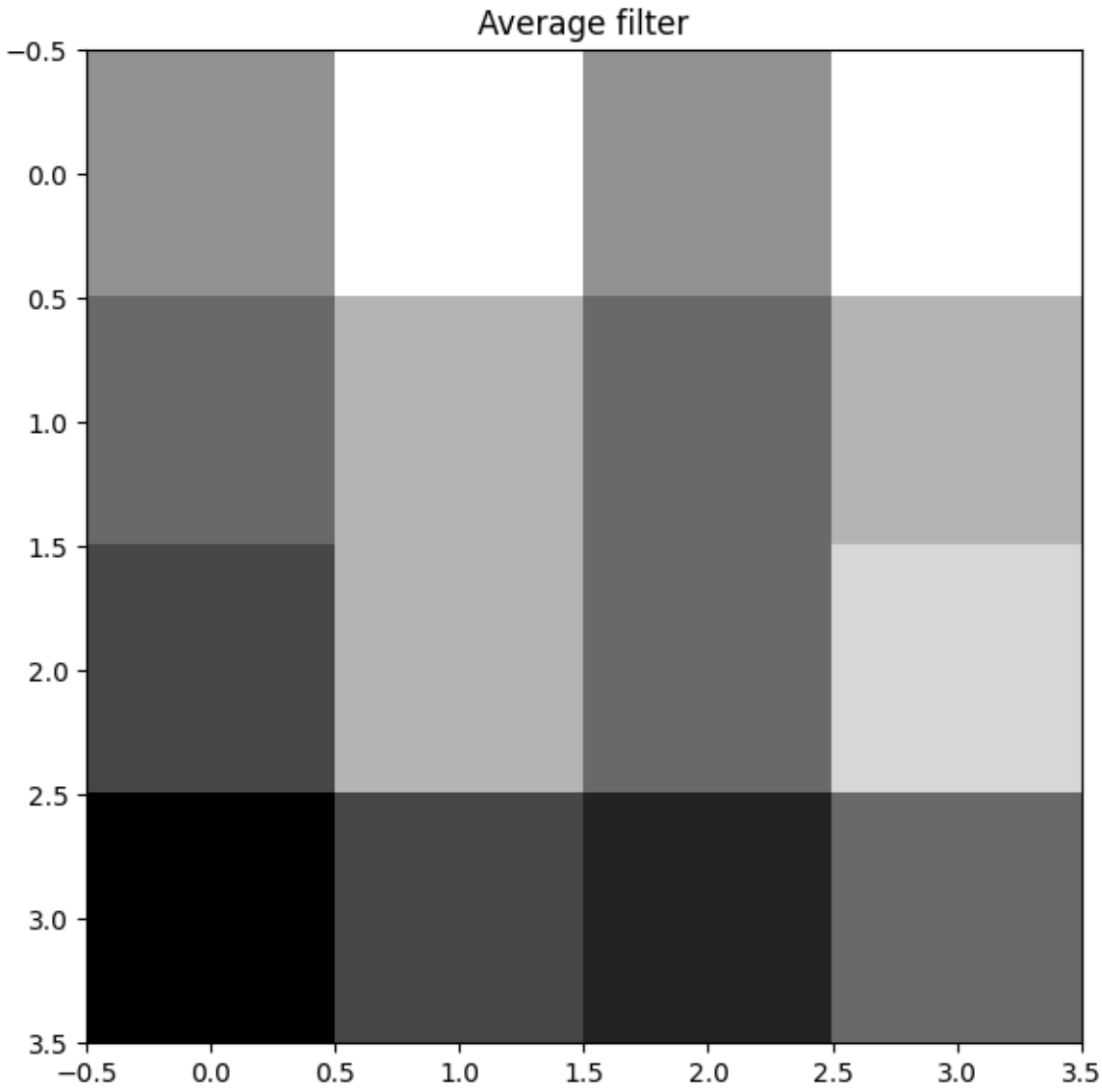
```
    dimension = 1 + w_kernel * 2
    height = dimension # or number of rows
    width = dimension # or number of columns
    kernel = np.ones((dimension, dimension), np.float32)/(1*(height*width))

    # Convolve image and kernel
    smoothed_img = cv2.filter2D(image,cv2.CV_8U,kernel)

    if verbose:
        # Show the initial image
        plt.subplot(121)
        plt.title('Noisy')
        plt.imshow(image, cmap='gray')
        plt.show()

        # Show the resultant one
        plt.subplot(122)
        plt.title('Average filter')
        plt.imshow(smoothed_img, cmap='gray')

    return smoothed_img
```

You can use the next snippet of code to **test if your results are correct**:

In [4]:
```
# Try this code
image = np.array([[1,6,2,5],[22,6,22,7],[7,7,13,0],[0,2,8,4]], dtype=np.uint8)
w_kernel = 1
print(average_filter(image, w_kernel, True))
```




```
[[ 9 12  9 12]
 [ 8 10  8 10]
 [ 7 10  8 11]
 [ 5  7  6  8]]
```



Expected output:

```
[[ 9 12  9 12]
[  8 10  8 10]
[  7 10  8 11]
[  5  7  6  8]]
```

Thinking about it (2)

You are asked to use the code cell below (the interactive one) and try **average_filter** using both noisy images `noisy_1.jpg` and `noisy_2.jpg`. Then, **answer the following questions**:

- Is the noise removed from the first image?

  I wouldn't say removed, but diminished (at least in the ones that are not the blurriest). When kernel aperture > 0, there's a certain degree of smoothing applied. The greater the aperture, the greater the smoothing (less noise), but also the greater the blur (edges are less differentiable), especially when talking about gaussian noise.

- Is the noise removed from the second image?

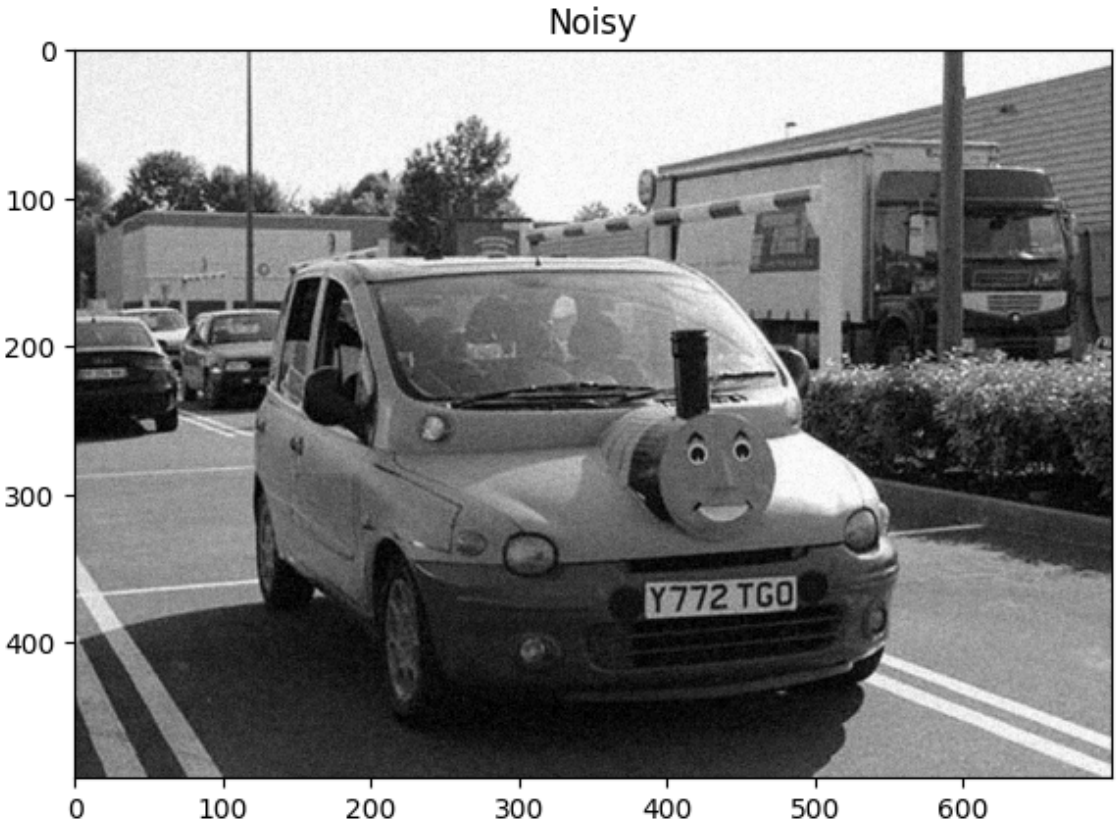
It is (very) slightly diminished, but it's still clearly there (much more than the gaussian one). Neighbourhood averaging is not the proper technique to remove considerably salt and pepper noise.

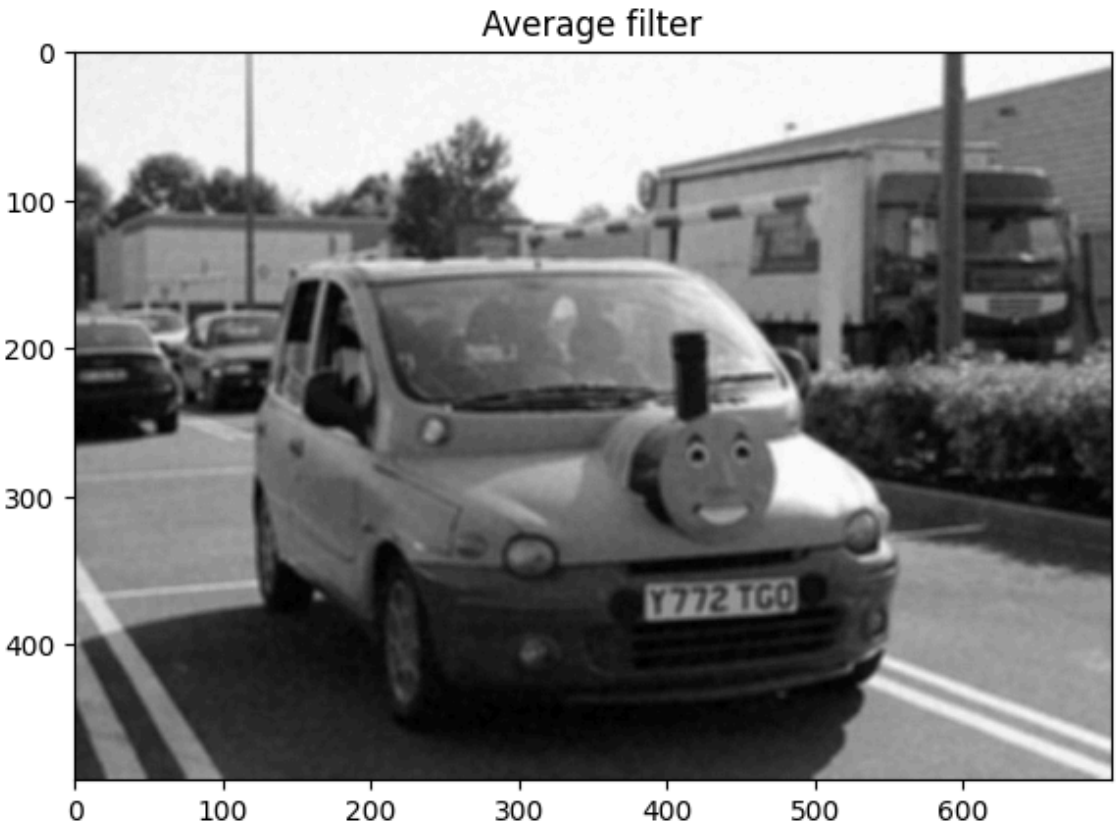
- Which value is a good choice for `w_kernel`? Why?

  For image 2 (salt and pepper noise), none. I would just use another technique, such as median filter. For image 1, something between 1 and 2 would be appropiate as it removes some noise and doesn't add a ton of blur, but there are better techniques (such as gaussian filter).

In [5]:
```
# Interact with the kernel size
noisy_img = cv2.imread(images_path + 'noisy_1.jpg', 0)
interactive(average_filter, image=fixed(noisy_img), w_kernel=(0,5,1), verbose=fi
```

Out[5]: w_kernel 2





b) Gaussian filtering

An alternative to neighborhood averaging is **Gaussian filtering**. This technique applies the same tool as averaging (a convolution operation) but with a more complex kernel.

The idea is to take advantage of the normal distribution for creating a kernel that keeps borders in the image while smoothing. This is done by giving more relevance to the pixels that are closer to the kernel center, creating a **neighborhood weighted averaging**. For example, considering a kernel with an aperture of 2 (5×5 size), its values would be:

0.003	0.013	0.022	0.013	0.003
0.013	0.059	0.097	0.059	0.013
0.022	0.097	0.159	0.097	0.022
0.013	0.059	0.097	0.059	0.013
0.003	0.013	0.022	0.013	0.003

For defining such a kernel it is used the Gaussian bell:

In 1-D:

$$g_\sigma(x) = \frac{1}{\sigma\sqrt{2\pi}} exp\left(-\frac{x^2}{2\sigma^2}\right)$$

In 2-D, we can make use of the *separability property* to separate rows and columns, resulting in convolutions of two 1D kernels:

$$g_\sigma(x,y) = \underbrace{\frac{1}{2\pi\sigma^2} exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)}_{g} = \underbrace{\frac{1}{\sigma\sqrt{2\pi}} exp\left(-\frac{x^2}{2\sigma^2}\right)}_{g_x} * \underbrace{\frac{1}{\sigma\sqrt{2\pi}} exp\left(-\frac{y^2}{2\sigma^2}\right)}_{g_y}$$

For example:

$$g = g_y \otimes g_x \rightarrow \begin{array}{|c|c|c|}\hline 1 & 2 & 1 \\ \hline 2 & 4 & 2 \\ \hline 1 & 2 & 1 \\ \hline \end{array} = \begin{array}{|c|}\hline 1 \\ \hline 2 \\ \hline 1 \\ \hline \end{array} \otimes \begin{array}{|c|c|c|}\hline 1 & 2 & 1 \\ \hline \end{array}$$

And because of the *associative property*:

$$\underbrace{f \otimes g}_{\text{2D convolution}} = f \otimes (g_x \otimes g_y) = \underbrace{(f \otimes g_x) \otimes g_y}_{\text{Two 1D convolutions}}$$

In this way, we do $2n$ operations instead of n^2, being n the kernel size. This is relevant in kernels with a big size, or if you have to apply this operation many times.

The degree of smoothing of this filter can be controlled by the σ parameter, that is, the **standard deviation** of the Gaussian distribution used to build the kernel. The bigger the σ, the more smoothing, but it could result in a blurrier image!

The σ parameter also influences the **kernel aperture** value to use, since it must be proportional. It has to be big enough to account for non-negligible values in the kernel! For example, in the kernel below, it doesn't make sense to increase its aperture (currently 1) since new rows/columns would have very small values:

1	15	1
15	100	15
1	15	1

ASSIGNMENT 3: Implementing the famous gaussian filter

Complete the `gaussian_filter()` method in a similar way to the previous one, but including a new input: `sigma`, representing the standard deviation of the Gaussian distribution used for building the kernel.

As an illustrative example of separability, we will obtain the kernel by performing the convolution of a 1D `vertical_kernel` with a 1D `horizontal_kernel`, resulting in the 2D gaussian `kernel`!

Tip: Note that NumPy defines mathematical functions that operate over arrays like exponential or square-root, as well as mathematical constants like `np.pi`. Remember the associative property of convolution.

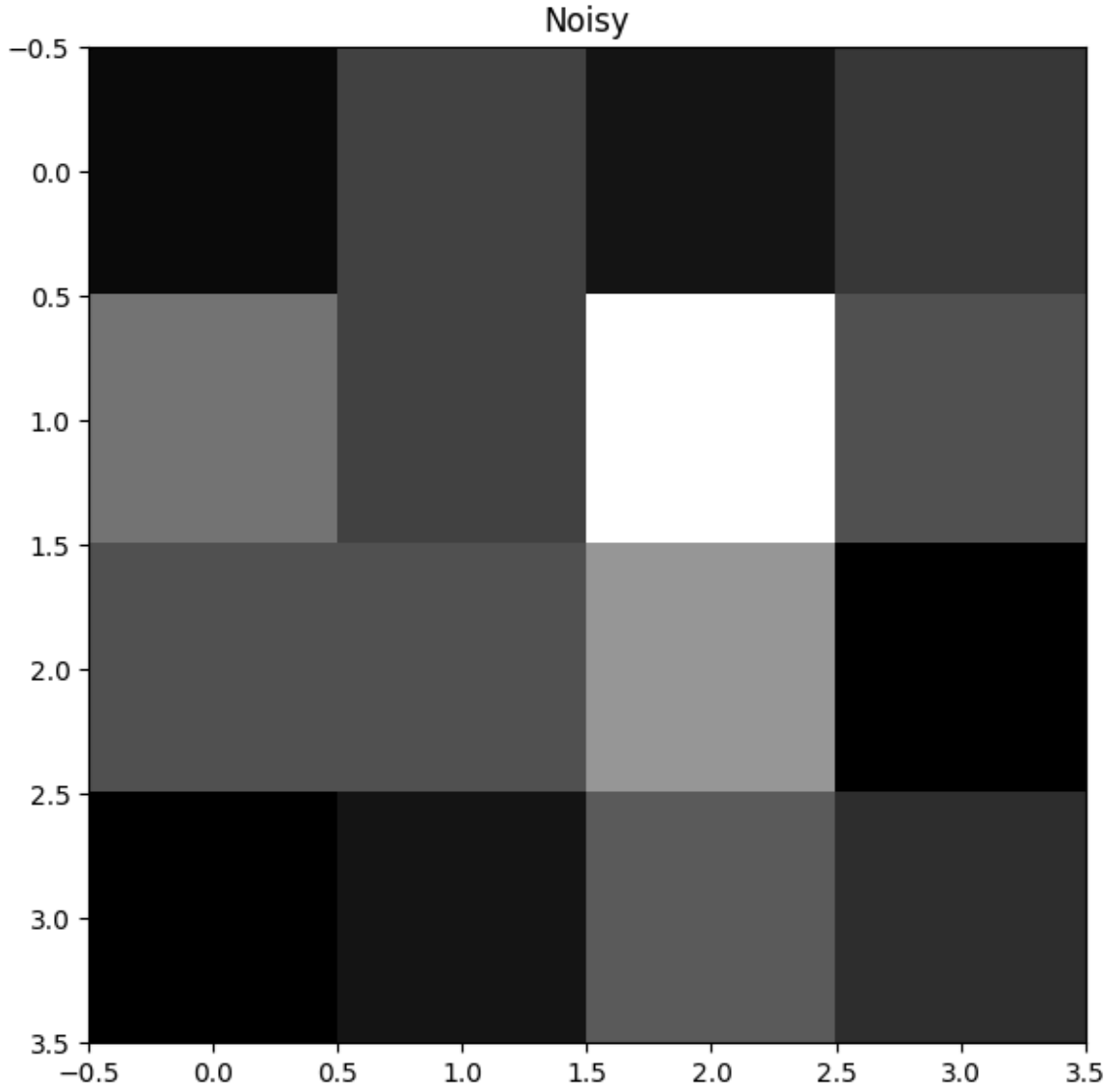
*Tip 2: The code below uses **List Comprehension** for creating a list of numbers by evaluating an expression within a `for` loop. Its syntax is: `[expression for item in list]`. You can find multiple examples of how to create lists using this technique on the internet.*

In [6]:
```python
# ASSIGNMENT 3
# Implement a function that:
# -- creates a 2D Gaussian filter (tip: it can be done by implementing a 1D Gaus
# -- convolves the input image with the kernel
# -- displays the input image and the filtered one in a 1x2 plot (if verbose=Tru
# -- returns the smoothed image
def gaussian_filter(image, w_kernel, sigma, verbose=False):
    """ Applies Gaussian filter to an image and display it.

        Args:
            image: Input image
            w_kernel: Kernel aperture size
            sigma: standard deviation of Gaussian distribution
            verbose: Only show images if this is True

        Returns:
            smoothed_img: smoothed image
    """
    # Write your code here!

    # Create kernel using associative property
    s = sigma
    w = w_kernel

    # Evaluates gaussian expression on [1-,0,1]; as w=1
    kernel_1D = np.float32([(1 / (sigma*np.sqrt(2*np.pi))*np.exp(- (z*z) / (2*si
    vertical_kernel = kernel_1D.reshape(2*w+1,1) # Reshape it as a matrix with j
    horizontal_kernel = kernel_1D.reshape(1,2*w+1) # Reshape it as a matrix with
    kernel = signal.convolve2d(vertical_kernel, horizontal_kernel) # Get the 2D
```

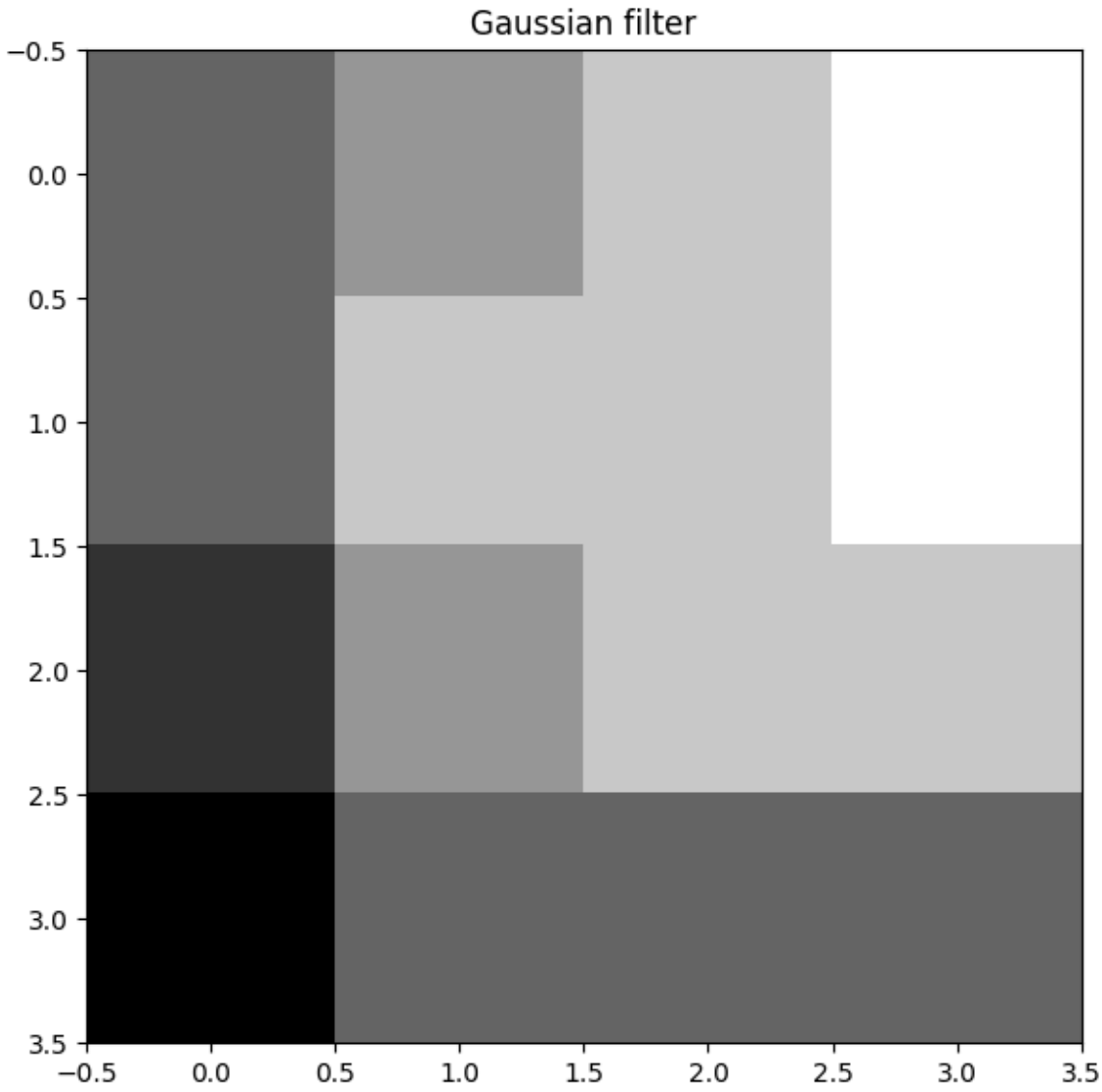
```
    #print(kernel_1D, "\n\n", vertical_kernel, "\n\n", horizontal_kernel, "\n\n"

    # Convolve image and kernel
    smoothed_img = cv2.filter2D(image,cv2.CV_8U, kernel)

    if verbose:
        # Show the initial image
        plt.subplot(121)
        plt.imshow(image, cmap='gray')
        plt.title('Noisy')
        plt.show()

        # Show the resultant one
        plt.subplot(122)
        plt.imshow(smoothed_img, cmap='gray')
        plt.title('Gaussian filter')

    return smoothed_img
```

Again, you can use next code to **test if your results are correct**:

In [7]:
```
image = np.array([[1,6,2,5],[10,6,22,7],[7,7,13,0],[0,2,8,4]], dtype=np.uint8)
w_kernel = 1
sigma = 1
print(gaussian_filter(image, w_kernel,sigma,True))
```




```
[[5 6 7 8]
 [5 7 7 8]
 [4 6 7 7]
 [3 5 5 5]]
```



Expected output:

```
[[5 6 7 8]
 [5 7 7 8]
 [4 6 7 7]
 [3 5 5 5]]
```

Thinking about it (3)

You are asked to try **gaussian_filter** using both noisy images `noisy_1.jpg` and `noisy_2.jpg` (see the cell below). Then, **answer following questions**:

- Is the noise removed from the first image?

  It's notoriously diminished, but not removed. As a result of the smoothing, the image also got blurrier.

- Is the noise removed from the second image?

  Just a tiny little bit, but if I had to give a definitive answer I'd say no. The result is inacceptable for a noise-remover filter (this result it's the one to expect when you use

gaussian noise filter on salt and pepper noise).

- Which value is a good choice for `w_kernel` and `sigma`? Why?

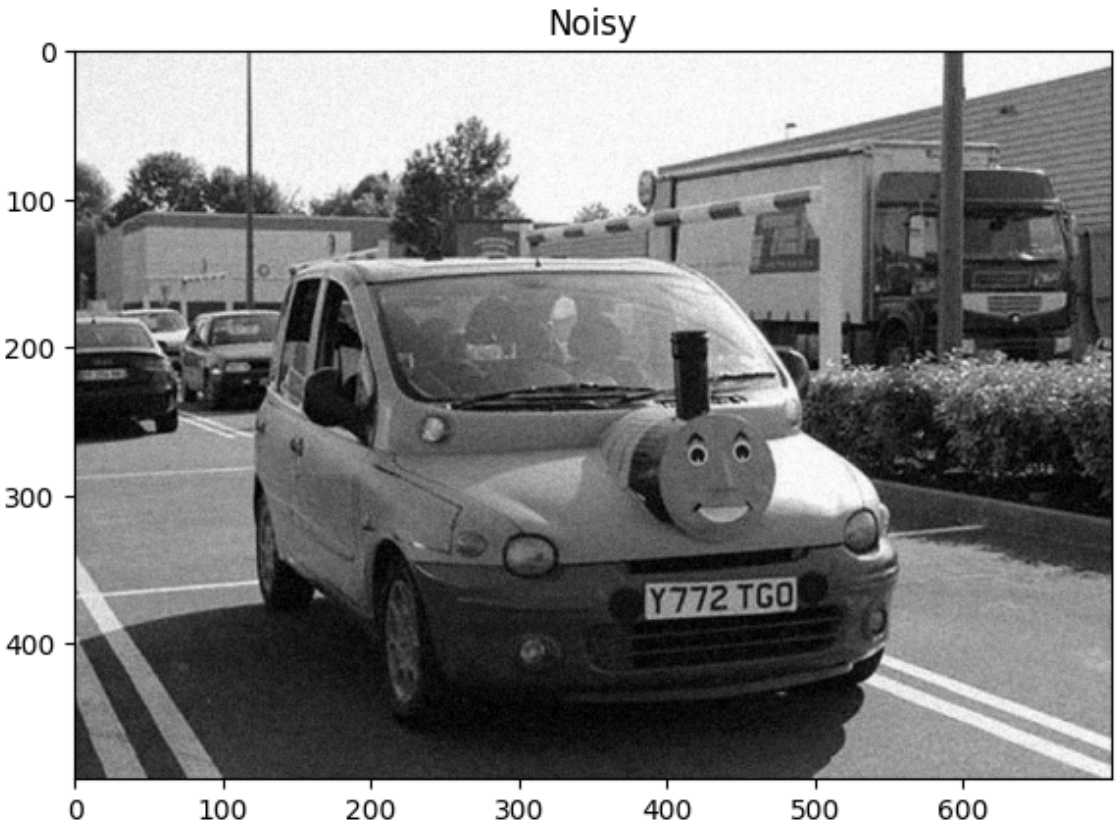
  For the second image, none; as this is not the proper technique for salt and pepper noise. For the first image, clearly a w value of 1 (as 2 or more introduces too much blur) and the sigma may vary between 1.9 and 2.2. The differences seen as sigma changes in this particular scenario are not astonishingly big, so trying to stay in middle values might be the wisest move.

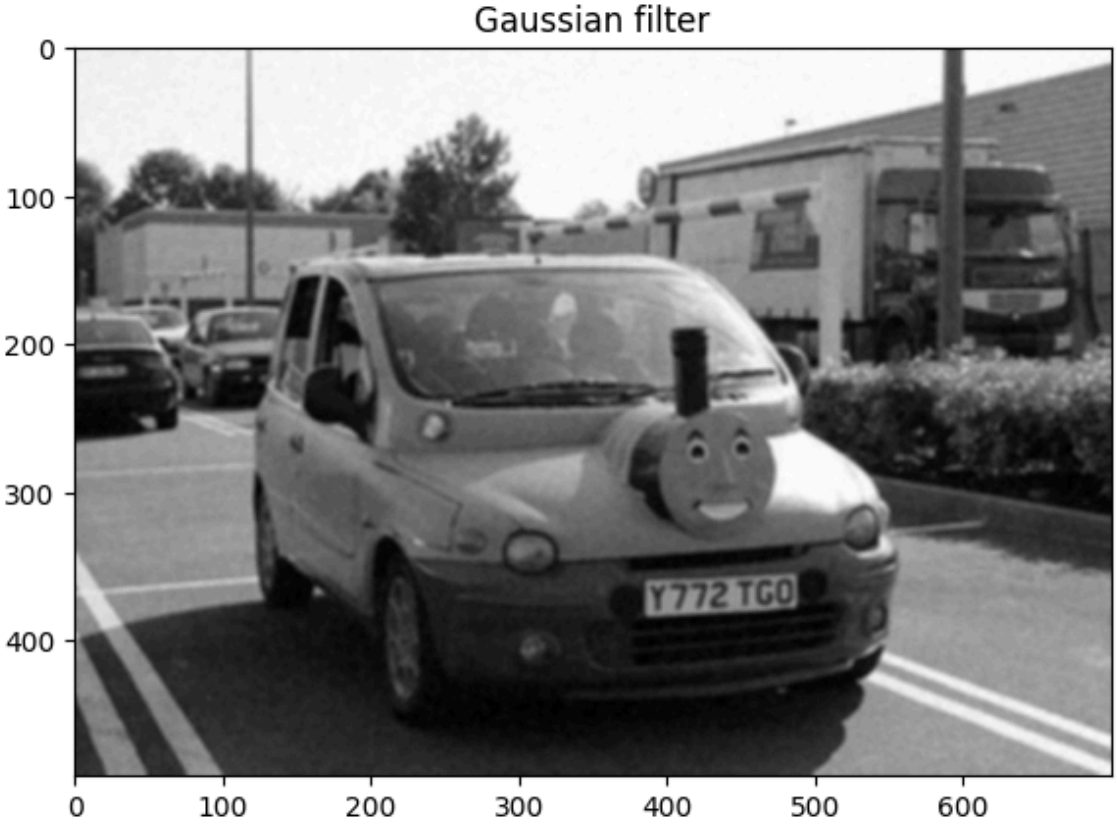
In [8]:
```
# Interact with the kernel size and the sigma value
noisy_img = cv2.imread(images_path + 'noisy_1.jpg', 0)
interactive(gaussian_filter, image=fixed(noisy_img), w_kernel=(0,5,1), sigma=(1,
```

Out[8]:

w_kernel 2

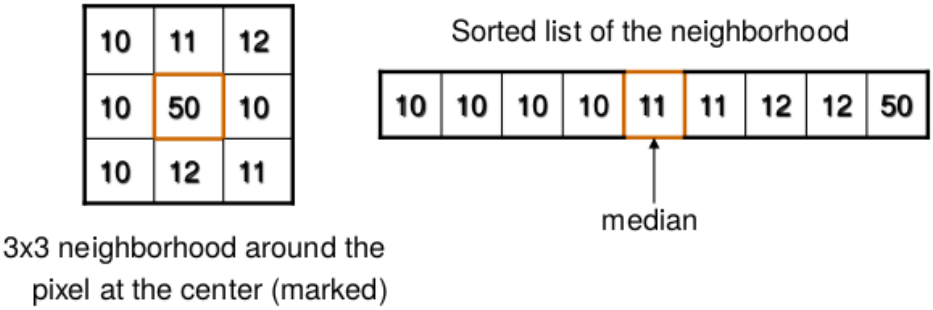
sigma 1.90





2.2.2 Median filter

There are other smoothing techniques besides those relying on convolution. One of them is **median filtering**, which operates by replacing each pixel in the image with the median of its neighborhood. For example, considering a 3×3 neighborhood:



Median filtering is quite good preserving borders (it doesn't produce image blurring), and is very effective to remove salt&pepper noise.

An **important drawback** of this technique is that it is not a linear operation, so it exhibits a high computational cost. Nevertheless there are efficient implementations like pseudomedian, sliding median, etc.

ASSIGNMENT 4: Playing with the median filter

Let's see if this filter could be useful for our plate number recognition system. For that, complete the `median_filter()` method in a similar way to the previous techniques. This method takes as inputs:

- the initial image, and
- the window aperture size (`w_window`), that is, the size of the neighborhood.

Tip: take a look at `cv2.medianBlur()`

In [9]:
```
# ASSIGNMENT 4
# Implement a function that:
# -- applies a median filter to the input image
# -- displays the input image and the filtered one in a 1x2 plot if verbose = Tr
# -- returns the smoothed image
def median_filter(image, w_window, verbose=False):
    """ Applies median filter to an image and display it.

        Args:
            image: Input image
            w_window: window aperture size
            verbose: Only show images if this is True

        Returns:
            smoothed_img: smoothed image
    """

    #Apply median filter
    smoothed_img = cv2.medianBlur(image, w_window * 2 + 1)

    if verbose:
```

```
        # Show the initial image
        plt.subplot(121)
        plt.imshow(image, cmap='gray')
        plt.title('Noisy')
        plt.show()

        # Show the resultant one
        plt.subplot(122)
        plt.imshow(smoothed_img, cmap='gray')
        plt.title('Median filter')

    return smoothed_img
```

You can use the next code to **test if your results are correct**:

In [10]:
```
image = np.array([[1,6,2,5],[10,6,22,7],[7,7,13,0],[0,2,8,4]], dtype=np.uint8)
w_window = 2
print(median_filter(image, w_window))
```

```
[[6 5 5 5]
 [6 5 5 5]
 [6 5 5 5]
 [6 4 4 4]]
```

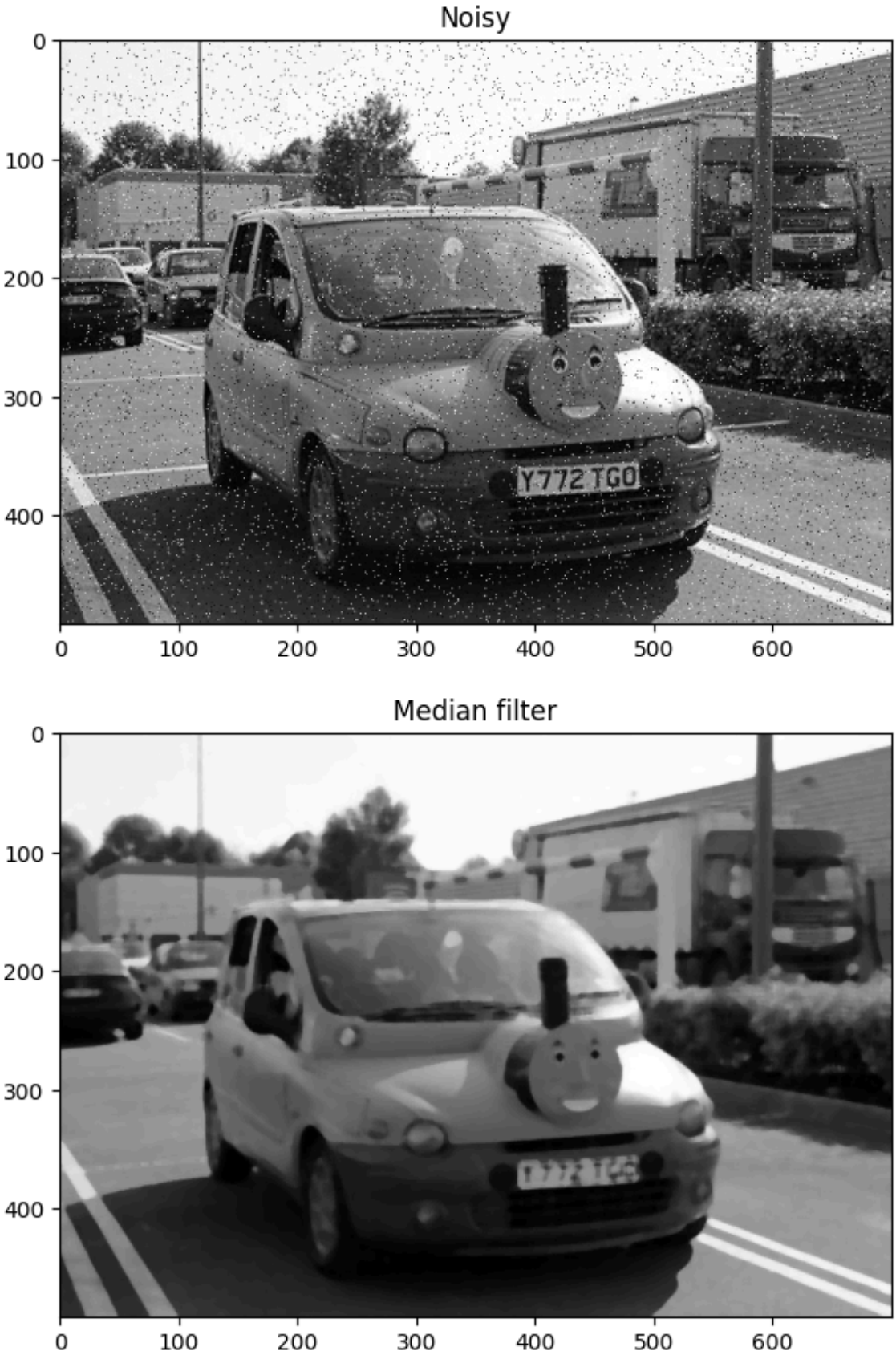
Expected output:

```
[[6 5 5 5]
 [6 5 5 5]
 [6 5 5 5]
 [6 4 4 4]]
```

Now play a bit with the parameters of the algorithm!

In [11]:
```
# Interact with the window size
noisy_img = cv2.imread(images_path + 'noisy_2.jpg', 0)
interactive(median_filter, image=fixed(noisy_img), w_window=(1,5,1), verbose=fix
```

Out[11]: w_window 3



Thinking about it (4)

You are asked to try **median_filter** using both noisy images `noisy_1.jpg` and `noisy_2.jpg`. Then, **answer following questions**:

- Is the noise removed from the first image?

  There's a diminishing of the noise to some extent, but it is negligible. If I had to give a definitive answer, I'd say no.

- Is the noise removed from the second image?

  Absolutely yes. The second image has salt and pepper noise, and the median filter technique is the one to implement when there's this kind of noise, as it achieves the best results.

- Which value is a good choice for `w_window`? Why?

  The best value in this case is 1. It removes every bit of salt and pepper noise and it hardly diminishes the detail. A greater value is not necessary, as the noise is already removed with `w_window = 1

2.2.3 Image average

Next, we asked UMA for the possibility to change their camera from a single shot mode to a multi-shot sequence of images. This is a continuous shooting mode also called *burst mode*. They were very kind and provided us with the sequences `burst1_(0:9).jpg` and `burst2_(0:9).jpg` for testing.

Image sequences allow the usage of **image averaging** for noise removal, the last technique we are going to try. In this technique the content of each pixel in the final image is the result of averaging the value of that pixel in the whole sequence. Remark that, in the context of our application, this technique will work only if the car is fully stopped!

The idea behind image averaging is that using a high number of noisy images from a still camera in a static scene, the resultant image would be noise-free. This is supposed because some types of noise usually has zero mean. Mathematically:

$$g(x,y) = \frac{1}{M}\sum_{i=1}^{M} f_i(x,y) = \frac{1}{M}\sum_{i=1}^{M}[f_{noise\_free}(x,y) + \eta_i(x,y)] = f_{noise\_free}(x,y) + \frac{1}{M}$$

This method:

- is very effective with gaussian noise, and
- it also preserves edges.

On the contrary:

- it doesn't work well with salt&pepper noise, and
- it is only applicable for sequences of images from a still scene.

ASSIGNMENT 5: And last but not least, image averaging

We want to analyze the suitability of this method for our application, so you have to complete the `image_averaging()` method. It takes:

- a sequence of images structured as an array with dimensions [sequence length × height × width], and
- the number of images that are going to be used.

Tip: Get inspiration from here: average of an array along a specified axis

In [12]:
```
# ASSIGNMENT 5
# Implement a function that:
# -- takes a number of images of the sequence (burst_length)
# -- averages the vale of each pixel in the selected part of the sequence
# -- displays the first image in the sequence and the final, filtered one in a 1
# -- returns the average image
def image_averaging(burst, burst_length, verbose=False):
    """ Applies image averaging to a sequence of images and display it.

        Args:
            burst: 3D array containing the fully image sequence.
            burst_length: Natural number indicating how many images are
                          going to be used.
            verbose: Only show images if this is True

        Returns:
            average_img: smoothed image
    """

    #Take only `burst_length` images
    burst = burst[0:burst_length] # The idea is that burst contains x images, an

    # Apply image averaging
    average_img = np.average(burst, axis=0) # I choose axis=0 because I want to

    # Change data type to 8-bit unsigned, as expected by plt.imshow()
    average_img = average_img.astype(np.uint8)

    if verbose:
        # Show the initial image
        plt.subplot(121)
        plt.imshow(burst[0], cmap='gray')
        plt.title('Noisy')
        plt.show()

        # Show the resultant one
        plt.subplot(122)
        plt.imshow(average_img, cmap='gray')
        plt.title('Image averaging')

    return average_img
```

You can use the next code to **test if your results are correct**:

In [13]:
```
burst = np.array([[[1,6,2,5],[10,6,22,7],[7,7,13,0],[0,2,8,4]],
                  [[7,7,13,0],[0,2,8,4],[1,6,2,5],[10,6,22,7]],
                  [[7,7,13,0],[0,2,8,4],[1,6,2,5],[10,6,22,7]]],dtype=np.uint8)
```

```
print(image_averaging(burst, 2))
```

```
[[ 4  6  7  2]
 [ 5  4 15  5]
 [ 4  6  7  2]
 [ 5  4 15  5]]
```

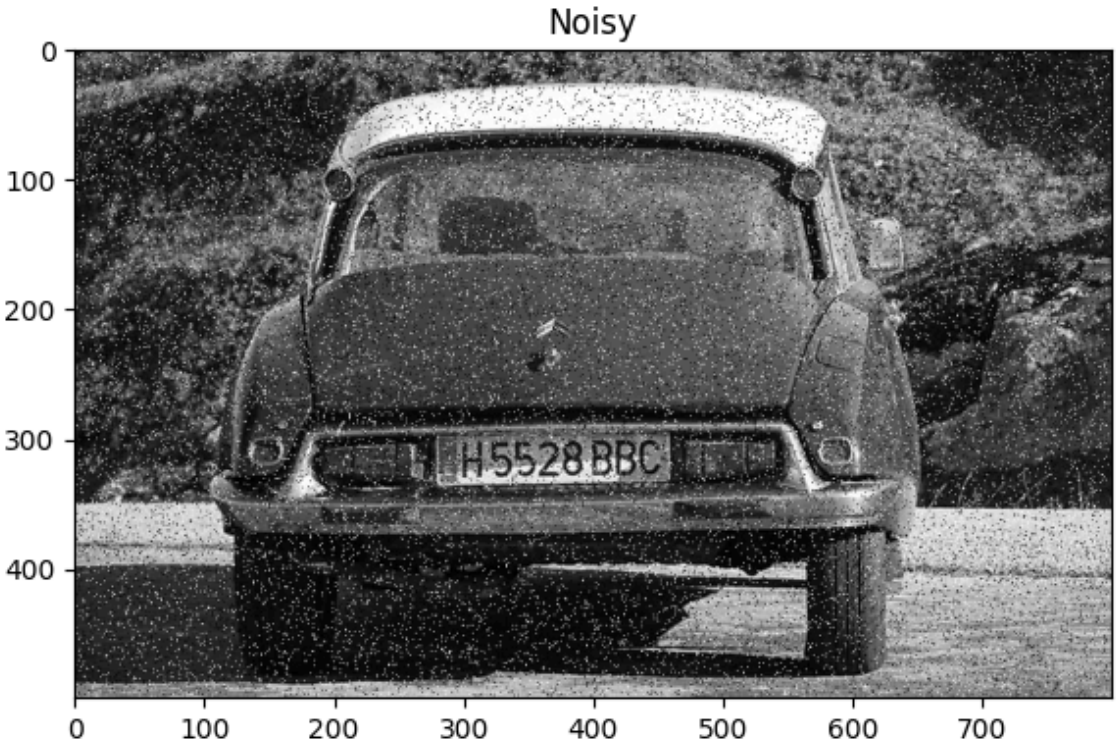
Expected output:

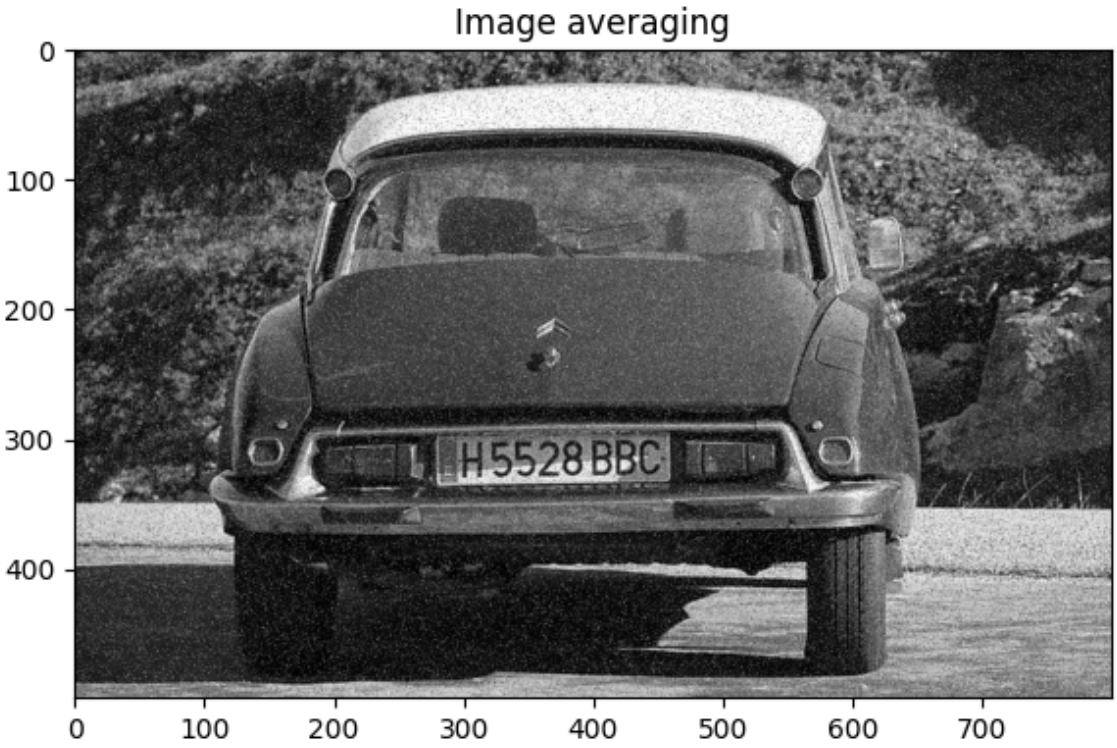
```
[[ 4  6  7  2]
 [ 5  4 15  5]
 [ 4  6  7  2]
 [ 5  4 15  5]]
```

Now check how the number of images used affect the noise removal (play with both sequences):

In [14]:
```
# Interact with the burst length
# Read image secuence
burst = []
for i in range(10):
    burst.append(cv2.imread('./images/burst2_' + str(i) + '.jpg', 0))

# Cast to array
burst = np.asarray(burst)

interactive(image_averaging, burst=fixed(burst), burst_length=(1, 10, 1), verbos
```

Out[14]: burst_length 5





Thinking about it (5)

You are asked to try **image_averaging** with `burst1_XX.jpg` and `burst2_XX.jpg` sequences. Then, **answer these questions**:

- Is the noise removed in both sequences?

  In burst 1, the noise is completely removed (well, almost completely removed), as it is gaussian noise and this technique is very effective with that kind of noise. The second

one is a salt and pepper noise, and it certainly helps (it gets notably reduced) but it doesn't remove the noise totally.

- What number of photos should the camera take in each image sequence?

  As many as possible. The higher the number of images, the closer the noise to zero. But, if we needed to find some middle ground in the whereabouts of effectiveness and efficiency, I'd say that generally between 7-10 photos should be enough (at least, in this case, the images from burst_length=seven onwards are fairly similar).

2.2.4 Choosing a smoothing technique

The next code cell runs the explored smoothing techniques and shows the results provided by each one while processing two different car license plates, **with two different types of noise. Check them!**

In [15]:
```
from time import perf_counter_ns

#Read first noisy image
im1 = cv2.imread('./images/burst1_0.jpg', 0)
im1 = im1[290:340,280:460]

# Read second noisy image
im2 = cv2.imread('./images/burst2_0.jpg', 0)
im2 = im2[290:340,280:460]

# Apply neighborhood averaging
n1_t = perf_counter_ns()
neighbor1 = average_filter(im1, 1)
n1_t = (perf_counter_ns() - n1_t) / 1e3
n2_t = perf_counter_ns()
neighbor2 = average_filter(im2, 1)
n2_t = (perf_counter_ns() - n2_t) / 1e3

# Apply Gaussian filter
g1_t = perf_counter_ns()
gaussian1 = gaussian_filter(im1, 2,1)
g1_t = (perf_counter_ns() - g1_t) / 1e3
g2_t = perf_counter_ns()
gaussian2 = gaussian_filter(im2, 2,1)
g2_t = (perf_counter_ns() - g2_t) / 1e3

# Apply median filter
m1_t = perf_counter_ns()
median1 = median_filter(im1, 1)
m1_t = (perf_counter_ns() - m1_t) / 1e3
m2_t = perf_counter_ns()
median2 = median_filter(im2, 1)
m2_t = (perf_counter_ns() - m2_t) / 1e3

# Apply image averaging
burst1 = []
burst2 = []
for i in range(10):
    burst1.append(cv2.imread('./images/burst1_' + str(i) + '.jpg', 0))
    burst2.append(cv2.imread('./images/burst2_' + str(i) + '.jpg', 0))
```

```
burst1 = np.asarray(burst1)
burst2 = np.asarray(burst2)

burst1 = burst1[:,290:340,280:460]
burst2 = burst2[:,290:340,280:460]

a1_t = perf_counter_ns()
average1 = image_averaging(burst1, 10)
a1_t = (perf_counter_ns() - a1_t) / 1e3
a2_t = perf_counter_ns()
average2 = image_averaging(burst2, 10)
a2_t = (perf_counter_ns() - a2_t) / 1e3

# Plot results
plt.subplot(521)
plt.imshow(im1, cmap='gray')
plt.title('Noisy 1')

plt.subplot(522)
plt.imshow(im2, cmap='gray')
plt.title('Noisy 2')

plt.subplot(523)
plt.imshow(neighbor1, cmap='gray')
plt.title(f'Neighborhood averaging, {n1_t:.2f} µs')

plt.subplot(524)
plt.imshow(neighbor2, cmap='gray')
plt.title(f'Neighborhood averaging, {n2_t:.2f} µs')

plt.subplot(525)
plt.imshow(gaussian1, cmap='gray')
plt.title(f'Gaussian filter, {g1_t:.2f} µs')

plt.subplot(526)
plt.imshow(gaussian2, cmap='gray')
plt.title(f'Gaussian filter, {g2_t:.2f} µs')

plt.subplot(527)
plt.imshow(median1, cmap='gray')
plt.title(f'Median filter, {m1_t:.2f} µs')

plt.subplot(528)
plt.imshow(median2, cmap='gray')
plt.title(f'Median filter, {m2_t:.2f} µs')

plt.subplot(529)
plt.imshow(average1, cmap='gray')
plt.title(f'Image averaging, {a1_t:.2f} µs')

plt.subplot(5,2,10)
plt.imshow(average2, cmap='gray')
plt.title(f'Image averaging, {a2_t:.3f} µs')

print(f'Neighborhood averaging average... {(n1_t + n2_t) / 2:.2f} µs')
print(f'Gaussian filter average.......... {(g1_t + g2_t) / 2:.2f} µs')
print(f'Median filter average............ {(m1_t + m2_t) / 2:.2f} µs')
print(f'Image averaging average.......... {(a1_t + a2_t) / 2:.2f} µs')
```

```
Neighborhood averaging average... 79.30 µs
Gaussian filter average.......... 91.60 µs
Median filter average............ 43.65 µs
Image averaging average.......... 260.45 µs
```

Thinking about it (6)

And the final question is:

- **What method would you choose** for a final implementation in the system? *Why?*

  *It depends on whether I care about the best possible result, the fastest possible (usable) result or something in the middle. When it comes to effectiveness, **for gaussian noise**; with the proper conditions (every image taken are images of the still object, not in movement), image averaging is the one that achieves the best results. It is, for gaussian noise, 235.5 / 91 = 2.58 times slower, though. For an implementation like this, in which we do not need to take thousands of photos, **it would be alright as 140 microseconds is a negligible time**. Things get serious when that 2.58 times slower is something like 1 minute vs 2.58 minutes. Or 1 hour vs 2.58 hours. I guess it just depends on the system in which you are going to use it. **And for the second image, the one with salt and pepper noise, I'd just use median filter**. It is*

significantly faster and achieves better results than the rest, so it is a no-brainer to use it when there's that kind of noise.

Conclusion

That was a complete and awesome job! Congratulations, you learned:

- how to reduce noise in images, for both salt & pepper and Gaussian noise,
- which methods are useful for each type of noise and which not, and
- to apply convolution and efficient implementations of some kernels.

If you want to improve your knowledge about noise in digital images, you can surf the internet for *speckle noise* and *Poisson noise*.

2.3 Image enhancement

Image enhancement is the process of adjusting a digital image so the resultant one is more suitable for further image analysis (edge detection, feature extraction, segmentation, etc.), in other words, **its goal is to improve the contrast and brightness of the image**.

There are three typical operations for enhancing images. We have already explored one of them in notebook *2.1 IP tools*: (linear) Look-Up Tables (LUTs). In this notebook we will play with a variant of LUTs and other two operations:

- Non-linear look-up tables (Section 2.3.1).
- Histogram equalization (Section 2.3.2).
- Histogram specification (Section 2.3.3).

Also, some color-space conversions are going to be needed. If you are not familiar with the YCrCb color space, **Appendix 2: Color spaces** contains the information you need to know about it.

Problem context - Implementing enhancement techniques for an image editor tool

We have all tried an image editor tool, sometimes without even knowing it! For example, modern smartphones already include an application for applying filters to images, cut them, modify their contrast, brightness, color temperature, etc.

One example of open source tool is the GNU Image Manipulation Program (GIMP). Quoting some words from its website:

> GIMP is a cross-platform image editor available for GNU/Linux, OS X, Windows and more operating systems. It is free software, you can change its source code and distribute your changes. Whether you are a graphic designer, photographer, illustrator, or scientist, GIMP provides you with sophisticated tools to get your job done. You can further enhance your productivity with GIMP thanks to many customization options and 3rd party plugins.

In this case we were contacted by UMA for implementing two techniques to be included in their own image editor tool! Concretely, we were asked to develop and test two methods that are also part of GIMP: **gamma correction** and **equalize**.

In [1]:
```
import numpy as np
import cv2
import matplotlib.pyplot as plt
import matplotlib
from ipywidgets import interactive, fixed, widgets
matplotlib.rcParams['figure.figsize'] = (20.0, 20.0)

images_path = './images/'
```

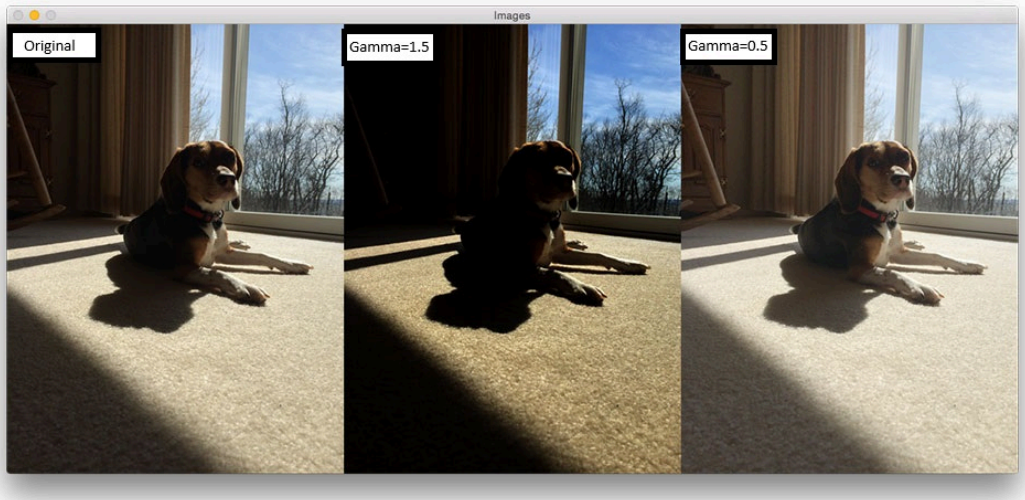
2.3.1 Non-linear look-up tables

Gamma correction, or often simply **gamma**, is a nonlinear operation used to adjust the luminance or brightness levels of an image. In other words, it is the result of applying an (already defined) **non-linear LUT** in order to stretch or shrink image intensities.

In this way, the gamma LUT definition for grayscale images, where each pixel i takes values in the range $[0 \dots 255]$, is:

$$LUT(i) = (\frac{i}{255})^{\gamma} * 255,\ \gamma > 0$$

The following images illustrate the application of gamma correction for different values of γ.



The role of γ:

- $\gamma < 1$: The image is lightened. Dark areas become brighter, enhancing shadow details.
- $\gamma = 1$: No change is applied; the output is identical to the input.
- $\gamma > 1$: The image is darkened. Bright areas become darker, which can reduce glare or overexposure.

ASSIGNMENT 1: Applying non-linear LUTs

Your task is to develop the `lut_chart()` function, which takes as arguments the image to be enhanced and a gamma value for building the non-linear LUT. It will also display a chart containing the original image, the gamma-corrected one, the used LUT and the histogram of the resulting image.

As users from UMA will use color images, you will have to **implement it for color images**. This can be done by:

1. **transforming** an image in the BGR color space **to the YCrCb one**,
2. then, **applying gamma LUT only to first band** of the YCrCb space (that's because it contains pixel intensities and you can handle it like a gray image), and
3. finally, as matplotlib displays RGB images (if verbose is True), it should be **converted back**. Also, return the resultant image.

Interesting functions:

- `np.copy()` : method that returns a copy of the array provided as input.
- `cv2.LUT()` : *function that performs a look-up table transform of an array of arbitrary dimensions.*
- `plt.hist()` *function that computes and draws the histogram of an array.* `numpy.ravel()` *is a good helper here, since it converts a n-dimensional array into a flattened 1D array.*

In [2]:
```
# ASSIGNMENT 1
# Implement a function that:
# -- converts the input image from the BGR to the YCrCb color space
# -- creates the gamma LUT
# -- applies the LUT to the original image
# -- displays in a 2x2 plot: the input image, the gamma-corrected one, the appli
def lut_chart(image, gamma, verbose=False):
    """ Applies gamma correction to an image and shows the result.

        Args:
            image: Input image
            gamma: Gamma parameter
            verbose: Only show images if this is True

        Returns:
            out_image: Gamma image
    """

    #Transform image to YCrCb color space
    image = cv2.cvtColor(image, cv2.COLOR_BGR2YCrCb)
    out_image = np.copy(image)

    # Define gamma correction LUT
    lut = np.array([((i / 255.0) ** gamma) * 255 for i in np.arange(0, 256)]).as

    # Apply LUT to first band of the YCrCb image
    out_image[:,:,0] = cv2.LUT(image[:,:,0],lut) # [:,:,0] selects every row, co
```

```
    if verbose:
        # Plot used LUT
        plt.subplot(2,2,3)
        plt.title('LUT')
        plt.plot(lut)


        # Plot histogram of gray image after applying the LUT
        plt.subplot(2,2,4)
        plt.hist(out_image[:,:,0].ravel(),256,[0,256])
        plt.title('Histogram')


        # Reconvert image to RGB
        image = cv2.cvtColor(image, cv2.COLOR_YCrCb2RGB)  # careful! RGB, not BG
        out_image = cv2.cvtColor(out_image, cv2.COLOR_YCrCb2RGB)

        # Show the initial image
        plt.subplot(2,2,1)
        plt.imshow(image)
        plt.title('Original image')

        # Show the resultant one
        plt.subplot(2,2,2)
        plt.imshow(out_image)
        plt.title('LUT applied')

    return out_image
```

You can use the next code to **test if results are correct**:

In [3]:
```
image = np.array([[[10,60,20],[60,22,74],[72,132,2]],[[11,63,42],[36,122,27],[37
gamma = 2
print(lut_chart(image, gamma))
```

```
[[[  6 112 110]
  [  6 151 138]
  [ 29  68 120]]

 [[ 10 122 105]
  [ 27  87 101]
  [ 25  92 104]]

 [[  0 127 126]
  [  1 122 122]
  [  0 122 127]]]
```

Expected output:

```
[[[  6 112 110]
  [  6 151 138]
  [ 29  68 120]]

 [[ 10 122 105]
  [ 27  87 101]
  [ 25  92 104]]

 [[  0 127 126]
```

```
[  1 122 122]
[  0 122 127]]]
```

Thinking about it (1)

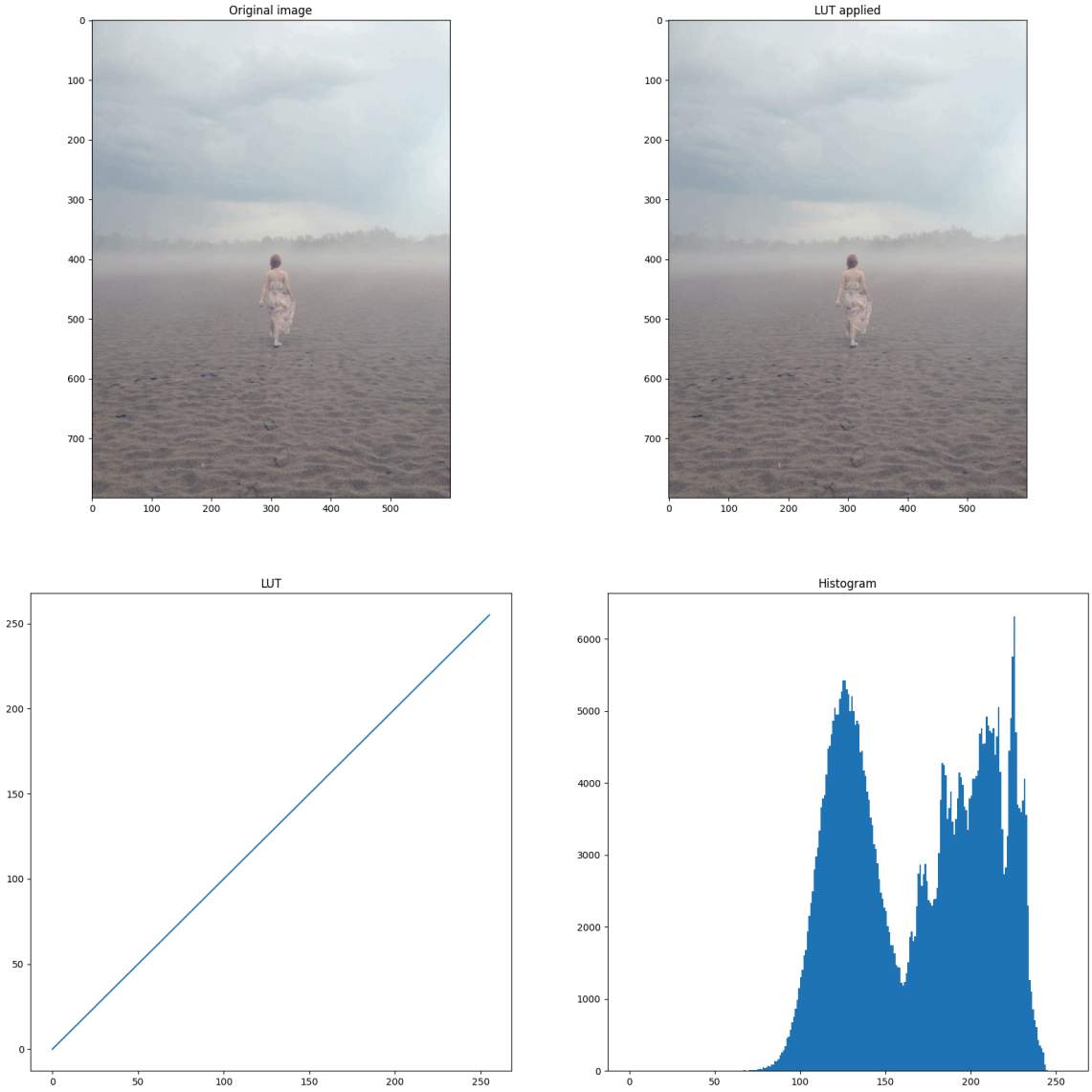
In the interactive code cell below, **you are asked to** explore how your new `lut_chart()` function works with `gamma_1.jpg` (an underexposed image) and `gamma_2.jpeg` (an overexposed image). Then, **answer the following question** (you can take a look at the LUT and the resulting histogram):

- What is happening when the *gamma value* is modified?

  *If gamma< 1, the image is lightened. **In the first picture**, the image is quite dark as it is underexposed. Lowering gamma below 1 (somewhere in the whereabouts [0.4, 0.6]) will apply the gamma function as a LUT, enriching the image with further detail now that stuff is more easily distinguishable in the darker spots. It, basically, exploits the dynamic range of the image better, as now intensities are better distributed. Something similar happens with **image 2**, but now it has overexposure (gamma> 1, the image is darkened). Applying a gamma in the range [2,3] will drammatically improve the detail, as now the dynamic range of the photo is better used.*

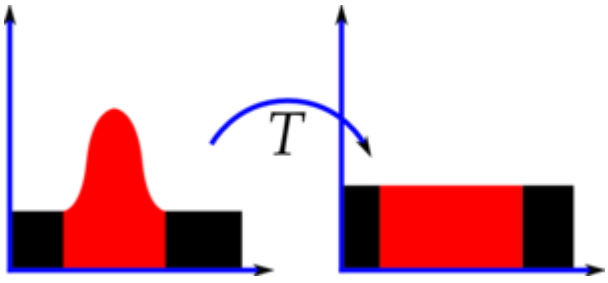
In [4]:
```
# Create widget object
gamma_widget = widgets.FloatSlider(value=1, min=0.1, max=5, step=0.1, descriptio

#Read image
image = cv2.imread(images_path + 'gamma_2.jpeg',-1)

#Interact with your code!
interactive(lut_chart, image=fixed(image), gamma=gamma_widget, verbose=fixed(Tru
```

Out[4]: Gamma: 1.00



2.3.2 Histogram equalization

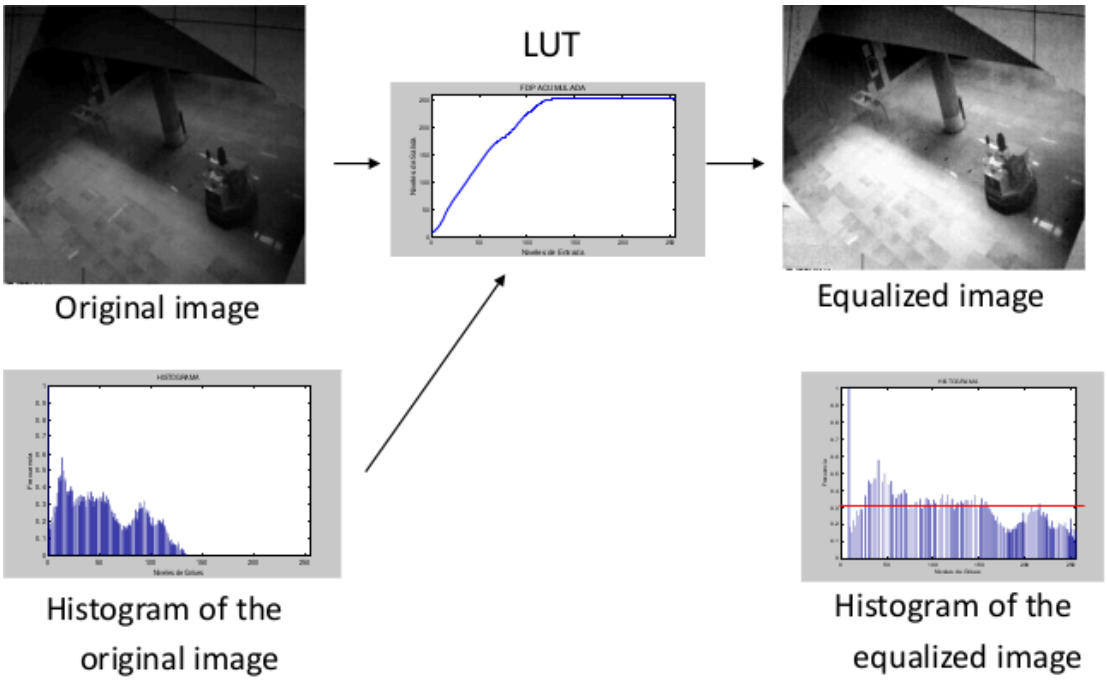
Histogram equalization is an image processing technique used to improve contrast in images. It operates by effectively spreading out the most frequent intensity values, i.e. stretching out the intensity range of the image so each possible pixel intensity appears the same number of times as every other value. This method usually increases the global contrast of images when its usable data is represented by close contrast values. This allows for areas of lower local contrast to gain a higher contrast.



To put an example, the **equalize** command from GIMP applies histogram equalization. But... how is this equalization achieved?

- First it is calculated the PMF (probability mass function) of all the pixels in the image. Basically, this is a normalization of the histogram.
- Next step involves calculation of CDF (cumulative distributive function), producing the LUT for histogram equalization.
- Finally, the obtained LUT is applied.

The figure below shows an example of applying histogram equalization to an image.



ASSIGNMENT 2: Equalizing the histogram!

Similarly to the previous exercise, **you are asked to** develop a function called `equalize_chart()`. This method takes a **color** image, and will display a plot containing:

- the original image,
- the equalized image,
- the original image histogram, and
- the equalized image histogram.

Tip: openCV implements histogram equalization in cv2.equalizeHist()

In [5]:
```
# ASSIGNMENT 2
# Implement a function that:
# -- converts the input image from the BGR to the YCrCb color space
# -- applies the histogram equalization
# -- displays in a 2x2 plot: the input image, the equalized one, the original hi
def equalize_chart(image, verbose=False):
```

```
    """ Applies histogram equalization to an image and shows the result.

        Args:
            image: Input image
            verbose: Only show images if this is True

        Returns:
            out_image: Equalized histogram image
    """

    #Transform image to YCrCb color space
    image = cv2.cvtColor(image, cv2.COLOR_BGR2YCrCb)
    out_image = np.copy(image)

    # Apply histogram equalization to first band of the YCrCb image
    out_image[:,:,0] = cv2.equalizeHist(image[:,:,0])

    if verbose:

        # Plot histogram of gray image
        plt.subplot(2,2,3)
        plt.hist(image[:,:,0].ravel(),bins=256,range=[0,256])  # careful! You mu
        plt.title('Original histogram')

        # Plot equalized histogram of the processed image
        plt.subplot(2,2,4)
        plt.hist(out_image[:,:,0].ravel(),bins=256,range=[0,256]) # careful! You
        plt.title('Equalized histogram')

        # Reconvert image to RGB
        image = cv2.cvtColor(image, cv2.COLOR_YCrCb2RGB)
        out_image = cv2.cvtColor(out_image, cv2.COLOR_YCrCb2RGB)

        # Show the initial image
        plt.subplot(2,2,1)
        plt.imshow(image)
        plt.title('Original image')

        # Show the resultant one
        plt.subplot(2,2,2)
        plt.imshow(out_image)
        plt.title('Equalized histogram image')

    return out_image
```

You can use the next code to **test if your results are correct**:

In [6]:
```
image = np.array([[[10,60,20],[60,22,74],[72,132,2]],[[11,63,42],[36,122,27],[37
print(equalize_chart(image))
```

```
[[[128 112 110]
  [128 151 138]
  [255  68 120]]

 [[159 122 105]
  [223  87 101]
  [191  92 104]]

 [[  0 127 126]
  [ 64 122 122]
  [ 32 122 127]]]
```

Expected output:

```
[[[128 112 110]
  [128 151 138]
  [255  68 120]]

 [[159 122 105]
  [223  87 101]
  [191  92 104]]

 [[  0 127 126]
  [ 64 122 122]
  [ 32 122 127]]]
```

Thinking about it (2)

We have developed our second image enhancement technique! Now try `equalize_chart()` with the `park.png` image in the code cell below. Then, **answer following questions**:

- What is the difference between the original histogram and the equalized one?

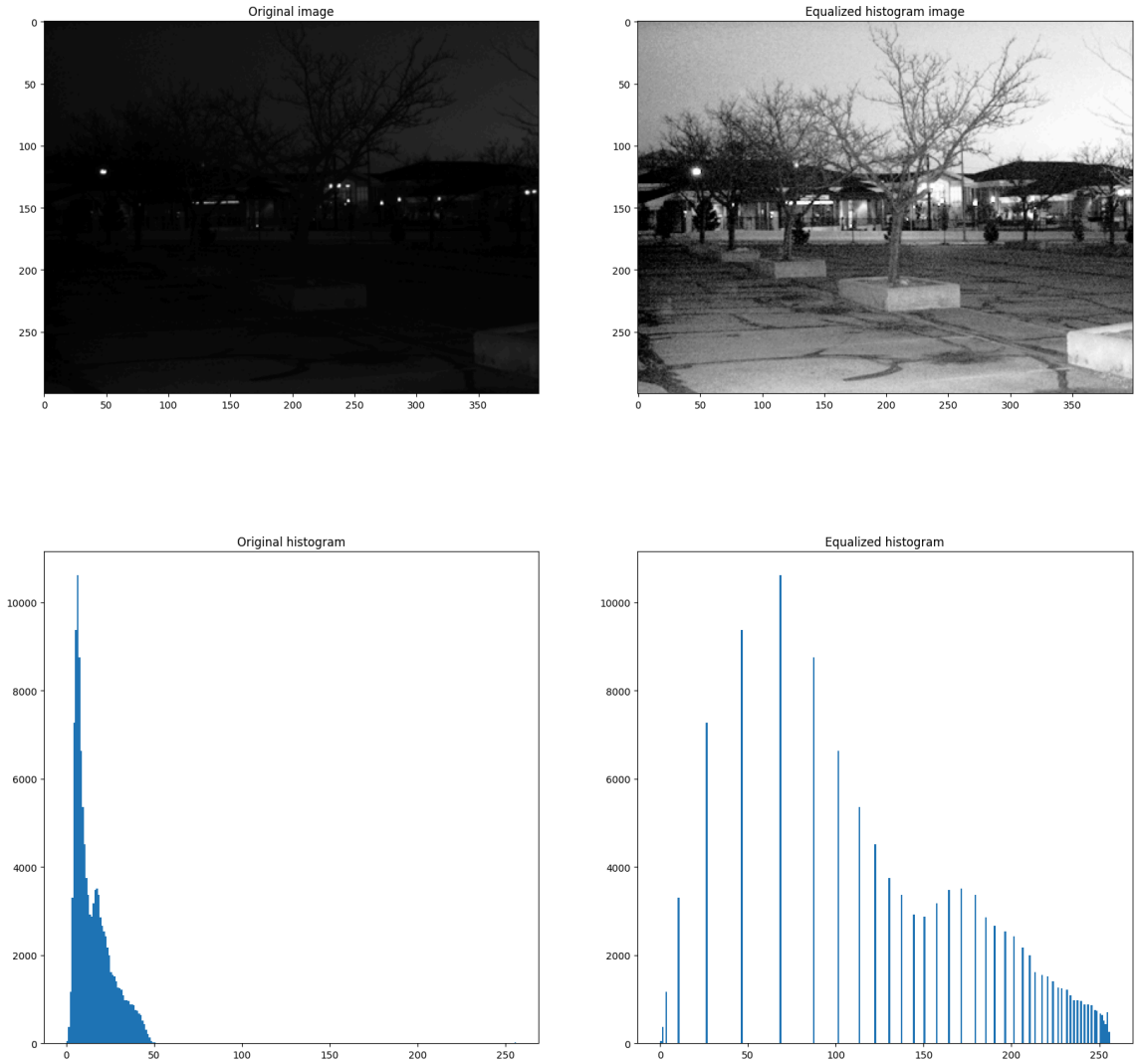
  In the first one, the histogram has a clear tendency to the darkest intensities. By equalizing it, those intensities are distributed through the histogram, resulting in a well-used dyncamic range (and, by extension, higher contrast). Now, we can distinguish the items in the dark photo easily.

- Is the final histogram uniform? why?

  No, it is not. There are wide gaps between intensity peaks, because the equalized histogram depends on the original image's pixel distribution. In other words, equalization only "stretches" the histogram in order to take advantage of the whole colouring range, which doesn't ensure a uniform histogram.

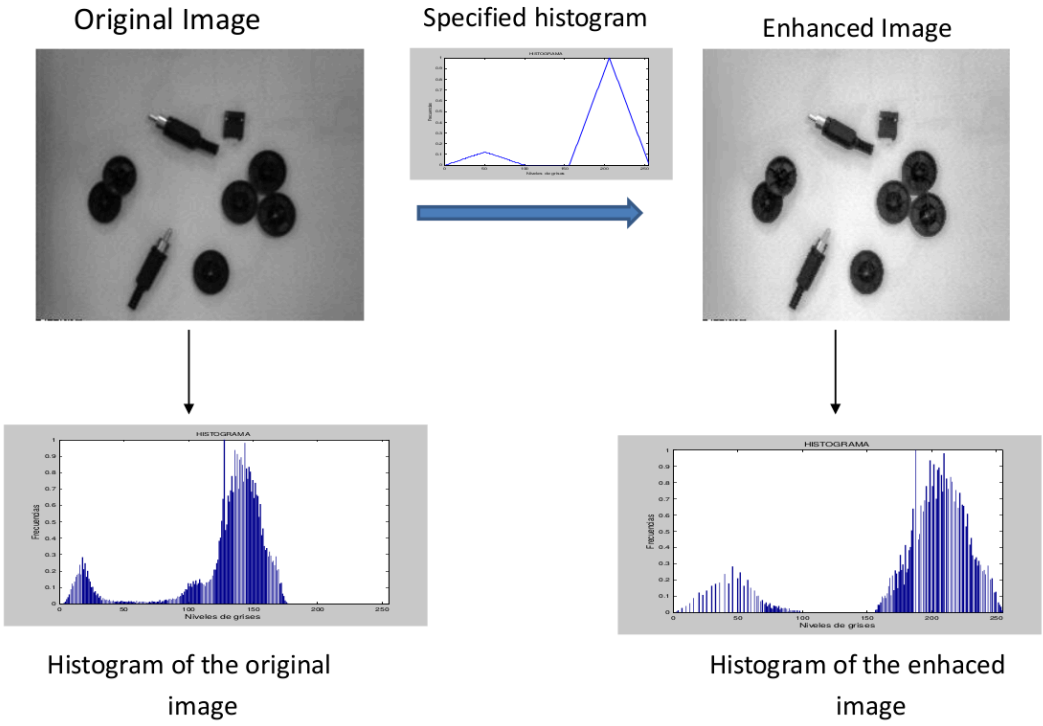
In [7]:
```
# Read image
image = cv2.imread(images_path + 'park.png',-1)

# Equalize its histogram
interactive(equalize_chart, image=fixed(image), verbose=fixed(True))
```

Out[7]:



2.3.3 Histogram specification

Histogram specification is the transformation of an image so that its histogram matches a specified one. In fact, the histogram equalization method is a special case in which the specified histogram is uniformly distributed.



Histogram of the original image

Histogram of the enhaced image

It's implementation is very similar to histogram equalization:

- First it is calculated the PMF (probability mass function) of all the pixels in both (source and reference) images.
- Next step involves calculation of CDF (cumulative distributive function) for both histograms (F_1 for source histogram and F_2 for reference histogram).
- Then for each gray level $G_1 \in [0, 255]$, we find the gray level G_2, for which $F_1(G_1) = F_2(G_2)$, producing the LUT for histogram equalization.
- Finally, the obtained LUT is applied.

ASSIGNMENT 3: Let's specify the histogram

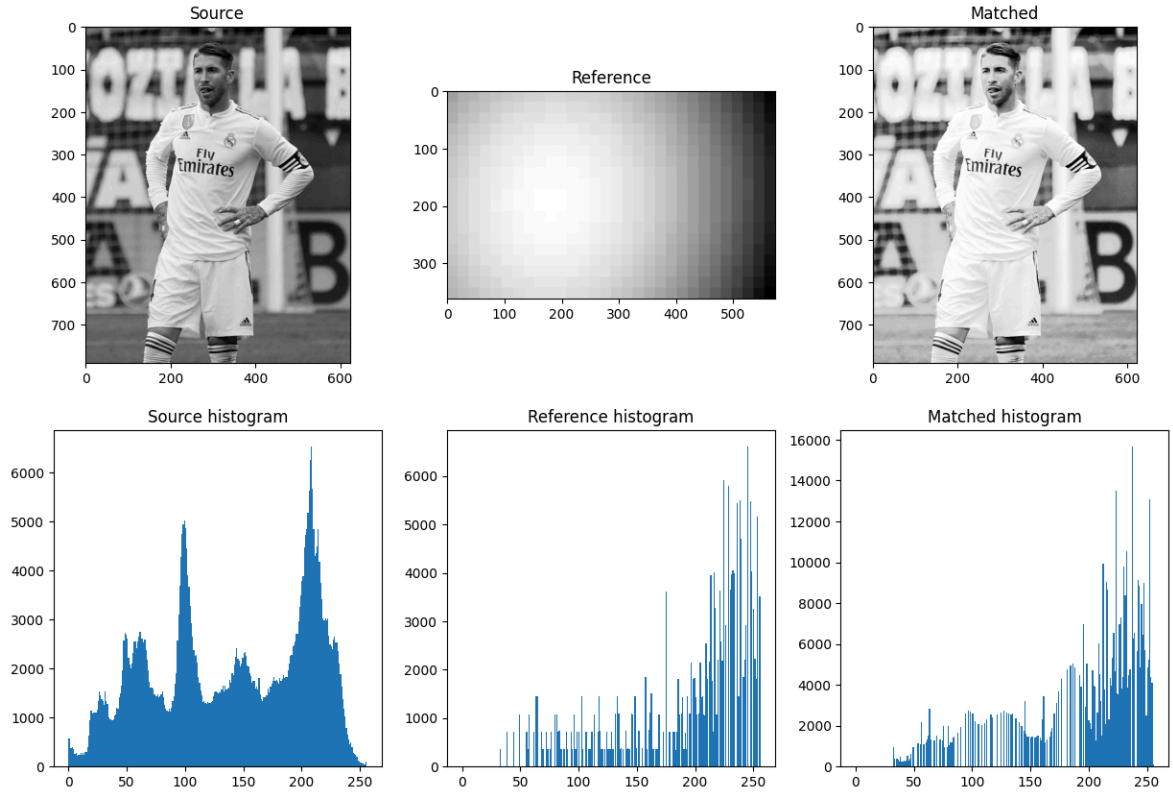
Apply histogram specification using the `ramos.jpg` (image to enhance) and `illumination.png` (reference) gray images. Then, show the resultant image along with input images (show their histograms as well).

Unfortunately, histogram specification is not implemented in our loved OpenCV. In this case you have to rely on the skimage.exposure.match_histograms() function from the also popular scikit-image library.

In [8]:
```
# ASSIGNMENT 3
# Write your code here!
from skimage.exposure import match_histograms

matplotlib.rcParams['figure.figsize'] = (15.0, 10.0)

image = cv2.imread(images_path + "ramos.jpg",0)
reference = cv2.imread(images_path + "illumination.png",0)
```

```
matched = match_histograms(image, reference)

# Plot results
plt.subplot(231)
plt.imshow(image, cmap='gray')
plt.title('Source')

plt.subplot(232)
plt.imshow(reference, cmap='gray')
plt.title('Reference')

plt.subplot(233)
plt.imshow(matched, cmap='gray')
plt.title('Matched')

plt.subplot(234)
plt.hist(image.ravel(),bins=256,range=[0,256])
plt.title('Source histogram')

plt.subplot(235)
plt.hist(reference.ravel(),bins=256,range=[0,256])
plt.title('Reference histogram')

plt.subplot(236)
plt.hist(matched.ravel(),bins=256,range=[0,256])
plt.title('Matched histogram')
```

Out[8]: Text(0.5, 1.0, 'Matched histogram')



Conclusion

Great! We are sure that UMA users are going to appreciate your efforts. Also, next time you use an image editor tool you are going to have another point of view of how things

work.

In conclusion, in this notebook you have learned:

- How to define a **gamma correction (non-linear) LUT** and to how to apply it to an image.
- How **histogram specification** works and its applications. When the specified histogram is uniformly distributed, we call it **histogram equalization**.

Extra

But this doesn't have to be the end, open GIMP and look through others implemented methods.

As you are learning about image processing, **comment how you think they are implemented from scratch.**