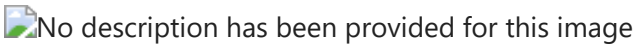
2.2 Smoothing

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.

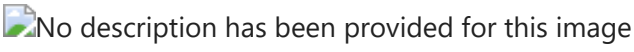
- **Gaussian** noise (intensities are affected by an additive zero-mean Gaussian error). Typical cause: Poor illumination or high temperatures that affect the electronics.



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?

  Claramente, se aprecia una diferenecia en el tipo de ruido, en la primera imagen se observa un ruido gaussiano y en la segunda se vislumbra un ruido salt and pepper.

- Why can this happen (the noise)?

  Puede deberse, en la primera imagen, a que el coche se encuentre en una zona mal iluminada o con niveles altos de calor. En el segundo caso a distintos errores en el

sensor de la camara, pixeles muertos o errores de transmisión.

- What could we do to face this issue?

  Podriamos probar a amplicar diferentes tecnicas de suavizado, para ver cual es más efectiva frente a los distintos tipos de ruido

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.

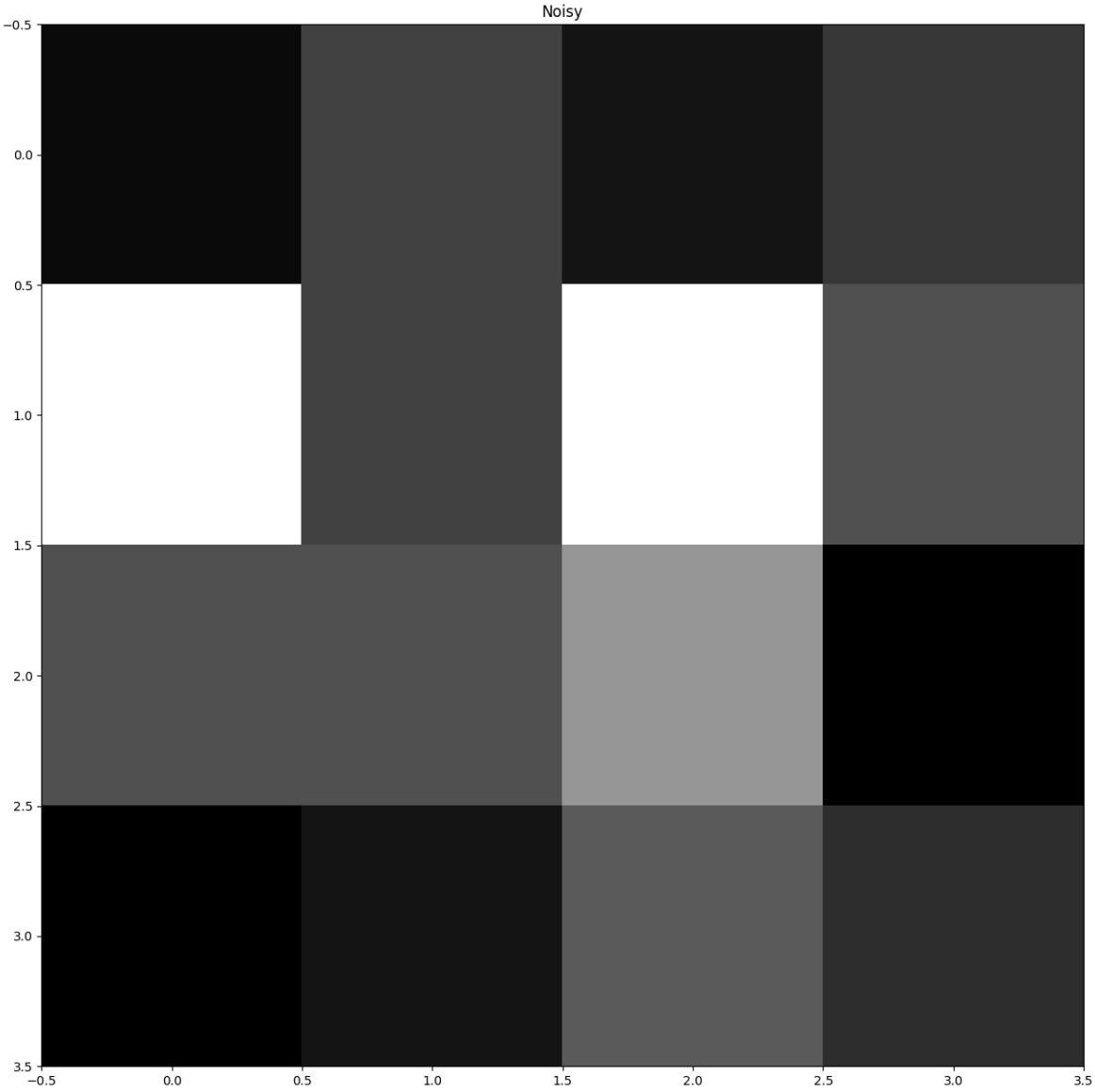
Tip: OpenCV defines the 2D-convolution cv2.filter2D(src, ddepth, kernel) method, where:

- *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)*

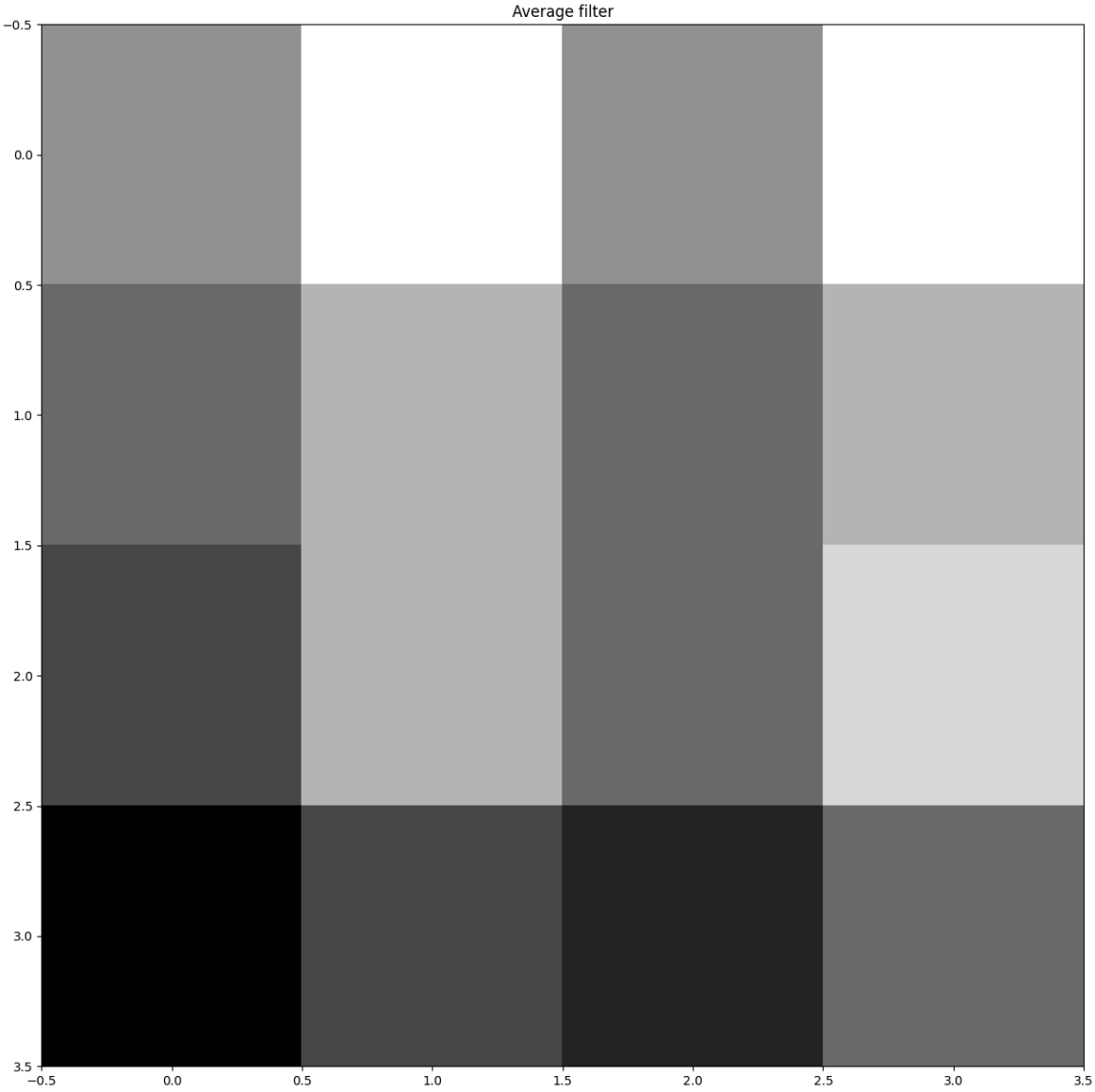
In [3]:
```
# ASSIGNMENT 2
# Implement a function that applies an 'average filter' to an input image. The k
# Show the input image and the resulting one in a 1x2 plot.
def average_filter(image, w_kernel, verbose=False):
    """ Applies neighborhood averaging to an image and display the result.

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

  En gran parte sí, con parametros más altos la imagen queda muy borrosa pero casi sin ruido.

- Is the noise removed from the second image?

  No, en la segunda imagen se sigue apreciando ruido.

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

Yo diría, que 3 es suficiente, elimina ruido y no hace demasiado borrosa la imagen.>

In [5]:
```
# Interact with the kernel size
noisy_img = cv2.imread(images_path + 'noisy_2.jpg', 0)
interactive(average_filter, image=fixed(noisy_img), w_kernel=(0,5,1), verbose=fi
```

Out[5]:
```
interactive(children=(IntSlider(value=2, description='w_kernel', max=5), Output
()), _dom_classes=('widget-inte…
```

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:

$$g = g_y \otimes g_x \rightarrow \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$$

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.

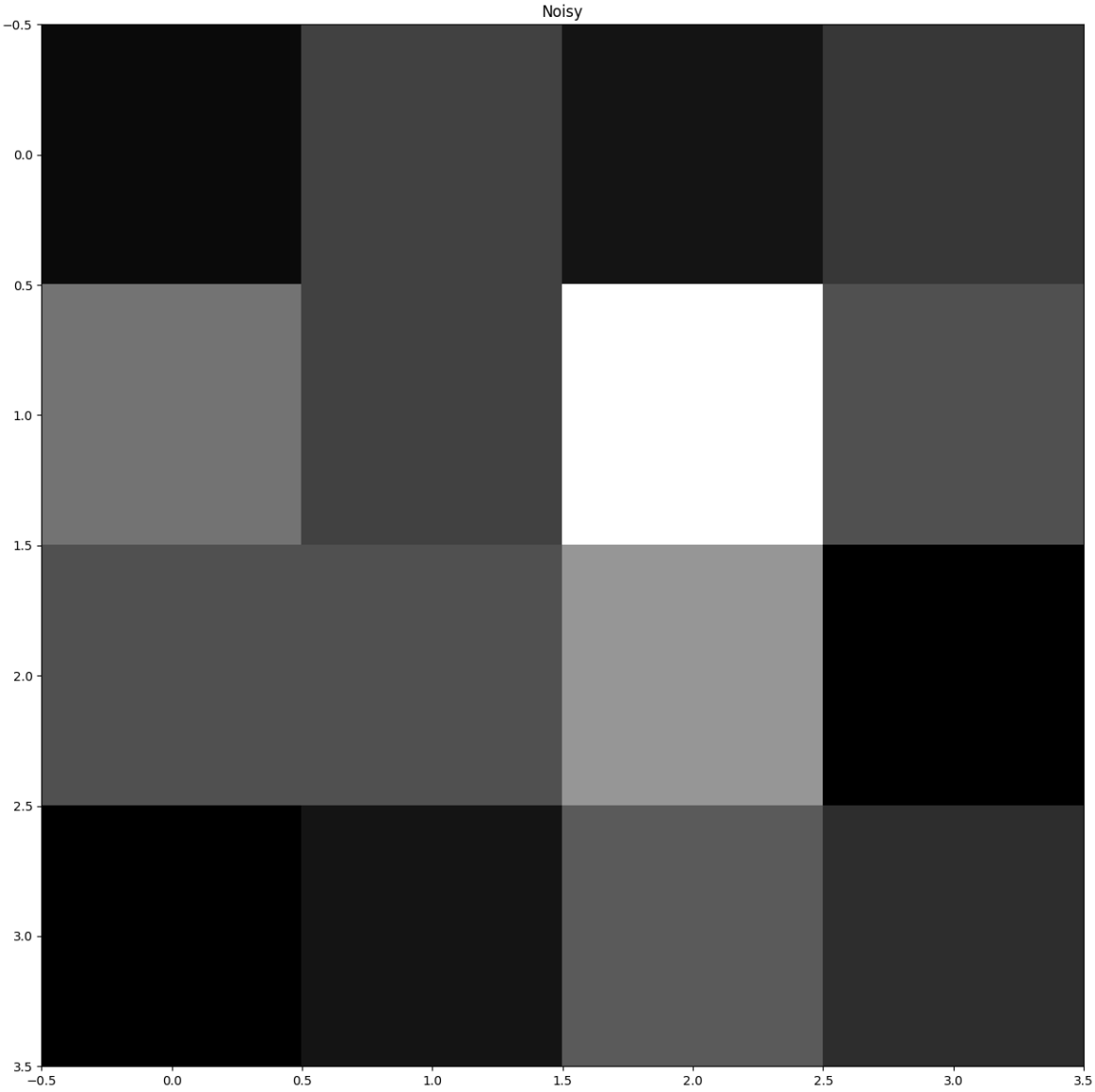
*Tip 2: The code below uses **List Comprehension** for creating a list of numbers by evaluating an expression within a `for` loop. Its syntax is: `[expression for item in list]`. You can find multiple examples of how to create lists using this technique on the internet.*

In [6]:
```
# 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
```

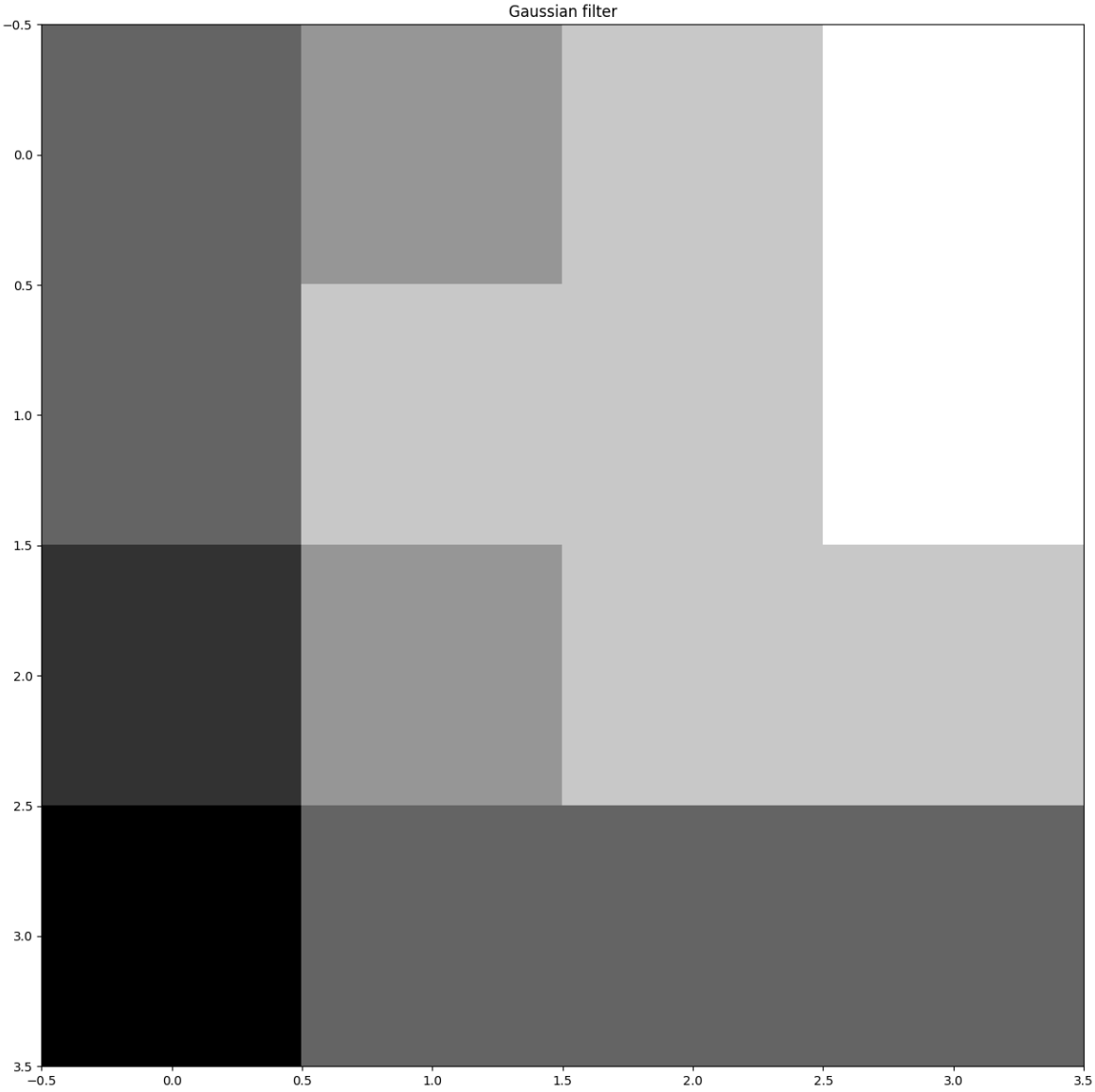
```
            w_kernel: Kernel aperture size
            sigma: standard deviation of Gaussian distribution
            verbose: Only show images if this is True

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

    # Create kernel using associative property
    s = sigma
    w = w_kernel
    kernel_1D = np.float32([(1/(s*np.sqrt(2*np.pi)))*np.exp(-(z**2)/(2*(s**2)))
    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

    # 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?

  Otra vez, el ruido se elimina bastante bien, pudiendo conseguir en este caso una imagen menos borrosa para ello.

- Is the noise removed from the second image?

  No, sigue estando presente

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

Para w_kernel con 2 es suficiente y utilizando un sigma 3 el resultado es aceptable.

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]:
```
interactive(children=(IntSlider(value=2, description='w_kernel', max=5), FloatS
lider(value=1.9, description='s…
```

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:

 No description has been provided for this image

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
```

```
    if w_window%2 == 0:
        w_window += 1
    smoothed_img = cv2.medianBlur(image,w_window)

    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 6 6 5]
 [7 7 6 5]
 [7 7 7 7]
 [2 7 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]:
```
interactive(children=(IntSlider(value=3, description='w_window', max=5, min=1),
Output()), _dom_classes=('widg…
```

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?

  En este caso da fallos con la primera imagen, no elimina el ruido. de hecho cuando aumentamos demasiado el tamaño del kernel, parece un cuadro hecho con pintura.

- Is the noise removed from the second image?

Sí, por completo incluso unicamente con un tamaño de 2.

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

  Un valor de 2 es el idóneo, no modifica demasiado la imagen y además elimina por completo el ruido salt and pepper.

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 $\times$ height $\times$ 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[:burst_length]

    # Apply image averaging
    average_img = np.average(burst,axis=0)

    # 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]:
```
interactive(children=(IntSlider(value=5, description='burst_length', max=10, mi
n=1), Output()), _dom_classes=(…
```

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?

  El gaussiano lo elimina por completo, el salt and pepper no del todo.

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

  En el gaussiano, con cinco imagenes, queda perfecta la imagen, el salt and pepper es un poco más complicado, con 10 imagenes el resultado es decente. Probé a meterle 50 imagenes pero claro en la carpeta nada más habia 10 imagenes así que no sirvió de nada.

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... 90.20 µs
Gaussian filter average.......... 102.95 µs
Median filter average............ 37.80 µs
Image averaging average.......... 257.00 µs
```

Thinking about it (6)

And the final question is:

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

  Es una pregunta muy complicada. El mejor en cuanto a resultado es claramente el image averaging, pero el tiempo que tarda en procesarese, es literalmente, 11 veces mayor en el caso del ruido gaussiano que el de median filter. Yo optaria por un punto intermedio, el gaussian filter, su coste es medio y lo resultados no están mal, solo que dejan la imagen borrosa. Aunque teniendo en cuenta que 339 microsegundos es nada para un humano, si estuvieramos hablando de un programa real para la etsi, claramente usaria el image averaging, pues no van a entrar tantos coches a la vez como para que 339 micro segundos suponga un problema.

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*.