CS6640: Image Processing
Project 2
Filtering, Edge detection and Template matching

Arthur Coste: *coste.arthur@gmail.com*

September 2012

Contents

1 Introduction

The objective of this second project is to get deeper in the image processing functions. Indeed, after having work on histograms which is a global processing of the image, we are going to work on a local processing which is filtering. The purpose of filtering can be multiples, reduce noise on a image for example or detect some important features of an image such as edges. We will in this work present different applications of filtering, the results we obtained and the code we implemented.

The implementation is also realized using MATLAB.

Note: Some images were provided for this project, for the MRI image I used, I don't have any right on it, it comes from a MRI database. The cameraman image belongs to the MIT, and I have to mention that the image has a Copyright from the Massachusetts Institute of Technology but it's a test image from MATLAB. I made all the other drawing or images so there is no Copyright infringement in this work.

The pictures are contained in the archive with the report so if you want to see them bigger they are in the picture directory.

Note: The following MATLAB functions are associated to this work:

- $convolution.m : OutputIm = convolution(InputImage, weight, display)$
- $convol\_separable.m : OutputIm = convol\_separable(InputImage, weightx, weighty, display)$
- $separable\_gaussian.m : OutputIm = separable\_gaussian(InputImage, sigma, display)$
- $Gaussian\_weights.m : OutputIm = gaussian\_weigths(n, display)$
- $edge\_detection.m : OutputIm = edge\_detection(InputImage, weightx, weighty, display)$
- $template\_matching.m : OutputIm = template\_matching(InputImage, template, threshold, display)$

2 Theoretical definitions

2.1 Image

A deep presentation of digital image has already been done in the introduction of the previous project so we will rely on what we have already presented.

We will just remind that a digital image can be considered as a numerical two dimensions array which is the reason why we can process them in the discrete space.

2.2 Mathematical approach

Filtering is an important step in image processing because it allows to reduce the noise that generally corrupt a lot of images. But filtering can also be used to perform other operations such as feature detections to extract the edges of objects in a image or to perform a template matching. These operations rely on a mathematical operation called convolution.

2.2.1 Convolution

Convolution is a mathematical operations on two functions: f and g to produce a third function which correspond of the area under the curve of the product of the two function f and g the continuous definition of convolution is given by:

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(\tau)g(t-\tau) = \int_{-\infty}^{+\infty} f(t-\tau)g(\tau) \tag{1}$$

Which can of course be converted into a discrete version to be applied to the discrete structure of an image.

$$(f * g)[n] = \sum_{m=-\infty}^{+\infty} f[m]g[n-m] = \sum_{m=-\infty}^{+\infty} f[n-m]g[m] \tag{2}$$

In our case, one of the previous function is going to be the input image array and the other function is going to be the filter kernel. A normalization step is clearly necessary to keep the intensity distribution invariant. Indeed, the kernel filter used should not modify the intensity distribution of the image. The implementation of this operation is presented in the implementation section.

2.2.2 Correlation

The cross-correlation is a measure of similarity of two signals or shapes. It's definition is really similar to convolution and is given by;

$$(f \circ g)(t) = \int_{-\infty}^{+\infty} f^*(\tau)g(t+\tau) \tag{3}$$

Which can of course be converted into a discrete version to be applied to the discrete structure of an image.

$$(f \circ g)[n] = \sum_{m=-\infty}^{+\infty} f^*[m]g[n+m] \tag{4}$$

2.2.3 Types of filter

In this project, we are going to work on different types of filters. The filters presented in this part are called averaging filters or low pass filter.

Averaging Kernel In this project, we are going to work on different types of filters. The first one is the smoothing kernel filter using a averaging kernel where all the elements have the same weight. The general definition of a $N \times N$ kernel is :

$$\mathbf{W_N} = \frac{1}{N^2} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 1 & 1 & \cdots & 1 \\ 1 & 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & 1 & \cdots & 1 \end{bmatrix} \tag{5}$$

Here are some examples of filter kernels we are going to use :

$$\mathbf{W_3} = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \qquad \text{and} \qquad \mathbf{W_5} = \frac{1}{25} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{6}$$

An important thing to notice here is that the dimension of the filter has to be odd. Because we want to modify the central pixel so it has to be surrounded by the same number of pixel which can't be achieved with a kernel of even dimension.

Gaussian Kernel As we presented in the previous project, the Gaussian distribution is widely used to model noise. So it seems pretty straightforward to use this distribution as a template for smoothing an image. The Gaussian distribution is a really interesting distribution and can be approximated easily using convolution. Let's remind the definition of this density.

$$\begin{cases} f : \mathbb{R} \to \mathbb{R} \\ x \to f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-1}{2}\left(\frac{(x-\mu)}{\sigma}\right)^2} \end{cases} \tag{7}$$

This definition can be generalized to multiples dimensions with the multivariate Gaussian model:

$$\begin{cases} f : \mathbb{R}^n \to \mathbb{R}^n \\ [X] \to f([X]) = \frac{1}{\sigma 2\pi^{\frac{N}{2}}\sqrt{|C|}} e^{\frac{-1}{2}\left((X-\mu)^T C^{-1}(X-\mu)\right)} \end{cases} \tag{8}$$

With C being the Covariance Matrix defined by:

$$\mathbf{C} = \begin{bmatrix} E\{X_1^2\} & E\{X_1X_2\} & E\{X_1X_3\} & \cdots & E\{X_1X_n\} \\ E\{X_1X_2\} & E\{X_2^2\} & E\{X_2X_3\} & \cdots & E\{X_2X_n\} \\ E\{X_1X_3\} & E\{X_2X_3\} & E\{X_3^2\} & \cdots & E\{X_3X_n\} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ E\{X_1X_n\} & E\{X_1X_n\} & E\{X_1X_n\} & \cdots & E\{X_n^2\} \end{bmatrix} \tag{9}$$

The Gaussian Kernel is composed of weights determined using the previous definition. But there exist an other way to generate those weights by using convolution. Indeed, if we use the standard vector $[1, 1]$ and apply the convolution on itself the first time and then on the result we get the coefficients of Newton's Binomial Theorem. Which according to the central limit theorem gives a fair approximation of a Gaussian distribution as the number of iterations increase. The following picture gives an illustration of it. The code used to generate this result will be presented in the implementation section. The result has been normalized and we can clearly see the convergence to a Gaussian distribution even after few iterations.

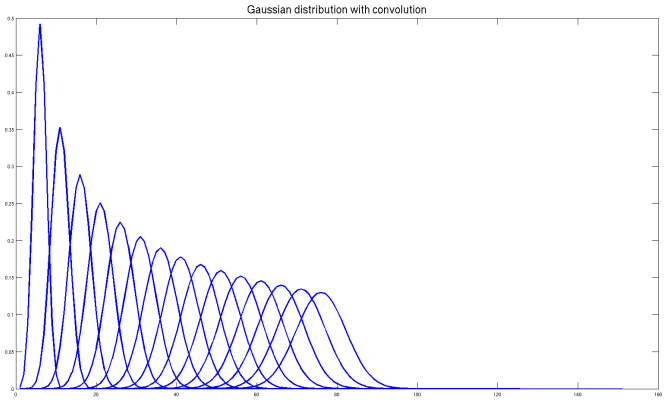

Figure 1: Convergence of convolution to a Gaussian distribution of iterations 10:10:150

Indeed, the Gaussian kernel is a kind of smoothing filters where the weights are different according to the position of the pixel in regard to the central pixel. The weights are determined using the standard deviation of the Gaussian law. Here is an example of a simple Gaussian smoothing kernel.

$$\mathbf{G_3} = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{10}$$

Figure 2: Summary of the two types of filters: (a) averaging box, (b) Gaussian

2.3 Filtering

2.3.1 Application of convolution

As presented in the previous part, the convolution is a local operation in which a filtering kernel is moving on the image to modify a pixel value according to the neighbours intensity. Here is a graphical explanation of the algorithm.

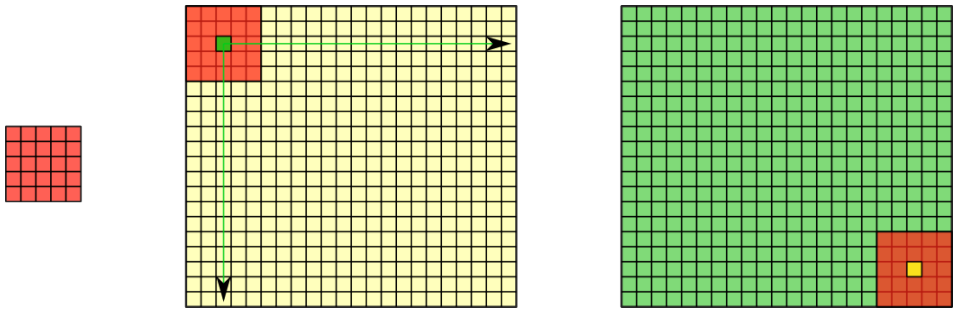

Figure 3: (a) smoothing kernel, (b) evolution of the kernel on the image, (c) Result of smoothing

2.3.2 Separability

Separability is a mathematical property of multidimensional convolution based on commutativity. Let's explain it, with the definition of the bi-dimensional convolution.

$$h[x,y] = f[x,y] * g[x,y] = \sum_{j=-\infty}^{+\infty} \sum_{i=-\infty}^{+\infty} f[i,j]g[x-i,y-j] \tag{11}$$

Let's assume that $f[x,y]$ is our kernel and has the separability property:

$$f[x,y] = f_1[x] \times f_2[y] \tag{12}$$

If we apply the previous result to the convolution we obtain:

$$h[x,y] = f[x,y] * g[x,y] = \sum_{j=-\infty}^{+\infty} \sum_{i=-\infty}^{+\infty} f_1[x] \times f_2[y] \times g[x-i,y-j] \tag{13}$$

$$h[x,y] = \sum_{j=-\infty}^{+\infty} f_2[y] \left[\sum_{i=-\infty}^{+\infty} f_1[x] \times g[x-i,y-j] \right] \tag{14}$$

If we plug in the previous equation the definition of the convolution given by:

$$(f * g)[x] = \sum_{m=-\infty}^{+\infty} f[m] \times g[x-m] \tag{15}$$

We finally prove that if the kernel is separable, we can perform the convolution in two steps with the following formula:

$$h[x, y] = f_2[y] * (f_1[x] * g[x, y]) \tag{16}$$

The following picture is showing how we can separate a kernel. (the star here stands for the matrix multiplication)

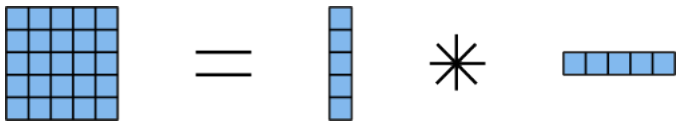

Figure 4: Separability of a 2D kernel in two 1D kernels

A good solution to know if a kernel is separable is to study the rank of the matrix. Indeed, if the rank is equal to one the kernel is separable. To determine that we just need to make sure that the Singular Value Decomposition of the Kernel only have a unique singular value.

2.3.3 Convolution issue

As you can see on the figure 3 (b), to fit perfectly in the image, the center of the kernel is not on the first pixel of the image. Indeed, we are always modifying the center pixel of the kernel so what to do if the kernel is lying in an area outside the image. This something we will have to take in account when we are going to implement the convolution because several solutions are possible to get over this issue.

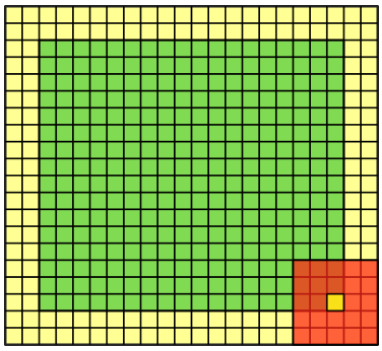

Figure 5: Result of implementation

The first solution to that problem will be to redefined the image, with a bigger size and surrounded by zeros or other constant value. This will allow us to really cover the all image with the kernel.

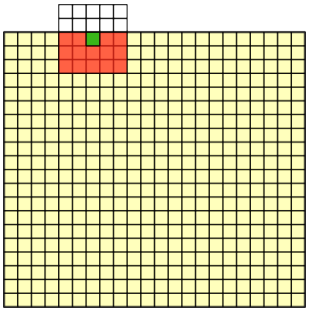

Figure 6: Possible solution with zeros outside the image

A second approach could be to implement a wrapping of the image. That is to say consider that the image is circular on both directions which gives it a spherical shape. The implication for the pixel of the kernel is presented on the following figure.

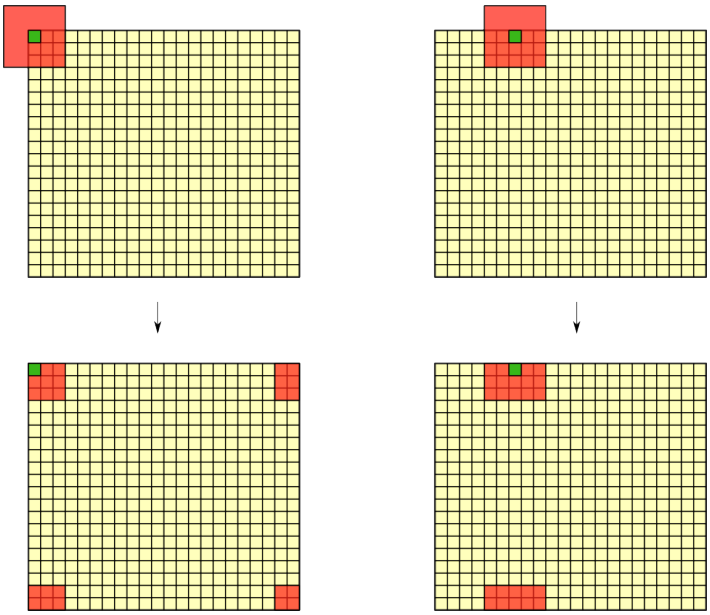

Figure 7: Possible solutions with wrapping according to the position of the kernel

There are probably other solutions, this is just few example to solve this issue. The solution to use will of course depend of the application and the purpose of the filtering operation.

2.4 Edge detection

2.4.1 Simple edge detection

The basis of edge detection is relying on the gradient of the image. In a continuous space the gradient is defined by :

$$\nabla(f) = grad(f) = \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} \tag{17}$$

The gradient of an image has two geometrical properties: its magnitude and its direction given by the angle between its two components:

$$\begin{cases} ||\nabla f|| = \sqrt{g_x^2 + g_y^2} \\ \alpha = \arctan\left(\frac{g_y}{g_x}\right) \end{cases} \tag{18}$$

In a digital image, the gradient is computed on a discrete structure so we need to compute it for every pixel with the following equations :

$$\begin{cases} g_x = \frac{\partial f(x,y)}{\partial x} = f(x+1,y) - f(x,y) \\ g_y = \frac{\partial f(x,y)}{\partial y} = f(x,y+1) - f(x,y) \end{cases} \tag{19}$$

Which gives us the following discrete kernel to apply in both x and y direction:

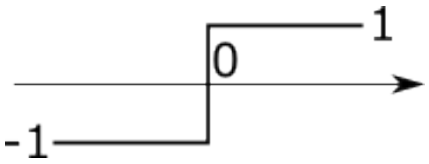

Figure 8: edge detection kernel

Let's also introduce Sobel's filter for edge detection that we are going to use as a comparison when we will implement the edge detection.

$$\mathbf{S_h} = \begin{bmatrix} -1 & 2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad \text{and} \quad \mathbf{S_v} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{20}$$

3 Filtering

3.1 Image Smoothing

In this part, the objective is to create a smooth effect on an Image. To do so, several methods exists, and the idea here is to consider a small patch which is going to move on the image and give to its central pixel the value defined by the pixel contained in it. See the previous section for a more theoretical definition. Smoothing is a technique used for reducing the noise on an image by blurring it. It's removing some small details from the image and could have interesting applications.

3.1.1 Direct smoothing

The direct smoothing is a reference to the processing we use to apply the filter. Indeed, as we explained in the theoretical part thanks to the mathematical properties of the convolution some filters can be apply separately. In this part we are going to implement the direct filtering method using the convolution between a bi-dimensional image and a bi-dimensional kernel. The implementation is presented in the implementation section. In this part we are going to run the function with different input images and different kernels to study the differences.

In a first part we are going to focus on a linear filter which is an averaging filter. This is the most intuitive way to reduce noise by relying on the idea that averaging the pixel values in a special neighbourhood should reduce the influence of the noise among those pixels.

Here is a first example of a filtering using W_3.

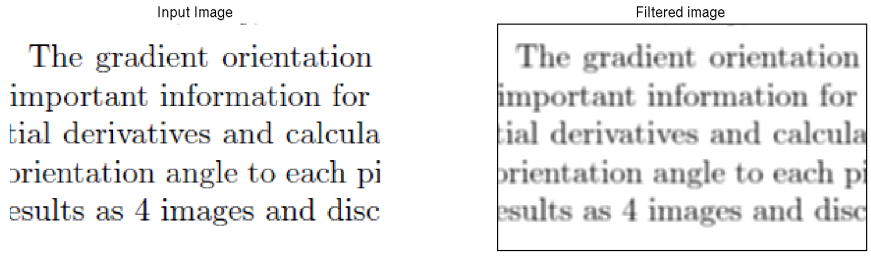

Figure 9: Smoothing of an original text image with W_3

Let's analyse the result. So first thing to say is that we have the 1 pixel black border as we were expecting due to the offset to prevent from filtering outside of the image. As we can also see, our implementation seems to be working pretty well because the result appears to be blurry. This image is good to see it because many characters have sharp edges on the original image and not that sharp if we zoom on it on the filtered image. This aspect is one of the most important aspect of applying a uniform weighted kernel.

Figure 10: Influence of smoothing on sharp edges

Indeed, on the previous figure we can clearly see that the 1 pixel edge of the T is now spread around. The explanation is that as soon as a border of the filter arrived on it, the neighbouring pixels are going to be affected because of the homogeneous weight of all the pixel in the kernel.

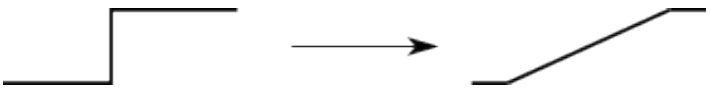

Figure 11: Transformation of edge due to smoothing

We can then take a bigger filter : W_5 to see what happen if the kernel gets bigger.

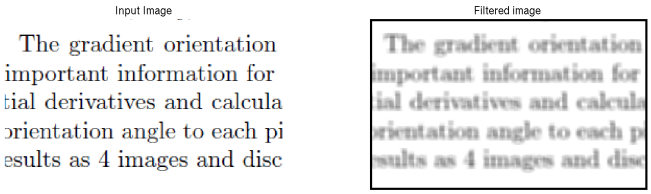

Figure 12: Smoothing of an original text image with W_5

When the kernel is getting bigger the image is getting blurrier and blurrier as we can see because the separation between letters is not clear any more. Indeed the bigger the kernel is, the bigger the blur is, because more pixels get into the computation of the value of one. In the case of a 5×5 kernel, the edge around the picture is composed of two pixels. So in a general $n \times n$ kernel, we will have a $\lceil \frac{n}{2} \rceil$ pixels edge. This first image with text was illustrating how smoothing was affecting sharp and small structures. We can also apply on it bigger images with bigger objects and structures. Here are few other examples.

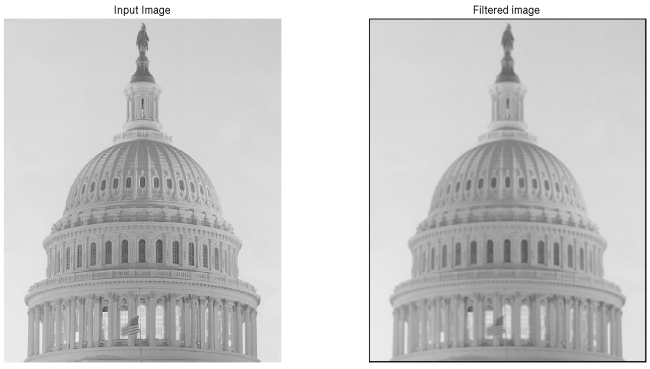

Figure 13: Smoothing of an original capitol image with W_3

Figure 14: Smoothing of an original cameraman image with W_3

Figure 15: Smoothing of an original cameraman image with W_5

If we analyse the previous results, we can clearly see on the capitol image, that sharp structures such as the small divisions of the windows are no longer visible because if they are 1 pixel big, they are averaging with 8 other pixel of background so they disappear little by little according to the size of the kernel. It also present within the example of the cameraman image where some clear structures of the camera are now really smooth with W_5 for instance.

3.1.2 Separable Filtering

This part is going to rely on the separability property we presented in the theoretical section. The kernel we are going to apply has to be symmetrical so we can decompose it with the following

formula:

$$w[x, y] = w_1[x, 1] \times w_2[1, y] \qquad \text{with } \times \text{ being the matrix multiplication} \tag{21}$$

In our case we are going to use again kernels W_3 and W_5 which are separable because they are symmetrical. Here is an example of separability for W_3, generalisable to any equally weighted squared kernels.

$$\mathbf{W_3} = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \qquad = \qquad \frac{1}{3}\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} * \frac{1}{3}\begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \tag{22}$$

The implementation of that part is shown in the implementation part let's show and discuss some results.

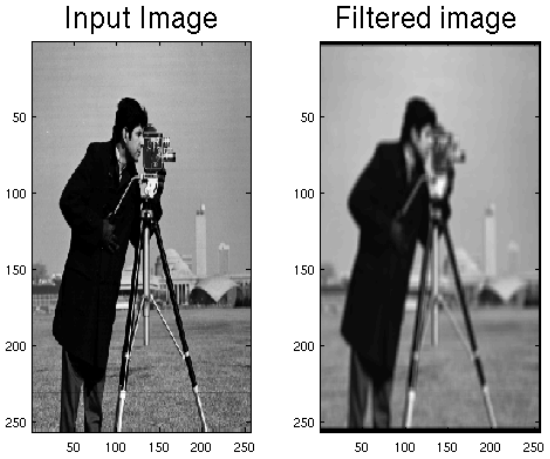

Figure 16: Smoothing of an original cameraman image with separable W_5

Comparing two images, like this one and the one shown in the previous section without having them side by side is not easy. So a good solution to know how different are two images is to subtract one from the other and display the resulting image. If it's black it means that the difference is equal to zero and that images are the same.

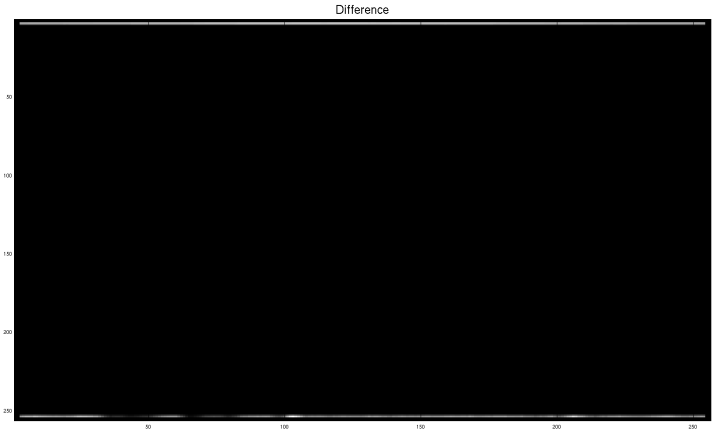

Figure 17: Difference between the two convolutions pipeline for smoothing with W_5

As we can see here, the two images looks exactly the same except on two pixel rows at the top and the bottom of the image. The explanation of this phenomenon is due to computation of the separable convolution is given in the implementation part.

3.1.3 Comparison between direct and separable filtering

In this section we are also going to apply the W_3 and W_5 kernel, so we should get the same resulting image. But the question here is more about computing performance. To quantify the difference between the two methods we are going to analyse the computation time difference between the two previous methods.
To do so, we use an embedded function of MATLAB called *tic-toc* which returns the duration to compute a block of instruction. Here, the studied bloc of instruction is relative to the computation of the two convolutions, we did not take in account the image input/output reading, displaying operations. We ran the comparison over several images using W_5, here are the results :

Image	Size	Direct Convolution	Separable Convolution
Cameraman	256×256	2.9 s	0.74 s
Capitol	470×370	7.6 s	1.82 s
Hand-hw	300×400	5.4 s	1.27 s
text	163×266	1.91 s	0.47 s

The duration are the mean duration over 5 executions of the code. Of course to be consistent we should do an average of more results and also use a local version of MATLAB. In this case, we used a network version so the calculation may be influenced by available bandwidth at the time of experiment. But this gives a pretty good illustration of the computation time difference. Indeed, computation time is dependent of Image size, the bigger the image, the longer the computation. The difference ratio between direct and separable convolution is around 4.

Which is a non negligible difference and comes from the fact that in direct convolution complexity is defined as $\mathcal{O}(width \times height \times n^2)$, were n is the dimension of a square kernel and *width* and *height* are the dimensions of the image. Indeed, if we look at the convolution implementation, we see that there are 4 nested ***for*** loops. While on the other hand, if we look at the separable convolution, each one of them has a $\mathcal{O}(width \times height \times n)$ so the total complexity is in $\mathcal{O}(width \times height \times n)$ which is better than $\mathcal{O}(width \times height \times n^2)$.

3.1.4 Separable Gaussian Filter

The Gaussian Filter is a really interesting filter because weights are not homogeneous and gives more influence to the closer pixels and less the others. We are going to implement the general formula given in the theoretical section, to create 1D Gaussian filter. We can also get some 2D kernels thanks to convolution because the Gaussian kernel is separable. In this part, the size of the kernel is going to be constrained by the choice of the standard deviation of the Gaussian filter. Indeed, we chose to use the standard $\pm 3\sigma$ as a reference so if σ equals one, the filter is going to have a $2 \times 3 \times \sigma + 1$ width and is going to be centred on $\lceil 3 \times \sigma + 1 \rceil$. The brace are used to represent the ceiling function. Indeed, to have a 5×5 kernel, you need to have $\sigma = \frac{1}{2}$ which is going to be centred on 3.

The implementation of this function is presented in the implementation section. Let's analyse some results.

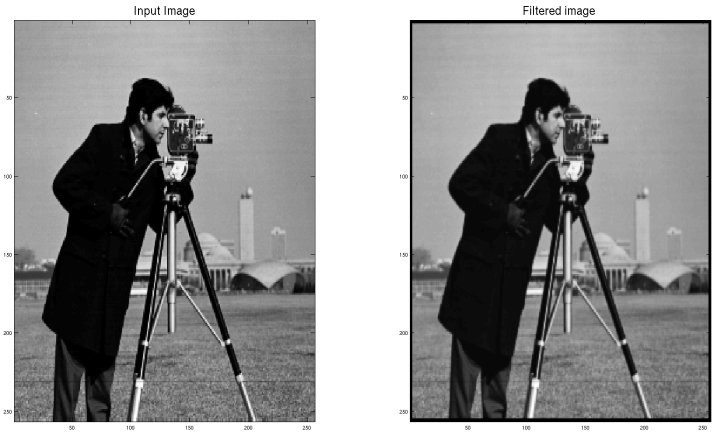

Figure 18: Smoothing of an original cameraman image with separable G_5

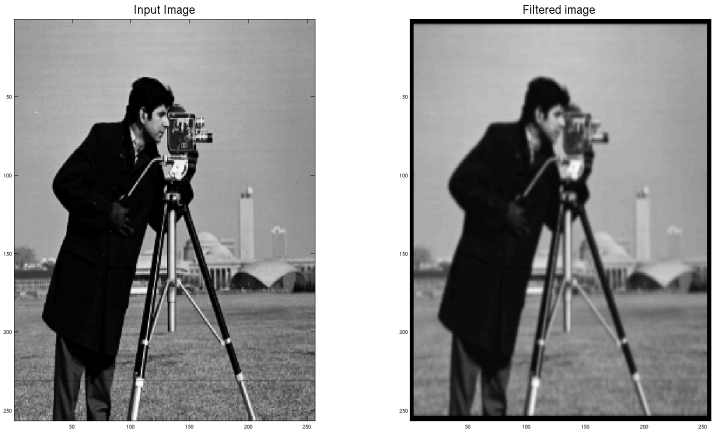

Figure 19: Smoothing of an original cameraman image with separable G_7

The first thing to notice here is that according to the formula we implemented, we have to have big enough kernels to see a significant result. Indeed, if we look at the first two kernels we can see that they are not influencing a lot the image. Because their respective sigma is 0.25 and 0.5. And this filters with very small sigma are not smoothing the image because the weights for the neighbours pixels are really small. On the other hand, if sigma gets bigger, so does the kernel, the smoothing effect is more visible and important. So, if we apply the kernel described in the project with $\sigma = 2$ we obtain:

Figure 20: Smoothing of an original cameraman image with separable G_{13}

So let's do a quick comparison with some results we got before for the same image. Here again

I'll use the difference between image to illustrate. If we compare the two filtering on the cameraman image with respectively W_5 and G_5 we can see that the image filtered with the Gaussian kernel appears to be less smooth. The reason is that the Gaussian weights distribution is not giving a weight as big as the one given by the averaging kernel to the far neighbours.

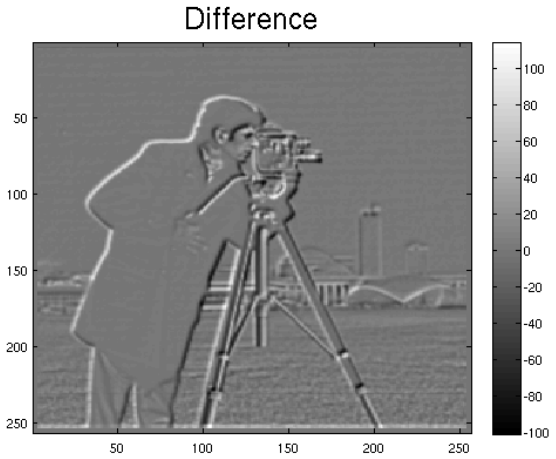

Figure 21: Smoothing of an original cameraman image with separable G_{13}

Indeed, if we look at the previous figure, the difference between the two images shows clearly that these differences are located on the edges of the objects. The grey is the showing pixel were the difference is close to zero. And that actually makes sense, because in a background environment differences are not going to be as huge as on a object because the result of convolution affect pixels the same way because intensity are close. While on the other hand, when you are on an edge, the Gaussian kernel will give more weight to the central pixel compared to the neighbours which will create the difference. To illustrate this point the idea is to go on the head of the cameraman to look at the difference.

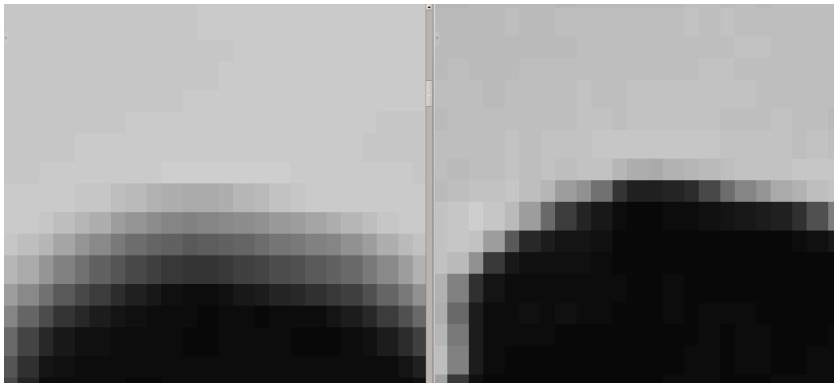

Figure 22: Comparison between homogeneous and Gaussian filtering on the head

On the previous figure, we can clearly see that the edge of the head is more clearly defined on the Gaussian smoothing than on the equal weight smoothing because of the influence of weights.

3.2 Noise Filtering

The idea of this section is to apply the two smoothing techniques we presented before to a noisy image to see what happens. To do so we are going to use the checker board image used in the previous project. The idea here is to provide an application of those techniques and also to illustrate some points presented before. Here are two example of the equal weight filtering applied to the checker board.

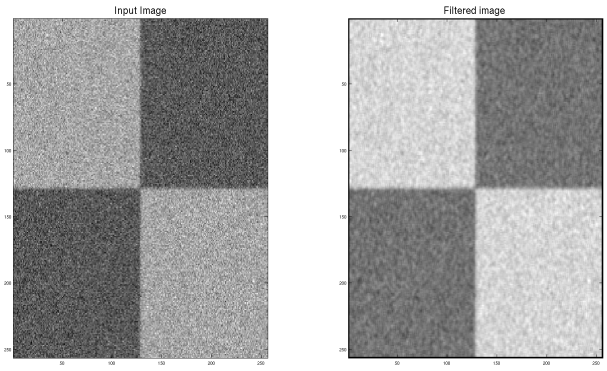

Figure 23: Checker board filtered with W_3

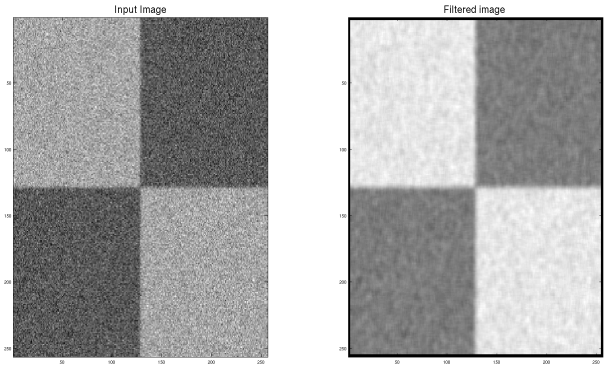

Figure 24: Checker board filtered with W_5

Here, we can clearly see that even with a small kernel like W_3 the noise is reduced and it gets better with W_5. So now, let's compare with the Gaussian kernel.

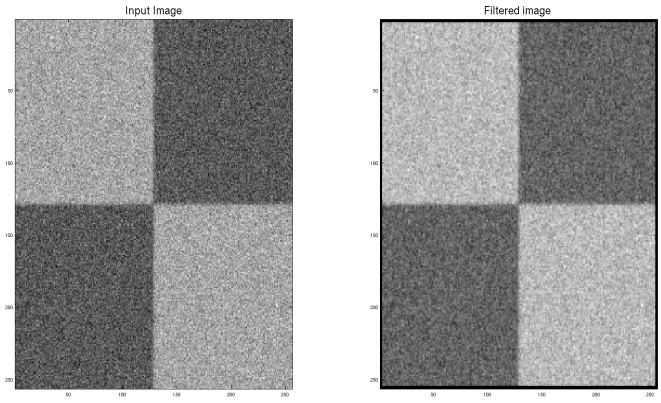

Figure 25: Checker board filtered with G_5

So, if we compare W_5 and G_5 we can see, that the blurring effect is much more effective with the equal weights than with the Gaussian filter, but the first one is blurring edges too while the Gaussian filter seems to keep them less blurry. An other interesting property we can see here if we use the functions we implemented in the previous project is that smoothing does not change the overall shape of distribution of intensities of the image. We generated the associated histogram for the previous images and the distribution stays Gaussian, but is slightly stretched and translated to brighter values.

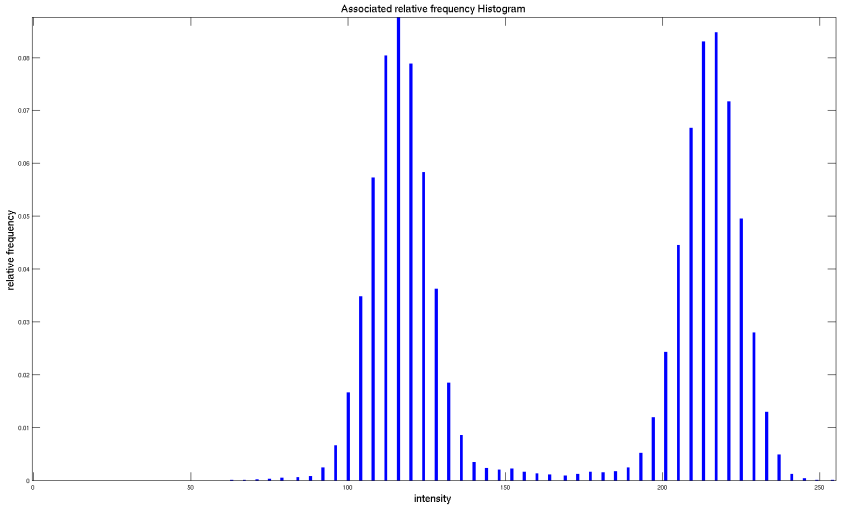

Figure 26: Histogram of Checker board filtered with Gaussian filter

So it seems that the choice of the filter really depends on the application, and if keeping a clearer edge is important or not. The best illustration possible would be using the text image where the difference is striking, if we use a 7×7 kernel.

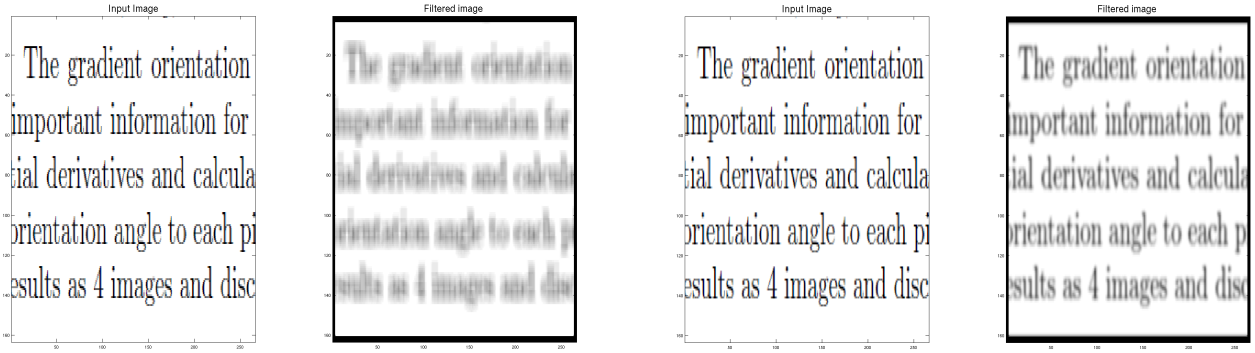

Figure 27: Comparison between W_7 and G_7

Here, we can see that with W_7 we are not able to read the next while we still can with G_7.

4 Edge detection

The objective of this part is to be able to extract some important features from an image such as edges.

4.1 Local edge filtering

In this part, we are using the kernel we introduced in the theoretical section and we are going to apply it on the image to see what can obtain. The implementation is presented in the implementation section. Firstly let's try with the horizontal vector $[-1, 0, 1]$:

Figure 28: Detection of vertical edges

As we can see we obtain an edge map composed mostly with the vertical edges, even if some diagonal and some horizontal are here. This is because when we apply the kernel, we compare two pixel in the horizontal direction so we find the places where vertical changes occur. This is mostly visible on the background of the image where we get the vertical structures of the buildings and also on the shoulder of the cameraman which is really horizontal and does not appear. The same reasoning can be inverted while we use the transposed vector so we expect to have the horizontal structure highlighted.

Figure 29: Detection of horizontal edges

Indeed, the previous statement is verified and we can clearly see it, specially on the head of the character, the camera or the building structure at the end. This was also a good way to prove that our implementation of convolution is working because we obtain the expected results. We can compare the two previous results with the use of the full 3×3 filters given in the book.

Figure 30: Detection of horizontal edges using Sobel filter S_h

The difference with the previous line filter is barely visible on this image, we have to zoom in to really see that this image is smoothed due to the weights used while the other one is not. Then we are going to generate the norm of the gradient and the angle associated. The edge map is presenting all the edges of the image because it's computed from both of the previous images. Here are some results that we are going to analyse.

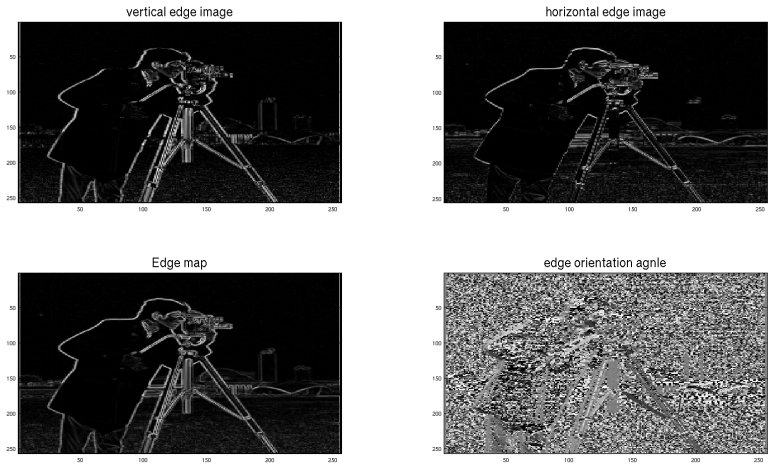

Figure 31: Cameraman edge analysis with dx, dy, gradient norm and angle

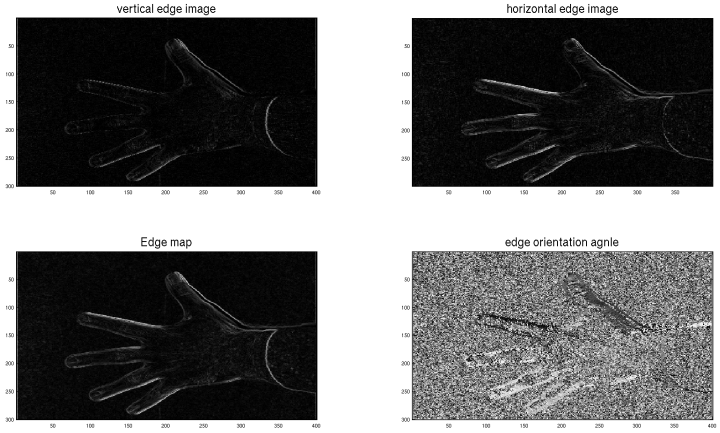

Figure 32: Hand edge analysis with dx, dy, gradient norm and angle

The edge map is pretty easy to analyse and understand in both cases because it represents the norm of the gradient so it's a combination of horizontal and vertical edges. While the angle map is harder to understand. It's in degrees in the range $[-180, 180]$. But if we look at it we can see that firstly there is a lot of noise and then that there is also areas where the color is the same. So it means that in these areas the gradient is the same. So the angle map gives us the area where the gradient vector has the same orientation, and on the previous image it fits pretty well the shape of the hand.

4.2 Edge filtering at specific scales

In this section we are going to use a slightly different pipeline to process an image to get edges. Indeed, we are firstly going to apply a Gaussian Filtering and then an Edge detection. Here are two examples with two different standard deviation of the Gaussian kernel we apply:

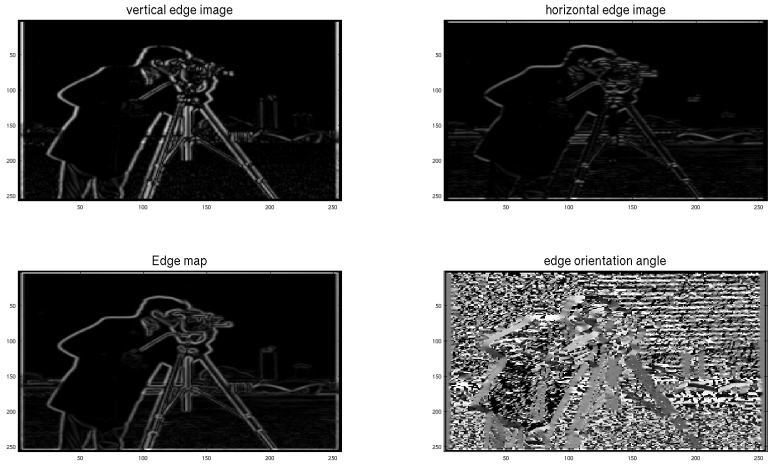

Figure 33: Cameraman edge analysis with dx, dy, gradient norm and angle previously filtered with G_7

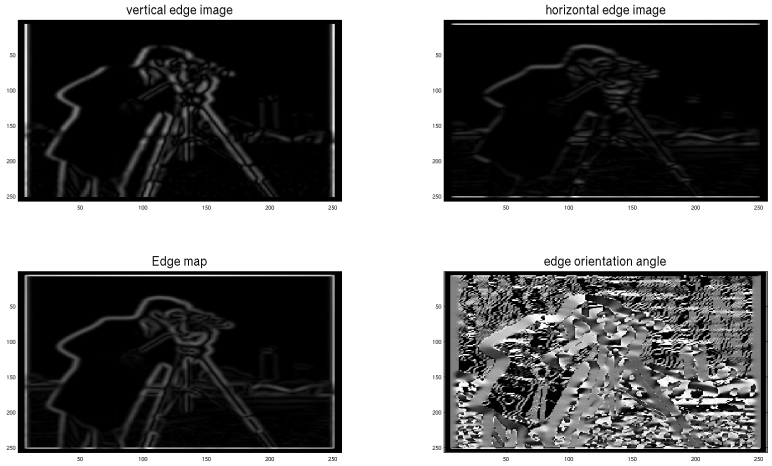

Figure 34: Cameraman edge analysis with dx, dy, gradient norm and angle previously filtered with G_{13}

If we compare the two previous results with the results obtained without smoothing, there is a lot to see and here the angle map is helpful to analyse. Indeed, applying a Gaussian smoothing with a large standard deviation is making the edges really blurry, so the edge detection is going to detect easily the biggest structures. In fact, we can see on the vertical and horizontal edge maps, that the buildings in the background are not as well defined as without filtering. And indeed, if the kernel gets bigger they become barely visible. But, on the other hand on the previous results, the edges on the main objects (cameraman, camera) are bigger so the gradient on them looks more homogeneous according to the angle map where the angle on them seems to be constant which was not obvious without smoothing. And they are also more visible because the noise in the background is more homogeneous due to smoothing.

5 Template matching

In this section, we are going to use the ideas and techniques presented before to perform a template matching. In this section, we are going to use the correlation between the image and the mask we are going to apply. First thing to do is to select an image and find a template. Many combination are possible we are going to present few of them. We are in a first time going to work with the shapes image. Let's in a first time consider that we want to locate the circles. The first solution would be to set as kernel mask the exact circle, which is going to be a huge mask. Or we can try to approximate it with a big enough square. This is what we are going to do in a first time and we will discuss it and then improve it. So here is our image and the resulting correlation with a 45×45 homogeneous kernel and the associated correlation image.

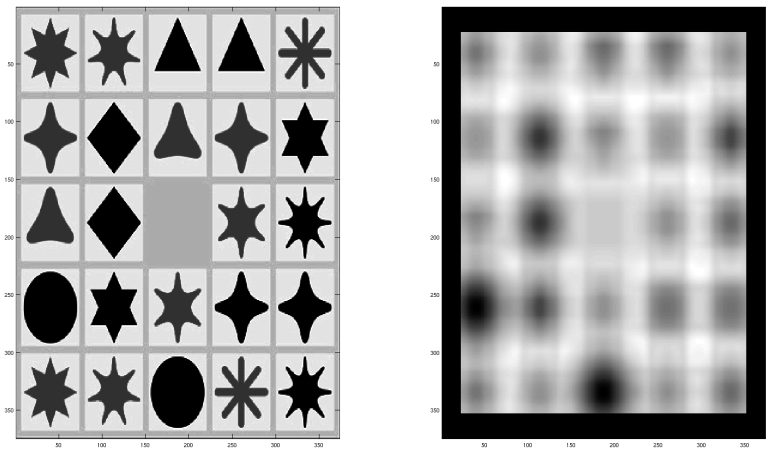

Figure 35: (a) Original shapes image (b)Correlation with W_{45}

Then to have a better view of the matching, let's invert the previous correlation image to have in white the places with maximum correlation.

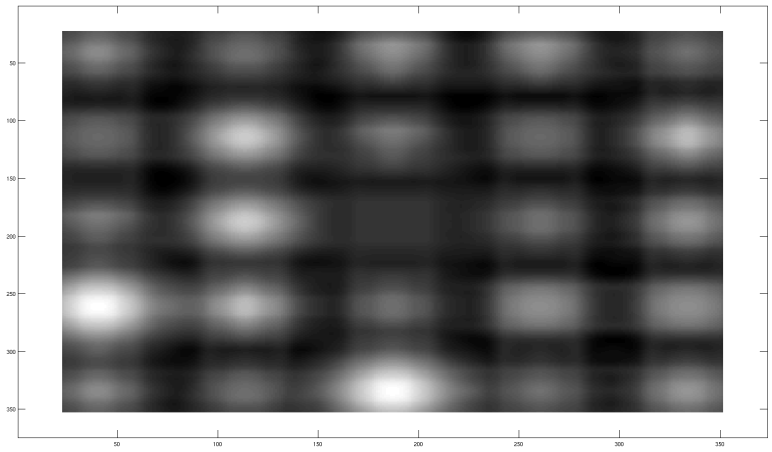

Figure 36: Inverted Correlation Image

As we can see on the previous image, there seems to be 2 spots where the correlation is big and maybe other few areas. The to big white areas are indeed the circles, but let's be sure of it by firstly looking at the height surface associated to this image.

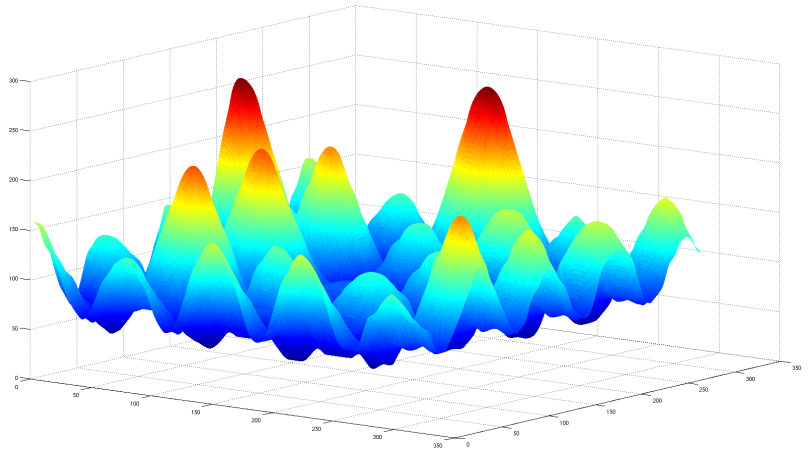

Figure 37: Correlation height map

Or we can secondly consider the threshold version of our correlation map.

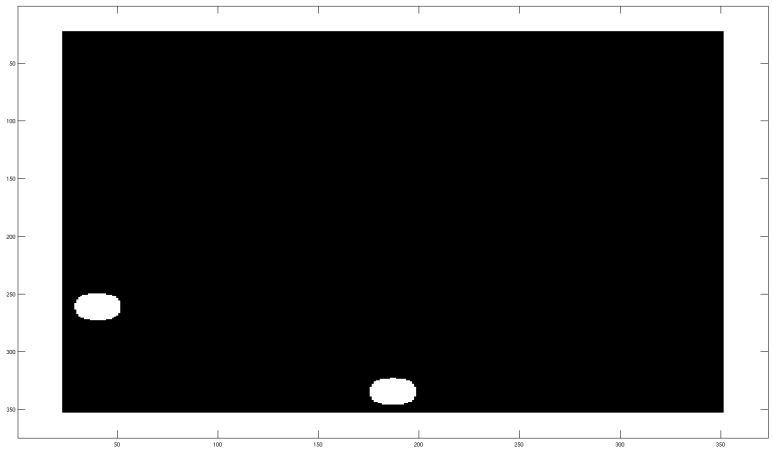

Figure 38: Binary correlation map

And indeed as we were expecting, the two spots we located before are the maxima of the correlation. So even with an approximation of our circle it's working pretty well. But there are few things to discuss here. Firstly if the approximation we took was bigger, *i.e* with a smaller square that would fit in more shapes than only the circles, our analysis would be wrong. Here is another inverted correlation map that shows the result with a 25×25 square. And as we can see there a more areas where the square fits so the correlation is high there.

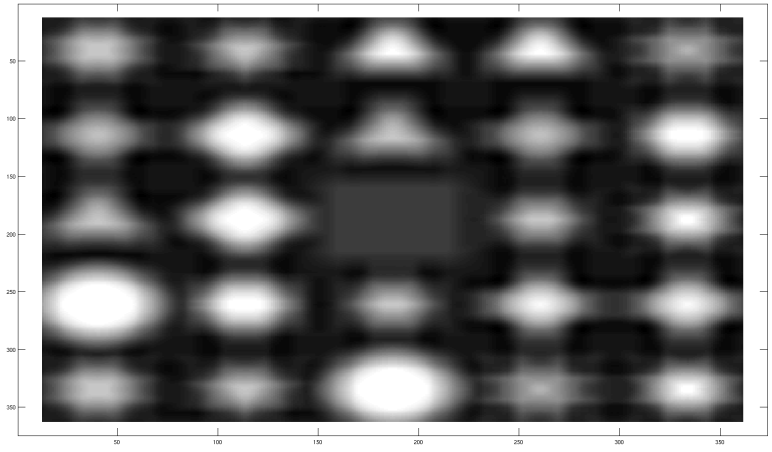

Figure 39: Inverted Correlation Image with W_{25}

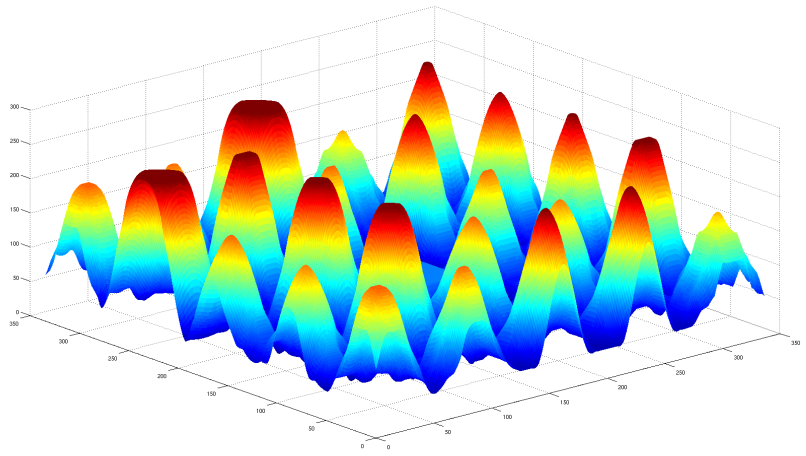

Figure 40: Correlation height map

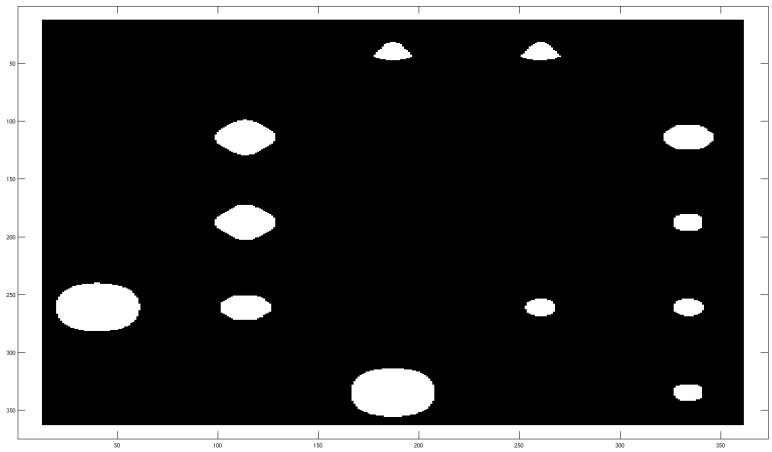

Figure 41: Binary correlation map

Another thing which is really important here is our method to deal with borders. Indeed, the bigger the kernel, the smaller the image. And in this precise case, we are looking for big structures, that we cannot always approximate so a convenient solution in this image would be to add a border around with half size of the kernel and with the background value. In this case, where our pattern as a really specific gray value it's not going to interfere or modify the image.

The image of the shape patterns is from far the easiest to analyse and to work with because shapes are big and the patterns are pretty easy to get and well separated from each other, but the only inconvenient is that their big size is going to require a longer computation time. So let's try with the text image and with the letter "a".

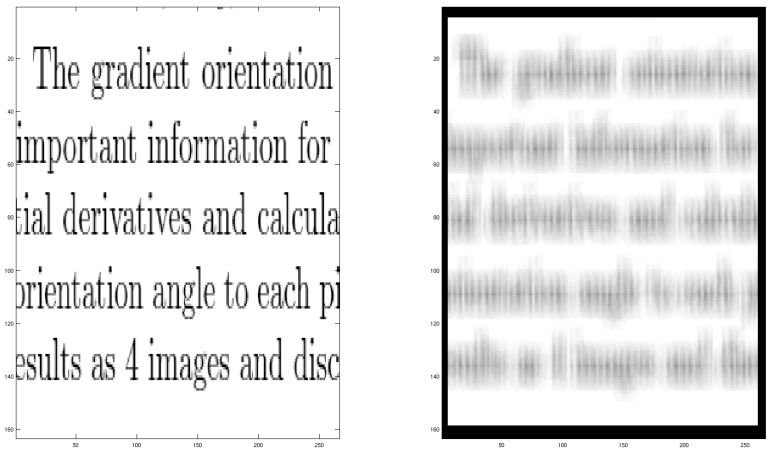

Figure 42: Text image and correlation map

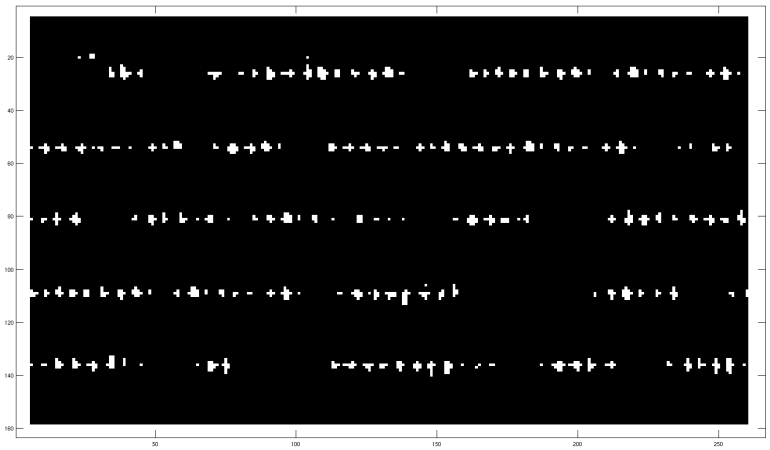

Figure 43: Text correlation map

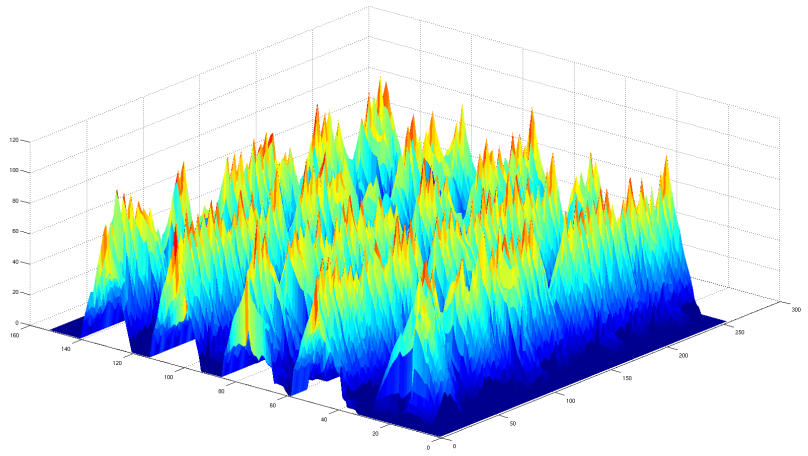

Figure 44: Text correlation height map

So as we can see on the previous results, here is harder to get exactly the position of "a" in the text image, because objects are smaller, closer to each other and not separable easily. An interesting way to see thresholding in this situation would consist in taking a virtual plan that the move up long the z axis of the surface. So the goal here is to find a good enough threshold that will give us the location we are looking for. Once we found a quite good threshold we can look at the location found and see if they match with what we are looking for. Here is the binary image resulting from a limit thresholding and the associate height map:

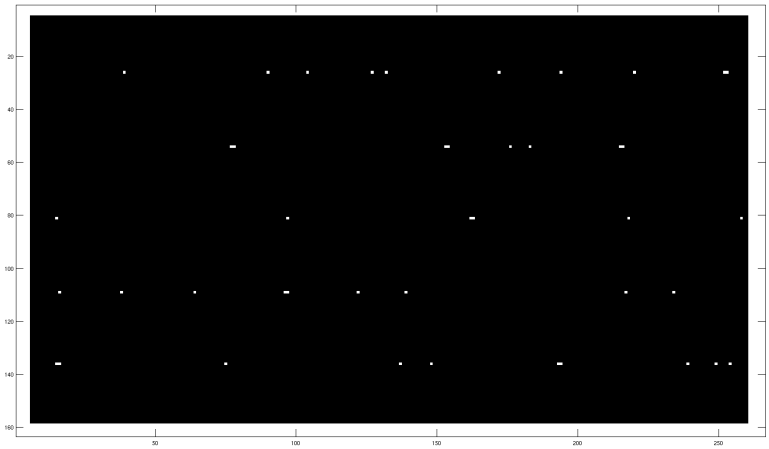

Figure 45: Text correlation map after threshold

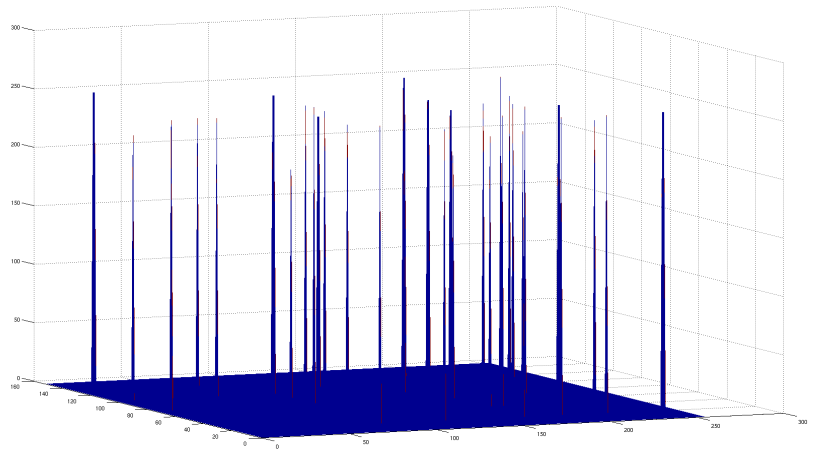

Figure 46: Text correlation height map after thresholding

The previous height map is like a "Dirac forest" where all the Dirac are representing a pixel where correlation with our template is high. If we analyse it we have 35 peaks while in reality we have 15 a in the image. So let see which of the peaks are actually representing a real a and what to the other represent.

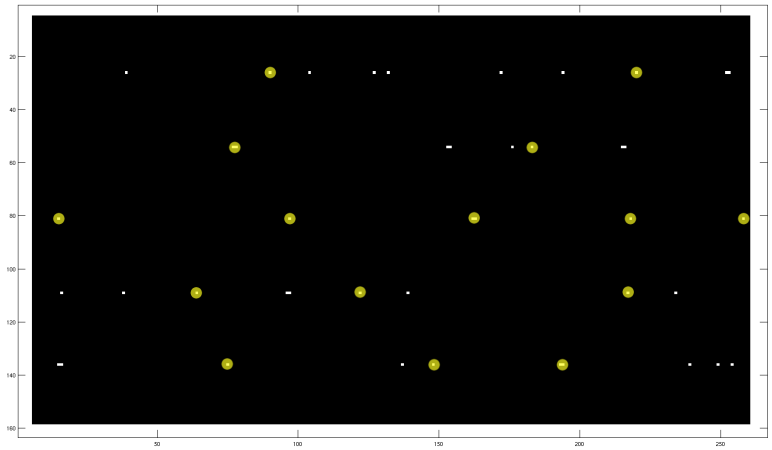

Figure 47: Highlight of pattern in text correlation map after threshold

So according to the previous result, we can see that all our $a's$ are detected, but also other letters. This is where it gets tricky because the threshold has to be high enough to remove as many

other letters as possible, but not too high to keep the $a's$. In the previous figure, if we increment the threshold value of a single unit, we lose several $a's$ which is not what we want, because their correlation value is lower than the correlation with other letters which could seem wrong but which actually does make sense if we consider that letters a really close and that all $a's$ of this image could be slightly different from each other. Indeed, it seems that the characters in the text are not homogeneous so all $a's$ could possibly be different. So the solution could be to threshold the image before. So we tried that aspect and we found out that it was not more effective. Here is the result:

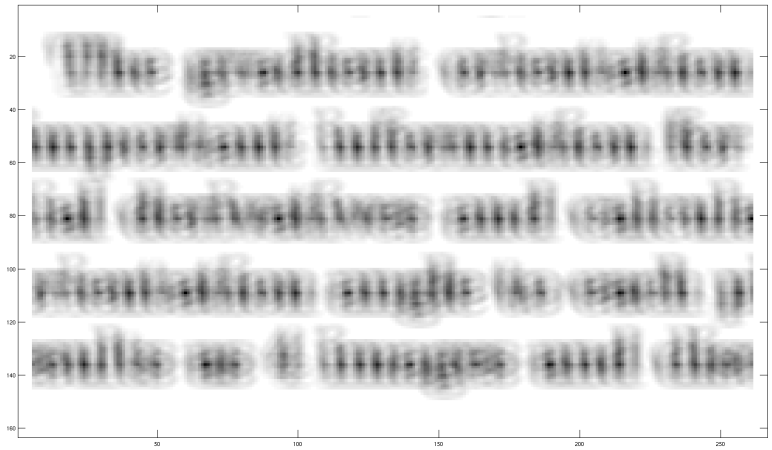

Figure 48: Binary text correlation map

So in this case, it seems pretty hard to have what we want, so based on the result with just few dots we can find the location of our $a's$ but also some other letters. Once we have that we could store the location of the remaining dots into a vector and try to compute the correlation only for those exact locations. We implemented this and we found out that the correlation of all these points was really close and we did not find a way to separate the $a's$ from the other letters.

6 Implementation

6.1 Convolution

Convolution is the most important function to implement in this project because it's the basis of all the filtering operations we need to perform. To implement the discrete convolution with MATLAB, we used the equation defined in the theoretical section:

$$h[x,y] = f[x,y] * g[x,y] = \sum_{j=-\infty}^{+\infty} \sum_{i=-\infty}^{+\infty} f[i,j]g[x-i,y-j] \tag{23}$$

Here is the core implementation of the convolution:

```
%walk through the image without the parts where the filter is not included
%completely in the image
for i = ceil(size(weight,1)/2):1:size(I,1)-size(-size(weight,1)+ceil(size(weight,1)/2))
     for j = ceil(size(weight,2)/2):1:size(I,2)-size(size(weight,2)+ceil(size(weight,2)/2)
         convol=0;
         %compute convolution for the neighbourhood associated to the kernel
         for a = 1:size(weight,1)
             for b=1:size(weight,2)

             convol = convol +
             (weight(a,b)*I(i-a+ceil(size(weight,1)/2),j-b+ceil(size(weight,2)/2)))/(255);

             end
         end
         new_im(i,j)=convol;
     end
end
```

Let's explain a tricky thing in this implementation: the definition of the correct boundaries of the loops.

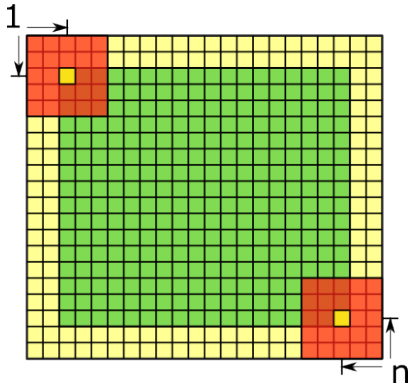

Figure 49: Explanation of the boundaries of the convolution

As we can see on the previous figure the starting point of the convolution is defined by the center point of the kernel. So we have to start computing the convolution from this point, because if not, we will face a situation where we need to read in undefined pixel (out of the array). So this is why the loops on the image respectively starts at the pixel corresponding to half the x dimension and y dimension of the kernel. Then the same idea applies for the last iteration which has to be located on the last pixel of the image.

Another aspect we have to take in account for the computation of the convolution is to flip the second signal and be sure that the right pixel are assigned for the multiplication. So we need to check if the multiplication is correct so let's consider a 3x3 kernel with starting position (2,2).

$\frac{a}{b}$	1	2	3
1	w(1,1)*I(3,3)	w(2,1)*I(2,3)	w(1,3)*I(1,3)
2	w(1,2)*I(3,2)	w(2,2)*I(2,2)	w(3,2)*I(1,2)
3	w(1,3)*I(3,1)	w(2,3)*I(2,1)	w(3,3)*I(1,1)

Which seems to be right because if we flip a 2 dimension matrix according to both axis here is what happen:

A	B	C
D	E	F
G	H	I

I	H	G
F	E	D
C	B	A

So it seems that the implementation seems correct.

$$OutputIm = convolution(InputImage, weight, display)$$

6.2 Separable Convolution

Due to the separability property of some kernel filters, we also implemented the separable convolution. The algorithm to do that is based on the following formula, which has been presented and explained in the theoretical section.

$$h[x, y] = f_2[y] * (f_1[x] * g[x, y]) \tag{24}$$

Indeed, as we can see on the previous equation, we cascade two convolutions. If we explained this aspect a bit more, we have a first convolution with the X component of the kernel which is going to create a first intermediary image (discussed in other sections) and then we have a second convolution with the Y component of the kernel which gives the result image. We will explain, and discuss those aspects later. Firstly let's present the implementation.

Here, the idea is to perform two convolutions between a vector and an image. So we implemented convolution between an image and a vector based on the previous code and run it twice.

```
for i = ceil(size(weight1,1)/2) :1: size(I,1)-size(weight1,1)+ceil(size(weight1,1)/2)
     for j = ceil(size(weight2,2)/2) :1: size(I,2)-size(weight2,2)+ceil(size(weight2,2)/2)
        convol=0;
         %compute convolution for the neighbourhood associated to the vector
         for b = 1:size(weight1,1)

             convol = convol + (weight1(b)*I2(i-b+ceil(size(weight2,2)/2),j));

         end
         new_convol(i,j)=convol;
     end
end

for i = ceil(size(weight1,1)/2) :1: size(I,1)-size(weight1,1)+ceil(size(weight1,1)/2)
     for j = ceil(size(weight2,2)/2) :1: size(I,2)-size(weight2,2)+ceil(size(weight2,2)/2)
         convol2=0;
             %convolution with vector on the image generated before
             for a = 1:size(weight2,2)

                 convol2 = convol2 + weight2(a)*new_convol(i,j-a+ceil(size(weight2,2)/2));

             end
         OutputIm(i,j)=convol2;
     end
end
```

The call for this function is :

$$OutputIm = convol\_separable(InputImage, weightx, weighty, display)$$

So we get here a close result from the previous one obtained with the direct convolution between the bi dimensional kernel and the image. But we needed to be sure that they were the same so we could use one or the other with no risks. To do so, we performed a very simple operation: the difference between the result of the direct convolution and the result from the separable convolution. Here is what we got:

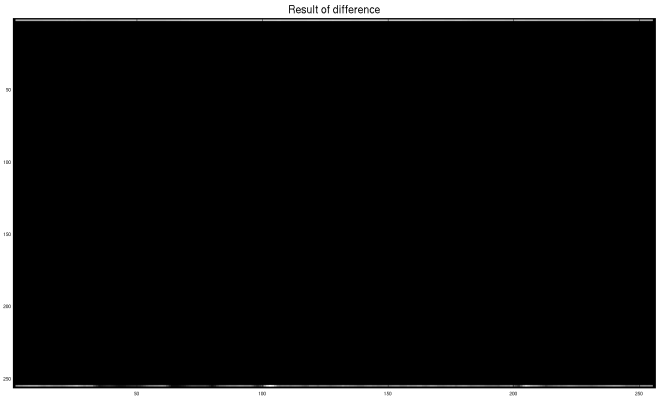

Figure 50: Difference image between the results of the two convolutions

So, according to what we have on that image, the result seems pretty good except on the top and bottom borders of the image. Indeed, we have here the result of the smoothing on the original image with the first kernel and then in the cascade we smooth the image where the border is defined with a zero. We can clearly see that the second convolution is done on the y axis because it affects rows. (Analogy with edge detection where a vertical filter shows horizontal edges).

6.3 Gaussian weights

6.3.1 Using formula

To implement our Gaussian filter we use the following formula and few tricks to make it work in MATLAB.

$$\begin{cases} f : \mathbb{R} \to \mathbb{R} \\ x \to f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-1}{2}\left(\frac{(x-\mu)}{\sigma}\right)^2} \end{cases} \tag{25}$$

In out program, the input parameter is the standard deviation σ that is conditioning all the kernel distribution. Indeed, by specifying a value of sigma and knowing that we want to be in the range $\pm 3\sigma$ it's then easy to derive the mean and the size of the vector. The mean is going to be 0 and according to the example, a σ of 2 should give a kernel of dimension 13.

```
for l=-(ceil(3*sigma)):1:ceil(3*sigma)
     g(l+(ceil(3*sigma))+1)=(1/(sqrt(2*pi)*sigma))*(exp(-0.5*((l-0)/sigma)^2));
end
```

We can of course check few things in this implementation, the first thing being the size of the filter if $\sigma = 2$. In this case we have a 13 elements signal. Then we can check the sum which is indeed equal to one, due to the use of the pdf's definition. The interesting thing to notice in this implementation is that σ is the only parameter conditioning the filter. For instance if we consider the G_3 filter we defined in the theoretical part we are not going to have it here. Indeed, we need to set a value of 0.25 to have a 3×3 kernel. And we have :

$$\begin{matrix} 0.00000011238 & 0.00033501293 & 0.00000011238 \\ 0.00033501293 & 0.99865949870 & 0.00033501293 \\ 0.00000011238 & 0.00033501293 & 0.00000011238 \end{matrix} \neq \begin{matrix} 0.0625 & 0.1250 & 0.0625 \\ 0.1250 & 0.2500 & 0.1250 \\ 0.0625 & 0.1250 & 0.0625 \end{matrix} \tag{26}$$

The difference is that in the second case, it's a 3×3 kernel which approximate a real Gaussian distribution, while in our case it's the real Gaussian kernel for the given standard deviation.

The call for this function is :

$OutputIm = separable\_gaussian(InputImage, sigma, display)$

This function is implemented based on the separable convolution presented before.

6.3.2 Using convolution

As presented in the theoretical section, there is an easy mathematical way to approximate the Gaussian distribution. This distribution relies on the convolution of the standard vector $[1, 1]$ to generate Newton's Binomial coefficients. The implementation just consists in a loop of convolution. And then based on separability properties of such a kernel, we can compute multidimensional Gaussian

kernels using matrix product.

The call for this function is :

$OutputIm = gaussian\_weigths(n, display)$

Here is the implementation to get the Gaussian normalized coefficients:

```
init = [1,1];
gauss = [1,1];
for i=1:100
     gauss = conv(gauss,init)/sum(2*gauss);
end
```

The results we get after few iterations of the previous code gives:

```
0.2500    0.5000    0.2500

0.1250    0.3750    0.3750    0.1250

0.0625    0.2500    0.3750    0.2500    0.0625

0.0312    0.1562    0.3125    0.3125    0.1562    0.0312

0.0156    0.0938    0.2344    0.3125    0.2344    0.0938    0.0156
```

Which is the pdf normalized version of :

```
1     2     1

1     3     3     1

1     4     6     4     1

1     5    10    10     5     1

1     6    15    20    15     6     1
```

Also well known as :

$$\binom{n}{k} = C_n^k = \frac{n!}{k!(n-k)!}. \tag{27}$$

And it also proves the fact that convolving box function will provide a Gaussian because [1,1] is the simple definition of a box function.
Then, to have a 2D kernel, we just need to use separability property associated with a Gaussian kernel :

```
gaussian2d = gauss'*gauss; % vector multiplication V^T * V = Matrix
```

Here are some results:

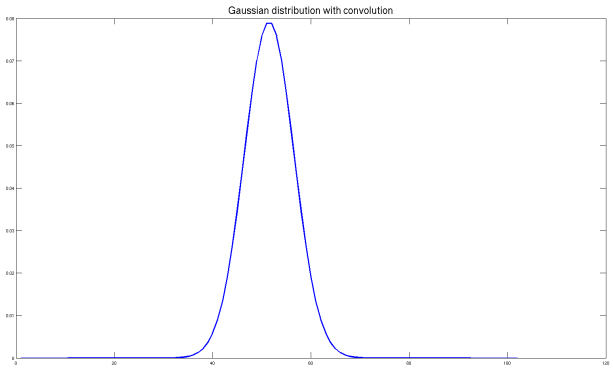

Figure 51: Generated Gaussian distribution with convolution

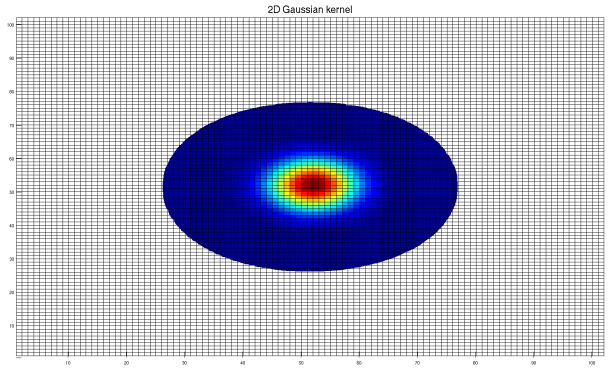

Figure 52: Multiplication of the previous 1D Gaussian to get a 2D Gaussian Distribution

The 3D view of the previous kernel.

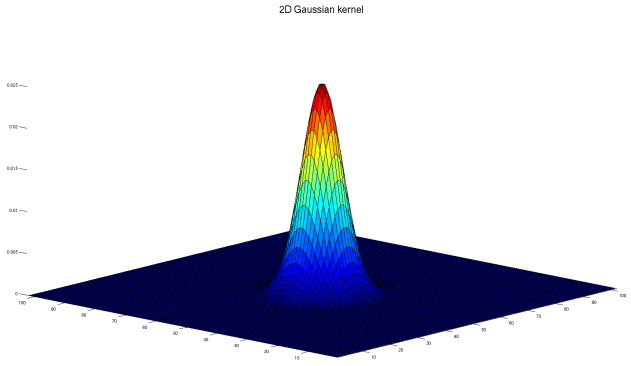

Figure 53: 3D view of the Gaussian kernel

Then to apply it on the image, we just need to use the resulting matrix $gaussian2d$, but the previous kernel is really big so we will use smaller ones and we will need to use the standard deviation as a parameter of the kernel.

6.4 Edge Detection

The edge detection algorithm is based on the previous separable convolution function and on the kernel we presented in the theoretical part. Here is the algorithm:

```
read image
convolution with horizontal part of the kernel
convolution with vertical part of the kernel
compute edge map and angle map
display results
```

Here is how we computed the edge and angle maps:

```
% compute edge map and angle map
for i = ceil(size(weighty,1)/2) :1: size(I,1)-size(weighty,1)+ceil(size(weighty,1)/2)
     for j = ceil(size(weightx,2)/2) :1: size(I,2)-size(weightx,2)+ceil(size(weightx,2)/2)

            norm_grad(i,j)=sqrt(new_im_separable2(i,j)^2+OutputIm(i,j)^2);
            angle(i,j)=atan(OutputIm(i,j)/new_im_separable2(i,j))*(360/pi);
     end
end
```

The call for this function is :

$OutputIm = edge\_detection(InputImage, weightx, weighty, display)$

Regarding the edge filtering at specific scale you need to compute first the Gaussian smoothing using the function presented before and then run this function.

6.5 Correlation

To be able to to perform template matching we rely on the computation of the correlation image. The following formula we implemented is:

$$(f \circ g)[n] = \sum_{m=-\infty}^{+\infty} f^*[m]g[n+m] \tag{28}$$

```
for i = ceil(size(weight,1)/2) :1: size(I,1)-size(weight,1)+ceil(size(weight,1)/2)
     for j = ceil(size(weight,2)/2) :1: size(I,2)-size(weight,2)+ceil(size(weight,2)/2)
         correl=0;
         %compute correlation for the neighbourhood associated to the kernel
         for a = 1:size(weight,1)
             for b=1:size(weight,2)

             correl = correl +
             (weight(a,b)*I2(i+a-ceil(size(weight,1)/2),j+b-ceil(size(weight,2)/2)));
```

```
            end
        end
        new_im(i,j)=correl;
    end
end
```

This part is not implemented as a function it's just a part of the template matching section.

6.6 Template Matching

In this section, the implementation relies on the correlation function presented before to compute the correlation map. Then we do a simple thresholding of the image on the inverted correlation image.

```
read image and template
compute correlation image
invert correlation image
threshold image
display results
```

Here is the implementation of the inversion and threshold steps:

```
%inverted correlation image
for i=1:size( new_im,1)
    for j=1:size( new_im,2)
        value= new_im(i,j);
        OutputIm(i,j)=255-value;
    end
end

% threshold image
for i=1:size( OutputIm,1)
    for j=1:size( OutputIm,2)
       value= OutputIm(i,j);
        if value <= threshold
        OutputIm2(i,j)=0;
        else
        OutputIm2(i,j)=255;
        end
    end
end
```

7 Conclusion

In this project we implemented the convolution which is a really important operation to perform smoothing and filtering. But we also saw that there is some issue regarding the way to implement this operation regarding the size of the image and the way how the kernel is going to interact with the image. Indeed, on the borders of the image, a part of the kernel lays on areas of the image that don't exists physically. So we have to come up with solutions to deal with this issue. The easiest one is not to consider a border in the image where the image is not completely defined. This is working well but reducing significantly the size of the image if the kernel gets bigger. So there is a trade off between the size of the kernel and its influence on the resulting size of the image. Another interesting solution would be to fill a border around the image with the background of the image so it does not interfere to much with the image and will not imply to reduce the size of the image.

We also had the opportunity to see that the mathematical property of separability could have a significant influence in computation. This is an important aspect that we have to take in account if the image and the kernel are big because the computation of convolution is time consuming so reducing it is always interesting. But this method is only available for separable kernel so this is the limitation.

We applied the blurring properties of smoothing to noisy images to see their influence. So we saw that smoothing noise makes it less important and does not change the initial distribution. The smoothing application is dependent of the application because average smoothing has a stronger effect on edges than the Gaussian.

The edge filter we implemented is quite good but could maybe be enhanced by using the Canny filter discussed in class. The fact of smoothing images before applying edge detection is making edges bigger so easier to detect for our simple edge detection filter.

The template matching application was really interesting because it has a lot of practical applications in the industrial world. Numerous industries are doing quality controls through image processing where they need to locate their object and then realize some measurements on it. So this is where pattern matching becomes powerful because it allows to find easily an object. Indeed, as we have seen through the text example if we are looking for an object that look like a lot to surrounding different objects it gets harder and detection has to be used in addition with other processing to clearly identify the desired object.

This project was a good illustration of the filtering techniques and processing that we can make in the Spatial domain of an image. This could be used as an interesting source to compare with the processing and filtering realized in the Frequency domain with Fourier analysis.

References

[1] Wikipedia contributors. Wikipedia, The Free Encyclopaedia, 2012.Available at: http://en.wikipedia.org, Accessed September, 2012.

[2] R. C. Gonzalez, R. E. Woods, Digital Image Processing, Third Edition, Pearson Prentice Hall, 2008.

List of Figures