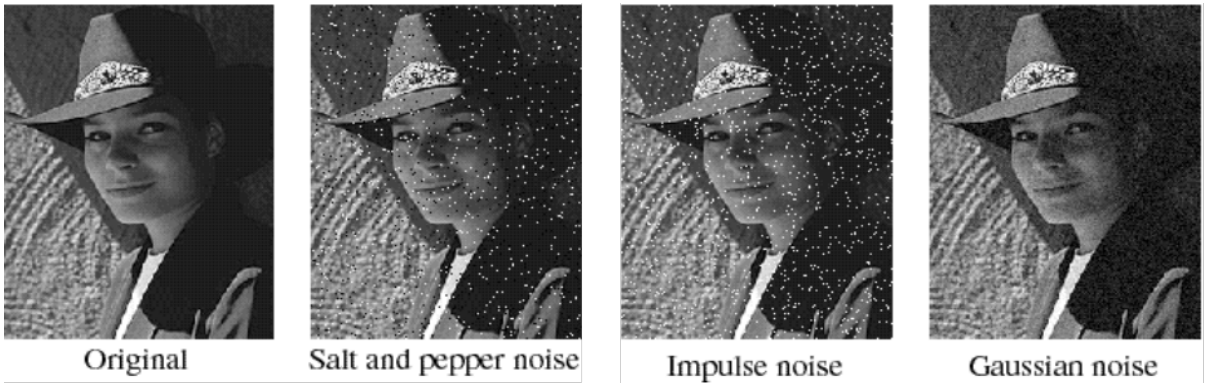
2.2 Smoothing

Images can exhibit different levels of *noise*: a random variation of brightness or color information. It is mainly produced by factors like the sensor response (more in CMOS technology), analog-to-digital conversion, *dead* sensor pixels, or bit errors in transmission, among others.

There are two typical types of noise:

- **Salt & pepper** noise (black and white pixels in random locations of the image) or **impulse** noise (only white pixels)
- **Gaussian** noise (intensities are affected by an additive zero-mean Gaussian error).



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, so different types of noise are appearing in its captured images.

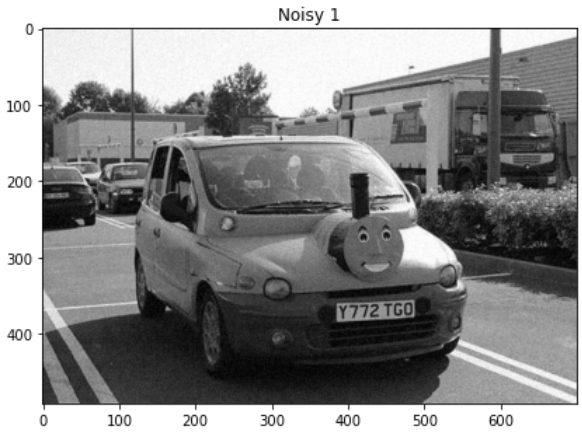
This way, UMA asked you again to provide some help with this problem!

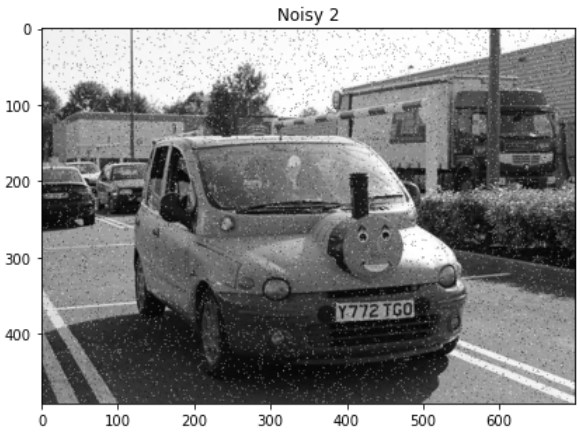
In [1]:
```
import numpy as np
from scipy import signal
import cv2
import matplotlib.pyplot as plt
import matplotlib
from ipywidgets import interactive, fixed, widgets
matplotlib.rcParams['figure.figsize'] = (15.0, 15.0)
import random

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 image (noisy_1.jpg) has Gaussian noise, the second one (noisy_2.jpg) has salt and pepper noise.

  La primera imagen (noisy_1.jpg) tiene ruido Gaussiano, la segunda (noisy_2.jpg) tiene ruido de sal y pimienta.

- Why can this happen?

  Speaking about the first image, this could happen because of the camera, or the conditions the photography was taken. The voltage and illumination of the sensor modify its heat, and in extreme conditions it produces Gaussian noise. In the other hand, salt and pepper noise can appear due to the analog-to-digital conversion, or dead pixesl in the camera sensor (dead pixels are pixels which dont't work properly so dont't give a correct color value).

  En la primera imagen, el ruido puede estar debido a la cámara, o a las condiciones en las que la fotografía fue tomada. El voltage y la iluminación del sensor modifican su calor, y en extremas condiciones se produce ruido Gaussiano. Por otro lado, el ruido de sal y pimienta puede por la conversión analógico-digital, o debidoa pixeles muertos en el sensor de la cámara (los pixeles muertos son aquellos que no funcionan correctamente y por tanto no dan un valor de color correcto).

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.

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
# 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, e
            verbose: Only show images if this is True

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

    # Create the kernel
    # Relation between w_kernel and height/width: 1 -> 3; 2 -> 5; 3 -> 7; n -

    height = 2*w_kernel+1 # or number of rows
    width = 2*w_kernel+1 # or number of columns
    kernel = np.ones((height, width), np.float32)/(height*width)
    #np.ones returns a matrix of the desired dimensions full of 1's

    # Convolve image and kernel
    smoothed_img = cv2.filter2D(image, cv2.CV_16S, kernel) #cv2.CV_16S becaus

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

        # 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))
```

```
[[ 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?

  We could say that noise is reduced in the first image (noisy_1.jpg), but it also gets blurred

  Podríamos decir que el ruido se reduce en la primera imagen (noisy_1.jpg), pero la imagen también se difumina.

- Is the noise removed from the second image?

  It isn't at all, because we see the black and white pixels (salt and pepper noise) in the convolved image.

  No se reduce en absoluto, porque en la imagen convolucionada seguimos viendo los pixeles negros y blancos propios del ruido de sal y pimienta.

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

  It may be w_kernel=1 (or w_kernel=2 maybe) because at this level noise is reduced and the image is not very blurred. If the value of w_kernel increases too much, the image will be too much blurred.

  Podría ser w_kernel=1 (o incluso w_kernel=2) porque en este nivel el ruido se reduce y la imagen no está muy borrosa. Si el valor de w_kernel se incrementa demasiado la imagen estára demasiado difuminada.

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=
```

In [6]:
```
noisy_img = cv2.imread(images_path + 'noisy_2.jpg', 0)
interactive(average_filter, image=fixed(noisy_img), w_kernel=(0,5,1), verbose=
```

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! 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 [7]:

```
# ASSIGNMENT 3

def gaussian(z, sigma):
    """ Returns the value of the Gaussian function for the given arguments

        Args:
            z: variable value in the 1-D Gaussian function
            sigma: sigma parameter of the 1-D Gaussian function

        Returns:
            value: value of the Gaussian function for x = z and sigma = sigma
    """
    return (np.exp(-(pow(z,2))/(2*pow(sigma,2))))/(sigma*np.sqrt(2*np.pi))

# Implement a function that:
# -- creates a 2D Gaussian filter (tip: it can be done by implementing a 1D Ga
# -- convolves the input image with the kernel
# -- displays the input image and the filtered one in a 1x2 plot (if verbose=T
# -- 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([gaussian(z, s) for z in range(-w,w+1)]) # Evaluate

    vertical_kernel = kernel_1D.reshape(2*w+1,1) # Reshape it as a matrix with
    horizontal_kernel = kernel_1D.reshape(1,2*w+1) # Reshape it as a matrix wi
    kernel = signal.convolve2d(vertical_kernel, horizontal_kernel) # Get the

    # Convolve image and kernel
    smoothed_img = cv2.filter2D(image, cv2.CV_16S, kernel)

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

        # 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 [8]:
```
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))
```

```
[[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?

  Yes, the noise of the first image (Gaussian noise) is removed using a Gaussian filter.

  Sí, el ruido de la primera imagen (ruido Gaussiano) se elimina utilizando un filtro Gaussiano.

- Is the noise removed from the second image?

  No, the salt and pepper noise can't be removed with a Gaussian filter.

  No, el ruido de sal y pimienta no puede ser eliminado con un filtro Gaussiano.

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

  In order to get a good reduction of Gaussian noise I would chose w_kernel=2, sigma=1,90. If the w_kernel value increases the image gets very blurred. If the value of sigma decreases the Gaussian noise is not reduced as much as we want.

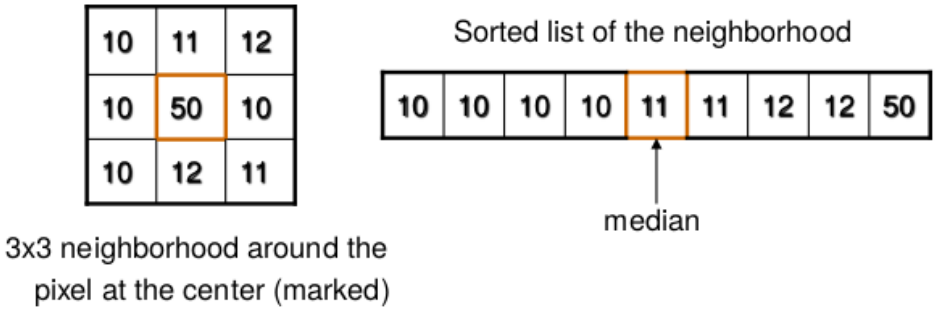
  Para obtener una buena reducción del ruido Gaussiano elegiría w_kernel=2, sigma=1,90. Si el valor de w_kernel se incrementa la imagen se vuelve muy difuminada. Si el valor de sigma se decrementa el ruido Gaussiano no se reduce tanto como queremos.

In [9]:
```
# 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=
```

In [10]:
```
noisy_img = cv2.imread(images_path + 'noisy_2.jpg', 0)
interactive(gaussian_filter, image=fixed(noisy_img), w_kernel=(0,5,1), sigma=
```

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 [11]:
```
# 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 =
# -- 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, 2*w_window+1)

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

        # 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 [12]:
```
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 [13]:
```
# Interact with the window size
noisy_img = cv2.imread(images_path + 'noisy_1.jpg', 0)
interactive(median_filter, image=fixed(noisy_img), w_window=(1,5,1), verbose=
```

In [14]:
```
noisy_img = cv2.imread(images_path + 'noisy_2.jpg', 0)
interactive(median_filter, image=fixed(noisy_img), w_window=(1,5,1), verbose=
```

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?

  No, it isn't because the median filter can't be used to remove Gaussian noise.

  No se reduce, porque el filtro de la mediana no puede utilizarse para eliminar ruido Gaussiano.

- Is the noise removed from the second image?

  Yes, it is. Median filter is a very good option to remove salt and pepper noise in an image. This can be observed in the interactive cells.

  Sí. El filtro de la mediana es una muy buena opción para eliminar el ruido de sal y pimienta de una imagen. Esto lo podemos observar en las celdas interactivas.

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

  I would choose w_window=1, because if the value is w_window is higher the smoothed image deforms, and we can't even read the car license plate.

  Yo elegiría w_window=1, porque si el valor de w_window es mayor la imagen suavizada se deforma, y no podemos ni siquiera leer la matrícula del coche.

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:

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

Average image — Noise Image

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

$$= f_{\text{noise\_free}}(x,y) + \frac{1}{M}\sum_{i=1}^{M} n_i(x,y)$$

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 [15]:

```
# 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
# -- 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
    """

    #Structure of burst: [ IMAGE1, IMAGE2, ... IMAGEn]
    #Structure of IMAGEn: [ROW1, ROW2, ... ROWn]
    #Structure of ROWn: [PIXEL1, PIXEL2, ... PIXELn]

    #Take only `burst_length` images
    burst = burst[0:burst_length]
    height = len(burst[0])
    width = len(burst[0][0])

    #Structure of pixel_union: [AVG_ROW1, AVG_ROW2, ... AVG_ROWn]
    #Structure of AVG_ROWn: [IMAGE1.ROW1.PIXEL1, IMAGE2.ROW1.PIXEL1, ... IMAGE
    pixel_union_img = np.empty((height, width, burst_length))

    for i in range(height):
        for j in range(width):
            for im_ind in range(burst_length):
                im = burst[im_ind]
                pixel_union_img[i][j][im_ind] = im[i][j]

    # Apply image averaging
    average_img = np.average(pixel_union_img, 2) #2 means averaging each pixel

    # 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') #could be burst[1] - burst[9]
        plt.title('Noisy')

        # 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 [16]:
```
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]]],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 [17]:
```
# Interact with the burst length
# Read image secuence
burst = []
for i in range(10):
    burst.append(cv2.imread('./images/burst1_' + str(i) + '.jpg', 0))

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

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

In [18]:
```
burst = []
for i in range(10):
    burst.append(cv2.imread('./images/burst2_' + str(i) + '.jpg', 0))

burst = np.asarray(burst)

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

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?

  Noise is especially reduced from the sequence 1 (the one with Gaussian noise); in this sequence, if we select a burst_length of 10 we can observe an averaged image with almost no noise. On the other hand, if we use this technique with the sequence 2 (the one with salt and

pepper noise) the changes are less significative and the noise doesn't reduce totally.

El ruido se reduce especialmente en la secuencia 1 (aquella con ruido Gaussiano); en esta secuencia, si seleccionamos un burst_length de 10 podemos observar una imagen promediada casi sin ruido. Por otro lado, si utilizamos esta técnica con la secuencia 2 (aquella con ruido de sal y pimienta) los cambios son menos significativos y el ruido no se reduce totalmente.

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

  In the sequence 1, 3 or 4 photos are enough to reduce Gaussian noise. In the sequence 2, we would need more than 10 photos, because using image average with burst_length=10 we can observe salt and pepper noise.

  En la secuencia 1, 3 o 4 fotos son suficientes para reducir el ruido Gaussiano. En la secuencia 2, necesitaríamos más de 10 fotos, porque al utilizar el promediado de imágenes con burst_length=10 aún observamos ruido de sal y pimienta.

2.2.4 Choosing a smoothing technique

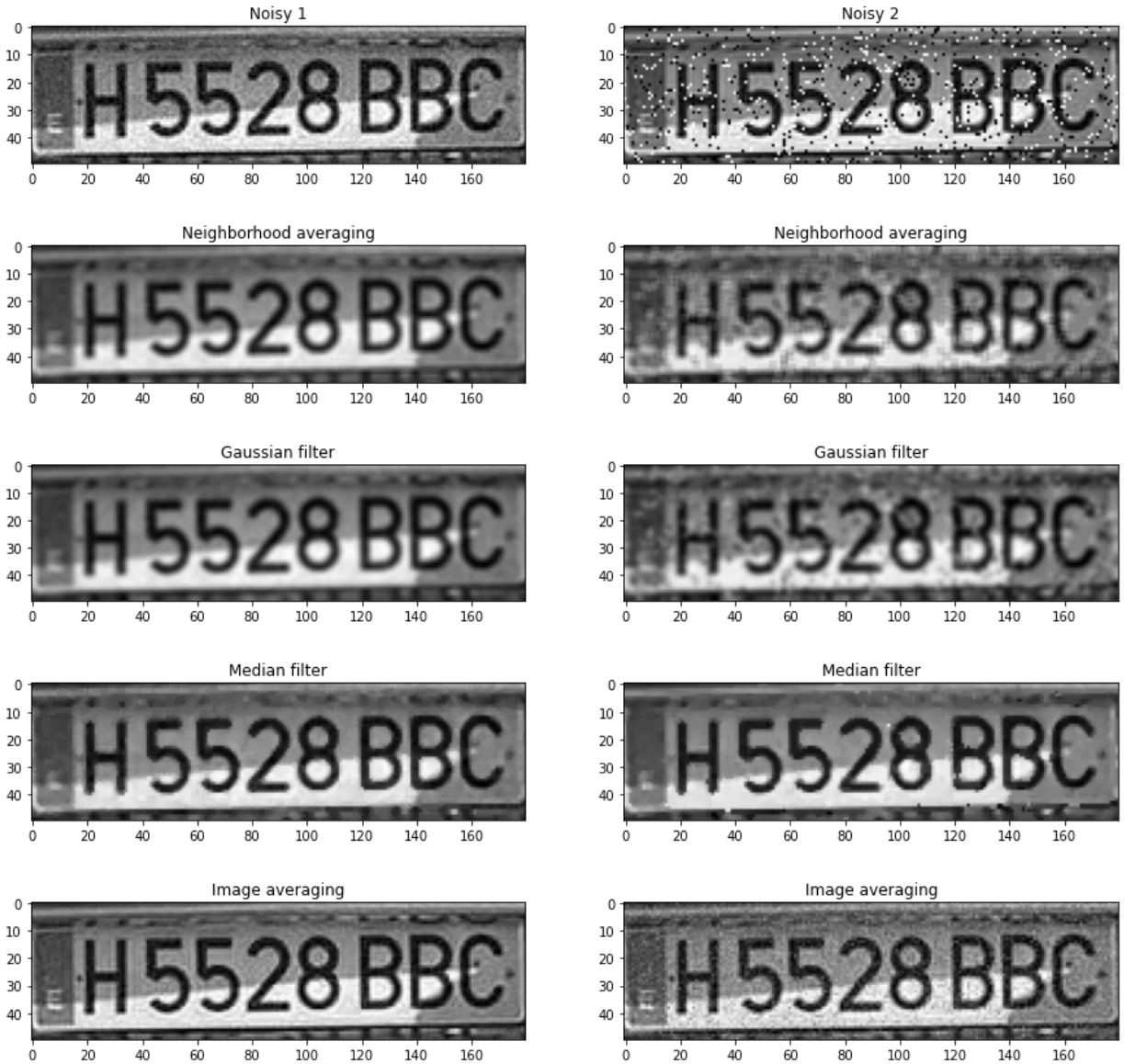
The next code cell runs the explored smoothing techniques and shows the results provided by each one while processing two different car license plates, **with two different types of noise. Check them!**

In [19]:

```
#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
neighbor1 = average_filter(im1, 1)
neighbor2 = average_filter(im2, 1)

# Apply Gaussian filter
gaussian1 = gaussian_filter(im1, 2,1)
gaussian2 = gaussian_filter(im2, 2,1)

# Apply median filter
median1 = median_filter(im1, 1)
median2 = median_filter(im2, 1)

# 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]

average1 = image_averaging(burst1, 10)
average2 = image_averaging(burst2, 10)

# 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('Neighborhood averaging')

plt.subplot(524)
plt.imshow(neighbor2, cmap='gray')
plt.title('Neighborhood averaging')

plt.subplot(525)
plt.imshow(gaussian1, cmap='gray')
plt.title('Gaussian filter')

plt.subplot(526)
```

```
plt.subplot(528)
plt.imshow(median2, cmap='gray')
plt.title('Median filter')

plt.subplot(529)
plt.imshow(average1, cmap='gray')
plt.title('Image averaging')

plt.subplot(5,2,10)
plt.imshow(average2, cmap='gray')
plt.title('Image averaging')
```

Out[19]: Text(0.5, 1.0, 'Image averaging')



Thinking about it (6)

And the final question is:

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

  *In order to reduce Gaussian noise I would choose the **image average** method, because it's the method that provides a higher accuracy image (more than the Gaussian filter); the problem is that it may be difficult to implement, because we would need several images of the same position of the plate. All in all, I would use the image average method, but if it can't be implemented I would use the **Gaussian filter**.*

  *In order to reduce salt and pepper noise I will chose the **median filter** method, with this method we get a clear image, without black and white pixels over it.*

  *If I had to choose a method to reduce noise (independently of what kind of noise was) I would choose the **median filter** method, because is the method that better results gets in the comparison above for both situations.*

  Para eliminar el ruido Gaussiano elegiría el método de **promediado de imágenes**, porque es el método que proporciona una imagen de más calidad (más que el filtro Gaussiano); el problema es que puede ser difícil de implementar, porque necesitaríamos varias imágenes de la misma posición de la matrícula. En resumen, elegiría el método de promediado de imágenes, pero si no es posible implementarlo utilizaría el **filtro Gaussiano**.

  Para reducir el ruido de sal y pimienta elegiría el método del **filtro de la mediana**, con este método obtenemos una imagen clara, sin pixeles blancos y negros.

  Si tuviera que elegir un método para reducir el ruido (independientemente de qué tipo de ruido fuese) elegiría el método del **filtro de la mediana**, porque es el método que mejores resultados obtiene en la comparación de arriba para ambas situaciones.

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*.