***Modeling activity*: Digital analysis of fingerprints.**

Nicolas CAVREL

Yassine FAKIHANI

Ben SARFATI

January-2019

Digital analysis of fingerprints

Ben Sarfati, Yasine Fakihani and Nicolas Cavrel

March 2, 2019

1 Intro

Every person has a very unique fingerprint. It is then a good way to identify someone. It happens for instance at the police station or on your smartphone. Let's say we have a database on a large amount of fingerprint, and then we get a new fingerprint, how to be sure that this fingerprint belongs to someone ? You also have to find a way to be sure that this specific fingerprint does not belong to anyone.
This is where we need some tools in image processing. Indeed, to go from one fingerprint (which are now consider as image) to another, and mostly because every images are registered in a different way, we need to compute some mathematical process on the new fingerprint to be able to say if this fingerprint belongs to the database or not.
We will then present those tools and the methods we implements to get a comparison as good as possible.

2 Basic Image Manipultion

2.1 Image saving, Pixel manipulation, Symetry

In this project, we'll deal with :
- Black and white image : intensity range from 0 (white) to 255 (black) for every pixels.
- Finite numbers of pixels : height versus width of the image.
- Matrix to represent the intensity values for every pixels.

We started using RGB image, but it was more complicated especially when we were dealing with loops. So, we assumed that all images we will process are gray scale.
For instance, to get the max or the min of intensity we computed two loops in order to check every pixel intensity and to return the max and the min. So, we coded the methods "return_max_intensity" and "return_min_intensity" of the class "image".

In this example it is easier to deal with gray scale. In deed, the loops have to be three times bigger each if we are dealing with RGB images instead of gray scale.

Moreover, The Open source computer vision (OpenCV) will be used in this project as an image processing library, thanks to the huge community using it and the great implementation in C++. We use also the Eigen library to deal with problems related with linear algebra.

Now we know all the classes and the libraries we will use to do this project. To get a fresh start with these libraries specially OpenCV, we implemented some functions like the ones to make a black and white square in a specific area of an image.

Also, in a way to manipulate the images, we started by implementing a method to compute the axis symmetries.

As a result for the (Oy) symmetry and the diagonal symmetry we get: (on the first figure)

Figure 1: It represent respectively the original weak finger image, the (Oy) symmetry and the diagonal symmetry

In this way (for instance regarding the (Ox) axis), we first must extract the pixel matrix of the original image and get the dimensions.
Then, using a new empty matrix we add pixel after pixel in the same order regarding the x-coordinates but decreasingly starting from the width for the y-coordinates.
the mathematical relationship being: $(x', y') = (x, width - y)$

2.2 Intensity Balancing

The user always applies a different pressure on the device. Mathematically speaking it means, every image has a different intensity. Speaking more pre-

cisely about fingerprint and in a way to compare two of them, there is a huge need of balancing the intensity. To understand what will be explain, it is important to have in mind that we will always start the process with images made by a strong pressure. It corresponds to a deeper gray. Later in this report, we will present a way to solve this problem using a decreasing function.

Let the function

$$c : \Re \to [0, 1]$$

which satisfies :

$$f(0) = 1$$

and

$$lim_{r\to\infty} c(r) = 0$$

We can first think of $r \to e^{-r}$ which has this two properties. But we also got the polynomials functions of r composed with the exponential: every function of the form $r \to e^{-r^n}$ is also valid.
But we also got others kind of functions, for instance: $r \to \frac{k}{k+r^n}$ with $n \in \aleph$ and $k \in \Re^*$ can also be used. With the additional value of having an extra setting parameter k.
We could even think about some stranger function like $r \to \frac{2}{\pi} arctan(r) + 1$ which have a different way to tend to zero.
For the function applying the c function to the pixels we used as parameters two variables of class *Coordinates* describing a rectangle on which the function will be applied. We also give it a threshold value that determines if the function should be applied (we apply it on pixel intensity higher than the threshold): this increases the contrast between black and white pixels. The function computes the coordinates of the center spot by itself. The corresponding function in the code is balance_intensity_exp for the exponential function and image::balance_intensity_quadratic for the quadratic one.

We can see that without any filtering with the threshold (actually the case where it is equal to 255, so the first one), the image to much altered by the c function. But with values around 150, it is possible to enhance the black and white contrast and to regularize the pixels intensity.
The two functions have a few differences, for instance the exponential balancing is "softer" than the one with the inverse function. But both can be used in different case.
Despite the fact that the filtering is working in some case, we can also see that the function is creating some "block" zones where a whole part is filled with black. these blocks are located on the top and the bottom on the furthest spots from the center.

Figure 2: Applying the function $r \to e^{-r2}$ with several values of threshold : from left to right 255, 200, 150 then 100, 50, 0(original with application point)

Figure 3: Applying the function $r \to \frac{1}{1+r}$ with the same threshold values.

2.3 Model improvements

What we have done might be consider as "good enough". It already balance the intensity. But we were keep thinking about it and then we realized that the projection of a finger on a sheet of paper looks more precisely to an ellipse than a circle. The function we used is called isotropic, which means it goes in all the directions in the same way. We had to consider that it is no longer isotropic but anisotropic which means that it is directionally dependent.

The c function can now decrease faster along the x axis than along the y axis. We can see that our problem is not anisotropic because if it was, the fingerprint would be round. And it is more of an ellipse.

We will now apply our c function as an ellipse grows. But in order to do that, we need to compute the two parameters of our ellipse : the (x_0, y_0) centered ellipse equation is $\frac{(x-x_0)^2}{a^2} + \frac{(y-y_0)^2}{b^2} = 1$
We can evaluate these parameters by measuring the distance of the furthest point from the center and the one of the closest point :

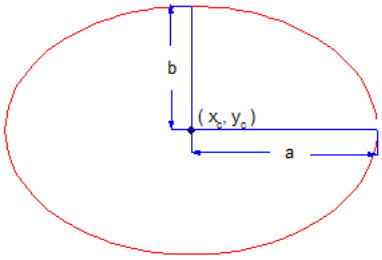

Figure 4: The two ellipse parameters, a and b

To compute it of our images, we first need to determine the pixels that are on the border of the fingerprint and then to compute their distances from the center point of the fingerprint. The border is returned by the function image::contour() by finding the first pixel on the left and the first pixel on the right of each line being smaller than a given threshold value. Here is the result :

Figure 5: weak_finger with its border pixels highlighted and its center point.

Computing the furthest and closest border point from the center we can now print the resulting ellipse by turning black the pixels verifying the equation :

$\frac{(x-x_0)^2}{a^2} + \frac{(y-y_0)^2}{b^2} = 1$, (x_0, y_0) being the coordinates of the center.

Figure 6: weak_finger with its ellipse and its center.

And the last step is to use the same a and b coefficients in the $c(x, y)$ function in order to make it grow with the ellipse shape. We will apply it to the previous $x \to e^{-r}$ function :

$$\Re^2 \to [0, 1]$$
$$(x, y) \to e^{-(\frac{(x-x_0)^2}{a^2} + \frac{(y-y)^2}{b^2})}$$

Let's apply this to the pixels intensity of weak_finger :

Figure 7: Anisotropic $c(x, y)$ function with the threshold values from left to right : 255, 200, 150, 100, 50, 0 (original with center).

Compared with the previous function which was isotropic, we can see that the top and bottom areas got much less "block zones" that we had before and

yet we still got a good enhancement of the image contrast.

Which shows that we successfully improved our model using an anisotropic function.

Another way to improve our model is to improve the ellipse approximating our fingerprint. Previously we were only computing the border pixels and then taking the closest and the furthest one to the center. And thus their distance to the center were defining our major and minor axis parameters.
But as we can see in the figure of weak_finger and its ellipse, the approximation isn't perfect and the minor axis parameters is too small...

We will now consider the error function defined as :

$$L(a,b) = \sum_{k=0}^{len(border)} (\frac{(i_k - x_0)^2}{a^2} + \frac{(j_k - x_0)^2}{b^2} - 1)^2$$

Where *border* is the vector of border points we computed before. The use of such an error function is justified by the fact that the ellipse equation is the following :

$$\frac{(x - x_0)^2}{a^2} + \frac{(y - y_0)^2}{b^2} = 1$$

So the closer $\frac{(i_k - x_0)^2}{a^2} + \frac{(j_k - x_0)^2}{b^2}$ is from 1, the smaller the error function is and the better is our ellipse estimation.

In order to simplify this expression, let's call $len(border)$ N. And our goal will be to minimize this error function finding the best (a, b) couple value.

In order to do that, we ill apply the gradient descent algorithm. This algorithm takes as paramaters a starting point x_0 and an accuracy threshold ϵ. And the recurrence formula is the following :

$$Computation\ of\ ||\nabla f(x_i)||$$

$$if\ ||\nabla f(x_i)|| > \epsilon\ :\ x_{i+1} = x_i - \alpha_i \frac{\nabla f(x_i)}{||\nabla f(x_i)||}$$

$$else\ :\ return\ x_i$$

In our case,

$$\nabla f(x_i) = \nabla L(a,b) = -\frac{4}{a^3} \begin{pmatrix} \sum_{k=0}^{N} (\frac{(i_k - x_0)^2}{a^2} + \frac{(j_k - x_0)^2}{b^2} - 1)(i_k - x_0)^2 \\ \sum_{k=0}^{N} (\frac{(i_k - x_0)^2}{a^2} + \frac{(j_k - x_0)^2}{b^2} - 1)(j_k - y_0)^2 \end{pmatrix}$$

The main issue with this algorithm is that even a good starting isn't that hard to find (for instance a solution "close" to the best one : the one we computed previously), finding an accuracy threshold which guaranties a good result is pretty hard. There are also the step coefficients α_i that are left to determine. Ideally, we would like decreasing step coefficients.

Let's start considering $\forall i \in \aleph, \alpha_i = \alpha \in \Re$, then the step is constant. We

can also test our algorithm by defining a finite number of iteration rather than a stop condition. And after 10000 iteration with a constant step of 0.01, we got this result :

Figure 8: Fingerprints with their improved ellipse and centers.

requires the partials derivative of L with regards to a and b :

$$\frac{\partial L}{\partial a}(a,b) = -\frac{4}{a^3}\sum_{k=0}^{N}(\frac{(i_k - x_0)^2}{a^2} + \frac{(j_k - x_0)^2}{b^2} - 1)(i_k - x_0)^2$$

and

$$\frac{\partial L}{\partial b}(a,b) = -\frac{4}{b^3}\sum_{k=0}^{N}(\frac{(i_k - x_0)^2}{a^2} + \frac{(j_k - x_0)^2}{b^2} - 1)(j_k - y_0)^2$$

By definition of the derivative, the vector

$$(\frac{\partial L}{\partial a}(a,b), \frac{\partial L}{\partial b}(a,b))$$

gives us the direction in which the L function is increasing. So the vector $-(\frac{\partial L}{\partial a}(a,b), \frac{\partial L}{\partial b}(a,b))$ gives the direction in which L is decreasing.

3 Image Comparison:

In order to compare two image, we have to be able to tell if one is a translation or a rotation of the other. And so, we need to be able to perform such a rotation or a translation on any image. We will see here some methods to do that :

3.1 Rotation and translation :

In Geometry, a translation is a map and a geometric transformation that moves every point of a figure by the same distance in a given direction. And a rotation is also a map, which concretely means a circular movement with a given angle of something around a central point that stays fixed.

We can define these motions models by some mathematical functions in order to apply them to image processing:

- Translation : The equation of a translation is:

$$T_{(h,k)}(x, y) = (x + h, y + k)$$

- Rotation: Let ω be this mathematical function; ω take as parameters the pixel coordinates (x,y), the center of rotation C(a,b) and the angle of rotation θ. Its prototype can be described like this:

$$\omega(x, y; a, b; \theta) = (x_0, y_0) = (a+(x-a)cos(\theta)-(y-b)sin(\theta), b+(x-a)sin(\theta)+(y-b)cos(\theta))$$

We coded in the class "image" a method to do the rotation -without any interpolation-. Its prototype is: "void rotation(Coordinates,float)". So, let's see an example of such a rotation of an image around its barycenter (using the method barycenter() of class image) and with an angle $\frac{\pi}{4}$:
As we can see; the result (which is the picture in the right) contains a lot of noise after rotation and an overall decrease in contrast.

In order to minimize losses in image quality, we are going to do some interpolation.

3.2 Interpolation

3.2.1 Bilinear interpolation:

The bilinear interpolation is an extension of linear interpolation for interpolating functions in two dimensions.
The goal is to find the value of the function f at the point (x,y), assuming that we know it's value at the four points $(x_1, y_1), (x_1, y_2), (x_2, y_1) and (x_2, y_2)$.
In image processing we consider that f(x,y) is the pixel intensity of the image at the position (x,y).
To do that, we write f under the form: f(x,y) = ax + by + cxy + d, such that a, b, c and d are constants that that must be determined from the four neighbors $(x_1, y_1), (x_1, y_2), (x_2, y_1) and (x_2, y_2)$, by solving the following linear system of 4 equations and 4 unknowns:

$$\begin{cases} f(x_1, y_1) = ax_1 + by_1 + cx_1y_1 + d \\ f(x_2, y_1) = ax_2 + by_1 + cx_2y_1 + d \\ f(x_1, y_2) = ax_1 + by_2 + cx_1y_2 + d \\ f(x_2, y_2) = ax_2 + by_2 + cx_2y_2 + d \end{cases}$$

A change of variable will be useful to solve this system, let's consider so the following variables: $dx = x–x_1$ and $dy = y–y_1$.

The new bilinear interpolation function is then written: $f(dx, dy) = adx + bdy + cdxdy + d$.
By introducing the notations: $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$, the matrix to be reversed becomes: $M = \begin{bmatrix} 0 & 0 & 0 & 1 \\ \Delta x & 0 & 0 & 1 \\ 0 & \Delta y & 0 & 1 \\ \Delta x & \Delta y & \Delta x \Delta y & 1 \end{bmatrix}$
It remains to introduce the following notations:

$$\Delta f_x = f(x_2, y_1) - f(x_1, y_1), \Delta f_y = f(x_1, y_2) - f(x_1, y_1), \Delta f_{xy} = f(x_1, y_1) + f(x, y_2) - f(x_2, y_1) - f(x_1, y_2)$$

The solution of this problem then comes directly:

$$f(dx, dy) = \Delta f_x \frac{dx}{\Delta x} + \Delta f_y \frac{dy}{\Delta y} + \Delta f_{xy} \frac{dxdy}{\Delta x \Delta y} + f(x_1, y_1)$$

We coded a method of the class "image" named "rotation_interpolation_bilinear" to do that.
If we apply this method to do a rotation of an image around its barycenter (using the method barycenter() of class image) and with an angle $\frac{\pi}{4}$ we find the following result:

Figure 9: A rotation with bilinear interpolation

As we can see the result is much better!
Concerning the complexity of this method: For an image of size $n*n$ the number of operation is of the order of $\Theta(n^2)$.

3.2.2 Bicubic interpolation :

For the bicubic interpolation we suppose that the function f and its derivatives f_x, f_y, f_{xy} are known at the four points (0,0), (1,0), (0,1) and (1,1), so the interpolated surface can be written as following:

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

So, the problem here is to find the value of the 16 coefficients a_{ij} using these 16 equations:

$$\begin{cases} f(0,0) = a_{00} \\ f(1,0) = a_{00} + a_{10} + a_{20} + a_{30} \\ f(0,1) = a_{00} + a_{01} + a_{02} + a_{03} \\ f(1,1) = \sum_{i,j=0}^{3} a_{ij} \\ f_x(0,0) = a_{10} \\ f_x(1,0) = a_{10} + 2a_{20} + 3a_{30} \\ f_x(0,1) = a_{10} + a_{11} + a_{12} + a_{13} \\ f_x(1,1) = \sum_{i,j=0}^{3} a_{ij} i \\ f_y(0,0) = a_{01} \\ f_y(1,0) = a_{01} + a_{11} + a_{21} + a_{31} \\ f_y(0,1) = a_{01} + 2a_{02} + 3a_{03} \\ f_y(1,1) = \sum_{i,j=0}^{3} a_{ij} j \\ f_{xy}(0,0) = a_{11} \\ f_{xy}(1,0) = a_{11} + 2a_{21} + 3a_{31} \\ f_{xy}(0,1) = a_{11} + 2a_{12} + 3a_{13} \\ f_{xy}(1,1) = \sum_{i,j=0}^{3} a_{ij} ij \end{cases}$$

After some calculation, we found that the solution of this system is:

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

In order to find derivatives, we used finite differences.
We computed the bicubic interpolation in the class "image" using an independent function named "interpolation_bicubique(Matrix4d,Coordinates)". Let's show an example of its execution:

Figure 10: A rotation with bicubic interpolation

As we can see, the result is better than the bilinear interpolation.
Finally, the complexity of the bicubic interpolation is of the order of $\Theta(n^2)$ for an image of size $n*n$. But it takes a lot of time to be executed than the bilinear interpolation.

3.2.3 To do better:

After computing rotation or translation, we used to lost a lot of information (mostly at the corners of the images).

For instance, if we compute a rotation of angle π and then of angle $-\pi$, we should get the same image as the original one.
However, we get the image on the 11^{th} figure.

Figure 11: Example of the information's lost after the computation of a rotation and a rotation back

This is why we had the idea of implementing a method using a bigger image, (it's a method of the class "image" named "bigger_image") :
During the computation of the rotation or the translation, we'll now work with an image nine times bigger than the original one. The original one being at the center of the new one as we can see on the 12^{th} figure.

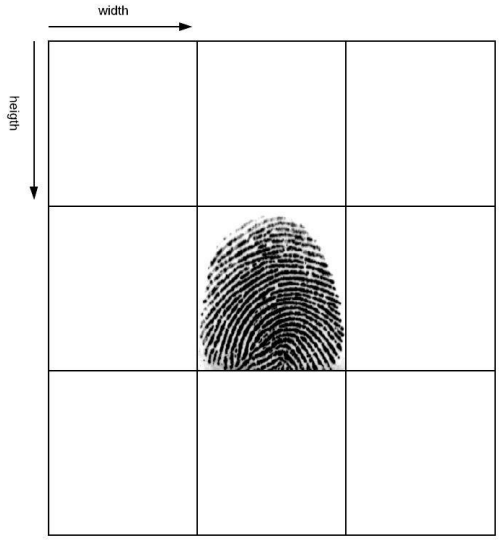

Figure 12: Bigger image in a way to save information

After all the computations, another method ((it's a method of the class "image" named "smaller_image")) short the image and return an image with

dimensions equals to the original one.
For the same example of the two rotations, we get this image :

Figure 13: The information is now protected

It is a good way to save information and we'll now be able to always work on the same image. Computing all kind of rotations and get back to the image without the looses of information because of the image sizes.

3.3 Recognizing a geometrical Warp

3.3.1 Loss Functions

In this part we will focus on ways to find if an image is the rotation and/or the translation of another one.
In order to do that, we will first need what we call a loss function. This kind of function models the similarities between two images : for instance if the to images are exactly the same, the loss function applied to this two images will be null. And the greater the value of the loss function, and the further away the two images are from each other.
So how to define such a function ? The first idea could be to take the absolute difference of the pixels intensities of the two images. So we would have a loss function such as :

$$L(f_1, f_2) = \sum_{i=1}^{height} \sum_{j=1}^{width} |f_1(i,j) - f_2(i,j)|$$

Where $f_1(i,j)$ represents the intensity of the pixel at position (i,j).
In our case we will prefer a quadratic function rather than an absolute difference

one, in order to have make very different pixels more important in the loss function. So the first function we will try is the following one :

$$L_1(f_1, f_2) = \sum_{i=1}^{height} \sum_{j=1}^{width} (f_1(i,j) - f_2(i,j))^2$$

But in the end, the most convenient function to estimate the error between to image would be a variance kind function defined by :

$$L_2(f_1, f_2) = \frac{(\sum_{i,j} f_1(i,j) - \overline{f_1})(\sum_{i,j} f_2(i,j) - \overline{f_2})}{\sqrt{(\sum_{i,j} f_1(i,j) - \overline{f_1})^2 (\sum_{x,y} f_2(i,j) - \overline{f_2})^2}}$$

Technically this function have to be maximized, but in order to apply the same algorithm for both loss function L_1 and L_2 we will use :

$$L_2(f_1, f_2) = -\frac{(\sum_{i,j} f_1(i,j) - \overline{f_1})(\sum_{i,j} f_2(i,j) - \overline{f_2})}{\sqrt{(\sum_{i,j} f_1(i,j) - \overline{f_1})^2 (\sum_{x,y} f_2(i,j) - \overline{f_2})^2}}$$

3.3.2 First naive algorithm

Once the loss function defined, the goal will be to minimize it by modifying one of the two images : let's say we want to know if f_2 is a warped image of f_1, then we will modify f_1 by rotating and translating it, until the loss function is minimized. The last step is to return the translation and rotation parameters found.
So the pseudo-code of the algorithm would be :

Algorithm 1 Naive algorithm

We got an image *image* to compare with another image *image_compare*
error $\leftarrow$ 1000000 (Initialised to an arbitrary big value)
rep $\leftarrow (0,0,0)$
for r $\in [0, nb\_test\_rot]$ **to** **do**Rotate the image *image* of $\frac{2r\pi}{nb_test_rot}$
for k $\in [-nb\_testx, nb\_testx]$ **to** **for** **to** **do***do*l $\in [-nb\_testy, nb\_testy]$
Translate the image *image* of $\frac{k*height}{nb\_testx}$ along Ox
Translate the image *image* of $\frac{l*width}{nb\_testy}$ along Oy
if loss_function(image,image_compare) ¡ error **then**
rep $\leftarrow (\frac{k*height}{nb\_testx}, \frac{l*width}{nb\_testy}, \frac{2r\pi}{nb_test_rot})$
error $\leftarrow loss\_function(image, image\_compare)$
Reset *image* to the initial image
return rep =0

This algorithm is working fine, and by fine I mean that the results are correct. But the main issue is the computation time. For a standard image of more 10000 pixels, it can takes several minutes of tens of minutes.

3.3.3 Optimized small square algorithm

So instead of trying to fit the whole image over the other one, we will now take a small square of the initial image, and try to fit this square on the other one.

This will effectively reduce the computation time as the loss function will have much less pixels to go through, and we will no longer need to translate the images.

In addition, this second method is also more realistic : in the reality we can imagine having only parts of a fingerprint, and being able to fit parts of one onto the other could sufficient to match the two fingerprints.

We would usually take a small square around the barycenter of the image in a way to ensure that the square is inside the fingerprint. So for instance for weak_finger :

Figure 14: Weak Finger and a small square of 50 pixels taken at its barycenter

So in this case the pseudo-code becomes :

This new algorithm is still improvable but compared to the previous one returns results much more accurate at fixed running time. Here's for instance an example of result with only 10 rotation test (nb_test_rot = 10) :

4 Image warping

4.1 Warping issues

This part of the project is focused on the modeling of a geometrical warp through two functions : the warping_rotation and the warping_translation functions. As their names suggest, we will deal with two kind of geometrical warping, the ones due to a rotation and the ones due to a translation of the finger.
These functions use a bilinear interpolation in the indirect way. Let's explain the algorithm :
We create an empty image of the same size of the image we wanna warp. For

Algorithm 2 Optimized algorithm

We got an image *image* to compare with another image *image_compare*
Extract a small square *small_square* of size *square_size* around the barycenter of image_compare
error $\leftarrow$ 0 (Initialised to an arbitrary big value : here all values are negatives)
rep $\leftarrow (0, 0, 0)$
for r $\in [0, nb\_test\_rot]$ **to** **do**Rotate the image *image* of $\frac{2r\pi}{nb_test_rot}$
    for k $\in [square\_size, height - square\_size]$ **to** **for to do***do*l $\in [square\_size, width - square\_size]$
        Extract a small square *tmp_small_square* of size *square_size* at position (k, l) on *image*
        if loss_function(small_square,tmp_small_square) ¡ error **then**
            rep $\leftarrow (\frac{k*height}{nb\_testx}, \frac{l*width}{nb\_testy}, \frac{2r\pi}{nb_test_rot})$
        error $\leftarrow loss\_function(image, image\_compare)$
        Reset *image* to the initial image
        return rep

Figure 15: The fingerprint on the left is the original image, the one on the middle is the one to compare with, and the one on the right is the image generated by the Optimized algorithm

each pixel of the empty image, we compute its antecedent pixel in the initial image. This antecedent pixel doesn't have integer values for coordinates, so we have to estimate its values with the interpolation. The bilinear interpolation goal is to link 4 points of a two dimensional space with a quadratic form. Which is the same as solving this equations system :

$$\begin{cases} f(x_1, y_1) = ax_1 + by_1 + cx_1y_1 + d \\ f(x_1, y_2) = ax_1 + by_2 + cx_1y_2 + d \\ f(x_2, y_1) = ax_2 + by_1 + cx_2y_1 + d \\ f(x_2, y_2) = ax_2 + by_2 + cx_2y_2 + d \end{cases} \tag{1}$$

The bilinear interpolation only depends on the four pixels values surrounding our "fictive" non integer pixel. Let's see what the result is for four surrounding pixels of values 255,127,127 and 0 :

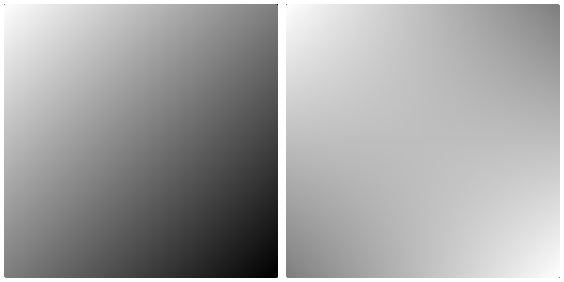

Figure 16: Interpolation result with corners pixels set as 255, 127, 127 and 0 (Left figure) and 255,127,127,255 (right figure)

4.2 different warping

Once we know how to interpolate between the four points, we can set the color of an in between pixel as the interpolate value :

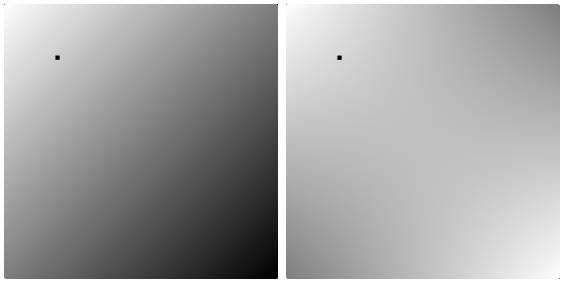

Figure 17: Result of the interpolation with a pixel highlighted in the top left, in this case the interpolation color would be white.

With the interpolation up and running the only thing remaining to do is to transform our pixel grid into another and take the interpolated value of the resulting grid :

The formulas used here are the following :

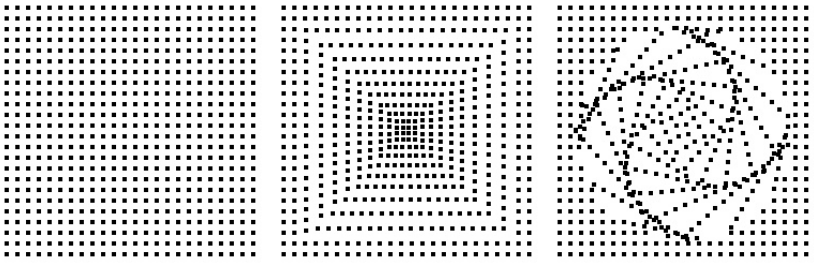

Figure 18: From left to right : Initial grid, Translation Warping grid, Rotation Warping grid

- Translation Warping :
$TW(x, y, \alpha_x, \alpha_y, r) = (x + \delta_x(x, y, \alpha_x), y + \delta_y(x, y, \alpha_y))$
Where $\delta_x(x, y, \alpha_x) = (x - x_0)(\frac{-\alpha_x}{r}||(x, y) - (x_0, y_0)||_\infty + \alpha_x)$ with (x_0, y_0) the application point (the point that will remain invariant by this function), α_x and α_y the strength of the warping with regards to x and y, and r the radius of the square on which we want to apply the function.

- Rotation Warping :
$RW(x, y, \alpha, r) = (x_0+(x-x_0)cos(-\theta(x, y, \alpha, r))-(y-y_0)sin(-\theta(x, y, \alpha, r)), y_0+(x - x_0)sin(-\theta(x, y, \alpha, r)) + (y - y_0)cos(-\theta(x, y, \alpha, r)))$

Where $\theta(x, y, \alpha, r) = \frac{-2\pi\alpha}{r}||(x, y) - (x_0, y_0)||_2 + 2\pi\alpha$ with α the strength of the rotation and r the radius on which the function should be applied.

And here's the result when we apply these functions to a real image :

Figure 19: From left to right : Initial image, Translation warped image, Rotation warped image.

4.3 Back to fingerprint

Now let's "squeeze" clean_finger.jpg in order to obtain its warped version :

We can see that we are able to compress the image especially in the center area. We are now able to manipulate our images to improve the fingerprint acquisition !

Figure 20: The original version of clean_finger and its compressed one.