

---

# COMPUTER VISION LABORATORY

SESSION 2: Image filtering and Fourier Transform

---

June 17, 2020

Arlotta Andrea 4089306

Cantoni Francesca 4698289

# Contents

<b>1</b>	<b>Abstract</b>	<b>2</b>
<b>2</b>	<b>Introduction</b>	<b>4</b>
<b>3</b>	<b>Theoretical information</b>	<b>6</b>
3.1	Filtering . . . . .	6
3.1.1	Linear filters . . . . .	7
3.1.1.1	Moving average . . . . .	9
3.1.1.2	Gaussian filter . . . . .	11
3.1.2	Non-linear filters . . . . .	13
3.1.2.1	Median filters . . . . .	13
3.2	Fourier Transform . . . . .	14
<b>4</b>	<b>Matlab</b>	<b>16</b>
4.1	Function tree . . . . .	16
4.2	Functions . . . . .	17
4.2.1	Generals . . . . .	17
4.2.2	Filtering . . . . .	18
4.2.3	Fourier Transform . . . . .	18
<b>5</b>	<b>Conclusion</b>	<b>19</b>
5.1	Execution time . . . . .	19
5.2	Test . . . . .	20
5.3	Results . . . . .	21

# Chapter 1

## Abstract

This report aims to show, through a Matlab script, noise manipulations, some examples of filters usage and Fourier Transform representations.

The code takes as input three images, which will be transformed in a gray-scale, if necessary, and returns a set of images, graphs and histograms that illustrate the following operations:

- **image corruption:** images are affected by Gaussian noise, with standard deviation equal to 20, and Salt&Pepper, with density equal to 20%
- **image reconstruction:** the corrupted images are processed by using a moving average, a low-pass Gaussian filter and a median filter, each one with  $3 \times 3$  and  $7 \times 7$  spatial supports
- **linear filtering:** the input images are processed with three different filters which use a spatial support of  $7 \times 7$  pixels
- **Fourier Transform:** magnitude representation, inside frequency domain, of the provided images and a low-pass Gaussian filter characterized by kernel's dimension equal to  $101 \times 101$  pixels and  $\sigma = 5$

## Description of the paragraphs:

Firstly the section '**Introduction**' gives to the reader a brief explanation about what noise corruption is, which are the most common types of noise and then it introduces the two main classes of filters employed in this laboratory session.

Secondly the section '**Theoretical information**' goes deeper on filtering application, describes the matrix operations usually performed in digital image processing, the Discrete Fourier Transform formula and its connection with image manipulation.

Chapter 4, called '**Matlab**', can be considered as a guide for the Matlab code that we implemented. Inside this section it is shown a function tree, for a better understanding of the structure of the program, and it is also provided a description of the main functions.

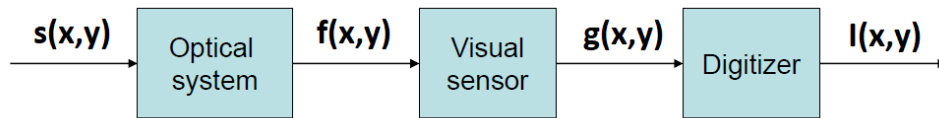
Inside '**Conclusion**' section are presented all the results, the screen-shots and the comments on the work done.

# Chapter 2

## Introduction

Digital images are representations of external world and the cleanliness of the information they communicate depends on the sensor and the circuitry of the camera. Therefore component specifications weight on the image quality.

As already mentioned in the previous report, an ideal monochrome image can be represent as the function  $I: [a, b] \times [c, d] \rightarrow [0, 1]$ . However, during the image formation process, the digital image can be corrupted by noises due to some changes with respect to the ideal conditions such as: light fluctuations, sensor noise, quantization effects, finite precision, etc..



**Figure 2.1:** Digital image formation system

Thus the digital image become the sum of a deterministic signal  $s(x,y)$  and the random variable noise  $ni$ , as follow:

$$I(x, y) = s(x, y) + ni \quad (2.1)$$

The common types of noise<sup>1</sup> are:

- **Gaussian noise:** affect the intensity of each pixel with a Gaussian normal distribution
- **Impulse noise:** replaces in random position pixels with white one
- **Salt and pepper noise:** as the previous one but the replacing pixels can be also black

In general filter processing is based on standard operators that map pixel values from the original image to another one. The main categories are:

- **Point operators:** each output pixel value only depends on the corresponding input ones (i.e. contrast adjustments and color transformations)
- **Area-based operators:** each output pixel value is determined by applying some algorithm to a small area that surrounds the corresponding input pixel. This is the main idea of filtering process.
- **Global operators:** the result of the operator, at a particular point inside the image, depends on the whole image (i.e. histogram and Fourier Transform)

In particular in this second laboratory session we apply different types of Area-based image filters<sup>2</sup> which can be subdivided as follows:

- **Linear filters:**
  - Moving average
  - Low-pass Gaussian filter
  - Sharping filter
- **Non-linear filter:**
  - Median filter

---

<sup>1</sup>The effects product by these noises on a generic image will be shown in subsection **5.3 Results** Figure 5.4.

<sup>2</sup>A better explanation of these types of filters will be provided in section **3.2 Filtering**.

# Chapter 3

## Theoretical information

### 3.1 Filtering

As mentioned in the previous chapter, filters are examples of neighborhoods operator which allow to obtain the output image as a weighted sum of a small area of the input pixel values.

With this particular image processing is possible to:

- accentuate edge
- add soft blur
- sharp details
- remove noise

In particular we are interested in *remove noise* action in order to reduce the effects produced by the intrinsic noise on the corrupted input image<sup>1</sup>.

We now proceed to describe the main classes of filters, their properties and effects on the provided images.

---

<sup>1</sup>During this laboratory session we voluntarily added to input images different types of noises to simulate very distorted condition.

### 3.1.1 Linear filters

These types of filters are Linear Shift Invariant (LSI) systems so, calling  $f_1$  and  $f_2$  two functions,  $g$  the resultant one,  $k$  a scalar value and  $h$  the mask of the spatial linear filter, they have the following main properties:

- **Linear:** output pixel values are obtain by a linear function of the input ones.

For this reason they obey to:

– *Superposition:*

$$h * (f_1 + f_2) = (h * f_1) + (h * f_2) \quad (3.1)$$

– *Scaling:*

$$h * (k * f_1) = k * (h * f_1) \quad (3.2)$$

- **Shift invariant:** output image is a shift-invariant function of the input one, so the output pixel values depends only on the chosen kernel dimension and filter coefficients

$$g(u, v) = f_1(u + k, v + l) \iff (h * g)(u, v) = (h * f_1)(u + k, v + l) \quad (3.3)$$

The previous characteristics can be represented in a compact way with the **cross-correlation**:

$$O(u, v) = I \otimes H = \sum_k \sum_l I(k, l) * H(u + k, v + l) \quad (3.4)$$

where  $I$  is the input image,  $O$  the output one and  $H$  the spatial filter.

Another way to perform this linear filtering is through the **convolution** operation:

$$O(u, v) = I \circledast H = \sum_k \sum_l I(k, l) * H(u - k, v - l) \quad (3.5)$$

in which the only difference from the previous one is that the offset in  $H$  has been reversed.



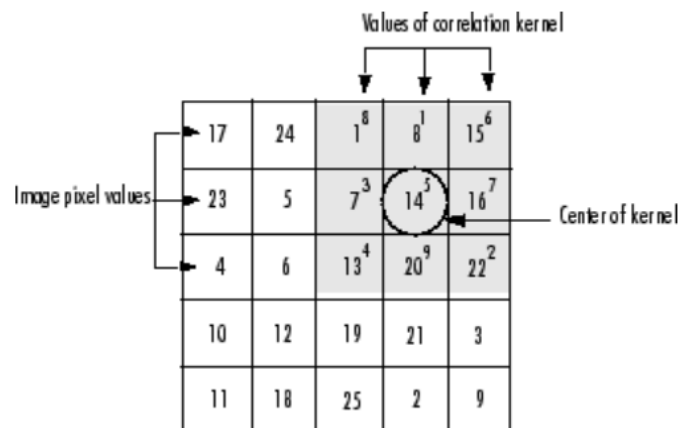
For sake of clarity let's consider two generic matrices one for the input  $I$  and the other one for the two-dimensional spatial filter  $H$ :

$$I = \begin{bmatrix} 17 & 24 & 1 & 8 & 15 \\ 23 & 5 & 7 & 14 & 16 \\ 4 & 26 & 13 & 20 & 22 \\ 10 & 22 & 19 & 21 & 3 \\ 1 & 28 & 25 & 2 & 9 \end{bmatrix} \quad H = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$$

### Cross-correlation:

Steps for performing the cross-correlation between  $I$  and  $H$  on the generic position (2,4):

1. slide the center element of the correlation kernel so that lies on top of the (2,4) element of  $I$
2. multiply each weight in the correlation kernel by the pixel of  $I$  underneath
3. sum the individual products obtain by the previous step



**Figure 3.1:** cross-correlation for obtain the value of the output pixel (2,4)

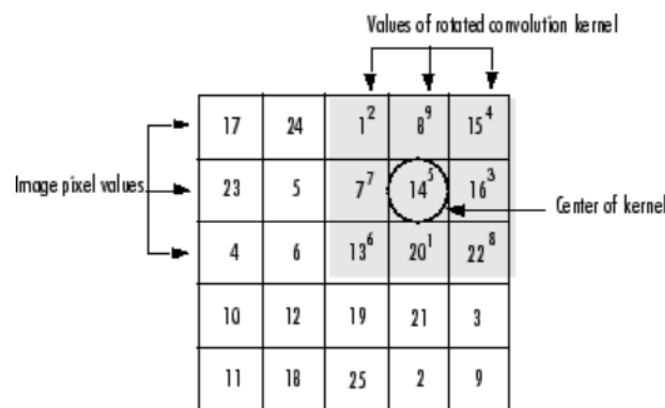
### Convolution:

Steps for performing the convolution between  $I$  and  $H$  considering the same generic output position (2,4):

1. rotate the correlation kernel 180 degrees about its center element to create a convolution kernel

2. slide the center element of the correlation kernel so that lies on top of the (2,4) element of I
3. multiply each weight in the rotated convolution kernel by the pixel of I underneath
4. sum the individual products obtain by the previous step

So when you apply a convolution you perform a cross-correlation between the input image matrix and the filter matrix flipped in both dimensions.



**Figure 3.2:** convolution for obtain the value of the output pixel (2,4)

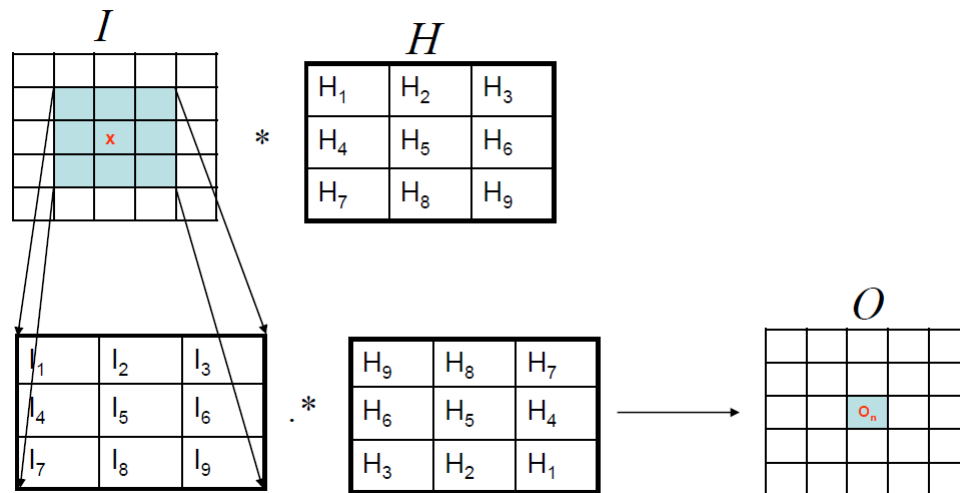
### 3.1.1.1 Moving average

The 2D linear filter moving average, or box filter, is one of the linear filters and it performs a simply sum among all the values obtained by the product element by element of the small neighborhoods of chosen pixel (red cross) inside the input matrix  $I$  and the flipped 2D kernel  $H$ . The result of this operation is the output pixel value associate to the position equal to the center of the convolution kernel.

Consider a generic 3x3 convolution matrix, the operation performed to obtain a single output pixel's value  $O_n$  is the following:

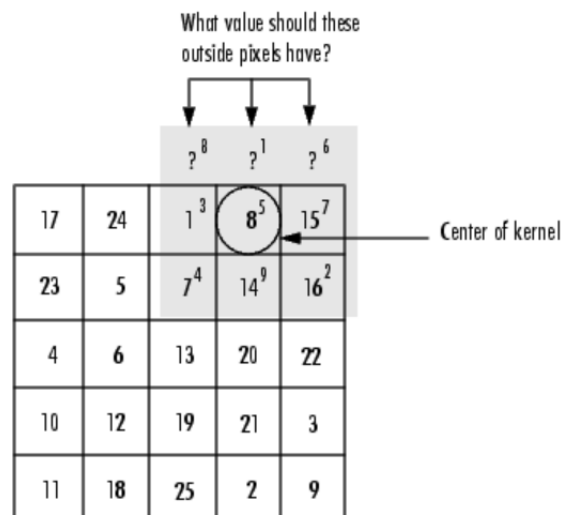
$$O_n = I_1 H_9 + I_2 H_8 + I_3 H_7 + I_4 H_6 + I_5 H_5 + I_6 H_4 + I_7 H_3 + I_8 H_2 + I_9 H_1 \quad (3.6)$$

It's easy now to understand that, in order to get all the desired values, is necessary to run the kernel over the whole input image.



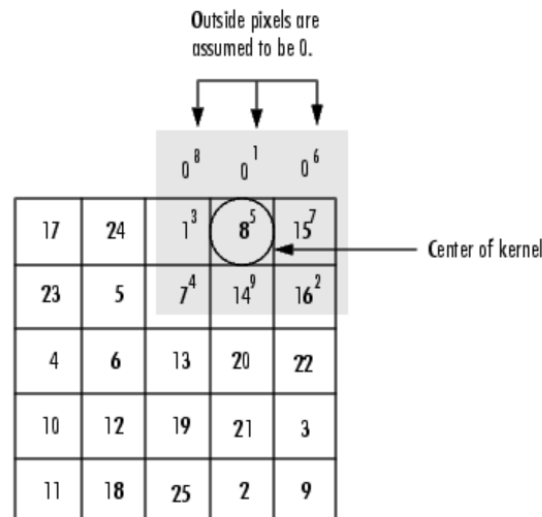
**Figure 3.3:** moving average performed on the neighborhoods of one pixel (red cross) with a 3x3 convolution filter

However this technique suffers from border effects issue due to the fact that for the boundary of the input image, is impossible to perform the convolution because a portion of the kernel is off the edge of the matrix  $I$ , as illustrated in the following figure:



**Figure 3.4:** issue due to the fact that the values of the Kernel fall outside the input image

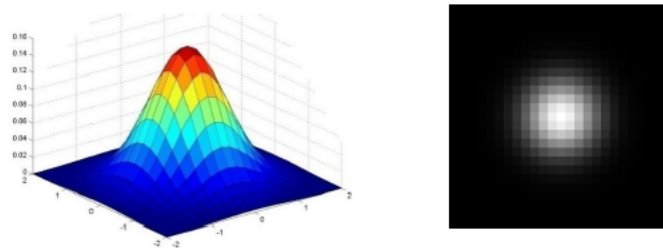
There are more than one types of padding for solve this problem. One of the simplest method is the **zero-padding** in which all the off of edge image pixels are fill assuming it as zero.



**Figure 3.5:** zero-padding method

### 3.1.1.2 Gaussian filter

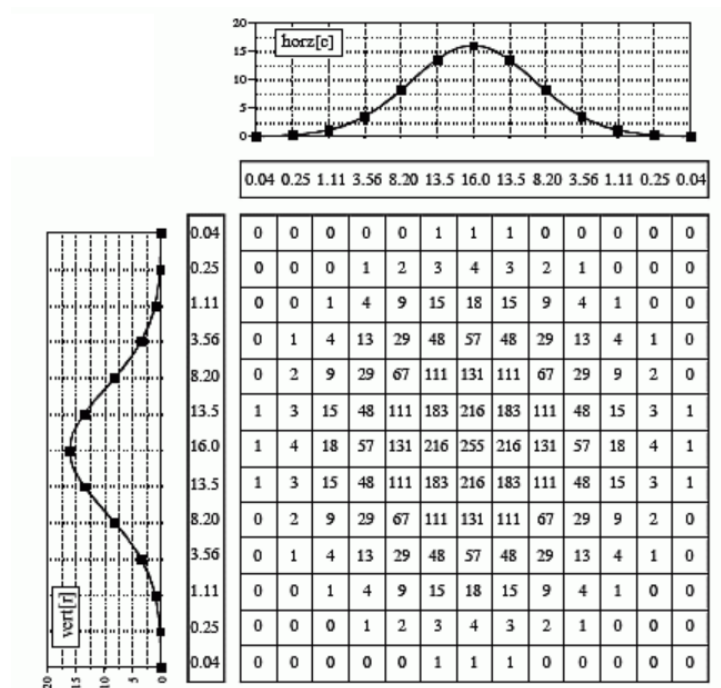
The Gaussian filter is another type of linear filter in which, during the weighted averaging, central pixels have more weight with respect to the neighbors so the coefficients are a 2D Gaussian.



**Figure 3.6:** Gaussian smoothing filter

The equation that describes the 2D unit area Gaussian filter is the following:

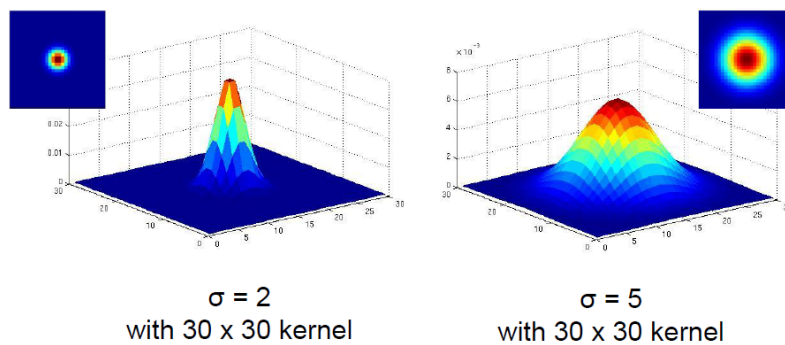
$$G(x, y)_\delta = \frac{1}{2\pi\delta^2} e^{-\frac{x^2+y^2}{2\delta^2}} \quad (3.7)$$



**Figure 3.7:** 2D Gaussian filter with 13x13 kernel

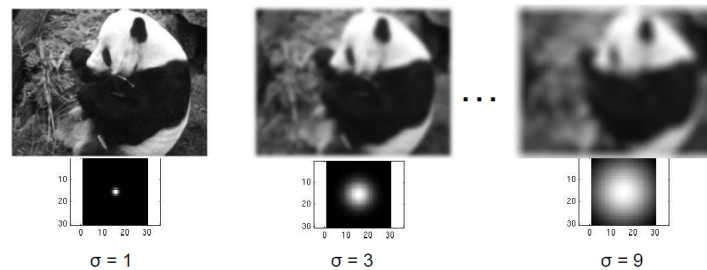
The two main parameters for this filter are:

- **Standard deviation  $\sigma$ :** square root of the variance  $\sigma^2$



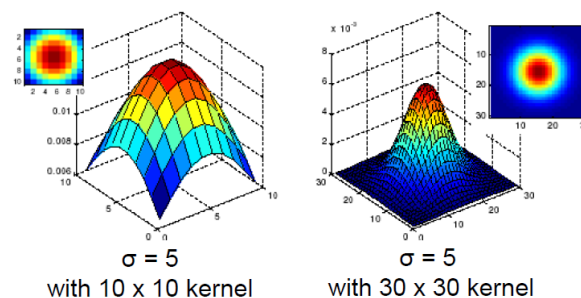
**Figure 3.8:** Gaussian filters with different sigma values

This parameter influences the intensity of the smoothing effect on the output image, as follow:



**Figure 3.9:** Different smoothing effects due to different values of sigma

- **Size of kernel:**<sup>2</sup> 2D filter's support dimension



**Figure 3.10:** Gaussian filters with different kernel dimensions

*REMARK: wrong choice of the size of the mask can produce a paraboloid function (left image) instead of a Gaussian one (right image).*

### 3.1.2 Non-linear filters

With these types of filters the output pixels' values are no more the weighted summation of some number of input pixels because the output can't be described as a linear function of the input.

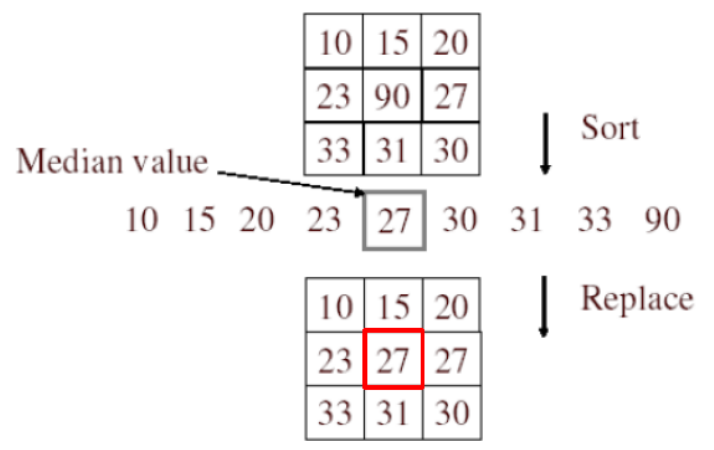
#### 3.1.2.1 Median filters

Median filter are one of the easiest non-linear digital filter and the main steps to process an image with it are:

<sup>2</sup>Gaussian function has infinite support, but discrete filters use finite kernels.

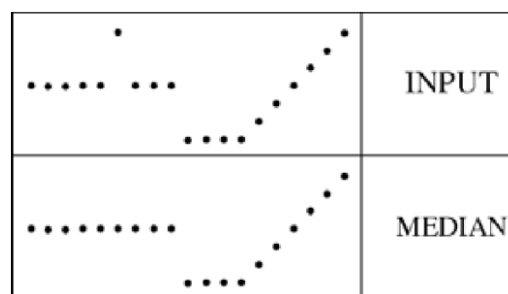
- choose the window size
- sort all the pixel values inside the window into numerical order
- replace the pixel considered with the middle pixel value

For sake of clarity:



**Figure 3.11:** Median filter with 3x3 window

Its performance is particularly effective in removing spikes and salt-and-pepper noise (impulsive noise) since these noises usually lies well outside the true value in the neighborhood. However is not that much better than Gaussian blur for high levels of noise.



**Figure 3.12:** Edge preserving and discards outliers of the median filter

## 3.2 Fourier Transform

The Fourier Transform is an important image processing tool which is used to decompose an image into its sine and cosine components.

Since we are only concerned with digital images we will consider the **Discrete Fourier Transform (DFT)**: this contains as much frequencies as the number of pixels in the spatial domain image.

For a generic image of size NxM, the 2D DFT is given by:

$$F(k, l) = \sum_{u=0}^{N-1} \sum_{v=0}^{M-1} f(u, v) e^{-j2\pi \frac{ku}{N} \frac{lv}{M}} \quad (3.8)$$

The DFT output produces a complex number valued output named **Fourier spectrum** and in polar form it can be described as follows:

$$F(u, v) = |F(u, v)| e^{j\phi(u, v)} \quad (3.9)$$

where its amplitude and phase are:

$$|F(u, v)| = \sqrt{R^2(u, v) + I^2(u, v)} \quad \phi(u, v) = \tan^{-1} \frac{I(u, v)}{R(u, v)} \quad (3.10)$$

Processing in Fourier transform domain is useful to isolate specific frequency portion of the image.

Moreover the convolution of two signals is the pointwise product of their Fourier transforms, so filtering in Fourier domain means to execute a product between the image and the filter inside frequency domain.

$$O(t) = I(t) \otimes h(t) \xrightarrow{\mathcal{F}} O(f) = I(f) H(f) \quad (3.11)$$



# Chapter 4

## Matlab

### 4.1 Function tree

The code consists of a *main.m* file that allow the user to choose the desired input image<sup>1</sup>: d

1. *callFunctions.m*

transforms input image into monochrome one, if necessary, and calls all the sub-functions in order to perform all the required actions

1.1. *addnoise.m*

main function that calls the 2 sub-functions related to noise actions and return the corrupted image

i. *addsaltandpepper.m*

adds salt&pepper noise to the input image

ii. *addgaussian.m*

adds Gaussian noise to the input image

1.2. *movav.m*

performs the moving average filter to the corrupted image and returns the filtered image

i. *matrix.m*

increases the size of the 2D moving average filter in order to have a better display of the mask

---

<sup>1</sup>It's possible to process more than once image contemporary.

1.3. *gaussfilt.m*

performs the Gaussian filter to the corrupted image and returns the filtered image

1.4. *medfilter.m*

performs the median filter to the corrupted image and returns the filtered image

1.5. *linfilter.m*

performs on the input image these linear elaborations: no change, one pixel shifted left, sharpening filter

i. *process.m*

perform the cross correlation between the input image and the given mask

1.6. *fourierTransform.m*

performs the DFT on the input monochrome image and displays magnitude of its spectrum

## 4.2 Functions

The main functions used in this second laboratory session have been divided into 3 subcategories: **Generals**, for general purposes, **Filtering**, related to filtering actions, and **Fourier Transform**, related to Discrete Fourier Transform actions.

### 4.2.1 Generals

- **imagesc(C):** displays the data in matrix C as an image that uses the full range of colors in the colormap
- **surf(C):** creates a three-dimensional surface plot of the matrix C
- **imhist(out, 256):** calculates the histogram for the grayscale image out using 256 bins

### 4.2.2 Filtering

- **conv2(INimage, 2D-kernel, 'same')** returns the two-dimensional convolution of matrices INimage and 2D-kernel with dimension equal to the first one
- **conv2(conv2(INimage, vector),vector')** performs two consecutive 1D convolutions, first horizontal and then vertical, in order to reduce the execution time
- **fspecial('gaussian', n, sigma)** returns a rotationally symmetric Gaussian low-pass filter of size n with standard deviation sigma
- **medfilt2(INimage, [m n])** performs a 2D median filter using g mxn spatial filter

### 4.2.3 Fourier Transform

- **fft2(double(INimage))** returns the two-dimensional Fourier Transform of a matrix using a Fast Fourier Transform algorithm. Using this function allow to reduce the complexity from  $N^2$  to  $N\log_2 N$ . This is a significant improvement, in particular for large images
- **fftshift(F)** rearranges a Fourier Transform F by shifting the zero-frequency component to the center of the array
- **abs(F)** returns the complex magnitude of each element in F
- **log(F)** returns the natural logarithm  $\ln(F)$  of each element inside F

# Chapter 5







## Conclusion

### 5.1 Execution time

In this section we analyze the performance time<sup>1</sup> of the program and in particular the time spent executing the whole script is approximately 3.848s<sup>2</sup>.

Analyzing the data obtained with the tool we can observe that *callFunctions('imagename')*, which call all the sub-functions that perform the desired image processing, is the one that uses most of the execution time approximately 2.986s.

The following table shows the other children functions that occupy most of the execution time:

Children (called functions)					
Function Name	Function Type	Calls	Total Time	% Time	Time Plot
<a href="#">movav</a>	function	2	0.548 s	18.4%	
<a href="#">addnoise</a>	function	1	0.384 s	12.9%	
<a href="#">gaussfilt</a>	function	2	0.375 s	12.6%	
<a href="#">linfilter</a>	function	1	0.360 s	12.1%	
<a href="#">medfilter</a>	function	2	0.306 s	10.3%	
<a href="#">fourierTransform</a>	function	1	0.221 s	7.4%	

**Figure 5.1:** performance time

<sup>1</sup>All these values have been evaluated using the Matlab tool *performance time*.

<sup>2</sup>The execution time has been estimated processing only one image, specifically the *panda.png* figure.

## 5.2 Test

In order to test the code we upload the RGB photo *panda.png*<sup>3</sup>, through a relative path, with these characteristics: width of 254 pixels, height of 190 pixels and dimension of 42,4 kByte. However it is possible to test the script with other two images by uncommenting the function *callFunction('imagename')* in the *main.m*.

The possible input images are shown below:



**Figure 5.2:** input image on which we will refer in the next subsection



**Figure 5.3:** other 2 possible input images

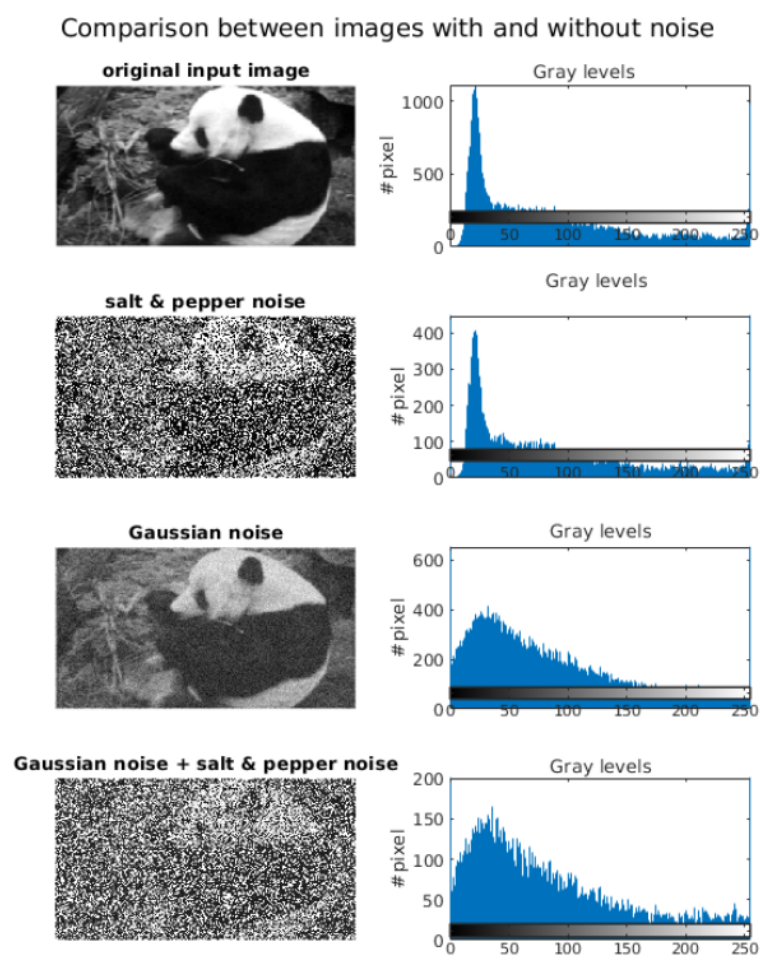
---

<sup>3</sup>We choose this image as test because in our opinion it displays the most representative results of the performed image processing.

## 5.3 Results

Retracing the structure of the code, we now present and comment the outcomes of our test:

1. `callFunctions()`
  - 1.1. `addnoise()`



**Figure 5.4:** comparison between different effects produced by the two types of noise

From the above figure it is possible to see how noise addition on the chosen grayscale image changes its histogram.

It is also important to remark that:

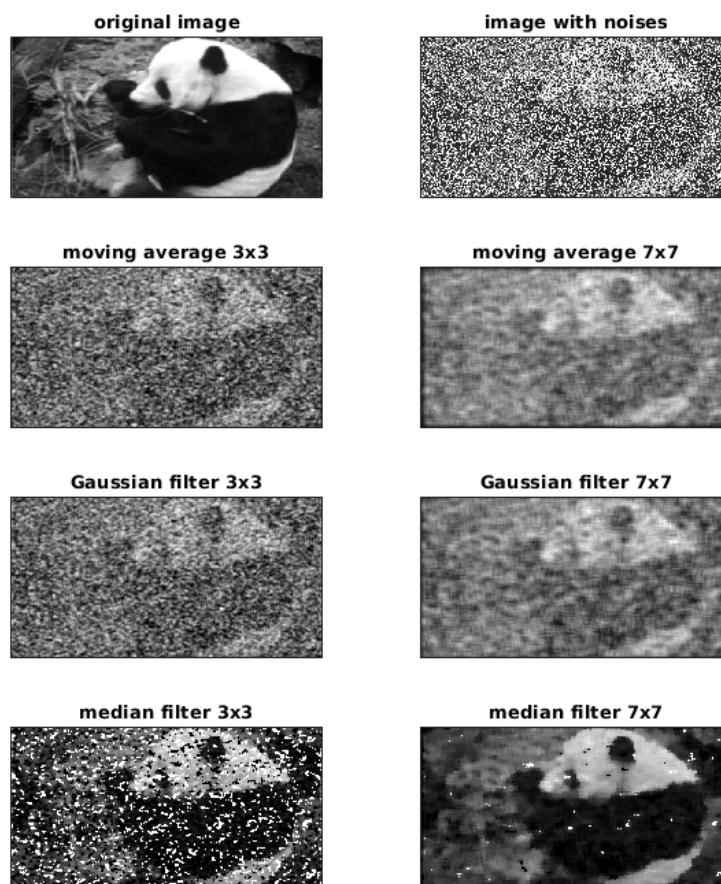
- *salt & pepper*: produces a random occurrences of black and white pixels with density equal to 20%
- *Gaussian noise*: changes the intensity of the output pixel's values based on the Gaussian normal distribution with  $\sigma$  equal to 20

From the fourth image we can observe that salt & pepper produces the preponderant visible deformation and that the histogram has more spread values than the initial one (Gaussian shape), so the intensity values are smoothed.

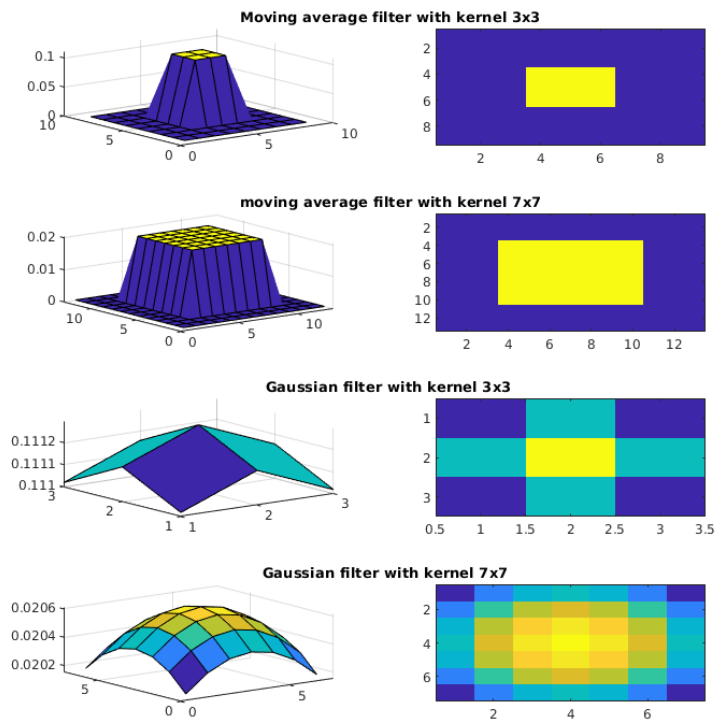
### 1.2. `movav()`, `gaussfilt()`, `medfilter()`

We have performed three different types of image filtering each one with 3x3 and 7x7 kernel dimension.

Comparison between image with noises and filtered images



**Figure 5.5:** different results obtained by three types of image filtering



**Figure 5.6:** different sizes and types of linear spatial filters

Figure 5.5 shows that the applied filters are smoothing type, for that reason they are powerful in noise reduction.

In particular it is possible to observe that median filter with  $7 \times 7$  kernel dimension is the most effective method to reduce these types of impulse noises<sup>4</sup>.

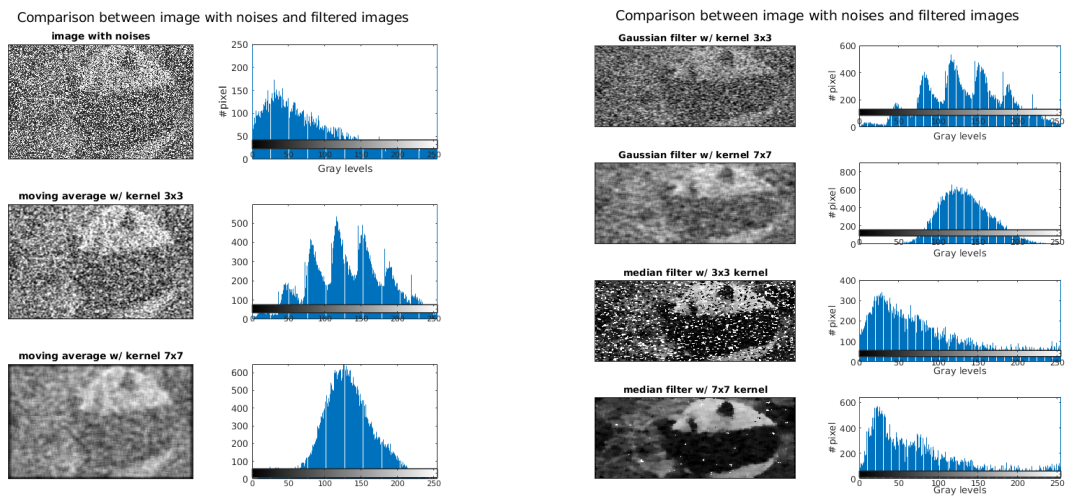
Figure 5.6 displays the kernel of the employed filters<sup>5</sup> from which is possible to notice that:

- *moving average filters* are surrounded by a frame of zeros in order to obtain a better prospective
- *Gaussian filters* seem to have a paraboloid surface due to the fact that the value of  $\sigma$  is a bit too high with respect to the kernel space. The Gaussian function could be restored by increasing the kernel dimension or by lowering the value of  $\sigma$

<sup>4</sup>The main problem of this filter is that it produces a visible patchy effect that can prevent you from preserving important details of the original image.

<sup>5</sup>Median filter cannot be displayed because is non-linear filter.

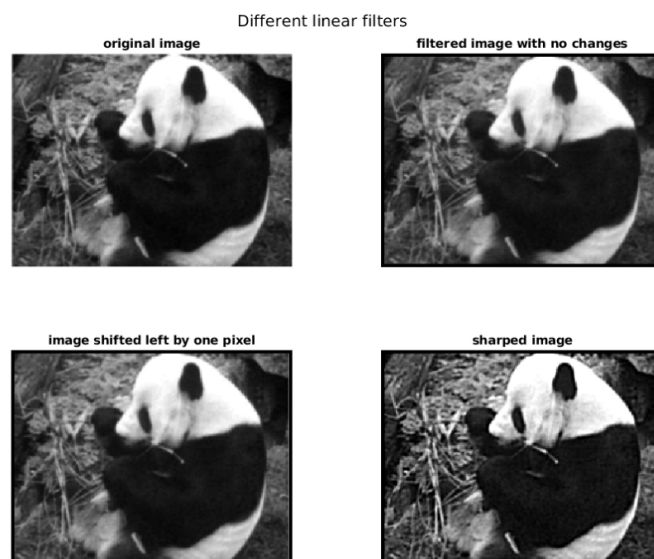




**Figure 5.7:** comparison of the results obtained by same filter with different kernel dimensions

Figure 5.7 emphasizes the relevance of the kernel dimension choice, in fact, using a  $3 \times 3$  mask, the filtering process is more localized, as it can be seen from the histograms. On the other hand  $7 \times 7$  mask treats the image in a more widely way.

### 1.3. `linfilter()`



**Figure 5.8:** linear filters image processing

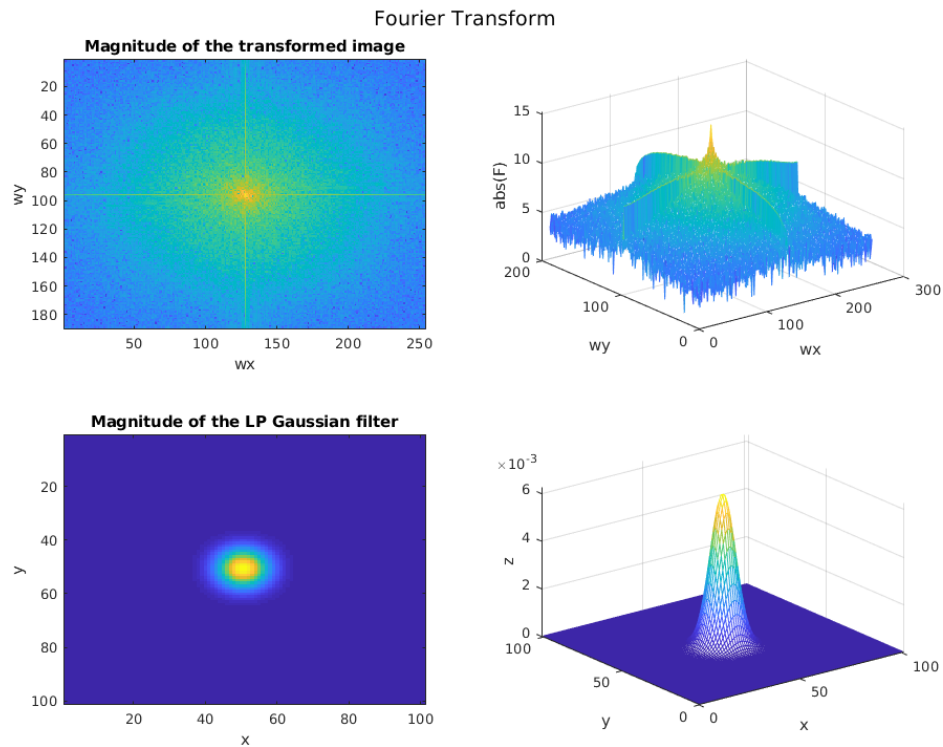
Figure 5.8 illustrates the results of three linear filtering processes with a spatial support of  $7 \times 7$  pixels.

When we perform these type of manipulations we execute a cross-correlation between the input image  $I$  and the relative mask  $A_n$ :

- **no changes:**  $O = I .* A_1$
- **shifted left by one pixel:**  $O = I .* A_2$
- **sharped image:**  $O = I .* A_3$

$$A_1 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad A_2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$A_3 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} - \frac{1}{49} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

1.4. `fourierTransform()`

**Figure 5.9:** Magnitude of the input image and a Low-Pass Gaussian filter

Magnitude provides most of the information of the geometric structure of the spatial domain image. However, if we want to re-transform the Fourier image into the correct spatial domain, it is important also to preserve its phase.