MSc - MSIAM

Second semester Modelling project

Digital analysis of fingerprints

Authors :
M. Romain COUDERC
M. William DIDIER
M. Théo LAMBERT
M. Théo MOINS

Teachers :
M. Régis PERRIER
M. Christophe PICARD

February 19, 2018

Contents

Introduction

The aim of the project is to find mathematical methods, that can find matching images of the same fingerprint, and to implement them. Because of the variety of what can happend during a real life acquisition of a fingerprint (dirty or wet finger, motion blur, ...), we need to find different models to fit the different ways noise is added to the image. In order to do so, we will use different geometrical and image processing tools. The associated code will be provided in C++.

We decided to treat 'Starter' 1, 3, 4 and 'Main course' 1 and 3 as mandatory work. During this whole project, our goal hasn't been to do as much questions as possible from the subject, but rather doing enough to be able to then pursue further the parts that we find the most interesting.

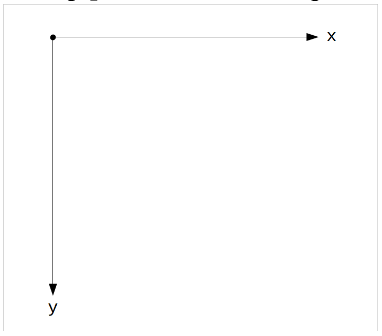
We decided at the beginning who would work on what, depending on the interests of each member. Our planning was updated every few days to take into account the potential delays or advances.

Chapter 1

Image Loading, Saving and Pixels Manipulation

1.1 Basics

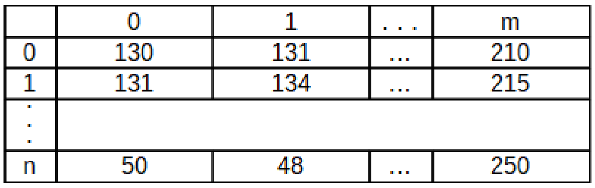
We decided to choose the following frame, which is the classical one used when manipulating pixels in an image :



In the PNG format, the intensity range in grey scale is from 0 to 255 (0 corresponding to black and 255 to white), so we chose to adopt the same convention. We have also implemented a function that can convert it to a value in $[0; 1]$. All the treatments we apply to the images are done with floating point values as intensities. We only switch back to integers when we write the image in a .png file.

Figure 1.1: From 0.0 to 1.0, the intensity scale of our images

In terms of library, OPENCV seemed to be the best option : it has a huge amount of features for image processing, and compared to others, their is a complete documentation with tutorials and an active community. The images are stored in a matrix class `Mat`, composed of `Char` (vector of `int`) type intensities.



	0	1	...	m
0	130	131	...	210
1	131	134	...	215
⋮				
n	50	48	...	250

Figure 1.2: Representation of a m*n image in OPENCV

1.2 Tests on a image

Our aim in this very first part is to implement different basic transformations like rotations, symmetries, etc... To check whether or not our code was working we decided not to test with fingerprint images at first, because they are almost squared and it's sometimes hard to tell if they have undergone the transformation we were expecting them to. To test the first functions of our program, we decided to work with the most famous image in the world of image processing : Lena.

Figure 1.3: First test image we used

1.3 Adding rectangles

Adding a uniform rectangle to the images is made through an important notion of OpenCV : selecting a Region Of Interest (ROI). The first thing to know is that when a copy of a Mat structure is made in OpenCV, the data isn't copied, but only the header pointing to the data. This means that if a copy of a Mat's data is modified, the original Mat's data is modified also because both headers point to the same data. The second thing to know, is that it is possible to select a ROI of an image. To begin, we define a region with a function such as *Rect*, *Circle*, *Ellipse*.... We then select the corresponding region of a Mat with a simple allocation :

```
void draw_uniform_rectangle (Mat image, Rect r, float color){
  image = image(r);
  image = Scalar(color);
}
```

This notion explains the implementation we have made to code this function. Usually, when we pass an image as a parameter of a function, we use pass-by reference to avoid copying the header. Here, we used pass-by copy, such that we could directly select the ROI using the parameter of the function. At any given time we have a maximum of 2 instances of the image.

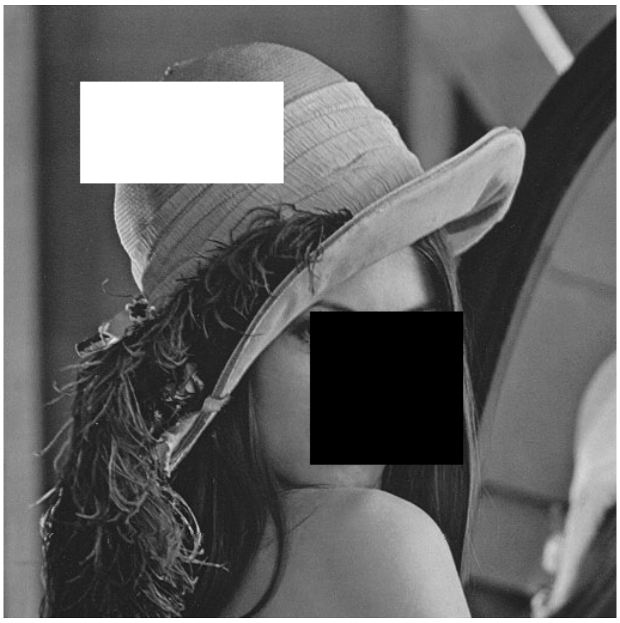


Figure 1.4: Lena with inserted rectangles

1.4 symmetries

We studied two basic symmetries : along the y axis and along both x and y axis.

For the symmetry along the y axis, the mathematical relationship between an image $f(x, y)$ and its symmetry $s(x, y)$ is : $s(x, y) = f(-x, y)$. Indeed, it is only a translation along the x axis. In order to implement it, we made the choice to create a `Mat` object of the dimensions of the original image, and then to swap the pixels using the relation $s(x, y) = f(x_{max} - x, y)$ to stay within the reserved memory.

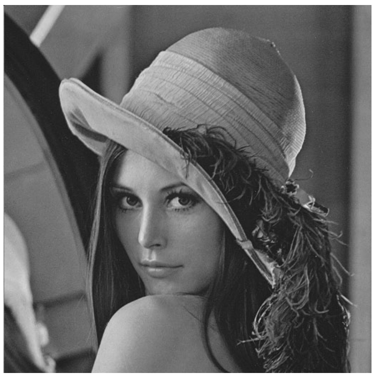


Figure 1.5: Lena after undergoing Y-axis symmetry

Concerning the symmetry along the x and y axis, we did almost the same thing as for the previous symmetry, but using $s(x, y) = f(y, x)$. The main change is that we had to swap also the dimensions of the `Mat` to avoid segmentation faults.

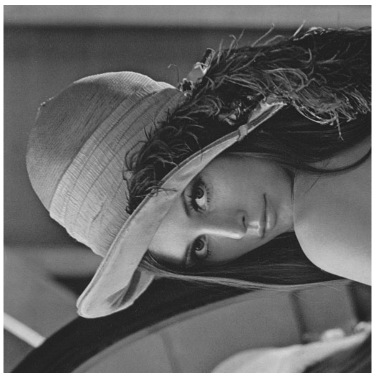


Figure 1.6: Lena after undergoing diagonal symmetry

1.5 How to obtain the outputs with our code

All of the outputs are obtained in a very simple way :

- Go to the "Digital-analysis" folder of the project
- Compile by executing the Compiler.sh script
- Run the Geometric.sh script with the image you want to apply the transformations to as a parameter `./Geometric.sh img/*.png`
- The result is stored in the "output" folder

Chapter 2

Pressure variations

2.1 Basic isotropic approach

Partie 2 : Vers un filtrage anisotropique Définir une ellipse sur laquelle on ne filtre pas Nécessaire pour la construction de l'ellipse : points limites centre de pression Trouver la fonction anisotropique
Conclusion : comparatif des différents résultats. Utilité ou non des différents travaux effectués.

The problem that we are facing is the following : People don't always press the sensor with the same strength when they use a fingerprint acquisition device. This is a problem if we want to compare two fingerprints : the information in the middle should be the same but there will be some information missing in the outer part of the image. We need to imagine a filter that allows us to model a change in the intensity of the pressure of the finger on the device. To do so, we tried and keep in mind what were the characteristics of a finger pressing a planar device :
-> The pressure at a given points depends on the point where the pressure is maximum
-> The information decreases with distance to this point increasing
-> We cannot make up information. Lower pressure means less contact between the finger and the plane which means less information.
-> The more constrast between black and white zones, the more recognizable the image is

2.2 Basic isotropic solution

Because of how the image is coded, the first solution that comes to mind is pretty basic. What we need to do is basically blacken the image when moving away from the pressure center. We decided to implement a solution that would compute a floating point coefficient c(x,y) and that would multiply the original intensity f(x,y) to give the new intensity g(x,y). Whatever the choices of implementation we will make in the future, we know that to treat each pixel like that the minimum complexity in time of this transformation is going to be $\mathcal{O}(n)$

As a first approximation, we used $c(r), r = d(x, x_s), x_s$ being the pressure center. The problem with this method is that it blackens the whole image, not only the grey parts. So we decided to apply a threshold to our image. If a pixel is "too white" then we set its value to full white (1.0). If not, then we apply the coefficient c(r) to its intensity.
In order to get an optimal threshold, we implemented the method described in **A threshold selection method from gray-level histograms**, *N. Otsu*, 1975, Automatica, which is based on variances computation over the grey-scale histogram of the image.

We decided to set the pixels that were close to being white to white to have more contrast on the output image. This way it looks as clean as possible.

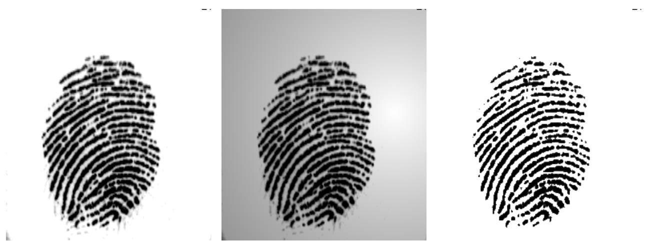


Figure 2.1: original image, then treated images without and with thresholding technique

This implementation is based on a very basic model and even though it gives a bit more contrast to the image, it doesn't seem to do much more. Although as we said in the intro to this part we cannot make up information that isn't there, there probably is a better way to get more from a weak image. In order to do so, we tried to implement different method that weren't based on applying a coefficient that only depends on r.

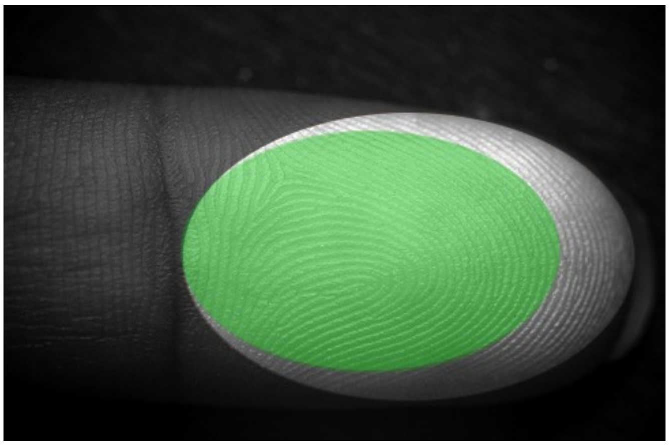
2.3 Dealing with the anisotropy : filtering around an ellipse

`Note: all the results and reasoning presented below supposed that the printing come from the right hand. A symmetry along the vertical axis crossing the pressure center cans be applied to deal with left hand fingers.`

2.3.1 Modelling the fingerprint and building an ellipse

Fingerprint modelling

To deal with anistropy, our idea was to create an elliptic zone on the fingerprint, on which no transformation will be applied. Indeed, because of the shape of the finger, some parts of the fingerprint tend to be more sensitive to weak pressure, as you can see in the illustration below :



In green : the elliptic zone considered as less sensitive.

The question is then to find how to parametrize the ellipse to make it fit as close as possible the shape of the fingerprint. First of all, we considered that the zone should be centered on the pressure center : it seems logical that the printing is correct where the pressure is high. It remains the semi-minor and semi-major axes to be determined.

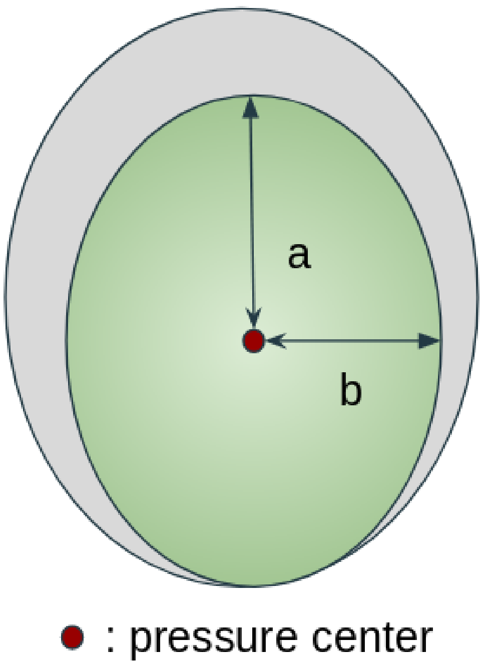


Figure 2.2: Geometric modelling of the fingerprint with the ellipse

According to the figure 2.2, and by denoting by (x_s, y_s) the coordinates of the pressure center, we get the following parametric equation for the ellipse:

$$\left(\frac{y-y_s}{a}\right)^2 + \left(\frac{x-x_s}{b}\right)^2 = 1$$

We denote by E the set of points that will not be considered for the transformations :

$$E = \left\{(x,y)| \left(\frac{y-y_s}{a}\right)^2 + \left(\frac{x-x_s}{b}\right)^2 \leq 1\right\}$$

We now have a first mathematical ground for our model. In order to test it on a real case, we had to find a way to compute approximately the pressure center from a fingerprint acquisition, and pick some values for a and b.

Computing the parameters

In this part we will focus on how we compute the pressure center of our fingerprint. We are therefore looking for the zone with the lowest intensity in the image *ie.* the most black zone. There is a minMaxLoc function in OpenCV that allows us to find the maximum and minimum intensity of an image, and also their location. The problem with this function is that it stops at the first extreme value it finds. You can see, circled in red, the points it returns as minLoc when applied to one of our fingerprints.

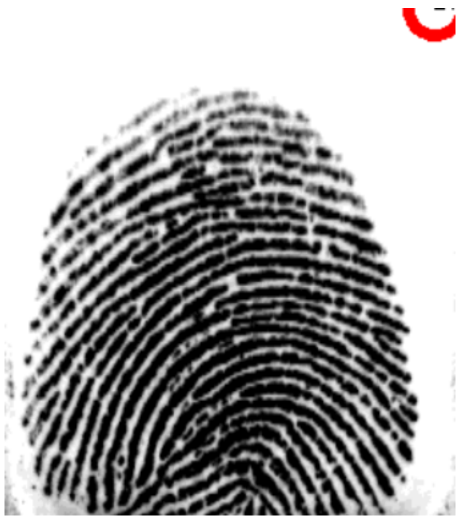


Figure 2.3: minMaxLox not being efficient

To solution this problem, we decide to make a copy of our image and to blur it. Therefore, there aren't any problems with small very black parts of the image interfering with the function. We decided that the most black pixel of the blurred image could be considered as the center of the high pressure zone, and therefore the pressure center. As you can see on the following images, the results are very conclusive. Even though we don't have any data about the actual pressure center of the acquisitions, the point is always were we expect it to be : more or less in the center lower part of the fingerprint.

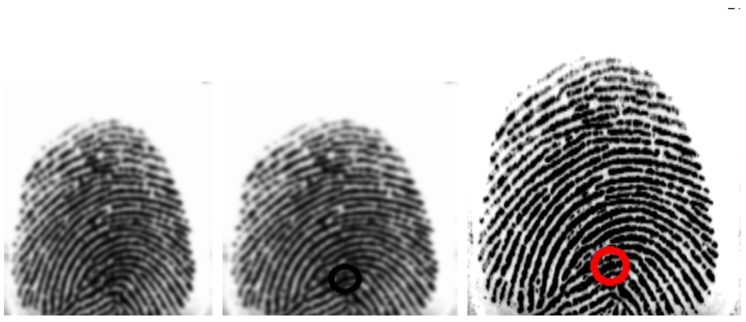


Figure 2.4: Three steps to make it work properly

We now have to decide how to set a and b, the two other parameters of the ellipse. We chose to compute the orange points on the following figure, which represent approximately the boundaries of the fingerprint :

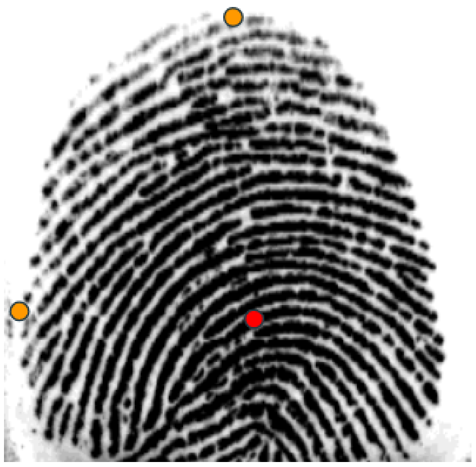


Figure 2.5: In red : the pressure center
In orange : our estimated boundaries

To find the two orange points we simply parse the image with a threshold that saves the first coordinates at which the intensity is high enough to be considered as part of the fingerprint.

We then decide arbitrarily, considering the shape of a finger, that $a \simeq \frac{2}{3} A$ and $b \simeq \frac{3}{4} B$. These parameters should be improved after, using statistical processing on a database for example, but constitute a quite correct beginning.

2.3.2 Creating the ellipse fitted to the fingerprint

Elliptical mask

The next step in our modelling is to apply the ellipse to the fingerprint. To do it, we created a Mat type object with the same dimensions as the image, that will contain the binary image of the ellipse. We denote by O the original image, and I the image of the ellipse. To compute I given the parameters, we apply the following function $\forall (x, y)$:

$$I(x, y) = 1 - \mathbb{1}_E \, (x, y)$$

The result for the image `clean_finger.png` when applying the process is shown on Figure 2.6.

We tried then to apply the isotropic filter previously detailed outside the zone covered by the ellipse, by including into the loop a condition depending on the color of the pixel

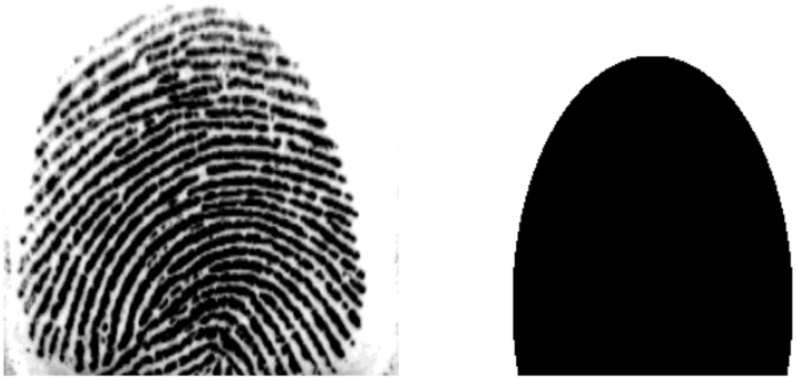


Figure 2.6: The original image and its elliptical mask

associated in the mask (white or black). As you can see on Figure 2.7, we get a high discontinuity at the boundaries of the ellipse, because of the way we defined the distance.

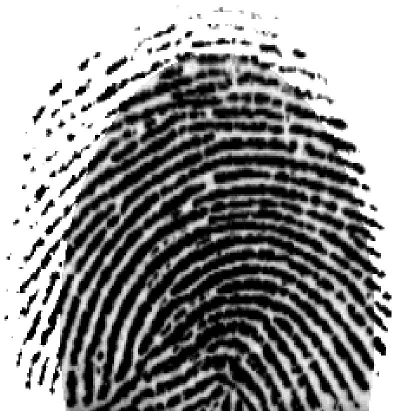


Figure 2.7: Discontinuity at the boundaries

Adapting the distance measurement

To face the issue of the discontinuity, we decided to cut the image into a sequence of layers of pixels, based on the ellipse, and then to compute the distance according to the layer number of each point. The idea is illustrated on the Figure 2.8.

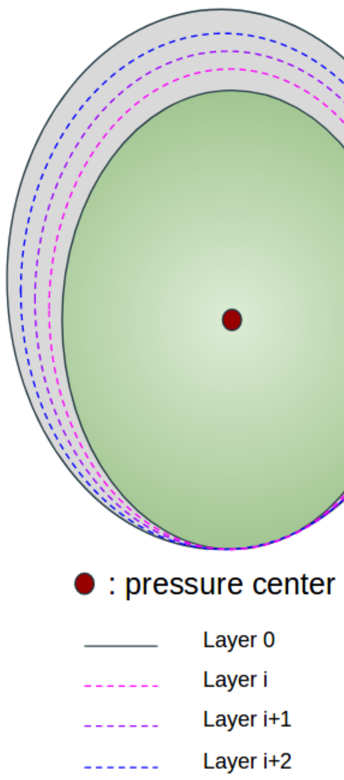


Figure 2.8: Elliptical layering

The recursive mathematical formula we found is :

$$E_0 = \left\{(x,y) \middle| \left(\frac{y-y_s}{a}\right)^2 + \left(\frac{x-x_s}{b}\right)^2 \leq 1\right\}$$

$$E_{i+1} = \Bigg(\{(x+1,y)\,|\,x > x_s\} \cup \{(x-1,y)\,|\,x < x_s\} \cup \{(x,y-1)\,|\,y < y_s\} \Bigg) \cap \bar{E}_i$$

$$(x,y) \in E_i$$

It is quite complicated, and the computation method isn't optimized, that is why we decided to simply extend the ellipse (by incrementing by 1 its parameters at each iteration).

2.3.3 Filtering

The algorithm we used is detailed below (Algorithm 1). The idea is to apply a coefficient to all of the pixels outside the ellipse at each iteration of the algorithm, in order to get a smooth result.

Algorithm 1 Anisotropic filtering

```
Require: a MAT type image (greyscale), the coordinates of the pressure center
  computation of the ellipse's parameters
  creating the ellipse
  for 0 to $max\_iterations$ do
    filtering outside the ellipse
    updating the parameters
    updating the ellipse
  end for
```

The first step is to estimate approximately the number of iterations that will be required to apply the filter properly. We decided to set it as the maximum of the distance (w.r.t x and y) between the pressure center and the boundaries of the fingerprint computed before. Even if this method can lead to a bit more calculations than necessary, we are sure to apply the filter to almost each point of the print.

The different parameters we used are listed below:

- the coefficient we chose to increase the intensity, called `intensity step` began from 1.00 and is incremented by 0.01 at each step.
- the size of the initial ellipse is around a quarter of the estimate dimensions of the fingerprint.
- the size of the axes of the ellipse are each increased by 1.

We started from a smaller ellipse than showned on Figure 2.6 to get a finer result, which is shown on Figure 2.9.
Remark : they only result from empirical tests, and we didn't had enough time to provide some analytical or statistical arguments to their value.

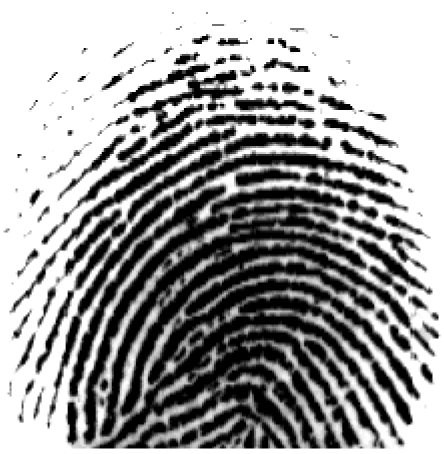


Figure 2.9: First anisotropic filtering

The result is quite satisfying, but we remarked that the result outside the initial ellipse tend to be too "binary" : indeed, because of the way we applied the coefficient (a product), the black zones of intensity 0 of the image are not modified by the filter. That is why we decided to add a small value, called `intensity floor`, to the values that has to be modified in spite of multiplying them by the coefficient. By this way, these intensities will also be modified during the other iterations.

The result shown on Figure 2.10 is better, but our filter cannot simulate the fact that the edges of the finger tend to be thiner at the boundaries in the case of a weak pressure. To reproduce that phenomenon, the mathematical tool that looked the most appropriate to us was the morphological filtering.

2.3.4 Using morphological filtering to simulate fading

Morphological filtering basics

The mathematical morphology is a mathematical theory that is frequently used in image processing. We will provide in our case some information about transformations on pixel sets using this tool.

We considered the set $E = \mathbb{R}^2$ and S a subset of E, that we will call "structuring element" (see on Figure for some examples). For $x \in E$, we denote by S_x the set defined by $\{s + x \,|\, s \in S\}$. The idea is first to put the center of the structuring element on each pixel fulfilling a property, such as a constraint on the color. Then, their intensity is changed w.r.t the shape of the structuring element.

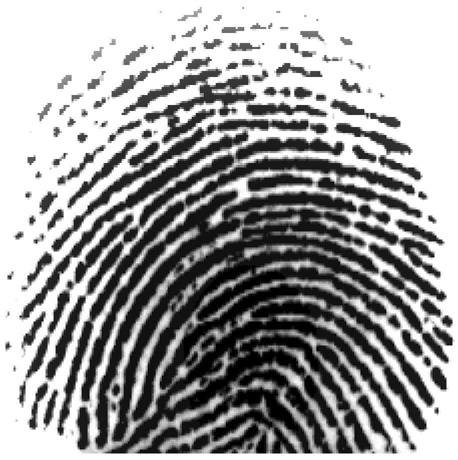
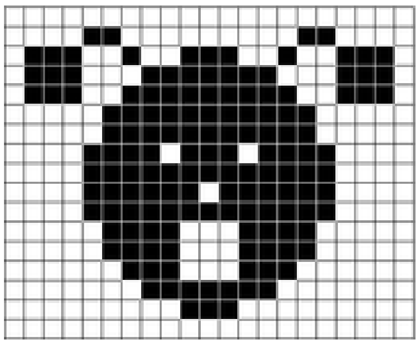


Figure 2.10: Elliptical layering

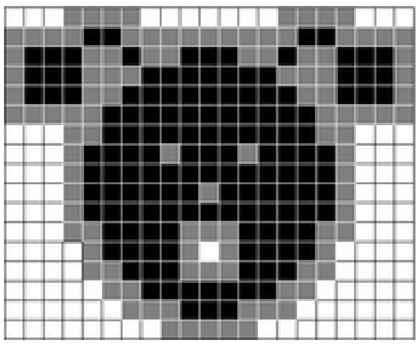
The two basic operations are the erosion and the dilation. The dilation w.r.t the structuring element S is defined by : $\delta_S(X) = X \oplus S = \bigcup_{x \in X} S_x$ and can be viewed as an extension of the pixels contained in X. The opposite operation (but not reversed) is called erosion, and is defined by : $\epsilon_S(X) = X \ominus S = \{x \,|\, S_x \subset X\}$. An example of such operations on a binary image and on the set of the black pixels is given on Figure 2.11.

`Note : the previous results can be generalized to grey-scale images.`

Original image



Dilation performed by a 3x3 square (grey pixels are part of the result)



Erosion performed by a 3x3 square (only black pixels constitute the result)

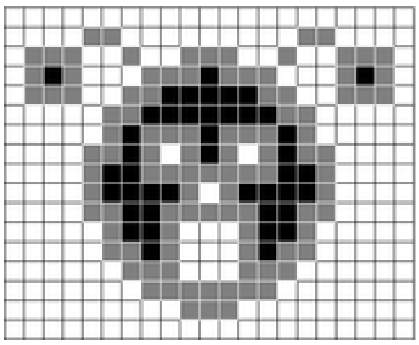


Figure 2.11: Illustration of morphological filtering operations

Source : *Morphologie mathématique*, Wikipédia

Applying the dilation

In order to improve the previous result (Figure 2.10), we used a pre-implemented dilation function adapted to grey-scale images (w.r.t light pixels). We applied it before the 'for' loop in the Algorithm 1 in order to preserve its effects on the input image.

To keep the elliptical modelling, we built an ellipse with other dimensions to select a zone on which the dilation has to be applied. As before, its parameters have been empirically set. The final result is shown on Figure 2.12.

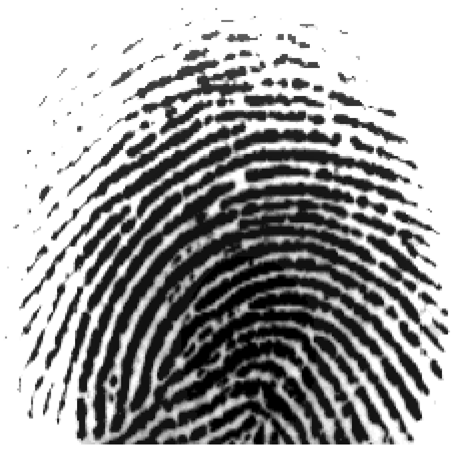


Figure 2.12: Final result

2.3.5 A quick look at complexity

We consider in this part an image of dimensions n x m. We will try to provide a very rough complexity analysis : the idea is to have a first look on the behaviour of our method from an algorithmic point of view. Regarding our problem, we can consider that $n \approx m$ (by cropping the image if necessary).

Time complexity

The most expensive part of our process is the loop in Algorithm 1, that can be estimated by a $O(n)$ cost. The three processes performed inside browse at most one time all the image's pixels : the approximate cost is so $O(n^2)$.
The global complexity is so roughly $O(n^3)$. Each operation can be optimized, but as a whole the general order cannot be lower than 3.

Space complexity

We suppose in this part that there is no unnecessary use of memory. From a spatial point of view, we only have to store three images of the size of the input image : the original image, the location of the result of each step (that can be updated in-place) and a location for the masks.
The global space complexity is so roughly $O(3n^2)$.

2.3.6 Conclusion

The final result is satisfying and consistent with the idea we have of a fingerprint with a weak pressure. However, it remains some points to upgrade and our approach must be widely completed. We listed below the main points we noticed :

- the transition between the eroded zone and the other part is noticeable, especially on the left.
- there is a lot of empirical coefficients in our method, without generalized process to get them. This issue hinder our approach to be applied as it is, for an industrial purpose for example. We thought about statistical processing to face that.
- we didn't have the time to work on a reversed method if it exists.
- we are not sure that the small size of the images considered compensate the space and time complexity of our approach.

2.4 How to obtain the outputs with our code

All of the outputs can be obtained in a similar way : compile and run the file `anisotropic_test` from the folder "tests". To do this :

- Place the terminal in the folder "build"
- Compile by typing `make`
- Run the file with the clean finger, by typing

  ```
  ./build/tests/anisotropic_test img/clean_finger.png
  ```

Figure 2.9 : You can comment lines from 99 to 104 in the file `elliptical_modelling.cpp` and compile the code, and set the global variable `INTENSITY_FLOOR` to 0 (initial value : 0.065).

Figure 2.10 : You can comment lines from 99 to 105 in the file `elliptical_modelling.cpp` and compile the code.

Figure 2.11 : You just have to run the test.

Chapter 3

Linear Filtering

For various reasons, (imprecise sensor, acquisition done too fast, etc), the output image of the fingerprint can be blurred, and therefore difficult to compare with the clean one. The aim of this part is to model as accurately as possible this perturbation, to understand what can be done if we want to correct the image. A classical way to create blur on an image is with a linear filter.

3.1 Convolution operation in 2D

Recall : 1D Convolution

We consider a discrete input signal $x = (x(n))_{n \in \mathbf{Z}}$. Here, the support of x will be finite, and we consider that $x(n) = 1$ where we don't define the signal.

For each $n \in \mathbb{Z}$, we can rewrite $x(n) = \sum_{k \in \mathbb{Z}} x(k)\delta(n-k)$

We transform the input into an output y by a linear and invariant system :

$$y(n) = S[x(n)] = S[\sum_{k \in \mathbb{Z}} x(k)\delta(n-k)] \overset{\text{linearity}}{=} \sum_{k \in \mathbb{Z}} x(k)S[\delta(n-k)] \overset{\text{invariance}}{=} \sum_{k \in \mathbb{Z}} x(k)h(n-k)$$

If we define $h(n) \overset{\text{def}}{=} S[\delta(n)]$. That's a way to deduce the definition of the convolution :

$$y(n) = x(n) * h(n) = \sum_{k \in \mathbb{Z}} x(k)h(n-k)$$

2D Version of convolution :

This approach of the definition of the discrete convolution allows us to generalize it easily in 2D. We are interested here in images, so the input signal is a matrix $x = x(n, m)_{(n,m)\in\mathbb{Z}^2}$.

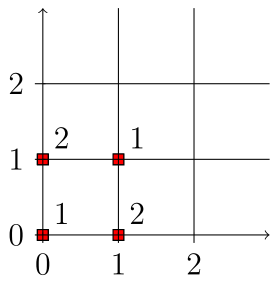
We introduce $(N_X, M_X) \in \mathbb{N}^2$, such as the dimension of x is $N_X \times M_X$.

In the same way as in 1D, we have $x(n, m) = \sum\limits_{i\in\mathbb{Z}} \sum\limits_{j\in\mathbb{Z}} x(i, j)\delta(n - i, m - j)$.

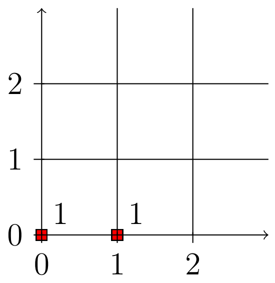
So with a system S linear and invariant, and $h(n, m) \overset{\text{def}}{=} S[\delta(n, m)]$, we can do the same calculation :

$$y(n, m) = S[x(n, m)] = \sum_{i\in\mathbb{Z}} \sum_{j\in\mathbb{Z}} x(i, j)h(n - i, m - j) = x(n, m) * h(n, m)$$

Illustrative example :



X



H

⚠In our implementation, the y-axis is in the other direction.

In this example :

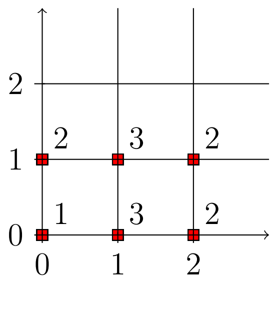
- $x(i, j) \neq 0$ if $i = 0, 1$ and $j = 0, 1$
- $h(n - i, m - j) \neq 0$ if $n - i = 0, 1$ and $m - j = 0$

So $y(n, m) \neq 0$ for $n = 0, 1, 2$ and $m = 0, 1$.

Remark : we can observe that y doesn't have the same dimension as x. In our case, we expect a result with the same dimension as the input, which comes down to computing $y(n, m)$ only on the support of x. Because of this constraint, we lose the commutative property of this operation.

But in reality, the dimension of y is $N_Y \times M_Y = (N_X + N_H - 1) \times (M_X + M_H - 1)$.

The result of this convolution is :



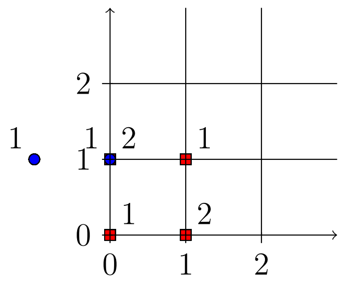
Y

Intuitive computation method :

Before the computation, it's necessary to flip both horizontal and vertical directions, in order to do the transform $h(i,j) \to h(n-i, m-j)$. Most of the time, this operation is not necessary because we use symmetric kernel.

Then, to compute $y(n,m)$: superimpose x and h so that $x(n,m)$ and $h(N_H-1, M_H-1)$ are superimposed, and then compute the sum of the products of the coefficients.

<u>Example :</u> for $y(0,1)$: $y(0,1) = 1 \times 0 + 1 \times 2 = 2$



Here, H is called the kernel, its size will be $(2P+1) \times (2P+1)$ for a practical reason. Indeed, to compute $y(n,m)$, it's more intuitive to superimpose $x(n,m)$ with the coefficient at the center of h, hence the necessity of odd dimension for h. This allows us to visualize as well as possible.

This comes back to using a shifted convolution :

$$x(n,m) * h(n,m) \overset{\text{def}}{=} \sum_{i \in \mathbb{Z}} \sum_{j \in \mathbb{Z}} x(i,j) h(n+P-i, m+P-j)$$

We implemented both convolutions (normal + shifted) on our code, since the shifted convolution only works for matrices of size $(2P+1) \times (2P+1)$ for h.

Because of this modelling, we chose the following way to deal with edges : we consider that every pixel around the image is a white pixel : $x(n, m) = 1$ if (n, m) are not between $(0, 0)$ and (N_X, M_X).

3.2 Naive algorithm

The "naive" algorithm is deduced from the intuitive way to calculate $x * h$:

Algorithm 2 Calculate $y = x * h$

```
Require: $x$ a matrix and $h$ a square matrix of odd dimension
  $P \leftarrow \frac{N_H-1}{2}$
  $BigX \leftarrow \text{Zeros}(N_X + 2P, M_X + 2P)$
  $BigX[P : N_X + P, P : M_X + P] \leftarrow x$
  for $i = 0$ to $N_X$ do
    for $j = 0$ to $M_X$ do
      $y[i, j] \leftarrow BigX[P - i : P + i, P - j : P + j] \otimes h$
    end for
  end for
```

Where we define $A \otimes B \overset{\text{def}}{=} \sum_{i,j} a_{i,j} b_{i,j}$.

We implement by ourselves the operation $\otimes$ for 2 matrices of same size. It's clear that the complexity of this operation is linear in dimension of the matrix : $\mathcal{O}(NM)$.

In our algorithm, we do this operation for every pixel of the image, with a matrix of the size of the kernel, so the complexity in time of the naive algorithm is $\mathcal{O}(N_X M_X N_H M_H)$.

Here, the edge problem is not too problematic, because the approximation will only affects P pixels on borders, and if we extend the image with white pixels, this approximation is quite satisfying.

The problem here is the complexity, that can be improved with another approach of the convolution.

3.3 Use of FFT

A classical way to compute the convolution product is through the Fourier Transform. In fact, we can compute the Discrete Fourier Transform (DFT) and the Inverse Fourier Transform (IDFT) with a relatively low complexity, thanks to fft and $ifft$. So with the formula $\widehat{f*g} = \hat{f}.\hat{g}$, we can deduce an algorithm with a lower complexity, but in our case of 2D matrices, we have to know the sense of "$X.H$".

Definition The DFT of $x(n, m)$ is the matrix X of the same dimension, such that :

$$X(u, v) = \sum_{m=0}^{M_X-1} \sum_{n=0}^{N_X-1} x(m, n) e^{-2i\pi(\frac{m}{M_X}u + \frac{n}{N_X}v)}$$

Let's now see how the relation between convolution and product translates into 2D.

Let's compute $Y(u, v) = DFT(y)(u, v)$, with $y = x * h$:

$$Y(u, v) = \widehat{x * h}(u, v) = \sum_{m=0}^{M_X-1} \sum_{n=0}^{N_X-1} y(n, m) e^{-2i\pi(\frac{n}{N_X}u + \frac{m}{M_X}v)}$$

Here, to understand the calculation, we have to write explicitly the sum indexes for the convolution : we based this on h dimension, because x will be extended in case of edge problems. Thus, we check if the indexes in h are between 0 and $2P$, so the result is :

$$\begin{aligned}
Y(u,v) &= \sum_{m=0}^{M_X-1}\sum_{n=0}^{N_X-1}\sum_{k=n-P}^{n+P}\sum_{l=m-P}^{m+P} x(k,l)h(n+P-k,m+P-l)e^{-2i\pi(\frac{n}{N_X}u+\frac{m}{M_X}v)}\\
&= \sum_{m=0}^{M_X-1}\sum_{n=0}^{N_X-1}\sum_{k=0}^{2P}\sum_{l=0}^{2P} x(P-k+n,P-l+m)h(2P-k,2P-l)e^{-2i\pi(\frac{n}{N_X}u+\frac{m}{M_X}v)}\\
&= \sum_{m=0}^{M_X-1}\sum_{n=0}^{N_X-1}\sum_{k=0}^{2P}\sum_{l=0}^{2P} x(P-k+n,P-l+m)h(k,l)e^{-2i\pi(\frac{n}{N_X}u+\frac{m}{M_X}v)}\\
&= \sum_{k=0}^{2P}\sum_{l=0}^{2P} h(k,l)\sum_{m=0}^{M_X-1}\sum_{n=0}^{N_X-1} x(P-k+n,P-l+m)e^{-2i\pi(\frac{n}{N_X}u+\frac{m}{M_X}v)}\\
&= \sum_{k=0}^{2P}\sum_{l=0}^{2P} h(k,l)e^{-2i\pi(\frac{k}{N_X}u+\frac{l}{M_X}v)}\sum_{m=0}^{M_X-1}\sum_{n=0}^{N_X-1} x(n-k+P,m-l+P)e^{-2i\pi(\frac{n-k}{N_X}u+\frac{m-l}{M_X}v)}\\
&= \sum_{k=0}^{2P}\sum_{l=0}^{2P} h(k,l)e^{-2i\pi(\frac{k}{N_X}u+\frac{l}{M_X}v)}\sum_{m=-l}^{M_X-1-l}\sum_{n=-k}^{N_X-1-k} x(n+P,m+P)e^{-2i\pi(\frac{n}{N_X}u+\frac{m}{M_X}v)}
\end{aligned}$$

To finish the calculation, we see that we have to the hypothesis of periodicity for the matrix x. We also have to extend the sum for h, and so to consider h on the same dimension as x : we have to do the DFT of $\left(\begin{array}{c|c} h & 0 \\ \hline 0 & 0 \end{array}\right)$.

We get :

$$\begin{aligned}
Y(u,v) &= \sum_{k=0}^{N_X-1}\sum_{l=0}^{M_X-1} h(k,l)e^{-2i\pi(\frac{k}{N_X}u+\frac{l}{M_X}v)}\sum_{m=0}^{M_X-1}\sum_{n=0}^{N_X-1} x(n+P,m+P)e^{-2i\pi(\frac{n}{N_X}u+\frac{m}{M_X}v)}\\
&= H(u,v).X(u,v)e^{2i\pi(\frac{P}{N_X}u+\frac{P}{M_X}v)}
\end{aligned}$$

Finally, the product we were looking for is the term by term product. We also observe that because of our shifted convolution, we have to shift X of P coefficients.

Finally, the steps to follow for computing $x * h$ are :

Algorithm 3 Calculate $y = x * h$

```
Require: $h$ a square matrix of odd dimensions
  $P \leftarrow \frac{N_H - 1}{2}$
  $Bigh \leftarrow \text{Zeros}(N_X, M_X)$
  $Bigh[0 : N_H, 0 : M_H] \leftarrow h$
  $x \leftarrow Shift(x, P)$ (shift $x$ of $P$ values)
  $X \leftarrow DFT(x)$
  $H \leftarrow DFT(Bigh)$
  $Y \leftarrow X \otimes H$
  $y \leftarrow IDFT(Y)$
```

Complexity

We can neglect the steps 1 and 2, because it can probably be done in constant time.

For DFT and IDFT : we use a function implemented on OpenCV, so we can't know the complexity. But we can suppose that the Cooley-Tukey algorithm is used :

Recall : For a vector of size n, the complexity for computing his DFT is in $\mathcal{O}(nlog(n))$ with Cooley-Tuckey.

For a 2D-vector :

We can write $X(u,v) = \sum_{n=0}^{N_X-1} \left(\sum_{m=0}^{M_X-1} x(n,m) e^{-2i\pi \frac{m}{M_X} u} \right) e^{-2i\pi \frac{n}{N_X} v} = \sum_{n=0}^{N_X-1} X_n^{(u)} e^{-2i\pi \frac{n}{N_X} v}$

Computing the coefficients $X_n^{(0)}, ..., X_n^{(M_X-1)}$ cost $\mathcal{O}(mlog(m))$ operations. Then, for each u, we compute the DFT of $X_n^{(u)}$, so we do $\mathcal{O}(mnlog(n))$ operations.

NB : with a similar reasoning, we can compute the 2D-DFT with $\mathcal{O}(mnlog(m))$ operations. So we can choose the lowest dimension of the matrix for the log.

For the step 4 : we multiply term by term each complex term of X and H, using a function of OpenCV called *mulspectrums*. We think that the complexity could be $\mathcal{O}(nm)$, but it's probably lower than that.

In a common case where the matrix is separable ($x(n,m) = x_1(n)x_2(m)$), we have $X(u,v) = X_1(u)X_2(v)$, and so it sufficient to compute 2 1D-DFT, which is of complexity $\mathcal{O}(nlog(n) + mlog(m)) = \mathcal{O}((n+m)log(n+m))$.

This reduction is possible for every kernel of rank 1.

Finally, if we consider a square matrix, we can approximate the complexity by $\mathcal{O}(n^2 log(n))$. If we compare with the complexity of the naive algorithm ($\mathcal{O}(n^2k^2)$, with k the size of h), it does not look far more efficient, and we see that for a matrix with a little kernel, it's better to choose the naive implementation (see Figure 3.1).

To obtain this curves, we use a time module on C++, and we write the result on a text file that we treat on Python. With *timeit* module, we easily verify that approximatively 10^7 operations per seconds are done on Python :

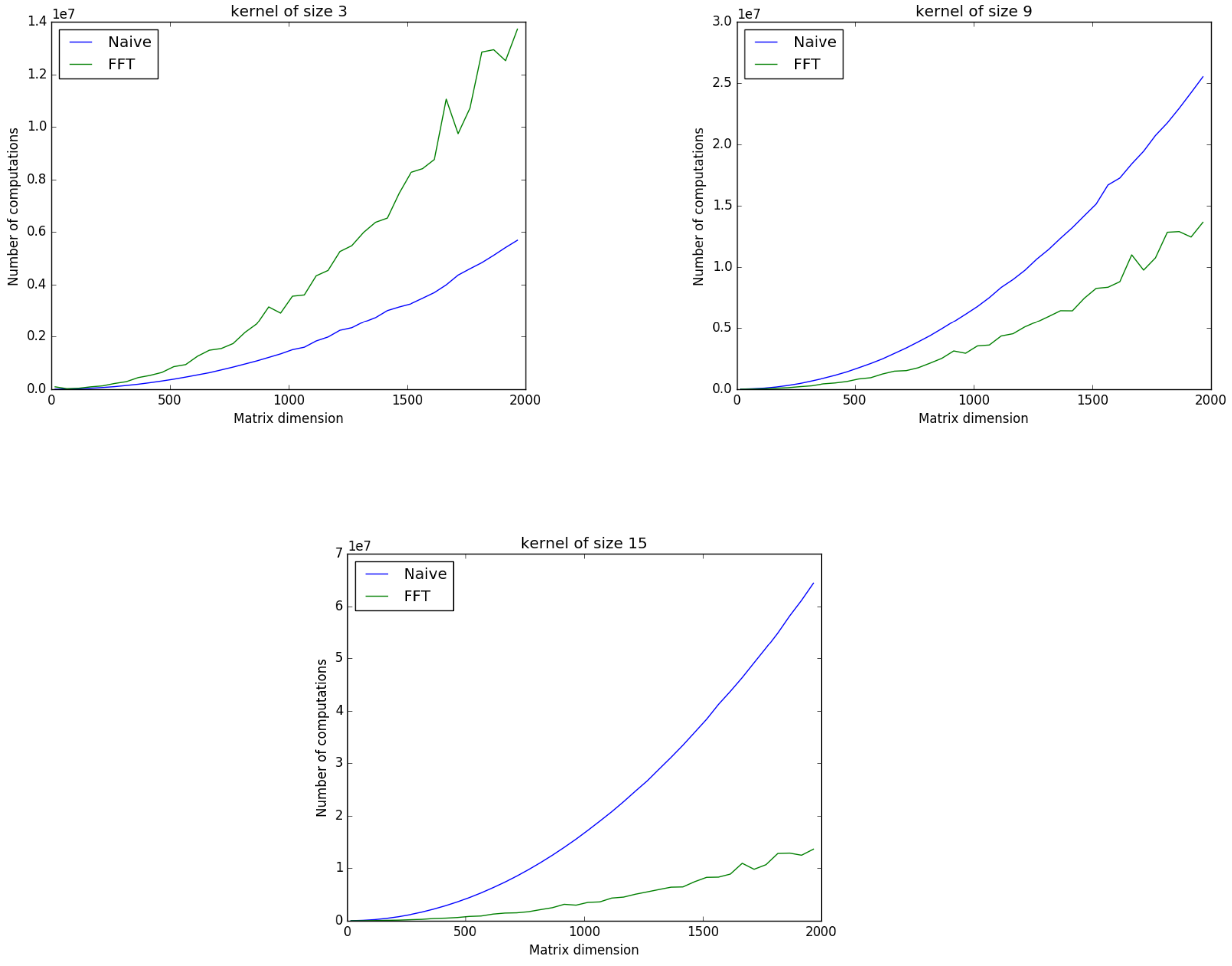
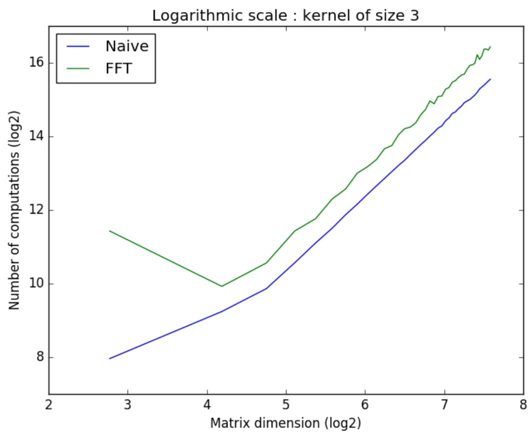
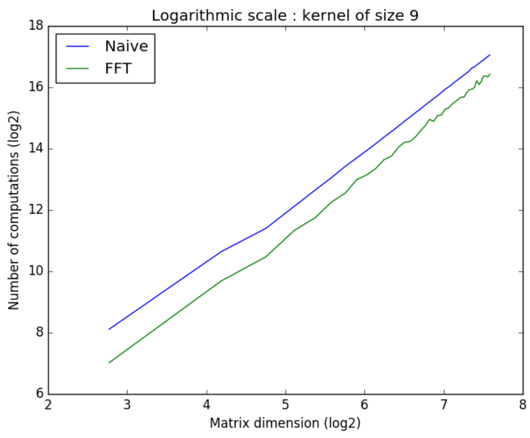


Figure 3.1: Complexity curve for 3x3, 9x9 and 15x15 kernels

If we plot in a logarithmic scale, we observe that both curve seems to have a similar complexity, the slope of the curves show a quadratic complexity (see Figure 3.2).

The ordinate at the origin gives the information of the coefficient in front of the order : the numeric results aren't really exploitable, but we see that the size of the kernel influences the naive algorithmmuch more than it does on the FFT implementation.





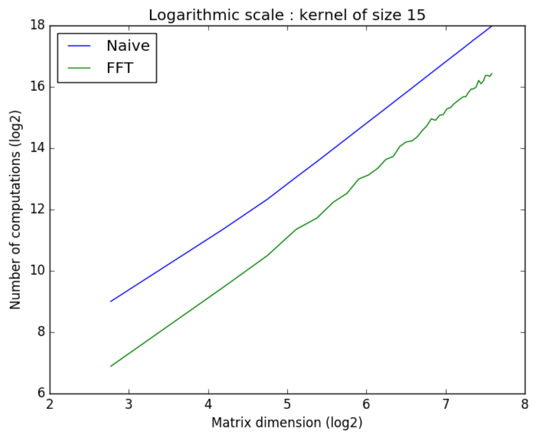


Figure 3.2: Logarithmic scale for 3x3, 9x9 and 15x15 kernels

If we compare to our theoretical results, we see that one of the curve is $\mathcal{O}(n^2)$, and the other is $\mathcal{O}(n^2 ln(n))$ (for a kernel of constant size). But the difference is difficult to show on logarithmic scale ; if we plot the curve $n \rightarrow n^2$ and $n \rightarrow n^2 ln(n)$, we don't have enough points to see the difference (see Figure 3.3).

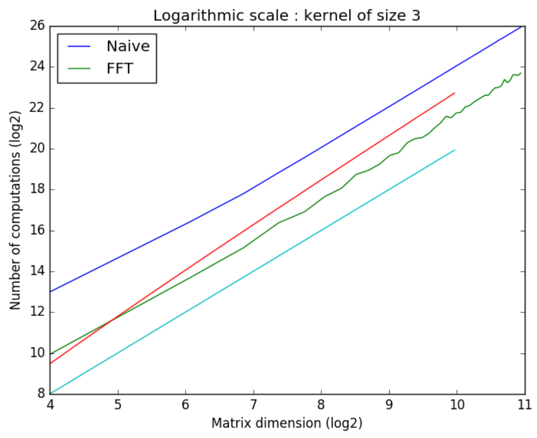


Figure 3.3: Logarithmic scale for a 15x15 kernel, with 2 theoretical curves

To conclude : For a big kernel, it's better to use the FFT algorithm, but for smaller kernels, the naive algorithm is more interesting.

Extension of the image to fix edge problem

In this version of the implementation of convolution, the use of DFT supposes a periodicity on both x and y-axes. That's why we enlarge the image before applying x-axis and y-axis symmetries to our original image. In that way, we have an image that is periodic on both axes, and therefore ready to be treated through DFT (see Figure 3.4).

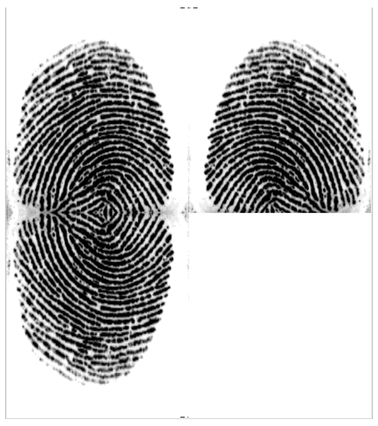


Figure 3.4: Extended image for FFT

3.4 Choice of the kernel

For each kernel h, it's important to have the sum of its coefficients equal to 1, because the input pixels are floats between 0 and 1, and we don't want to modify the contrast.

Normalized kernel

We consider a kernel where are pixel are equal to $\frac{1}{N_h M_h}$. Exemple with a 3×3 kernel :

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

The main benefits of this class of kernel is that we can have one of any dimension (not necessarily square), and that is symmetric according to x-axis and y-axis (swapping coefficients is useless).

Each output pixel is the mean with the same weight of its kernel neighbors, so the result is effectively blurred (see Figure 3.5).

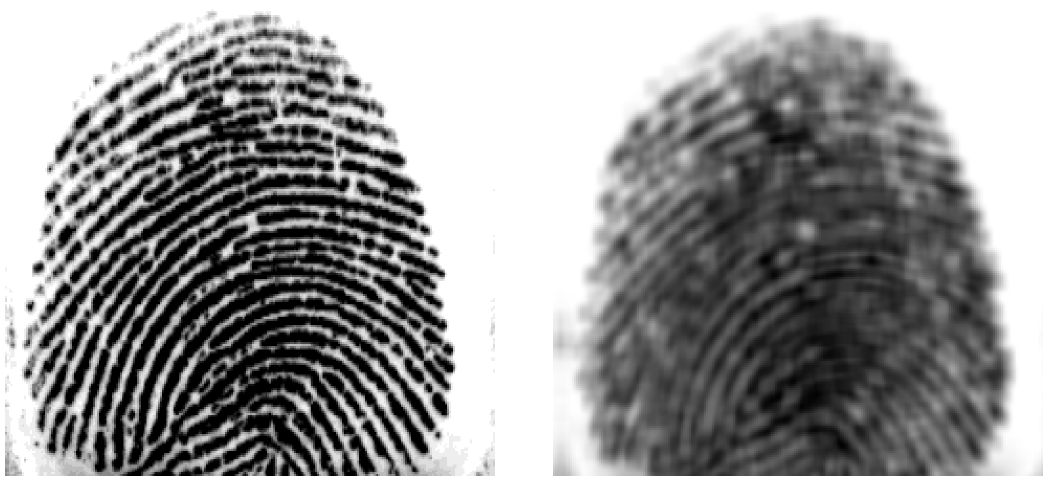


Figure 3.5: Input and Output imput with normalized kernel

Gaussian kernel

For a kernel $(2P+1) \times (2P+1)$, we fill the kernel with the formula :

$$h(i,j) = A.exp\left(\frac{-(i-P)^2}{2\sigma_x^2} + \frac{-(j-P)^2}{2\sigma_y^2}\right)$$

With A being the coefficient of normalization.

Here, the kernel is also symmetric according to both axis. Compared to a normalized kernel, it gives more importance to the nearest pixels. Most of physical phenomenons of blurring are modelled with a Gaussian blur. Moreover, it is possible to control the blur by varying deviations σ_x and σ_y.

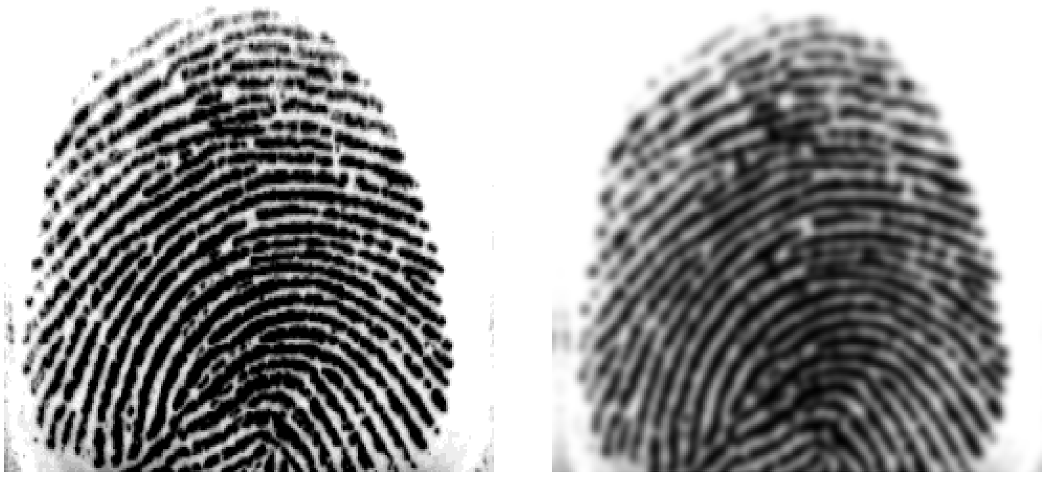


Figure 3.6: Input and Output imput with Gaussian kernel

How to choose the right kernel ? For the future, we will always choose a Gaussian kernel, but we still have to choose the parameters : the size of the kernel, and the horizontal and vertical deviations.

`For the dimension :`

It's clear that with a Gaussian kernel and a normalized kernel of the same size, the normalized one will always give a more blurred result ; the maximum of the Gaussian kernel is always on the center of the matrix, whereas all the pixels has the same weight with a normalized kernel. If $(\sigma_x, \sigma_y) \longrightarrow (+\infty, +\infty)$, the Gaussian kernel is a normalized kernel (all the coefficients are equals), and if $(\sigma_x, \sigma_y) \longrightarrow (0,0)$, we don't have any blurring effect, because all coefficients of the kernel are null except the central coefficient.

The blurring begins to be really pronounced with a 9×9 normalized kernel, so we choose the same dimensions for the Gaussian kernel. With $\sigma_x = \sigma_y = 15$, we can consider that the Gaussian kernel is a normalized one.

It can also be interesting to accentuate the blur only on one direction (see Figure 3.8 and 3.9)

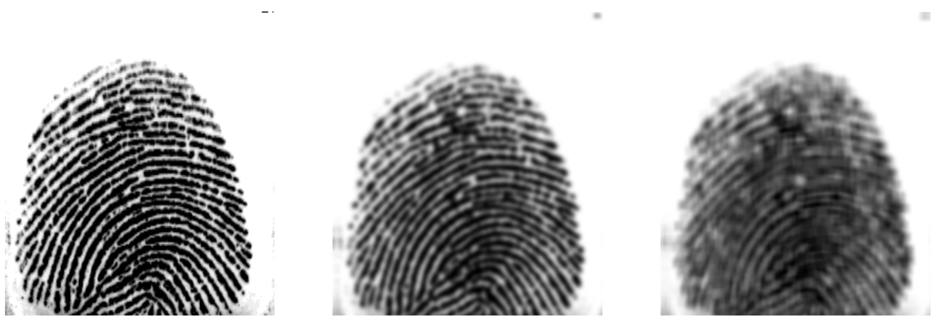


Figure 3.7: Input and two Outputs with $(\sigma_x, \sigma_y) = (2, 2)$ and $(\sigma_x, \sigma_y) = (15, 15)$

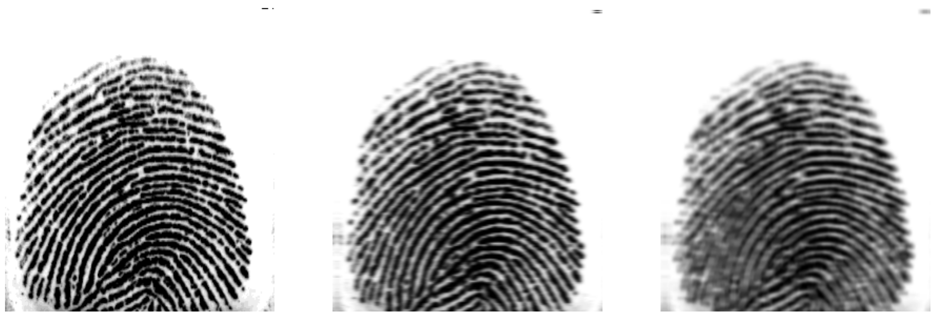


Figure 3.8: Input and two Outputs with $(\sigma_x, \sigma_y) = (2, 0.2)$ and $(\sigma_x, \sigma_y) = (15, 2)$

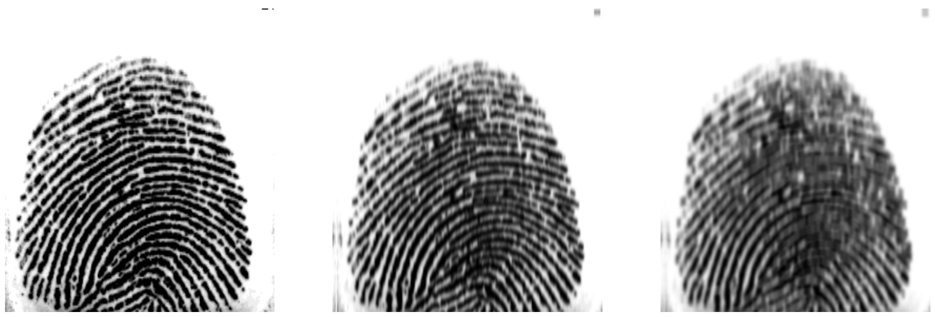


Figure 3.9: Input and two Outputs with $(\sigma_x, \sigma_y) = (0.2, 2)$ and $(\sigma_x, \sigma_y) = (2, 15)$

Justification of the approximation

Each coefficient of a Gaussian kernel is

$$G(x,y) = A.exp\left(\frac{-(x-P)^2}{2\sigma_x^2} + \frac{-(y-P)^2}{2\sigma_y^2}\right) = A.exp\left(-\frac{1}{2}\left(\frac{(x-P)^2}{\sigma_x^2} + \frac{(y-P)^2}{\sigma_y^2}\right)\right)$$

We recognize in the exponential the equation of an ellipse centered on the center of the matrix : we call "dispersion ellipse" of order k the ellipsoid $\frac{(x-P)^2}{\sigma_x^2} + \frac{(y-P)^2}{\sigma_y^2} = k^2$

For coefficients that verify $k = 1$, we have $G(x,y) = A.e^{-\frac{1}{2}} \simeq 0.60A$, which means that we have 0.6 times the value in the center of the matrix.

If we want to have coefficients on the corner of the matrix that belong to this ellipse, we want $\sigma_x = \sigma_y = \sqrt{2}P$. In practice, it is difficult to see the difference when $\sigma_x \geq P$, so we can approximate the Gaussian kernel with the normalized one when $\sigma_x \geq P$ and $\sigma_y \geq P$.

Modification of the kernel norm

Until now, we imposed as a necessary condition to have kernels which norm is equal to 1, in order to do the mean of the coefficient correctly. Let's now see what happens if $||h||_1 \geq 1$

To understand that, we can do an analogy with a mean of marks : the common way to compute the mean is to sum all marks, and then to divide by the number of marks. If we have 4 marks but we only divide by 3 the sum of them, the mean will be biased, and higher than expected. In this example, the norm of the vector is $\frac{1}{3} + \frac{1}{3} + \frac{1}{3} + \frac{1}{3} = \frac{4}{3}$.

Here, we can do the same, by multiplying the Gaussian normalized kernel by a constant greater than 1 : and as a result, we will obtain whiter pixels (because the intensity is higher), which can be an interesting effect because it can simulate a weak pressure (see Figure 3.10). Consequently, we can also obtain pixel with an intensity greater than 1 on output, but when the conversion into integer is done, the intensity is brought to 1.

We can also change the energy for vertical and horizontal blurring, to obtain other blurred image from the original one.

Figure 3.10: Outputs with a 11×11 Gaussian kernel where the dispersion goes from 0 to 5, and the energy from 1 to 6

3.5 With a varying kernel

If we look at blurred image, we observe that the blurred effect is more pronounced on the fingerprint surrondings. Thus, if we change the kernel for each pixel we apply convolution, we obtain a varying blurred depending on the coordinates.

To change the kernel for each pixels, it is intrinsically mandatory to use the naive implementation, that do the operation pixel by pixel. If we use DFT, we lose the spatial information, that's why it's impossible to use the FFT version.

Here, we have a problem similar to the "Pressure Variation" chapter, and so we can use the implementation done in this part to compute the pressure center and the ellipsoid that approximate the fingerprint. So, with theses functions, we can obtain the pressure center (x_0, y_0) and the two semi-axes (a_x, a_y) of the fingerprint.

How to change the kernel with the coordinates

To have a varying blur effect, the easiest way is to use a Gaussian kernel, and to make the two parameters σ_x and σ_y functions of the coordinates.

In fact, we want that : $(\sigma_x, \sigma_y) \to (0,0)$ when $(x,y) \to (x_0, y_0)$ and $(\sigma_x, \sigma_y) = (P,P)$ when (x,y) belongs to the ellipsoid of semi-axes (a_x, a_y). We can deduce a formula for(σ_x, σ_y) to verify those conditions :

$$\sigma_x(x,y) = \sigma_y(x,y) = P\left(\frac{(x-x_0)^2}{a_x^2} + \frac{(y-y_0)^2}{a_y^2}\right)$$

That way, we obtain kernels that are equal on each ellipsoid centered on the pressure center, and that maximize the blurring effect on boundaries.

The algorithm is the following :

Algorithm 4 Convolution with a varying kernel

```
Require: $x$ a matrix and $N_H$ and odd integer
  $P \leftarrow \frac{N_H-1}{2}$
  $BigX \leftarrow \text{Zeros}(N_X + 2P, M_X + 2P)$
  $BigX[P : N_X + P, P : M_X + P] \leftarrow x$
  $(x_0, y_0) \leftarrow \text{PressureCenter}(x)$
  $(a_x, a_y) \leftarrow \text{SemiAxes}(x)$
  for $i = 0$ to $N_X$ do
    for $j = 0$ to $M_X$ do
      $\sigma \leftarrow P\left(\frac{(i-x_0)^2}{a_x^2} + \frac{(j-y_0)^2}{a_y^2}\right)$
      $h \leftarrow \text{GaussianKernel}(N_H, \sigma_x = \sigma, \sigma_y = \sigma)$
      $y[i,j] \leftarrow BigX[P-i : P+i, P-j : P+j] \otimes h$
    end for
  end for
```

We can see the result of this algorithm on the Figure 3.11 : the blurred effect is more significative on the borders of the image.

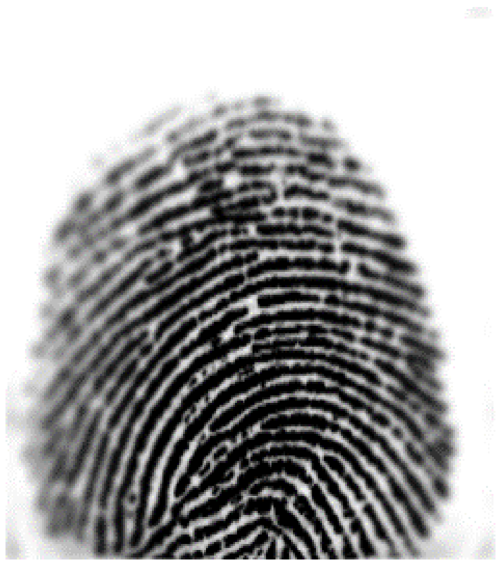


Figure 3.11: Output image of the Algorithm 4 with $N_H = 11$

It is possible to change the increasing of the blur by replacing $P\left(\frac{(x-x_0)^2}{a_x} + \frac{(y-y_0)^2}{a_y}\right)$ by $P\left(\frac{(x-x_0)^2}{a_x^2} + \frac{(y-y_0)^2}{a_y^2}\right)^r$, $r \in \mathbb{R}$, in order to raise faster or slowler the blur effect.

A similar way to vary the blur effect on the image is by varying differently σ_x and σ_y, in order to have a dependency of vertical blur (resp. horizontal) only in function of y (resp. x) coordinate :

$$\sigma_x(x) = P\frac{(x-x_0)^2}{a_x^2}, \sigma_y(y) = P\frac{(y-y_0)^2}{a_y^2}$$

This makes a blur effect different from the previous one (see Figure 3.12).

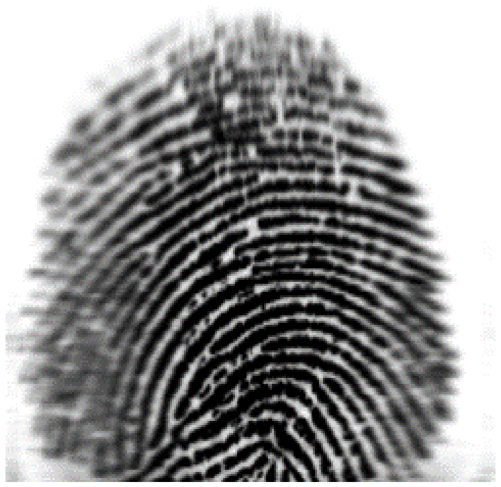


Figure 3.12: Output image of the Algorithm 4 with the separate dispersion and $N_H = 11$

We can also make a varying vertical blur or varying horizontal blur, by replacing the formula by $\sigma_x(x) = \sigma_y(x) = P\frac{(x-x_0)^2}{a_x^2}$ or $\sigma_x(y) = \sigma_y(y) = P\frac{(y-y_0)^2}{a_y^2}$. In the same way, every formula of this type can be replace by changing the power of the expression without P, to increase of decrease the speed of blur effect.

Varying kernel with varying energy

For every of these kernels, we can also vary their energy, and we can do this for every pixels depending of the coordinate, in order to have an energy equal to 1 on the pressure center (conservation of the energy), and an energy higher on the border, where the information is possibly lost : this bring us to the following formula, for each pixels of coordinate (x, y) of the input image :

$$||h||_1(x,y) = 1 + P\left(\frac{(x-x_0)^2}{a_x} + \frac{(y-y_0)^2}{a_y}\right)$$

.

If we combine this formula for $||h||_1$ with $\sigma_x(x) = P\frac{(x-x_0)^2}{a_x^2}$ and $\sigma_y(y) = P\frac{(y-y_0)^2}{a_y^2}$, we obtain the Figure 3.13 :

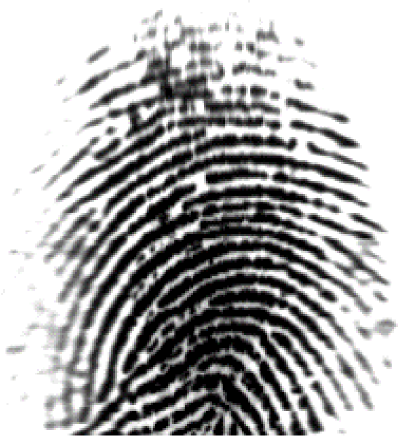


Figure 3.13: Output image of kernel with varying dispersions and energy, with $N_H = 11$

And as the formula for the norm is very similar to the formula for dispersion, every variations we propose could also be done here.

3.6 How to obtain the outputs with our code

All of the outputs are obtained in a similar way : compile and run a test from the folder "tests", having previously fix the parameters. To do this :

- Go to the "Digital-analysis" folder of the project
- Compile by executing the Compiler.sh script
- Run the file with the clean finger, by typing
  `./build/tests/test_name img/clean_finger.png`
- The result is stored in the "img" folder

Figure 3.5 : With `test_convol_fft.cpp`, setting the kernel line 36 with a normalized kernel :

```
Mat kernel = Normalized_kernel(11,11);
```

Figure 3.6 : With `test_convol_fft.cpp`, setting the kernel line 36 with a normalized kernel :

```
Mat kernel = Gaussian_kernel(11, 7, 7, 1);
```

Figure 3.7, 3.8 and 3.9 : Same as previously, changing the second and the third parameter of `kernel` (corresponding to σ_x and σ_y).

Figure 3.10 : We can do the same as previously, because the last parameter correspond to the energy. But there is also another file that create all the outputs : the file `text_convol_shifted.cpp` contain a loop that create 12 images with a progressive blur. Here, changing the kernel line 32 by `Mat kernel = Gaussian_kernel(11, ii, ii, ii+1);` allow to obtain the result.

Figure 3.11 and Figure 3.12 : With `test_convol_xy.cpp`, and the two outputs can be obtained by changing the line 235 :

`Mat H = Gaussian_kernel(size_h, sigma+sigma2, sigma+sigma2, 1)` for Figure 3.11,

and `Mat H = Gaussian_kernel(size_h, sigma, sigma2, 1)` for Figure 3.12.

Figure 3.13 : With `test_convol_xy.cpp`, changing

`Mat naive = Convol_Shifted_xy(image, 11)`

by `Mat naive = Convol_Shifted_xy_energy(image, 11)`.

3.7 Conclusion for blurring kernel

If we look at our results, we can be satisfied overall. We are precise on the simulation of the blur effect, and that makes the modelling accurate, that is to say not far from a possible output image that we can obtain.

But we also realize that we moved the problem to the choice of parameters : in fact, we are able to obtain a lot of different blurs, and the problem is now to choose the one we want. As long as we've been working on the project, we've accumulated parameters that we can't rigorously fix, since there isn't only one possible blurred image. In real life, many different blurred image can be encountered, because there are so many reasons for an acquisition to be blurred :

- The intensity of the blur
- How the blur depend of the coordinate (horizontal, vertical, both)
- How fast we have a blur effect when we move away from the pressure center
- The possible loss of intensity on the border
- etc.

All of theses are parameters that can change in function of the needs. With these degrees of freedom, we can construct an image bank of output images from one clean image, with different parameters. That way we could compare a potentially blurred acquisition to many simulations of possible blurs applied to a single clean image.

Chapter 4

Image rotation

4.1 Basic rotation

The rotation is a basic image transformation. One could believe that the implementation is also easy but it is not the case. First of all, we are going to define clearly what is a rotation from a mathematical point of view. The coordinates of a point (x, y) after rotation of angle θ centered on the point (tx, ty) are defined by:

$$x' = cos(\theta) * x - sin(\theta) * y + tx$$
$$y' = sin(\theta) * x + cos(\theta) * y + ty$$

The image being stored in matrices where each pixel is coded at the coordinate (i, j), these coordinates are integer. Once the rotation applied, we get a floating type coordinates. The intuitive solution is to take the floor of the value of each resulting coordinate. The result of this method is shown on Figure 4.1.

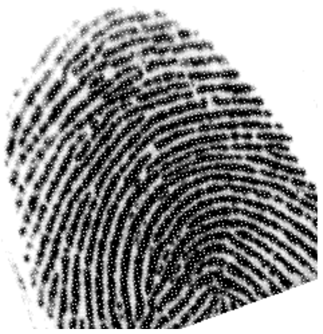


Figure 4.1: Basic rotation of an image

The result is disappointing, the processed image is full of white holes. The explanation is quite simple, when we take the floor of coordinates we lose a part of the information of the image. For example if we take two pixels whose the coordinates are after rotation (25.126, 45.515) and (25.785, 45.125), after taking the floor, they are coded at the same coordinate (25, 45). So we have a loss of information : two pixels went to the same sport, and that implies that there will be an empty sport in the image. The first idea consists on simply replacing the white pixel hole by the pixel intensity of their right neighbour. The second idea we had to solve the problem was to replace the white holes by the mean of the intensity of the neighbouring pixels. The result is shown on Figure 4.2 :

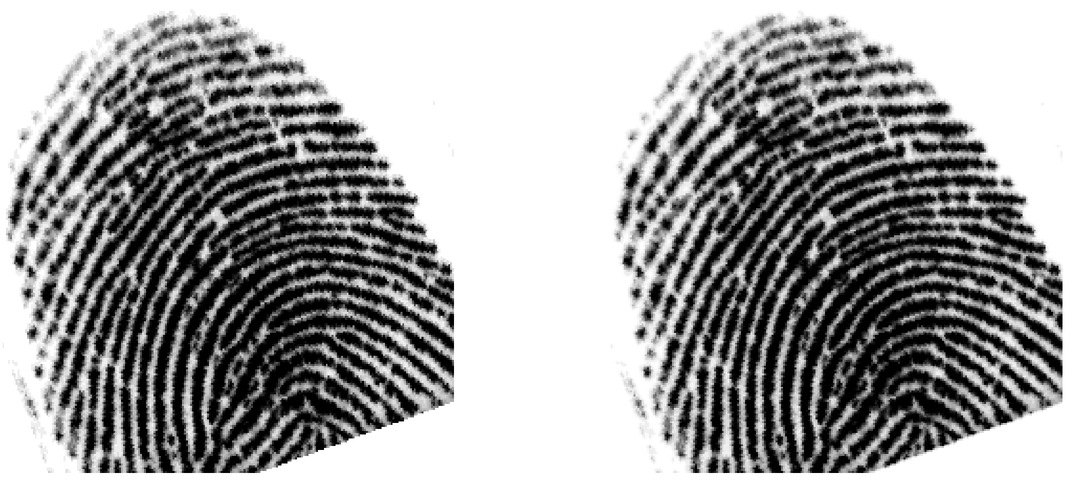


Figure 4.2: Replace by the right neighbour and replace by the mean of the four around neighbours

The result is obviously better but we can still observe the lack of accuracy of this method. Another method must be tried to decrease the loss of information.

4.2 Rotation from source

The principle is the same, we process for each pixel the rotation. Nevertheless, the method is different. Instead of taking the intensity of the pixel (i, j) and calculating the new coordinates (i',j') to set the new intensity. We take the opposite angle, and we calculate the new dimensions of the image. For each pixel of the new image, we compute the new coordinates (i',j'), we take the intensity of the source image at this coordinates and we put this intensity in the new image at the coordinate (i,j). This method does exactly the same rotation as the previous one. Yet there is a great benefit : thanks to this method, all the pixels are coded, without losing any information. We get the result shown on Figure 4.3.

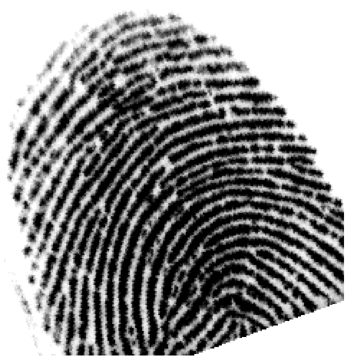


Figure 4.3: Rotation of an image from source

Some pixels still have the intensity of their neighbour, due to the fact that we still take the floor when doing the rotation. A way to improve the accuracy of the image is to do an interpolation. We will present three manners to compute this interpolation : bilinear, weighted and bicubic.

Weighted interpolation The weighted interpolation is the first method which came to our minds. The idea is quite simple : we calculate the distance between the real coordinate of a pixel after rotation (we don't take the floor) and the four neighbours which surround it.

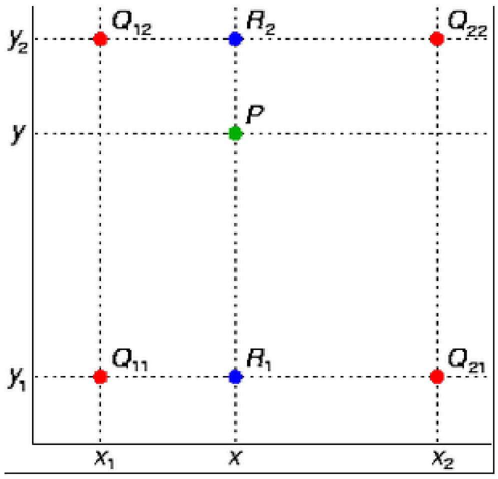


Figure 4.4: Pixel after rotation

On the Figure 4.4 the pixel after rotation is the point P and its four neighbours are the points Q_{12}, Q_{22}, Q_{11} and Q_{21}. The method consists in computing the four distances between P and the different Q points. Then, we take back the intensity of the four Q points and we make a mean of the four intensities weighted by the distance between P and each neighbour to get the intensity of P. So the mathematical formula is :

$$I_p = \frac{\frac{I_{11}}{D_{11}} + \frac{I_{12}}{D_{12}} + \frac{I_{21}}{D_{21}} + \frac{I_{22}}{D_{22}}}{\frac{1}{D_{11}} + \frac{1}{D_{12}} + \frac{1}{D_{21}} + \frac{1}{D_{22}}}$$

This idea has a better accuracy and the defections are invisible at naked eye. The result is given on Figure 4.5.

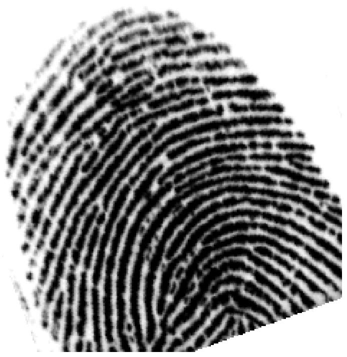


Figure 4.5: Rotation of the image with a weighted interpolation

Bilinear interpolation The bilinear interpolation is an extension of the linear interpolation. The idea is to do the interpolation in both directions. Here, the function that we want to interpolate is the intensity of the rotated pixel. As we can see on the Figure 4.3, we know the value of intensity at the points Q_{12}, Q_{22}, Q_{11} and Q_{21}, we just have to interpolate the intensity in both directions first along x then along y. So along x we have:

$$I(x, y_1) = \frac{x_2 - x}{x_2 - x_1} I_{11} + \frac{x - x_1}{x_2 - x_1} I_{21}$$

$$I(x, y_2) = \frac{x_2 - x}{x_2 - x_1} I_{12} + \frac{x - x_1}{x_2 - x_1} I_{22}$$

Then we apply the interpolation along y and we get:

$$I(x, y) = \frac{y_2 - y}{y_2 - y_1} f(x, y_1) + \frac{y - y_1}{y_2 - y_1} f(x, y_2)$$

We define in our code :

$$dx = \frac{x - x_1}{x_2 - x_1}$$

$$dy = \frac{y - y_1}{y_2 - y_1}$$

That's why we get the following equation :

$$I(x, y) = (1 - dx)(1 - dy)I_{11} + dx(1 - dy)I_{21} + dxdyI_{22} + (1 - dx)dyI_{12}$$

And the result on the rotation image is shown on Figure 4.6.

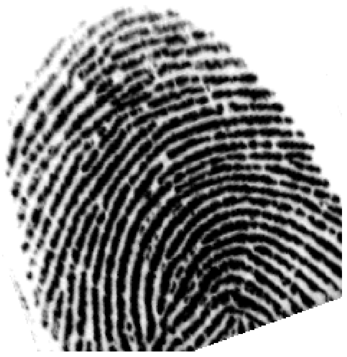


Figure 4.6: Rotation of the image with a bilinear interpolation

Bicubic interpolation The bicubic interpolation is the 2D version of the cubic interpolation. The interpolated surface is smoother when we use bilinear or weighted interpolation. Indeed, instead of using the 4 neighbouring pixels, we use for this interpolation the 16 neighbouring pixels of the interpolated point. The equation we have to solve is the following one :

$$I(x,y) = \sum_{i=0}^{3}\sum_{j=0}^{3} a_{ij}x^i y^j$$

So the first step is to find the 16 coefficients a_{ij}. We use not only the value of intensity at Q_{12}, Q_{22}, Q_{11} and Q_{21} but also the value of the derived intensities with respect to x (I_x), with respect to y (I_y) and with respect to both (I_{xy}).
Yet we have to use a numerical scheme to compute this derivative, due to the fact that the coordinate are integers. That's why we use :

$$I_x(x,y) = \frac{I(x_2,y_1) - I(x_1-1,y_1)}{x_2-(x_1-1)}$$
$$I_y(x,y) = \frac{I(x_1,y_2) - I(x_1,y_1-1)}{y_2-(y_1-1)}$$

$$I_{xy}(x,y) = \frac{I(x_2,y_2) - I(x_2,y_1-1) - I(x_1-1,y_2) + I(x_1-1,y1-1)}{(x_2-(x_1-1))(y_2-(y_1-1))}$$

Then finding the 16 coefficients all comes down to solving the following matricial equation:

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -3 & 3 & -2 & -1 \\ 2 & -2 & 1 & 1 \end{bmatrix} * I * \begin{bmatrix} 1 & 0 & -3 & 2 \\ 0 & 0 & 3 & -2 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & -1 & 1 \end{bmatrix}$$

with $I = \begin{bmatrix} I(x,y) & I(x,y+1) & I_y(x,y) & I_y(x,y+1) \\ I(x+1,y) & I(x+1,y+1) & I_y(x+1,y) & I_y(x+1,y+1) \\ I_x(x,y) & I_x(x,y+1) & I_{xy}(x,y) & I_{xy}(x,y+1) \\ I_x(x+1,y) & I_x(x+1,y+1) & I_{xy}(x+1,y) & I_{xy}(x+1,y+1) \end{bmatrix}$.

And finally we have :

$$I(x,y) = \begin{bmatrix} 1 & dx & dx^2 & dx^3 \end{bmatrix} \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} 1 \\ dy \\ dy^2 \\ dy^3 \end{bmatrix}$$

With this interpolation we got for the rotation the result shown on Figure 4.7.

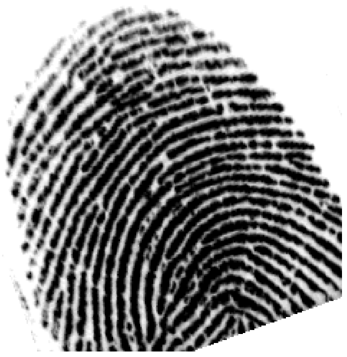


Figure 4.7: Rotation of the image with a bicubic interpolation

4.3 Performance of the different methods

Accuracy `Note: All the following results have been obtained with a previous version of our code. Since it was extremely hard to read, we then implemented the Rotation class. Without being able to trace the precision loss, our accuracies are more or less 0.5% lower with the actual version of the code.`
We can see that the different interpolations all have good results : the rotated image is well reconstituted. But between the different images, it is hard to tell what interpolation have the best accuracy. To answer this question, we have decided to compare our rotation results with the rotations made by OpenCV that we'll consider as our reference. The way to compare two rotated image is to sum the absolute difference for each pixel's intensity between our rotation and the rotation of OpenCV. We then divide by 255 and we multiply by 100 to get a percentage. Let M be the matrix of OpenCV and N our result matrix. We have the following formula:

$$accuracy = 100 - 100 * \frac{\sum_i \sum_j |m_{i,j} - n_{i,j}|}{255}$$

The results for the different interpolations are depicted on the graph 4.8.

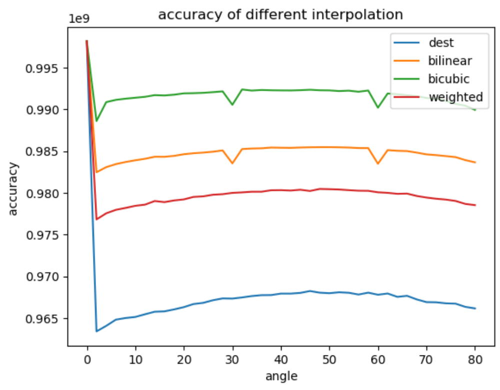


Figure 4.8: Accuracy of the different methods depending on the angle of the rotation

We may analyze the different results. First of all, we can see that the most accurate interpolation is the bicubic one, then the bilinear and the weighted one and finally the rotation to destination. Nevertheless, we may notice that all these methods have an accuracy over 95 percents, whatever the angle of the rotation.
Yet there are still some problems remaining. First, we cannot measure the accuracy for angles over 80 because of an issue with the function `Rect` of OpenCV. Indeed, we can make the rotation with an angle over 80, but we cannot take the rectangle with the same size as the initial image. Nevertheless, we did the same calculations for angle from 0 to -80 and we got the same results. Another problem is the fact that around 30 and 60, we face a loss of accuracy for the bilinear and bicubic interpolations. When we draw the rotated image for this angle and with these interpolations, there is a vertical dotted line and a horizontal dotted line which appear. That seems to be due to a non definition of dx and dy but we didn't find the source of the problem.

Complexity We have studied the accuracy of the different method but when we want to analyze some fingerprints we must make that quickly enough. Indeed, in an industrial context the complexity of an algorithm is very important. So we are going to analyze for each method its complexity theoretically and in the facts. Let's draw a comparison between the rotation from source and the rotation to destination. For the rotation to destination we have (with matrix of size (n,n) :

$$computations = n^2 * \begin{cases} 4 & +/- \\ 4 & sin/cos \end{cases}$$
$$= 8n^2$$

For the rotation from source we must run through the dimension of the result matrix that's to say: $n(cos(\theta) + sin(\theta))$, we may upper bound $(cos(\theta) + sin(\theta)$ by $\sqrt{2})$ the rest of the complexity is the same so we have for the rotation from source:

$$computations = \sqrt{2} * 8n^2$$

When we draw the complexity with the times we find these curves:

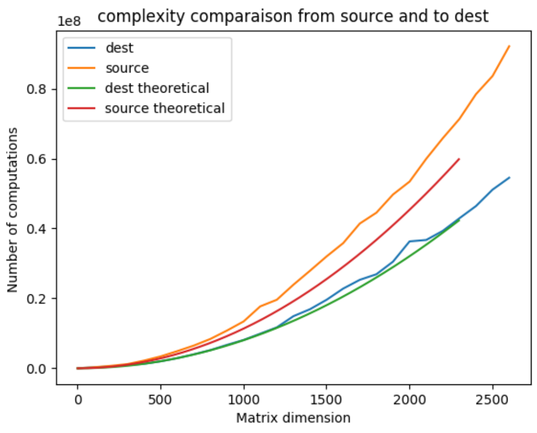


Figure 4.9: Compare complexity between from source and to destination method rotation

So our theoretical calculations was quite good. Then we have to try with the different interpolations. First we want to compare the rotation from source with the rotation from source with a weighted interpolation. For the weighted interpolation as we observe the four neighbouring pixels, we may think that the number of computation should be multiplied by 4. Yet, in practice the number of computation is not much greater than without weighted interpolation. Indeed we get:

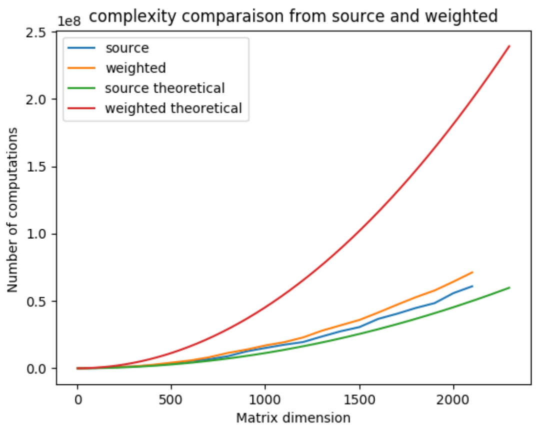


Figure 4.10: Compare complexity between from source and weighted interpolation rotation

And it is the same when we use the bilinear interpolation where we also use the four neighbouring pixels :

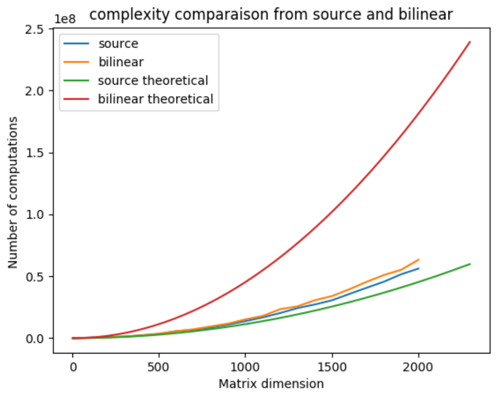


Figure 4.11: Compare complexity between from source and bilinear interpolation rotation

We don't know exactly why the complexity of these interpolations is so close to the original algorithm's. Nevertheless what is very important, is that with a few more computations, we have much better accuracy.

Finally, we can compute the complexity for the bicubic interpolation. For this interpolation, we use the 16 neighbouring pixels, theoretically we should multiply the complexity of the rotation from source by 16. Yet, with the two precedent curves, we saw that these modelling is not so good. If we divided by 2 the previous theoretical complexity result for the interpolation, we are closer to the practical result (even if it is not exact). That's why we have decided to take for the complexity of bicubic interpolation only the complexity of the algorithm from source multiply by $\frac{16}{2} = 8$. And we obtain the following result:

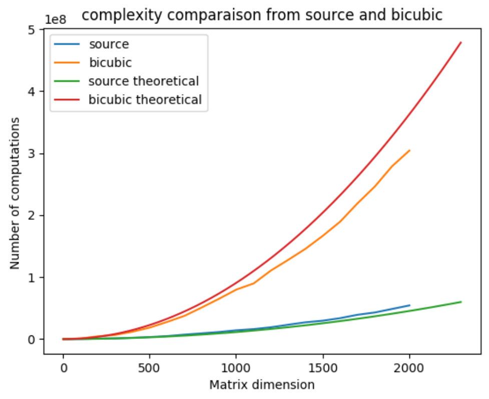


Figure 4.12: Compare complexity between from source and bicubic interpolation rotation

We may see that the complexity is theoretically better defined. Yet, the most interesting thing is that the complexity of the algorithm with the bicubic interpolation is much greater than the algorithm from source. Given that the accuracy of the bicubic interpolation is only 0.5 percent better than the bilinear interpolation but also 8 times slower, we can consider that the bilinear interpolation is the most efficient method for our fingerprint analysis.

4.4 Application for skin elasticity

All these methods are implemented to solve the problem of skin elasticity, indeed while we put our finger on the detector, because of the skin elasticity the fingerprints can rotate locally. That's why we want to apply an inverse rotation to correct this problem. To do so, we have to use a function which is going to decrease the angle in function of the distance of the center of rotation. We are going to use a Gaussian function defined by :

$$f(\theta) = \theta * e^{\frac{-d^2}{1000}}$$

where d is equal to :

$$d = \sqrt{(cx - x)^2 + (cy - y)^2}$$

where (x,y) is the coordinate of the current point and (cx,cy) is the coordinate of the center of rotation.

When we apply this function we obtain this:

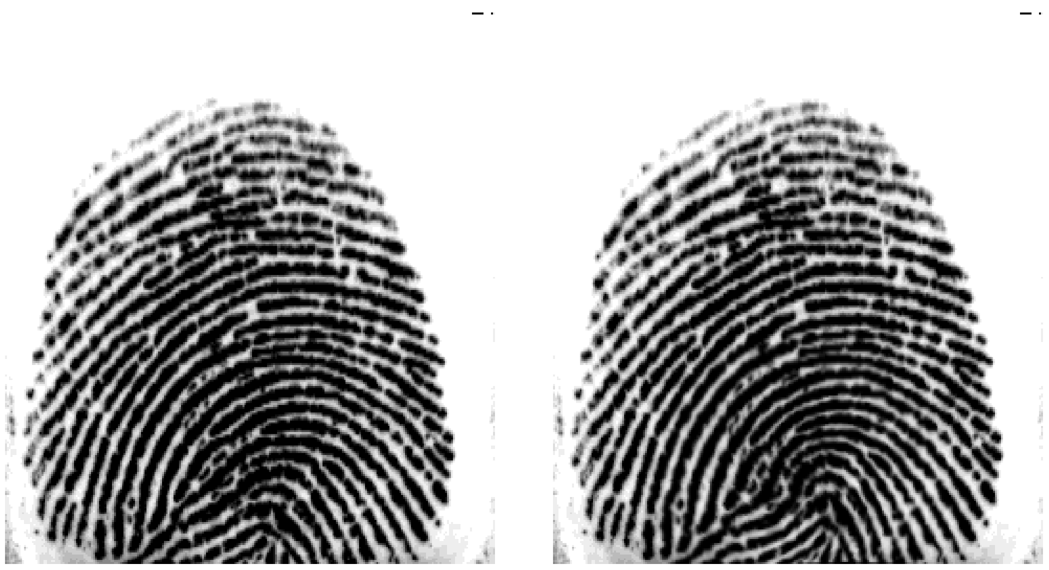


Figure 4.13: Original image and the locally image rotated

This output is obtained with a previous version of our code. The newly implemented Rotation class doesn't allow us to perform such a result. The previous code isn't in the actual code in our gitlab page but can be found in version anterior to the 15th of February.

4.5 How to obtain the outputs with our code

All of the outputs are obtained in a similar way : compile and run a test from the folder "tests", having previously fix the parameters. To do this :

- Go to the "Digital-analysis" folder of the project
- Compile by executing the Compiler.sh script
- Run the file with the clean finger, by typing
  `./build/tests/test_name img/clean_finger.png`
- The result is stored in the "img/image_rotation" folder

Figure 4.1, 4.2, 4.3, 4.5, 4.6 and 4.7 : With `test_rotation.cpp`

Figure 4.9, 4.10, 4.11 and 4.12 : We use `test_rotation_accuracy.cpp` and `test_complexity_rota` and the python file associated. Yet, that necessitates several more complex manipulations.

Figure 4.13 : With `test_elasticity.cpp`

Chapter 5

Conclusion

The purpose of the modelling activity was to work on fingerprint recognition, by reproducing specific cases (weak pressure, motion blur, ...) using mathematical tools. We focused on the issues of a weak pressure, motion blur, and motion warps.

Concerning the modelling of the weak pressure, we arrived to quite satisfying results even if there is still work to do. The code has to be optimized, and some adjustments have to be done, as described in the concluding part of the second chapter. The motion blur part has been done rigorously from a mathematical point of view, and the implementation was successful. The same statement could be done for the motion warps, even if we did not success in going further. However, we did not have time to work on reverse operations, which is in our sense an important step in an industrial context.

From a more general point of view, we are satisfied of the way we managed our project : our organization was correct and we respected the deadlines going deeper in several points. We also learned to provide a structured and documented code, which is a burning issue for industrial work. This was also our first opportunity to put into practice what we studied in classes and a first foretaste of what is R&D in a professional context.

Sources

Finding the pressure center :

- `https://www.pyimagesearch.com/2014/09/29/finding-brightest-spot-image-using-pyth`

Fingerprint recognition :

- **A high performance fingerprint liveness detection method based on quality related features**, *Fernando Alonso-Fernandez, Josef Bigun, Julian Fierrez, Hartwig Fronthaler, Klaus Kollreider, Javier Ortega-Garcia*

Thresholding method :

- **A threshold selection method from gray-level histograms**, *N. Otsu*, 1975, Automatica

Linear Filtering :

- `http://www.songho.ca/dsp/convolution/convolution.html`
- `http://xmcvs.free.fr/astroart/Chapitre4.pdf`
- `http://master-ivi.univ-lille1.fr/fichiers/Cours/seance7-convolution.pdf`
- `http://eeweb.poly.edu/~yao/EL5123/lecture4_2DFT.pdf`
- `http://people.rennes.inria.fr/Olivier.Sentieys/teach/filtragepourlesnuls.pdf`
- A lot of pages from `wikipedia.org` !

Morphological Filtering :

- `https://fr.wikipedia.org/wiki/Morphologie_math%C3%A9matique`

- **Digital Image Processing**, *Rafael C. Gonzalez, Richard E. Woods*, 2008, Pearson
- *Jean Serra, Luc Vincent*, **An overview of morphological filtering**, **Circuits, Systems and Signal Processing**, Vol. 11, No. 1, 1992

Rotation:

- `http://polymathprogrammer.com/2008/10/06/image-rotation-with-bilinear-interpolation/`
- `https://en.wikipedia.org/wiki/Bicubic_interpolation`
- `https://en.wikipedia.org/wiki/Bilinear_interpolation`