Communauté
UNIVERSITÉ Grenoble Alpes

Digital Analysis of fingerprints

ROMAN BREDEHOFT - SAMUEL DUCROS
JACOPO IOLLO - FABIEN POURRE
March 9, 2020

Contents

Abstract

The goal of our project is to find mathematical filters and models which best simulate artifacts that could occur during a real fingerprint acquisition : rotation, strong and weak pressure, moist and dry skin, skin warp and motion blur.

We used several mathematical approaches and adapted them with specific input parameters in order to best match qualitatively the expected images : some basic geometrical image processing formulas, anisotropic functions, separable filters, convolution, Fourier Transform, morphological operations, the gradient descent method and more.

We also implemented an algorithm that finds from an transformed input image what transformations it has been subjected to by finding its normal acquisition. Different techniques are described since the more naive one are very inefficient. We in fact tried to optimize most of our implementations since time complexity is often a real issue.

Our results are very acceptable, qualitatively and quantitatively, even though we always can improve many aspects, especially in terms of complexity. Some extension also could have been thought of to make some results a bit more realistic but what we got is fine to a certain extend.

Introduction

The goal of our project is to find mathematical filters and models which best simulate artifacts that could occur during a real fingerprint acquisition. It can be a rotation, a strong or weak pressure and many more. Here is an image (**Figure 1**) the summarizes the ones that we took care of.

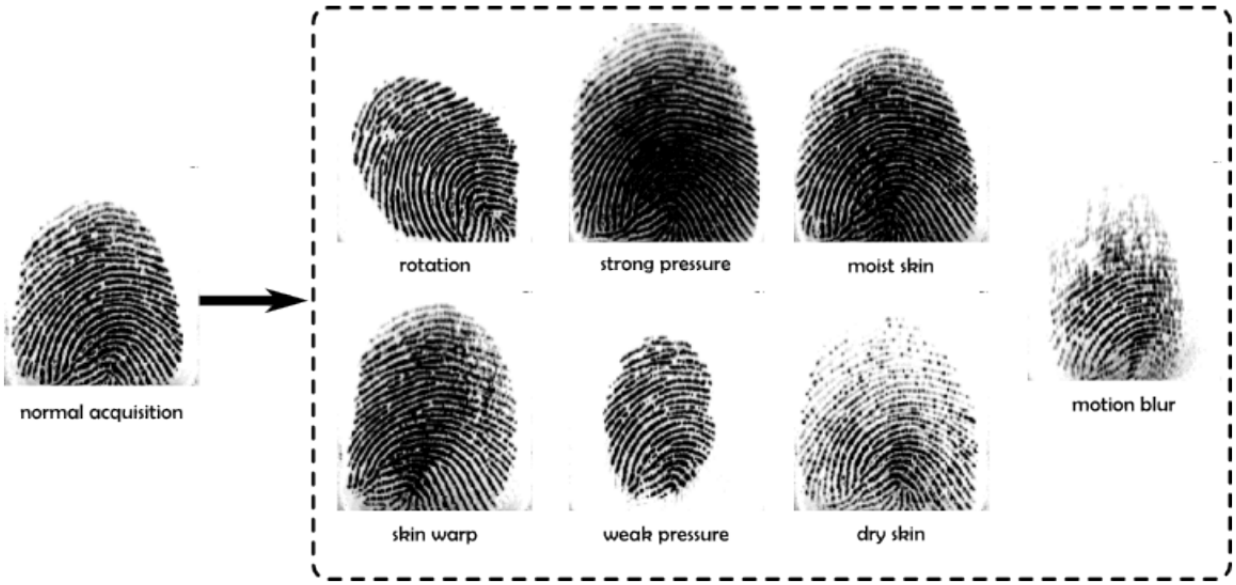

Figure 1: Several physical artifacts that we want to simulate

Such work in image recognition can have multiple applications in certain scientific disciplines, especially since it could be generalized to any black and white images. In our case, for example, the idea would then be to use our results in fingerprints recognition algorithms in order to better their efficiency. It could help data scientists to properly create data bases by better categorizing several types of fingerprints before training their Artificial Intelligence algorithm on it. In fact, of the main problem in Data Science is that we cannot always control the data itself that easily and the the idyllic situation would be having only ”clean data”, where pictures have all been taken the same way for example. Being able to reproduce natural artifacts and then compare them to real images before simulate its original acquisition is then a real strength when it is done correctly.

Nonetheless, an algorithm’s usefulness not only depends on the results but also a lot on its effectiveness. In fact, trying to optimize computations in time, by using different mathematical approaches in order to reduce the time complexity, is one of the most important aspect of computer science. If an image recognition software is slow then its

applications can be pretty limited and that would make the whole work not that interesting. Consequently, in the report, we attempted to deliver the fastest implementation we could find and complete while keeping results that seemed acceptable.

In fact, for some of the filters, the most difficult part was to find the right parameters that would really simulate the natural artifacts quite realistically. In this manner, the following pages not only explains the theory around the project but also how to adapt it in practice in order to obtain images that looked like the ones in **Figure 1** through several analysis of our solutions.

1 Image Loading, Saving and Pixels Manipulation

1.1 Conventions

For the project, we only consider grayscale images with the format `png`. The pixels intensity values are then integers between 0 and 255. For the calculus, we map these value into floats between 0 and 1, and we finally map these values back in $[0, 255]$ to save and display the images. We chose to use a frame as in **Figure 2**, with O the top left pixel of coordinates $(0, 0)$.

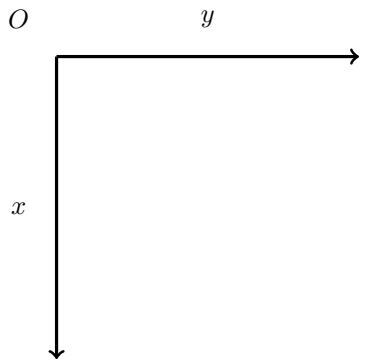

Figure 2: Chosen frame for images: the top left pixel is the origin

We will focus on the basis of image manipulation in C++ using the `OpenCV` library. We chose this library because some of us were already used to it but also because this is a well-known optimized library for image processing.

1.2 The class `Image`

We built a class `Image` which contains 3 attributes: a matrix `im` (Opencv basic image type `Mat`) which contains the values of the pixels of the image loaded and 2 unsigned integers, `height` and `width`, which correspond to the height and the width of the image in terms of the number of pixels. We built several basic methods we can use at every moment:

- 3 different constructors which respectively take into account the file name to load the image; a matrix that represents an image; or 2 unsigned integers which correspond to the height and the width of a new image,
- a copy constructor,
- `void display()` which displays the image in a new window,
- `void save(const string& name)` which saves the image in the file named `name` into a folder named `results`,
- `void map_to_norm()` and `void map_to_base()` which respectively map all the pixels from integers between 0 and 255 to floats in the interval $[0, 1]$ and from floats between 0 and 1 to integers between 0 and 255. In C++, integers between 0 and 255 are represented by the type `uchar`, which corresponds to a single byte. **We have to be cautious here because in the interval $[0, 255]$, the whiter pixel corresponds to the value 255 and the blacker one to 0. In the interval $[0, 1]$, it is the opposite, the whiter pixel corresponds to the value 0 and the blacker one to the value 1.** For all this project, we will only consider cases where the pixel type is `float`, by converting images into that type.

We created the method `Image rectangle(int xa, int xb, int ya, int yb, uchar color)` which copies the image and changes the pixels' values inside a rectangle defined by the top left point $A = (x_a, y_a)$ and the bottom right point $B = (x_b, y_b)$. This rectangle is filled with a color `color`, an integer between 0 and 255. In **Figure 3**, we have the original image with two of these rectangles.

1.3 The symmetry transform

Here, we want to apply some symmetry transforms at the image. First, we want to apply a symmetry transform along the height axis. Knowing the image's height and width, the symmetry function $s(x, y)$ of the image $f(x, y)$ is :

$$s(x, y) = f(x, width - y)$$

Figure 3: White and black rectangles

Figure 4: Original image

Figure 5: Symmetry of the image along the height

We then obtain **Figure 5**:

We now want to apply a symmetry transform along the image's diagonal, the one that verifies y=x as in **Figure 6**. The symmetry function is then $t(x, y) = f(y, x)$, as shown in **Figure 8**.

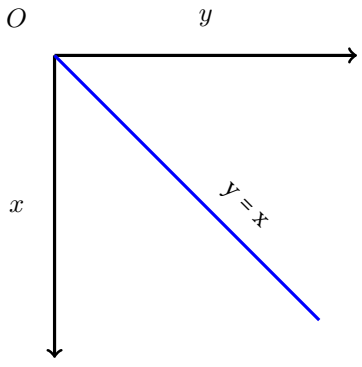

Figure 6: Chosen frame for images: the top left pixel is the origin

The matrix which represents the symmetry along the diagonal axis is: $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ and its determinant is -1. As a rotation can be written $\begin{bmatrix} \cos(\theta) & \text{-}\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$ with a determinant equal to 1, even if we apply the transposition to the matrix, this pixel operation is not a rotation.

1.4 Weak pressure simulation

Here we want to simulate a fingerprint done with a weaker pressure than the normal acquisition. The goal is to apply a coefficient function which decreases the intensity of the pixels anisotropically as they get far from the pressure spot.

Figure 7: Original image

Figure 8: Symmetry of the image along the diagonal y=x

First, we assume that the coefficient function is isotropic. We can then define $c(r)$ where r is the euclidean distance from the center of the pressure spot and the pixel. This function has to be equal to 1 at $r = 0$ and to monotonically decrease as $r \mapsto \infty$ to 0. We suggest, $\forall a, b, p \in \mathbb{R}_*^+$:

- $c(r) = e^{-ar^p}$
- $c(r) = \dfrac{1}{ar^p + 1}$
- $c(r) = \dfrac{1 + e^{-b}}{1 + e^{a(r-b)^p}}$

Before implementing these functions, we first have to find the pressure spot's coordinates. We define this spot as the barycenter of all the "black pixels", pixels with intensity superior to a specific threshold. After a few testing with different values, we arbitrarily set this threshold to 0.6, since all values are in the range $[0, 1]$, with 1 being the blacker pixel. We then find the barycenter shown in **Figure 9** thanks to the white rectangle.

Figure 9: Pressure spot of the fingerprint

We implemented the preceding functions and obtained **Figure 10**, **Figure 11** and **Figure 12**.

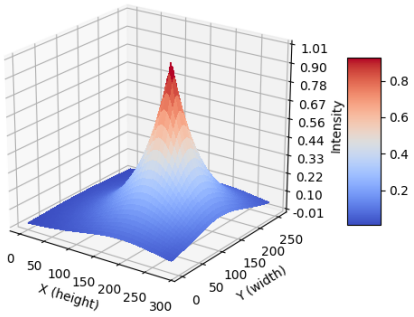

Figure 10: $r \mapsto e^{-\frac{r}{50}}$

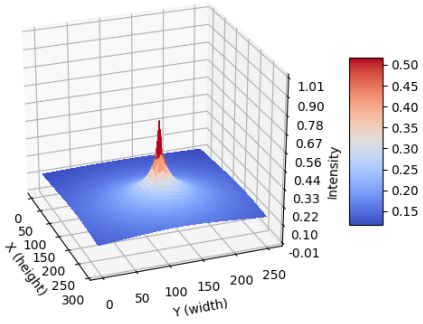

Figure 11: $r \mapsto \frac{1}{0.2r+1}$

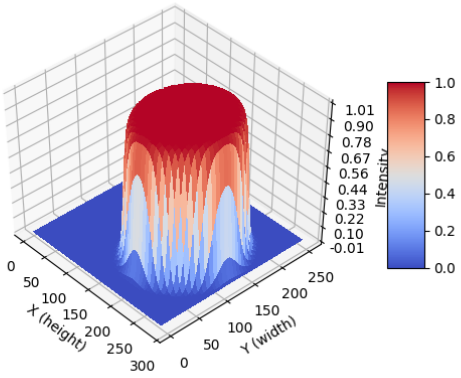

Figure 12: $r \mapsto \frac{e^{-85}}{1+e^{0.4*r-85}}$

We then applied these functions to the clean finger image and got **Figure 13**, **Figure 14** and **Figure 15**.

Figure 13: $r \mapsto e^{-\frac{r}{50}}$ applied to the clean finger

Figure 14: $r \mapsto \frac{1}{0.2r+1}$

Figure 15: $r \mapsto \frac{e^{-85}}{1+e^{0.4*r-85}}$

All of our functions are giving circular shapes while the expected image `weak_finger.png`, provided in the subject, always has an elliptical one.
By observing this image at issue, we noticed that the pixel intensity transform seems anisotropic. Anisotropy is the property of being directionally dependent, which implies different properties when directions differ. Since our fingers have some kind of elliptical shape, we assumed that this property must be considered.

We then decided to adapt the last function used, $c(r) = \dfrac{e^{-85}}{1 + e^{0.2*r-85}}$, so that it becomes anisotropic. This choice was only made because the **Figure 15** outcome seems the most natural one.

To take this property into account, we first write the equation of an ellipse with the image's barycentre $P = (x_p, y_p)$, or as we called it, the pressure spot, and r, the distance from one pixel at the position $(x, y))$ from P, as followed : $a, b \in \mathbb{R}^*$: $\forall x, y \in \mathbb{R}, r^2 = \left(\dfrac{x - x_p}{a}\right)^2 + \left(\dfrac{y - y_p}{b}\right)^2$. Wore conveniently, we will use $r = \sqrt{\left(\dfrac{x - x_p}{a}\right)^2 + \left(\dfrac{y - y_p}{b}\right)^2}$ in the chosen function. That way, with a, b well chosen, the image returned will be an ellipse and not a circle.
We took $a = \sqrt{2}$ and $b = \dfrac{5}{6}$ so that the resulting image corresponds to the expected one, `weak_finger.png`,

Now we need to choose a direction, since we saw before that a anisotropic function depends on that. The default direction would be a null angle $\alpha = 0$ and gives an horizontal ellipse pointing to the right as in **Figure 16** and in **Figure 17**:

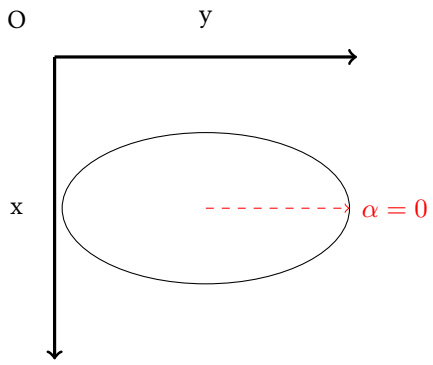

Figure 16: Horizontal ellipse for $\alpha = 0$

Figure 17: $r \mapsto c(r)$ with the angle $\alpha = 0$

Thus, we may have to rotate the ellipse with an angle $\alpha \in [0, 2\pi]$. Let a point $M = (x_m, y_m)$ at distance r_m from the pressure spot $P = (x_p, y_p)$, there must be a $\theta \in [0, 2\pi]$ such that the polar coordinates of M around the spot pressure P are $x_m - x_p = -r_m \cos(\theta)$ and $y_m - y_p = r_m \sin(\theta)$, as in **Figure 18** and **Figure 19**.

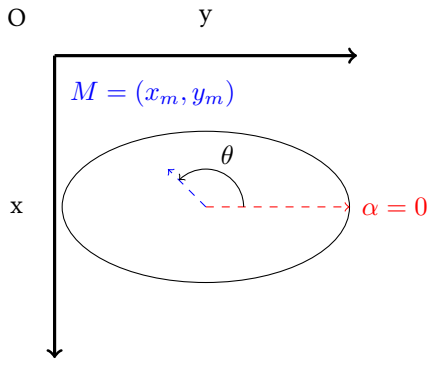

Figure 18: Polar coordinates with $\alpha = 0$

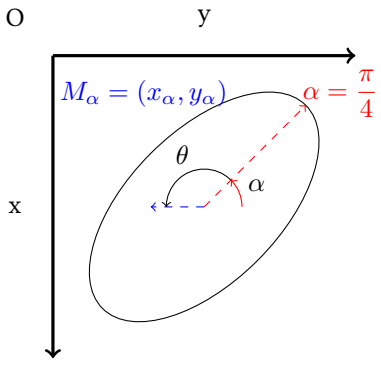

Figure 19: Polar coordinates with $\alpha = 0$

We then apply a rotation of an angle α to this point, giving a new one, $M_\alpha = (x_\alpha, y_\alpha)$, thanks to **Equation 1** and **Equation 2**.

$$\begin{aligned} x_\alpha - x_p &= -r_m \sin(\theta + \alpha) \\ &= -r_m \left(\sin(\theta)\cos(\alpha) + \sin(\alpha)\cos(\theta)\right) \\ &= (x_m - x_p)\cos(\alpha) - (y_m - y_p)\sin(\alpha) \\ \implies x_\alpha &= (x_m - x_p)\cos(\alpha) - (y_m - y_p)\sin(\alpha) + x_p \end{aligned} \tag{1}$$

$$\begin{aligned} y_\alpha - y_p &= r_m \cos(\theta + \alpha) \\ &= r_m \left(\cos(\theta)\cos(\alpha) - \sin(\theta)\sin(\alpha)\right) \\ &= (y_m - y_p)\cos(\alpha) + (x_m - x_p)\sin(\alpha) \\ \implies y_\alpha &= (y_m - y_p)\cos(\alpha) + (x_m - x_p)\sin(\alpha) + y_p \end{aligned} \tag{2}$$

We thus have M_α a point on which to apply the coefficient function and $r_{M_\alpha} = \sqrt{\left(\frac{x_\alpha - x_p}{\sqrt{2}}\right)^2 + \left(6 * \frac{y_\alpha - y_p}{5}\right)^2}$. Therefore, for each pixel $M = (i, j)$ and $\alpha \in [0, 2\pi]$, we apply the coefficient function to $M_\alpha = (i_\alpha, j_\alpha)$ with $i_\alpha = (i - x_p)\cos(\alpha) - (j - y_p)\sin(\alpha) + x_p$ and $j_\alpha = (j - y_p)\cos(\alpha) + (i - x_p)\sin(\alpha) + y_p$, with $r_{i,j,\alpha} = \sqrt{\left(\frac{i_\alpha - x_p}{\sqrt{2}}\right)^2 + \left(6 * \frac{j_\alpha - y_p}{5}\right)^2}$

In order to obtain a nice result, we also need to filter the pixels on which we apply the coefficient function. Indeed, the fingerprint is cleaner with a higher contrast as the pressure is weaker. To do so, we decided to apply the function to pixels with intensity higher than 0.5. The other pixels' intensity is lowered to 0.

Finally, the function we apply to the image at the pixel (i, j) with intensity $I(i, j)$ is:

$$c_{anisotropic}(i, j, \alpha) = \begin{cases} 0 & \text{if } I(i,j) < 0.5 \\ c(r_{i,j,\alpha}) & \text{else} \end{cases}$$

In the expected image `weak_pressure.png`, the direction of the ellipse seems to be a bit less than $\frac{\pi}{2}$. We take an angle $\alpha = \frac{\pi}{2} - 0.15$ in **Figure 21**. The result looks like the original acquisition of the weak pressure fingerprint, but we can notice that the actual acquisition is not a absolute ellipse but contains some variations at bounds. Our function which clears the whitest pixels make the result fingerprint closer to the acquisition. In conclusion, this method gives a result close to the reality, without considering enough bounds variations.

Figure 20: Weak pressure in practice

Figure 21: $r \mapsto \frac{e^{-85}}{1 + e^{0.2*r-85}}$ with the angle $\alpha = \frac{\pi}{2} - 0.15$

2 Geometrical Warps

In this section we will first focus on the relationship between the images in **Figure 22**.

Figure 22

2.1 Introduction

This geometrical operation is a rotation of an angle θ around the image's center that we will consider here as the origin for our coordinates. This choice was made only to make things easier to understand; mathematically, this change of origin doesn't change much since it's only made thanks to a simple translation.

Acknowledging that, with the (x, y) the coordinates of a pixel's center, we can express the new pixel's center coordinates (x', y') after making a rotation of an angle $\theta \in [-\pi, \pi]$ as **Equation 5.5** :

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{3}$$

Rotating an image is not as easy as it may seems. Generally, the center coordinates (x', y') of a rotated pixel will not correspond to an actual pixel center of the rotated image. As shown in **Figure 23**, it usually overlaps four other pixels. So we can't just assign pixel values to the rotated image that easily.

Moreover, when we rotate an image, we need also to change the dimensions of the array that stores it if we don't want to cut off its angles (**Figure 24**). The new dimensions H and L can be obtained by doing some basic trigonometry: $H = l\sin\theta + h\cos\theta$ and $L = h\sin\theta + l\cos\theta$.

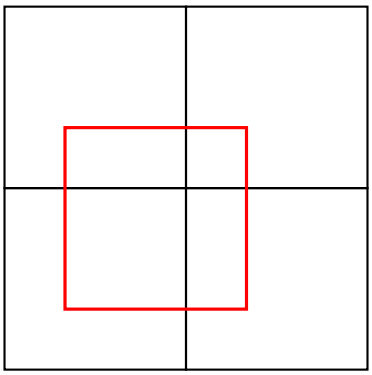

Figure 23: In red, an example of a rotated pixel overlapping four pixels of the rotated image

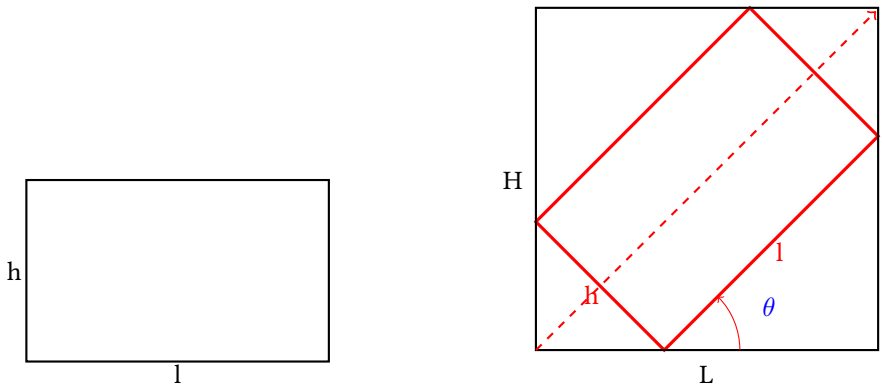

Figure 24: In dark, the window required to plot all the image with no rotation (left) and rotated by $\theta = \frac{\pi}{4}$ (right)

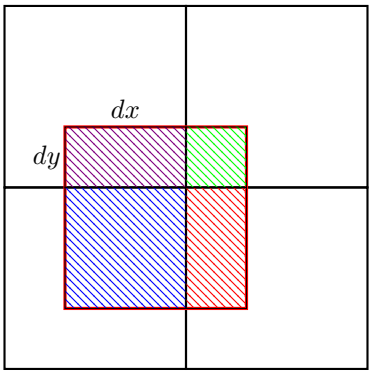

Figure 25: Four areas used to do the weighted average

2.2 Rotation by Area mapping

To overcome those problems, we implemented an algorithm called Rotation by Area Mapping [Blo]. The main idea is to iterate on the rotated image array, with its new dimensions. By doing the inverse rotation of a destination pixel, we find the it overlaps as in Figure 23. We then compute the weighted average of those source pixels with the proportion of area covered by our inverse rotated pixel (**Figure 25**) and assign this value to the destination pixel.
This method enables us to have strictly one value per destination pixel and has a complexity in $\mathcal{O}(nm)$, the rotated image's number of pixels.

2.3 Validation examples

To validate our code, we rotated the same images for different angles, with $|\theta| \leq \frac{\pi}{2}$ or $|\theta| \geq \frac{\pi}{2}$, positive or negative, as shown in **Figures 26, 27 and 28**.

To further validate our algorithm we did a rotation of θ and then a rotation of $-\theta$ on the obtained image. We recovered the same orientation of the image but on a different size window, as we can see in **Figure 29**.

The best parameter we found to mimic the expected image $warp1\_finger.png$ is $\theta = -\frac{\pi}{4}$, giving **Figure 30**.

2.4 Local warps

In this section we aim to replicate the skin elasticity property. The main idea is to do a local rotation a certain point. The more we move away from this specific point, the less pixels are subjected to this rotation. So our idea was to simulate a rotation matrix which would tend to the identity matrix when moving away from the rotation center. We use again some functions defined in the 1st part :

$$\theta(d, a, p) = \theta_0 e^{-(d/a)^p}$$

Figure 26: Rotations de $\frac{\pi}{4}$ et $-\frac{\pi}{4}$

Figure 27: Rotations de $\frac{3\pi}{4}$ et $-\frac{3\pi}{4}$

Figure 28: Rotations de $\frac{\pi}{6}$, $\frac{\pi}{8}$ et $\frac{5\pi}{6}$

With d the distance from the rotation's center, a representing the length were we would want $\theta \approx \theta_0$ and p the speed of $\theta \xrightarrow[d\to+\infty]{} 0$, we have that

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \xrightarrow[d\to+\infty]{} \mathcal{I}_2$$

We can reuse our rotation algorithm by substituting θ by $\theta(d, a, p)$. See Figure 31 for some random examples.

Now, in order to correctly approximate `Warp2_finger.png`, we need to find the right parameters.

Figure 29: Original Image, then a rotation of $\frac{\pi}{6}$ followed by $\frac{-\pi}{6}$ and the error betwen the original image and the inverse rotation

Figure 30: Warp1_finger and our approximation

Figure 31: Random wraps

Figure 32: Warp1_finger and our approximation

3 Linear filtering

3.1 Introduction

A convolution is a linear transformation widely used on images, taking advantage of the structural properties of the data. The following is a quick recap on the topic, for a further reading we would recommend [DV16], where most of our figures are taken from.
A discrete convolution of an image X of size (N, M) and an impulse kernel H of size (N_H, M_H) can be expressed as below, where image coefficients are taken null when they fall out of the image :

$$(X * H)[x, y] = \sum_j \sum_i X[x - i, y - j] H[i, j] \tag{4}$$

Convoluting an image with a kernel can be represented as in **Figure 33**. For each step, the product between a kernel pixel and the image pixel it overlap is calculated and the sum of them is the value assigned to the convolution in this position. Sometimes a kernel pixel fall out of the image, to prevent this from happening we can add 0 at the end and at the beginning of each axis. This is called 0-padding. The output shape of a convolution over an image is a function of

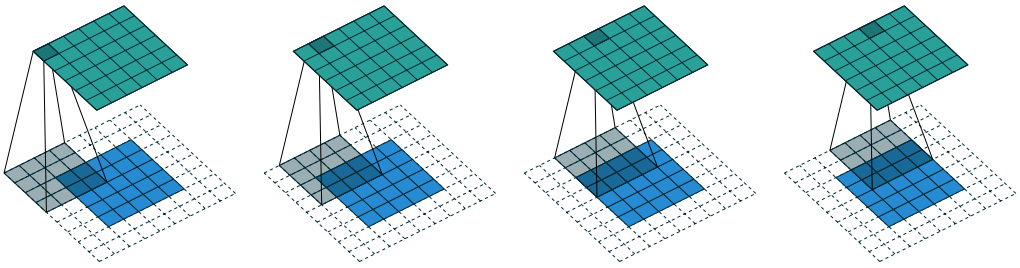

Figure 33: Convolution of a 4x4 kernel over a 5x5 image

the following parameters:

- k_j kernel size along axis j
- i_j input size along axis j
- s_j stride: step with which we move the kernel along axis j
- p_j zero padding: number of zeros added at the begging and at the end of an axis j

For example, in **Figure 33** we have a 4×4 kernel over a 5×5 input. We move the kernel one pixel at the time ($s = 1$) and we padded 2 zeros at the border ($p = 2$). In **Figure 34** we can see an example where $s = 2$.

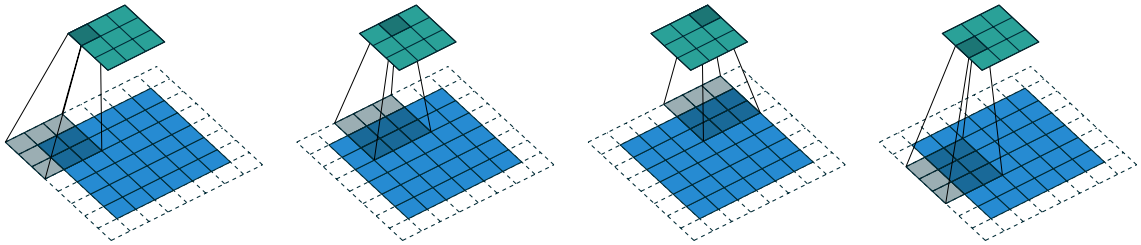

Figure 34: Convolution with $s = 2$

Output size From now on, we will focus only on convolutions with $s = 1$. In this case, the output dimension o on an axis can be obtained with:

$$o = (i - k) + 2p + 1$$

Because we don't want to change the image size, we need to define our padding parameter p such that $o = i$. This is why we will use mostly odd shaped filters, so we can define $p = \frac{k-1}{2}$.

3.2 Implementation

Convolution can be implemented in several ways. In the following, we will present a naive algorithm, a special algorithm in the specific case when the filter is separable and last but not least the convolution implemented with FFT.

3.2.1 Naive Algorithm

The naive Algorithm is the straight forward implementation of the formula. The only difficulty here is to implement the 0-padding in order to preserve the size of the image. The images in **Figures 35, 36 and 37** show the results of the convolution of the original image with respectively a 5×5 Laplacian filter and a 5×5 Blur filter as following:

5×5 **Laplacian filter**

$$H_{lapl} = \begin{bmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & -2 & -1 & 0 \\ -1 & -2 & 16 & -2 & -1 \\ 0 & -1 & -2 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{bmatrix}$$

5×5 **Blur filter**

$$H_{blur} = \frac{1}{273} \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

Figure 35: Original image

Figure 36: 5x5 Laplacian filter applied with the naive convolution algorithm

Figure 37: 5x5 Blur filter applied with the naive convolution algorithm

3.2.2 Separable Kernel

Separable filters Let H be the matrix of a filter, a filter is called separable when we can write $H(i,j) = h_1(i)h_2(j)$. This would mean being able to write H as:

$$H = \begin{pmatrix} h_1(1) \\ \vdots \\ h_1(k_x) \end{pmatrix} \begin{pmatrix} h_2(1) & \cdots & h_2(k_y) \end{pmatrix}$$

We implemented an algorithm that compute this separation for us. Our idea was to find a j such as $h_2(j) \neq 0$ (this just means that a whole column is not $(0 \cdots 0)$) and write H as

$$H = UV = \begin{pmatrix} h_1(1)h_2(j) \\ \vdots \\ h_1(k_x)h_2(j) \end{pmatrix} \begin{pmatrix} \frac{h_2(1)}{h_2(j)} & \cdots & \frac{h_2(k_y)}{h_2(j)} \end{pmatrix}$$

We can easily compute those vectors : U is just a column of our matrix and V coefficient can be obtained by along any row :

$$V(j) = \frac{H(.,j)}{H(i,j)}$$

Algorithm This algorithm takes profit of the separable property of some filters. The idea is then to write (4) as (5), which can be obtained by doing 2 successive 1-D convolutions. The complexity is then in $\mathcal{O}(nm(k_x + k_y))$ instead of $\mathcal{O}(nmk_xk_y)$ for the naive implementation.

$$\sum_i h_1(i) \sum_j h_2(j) X(x-i, y-j) \tag{5}$$

The algorithm is the straightforward implementation of the following mathematical steps:
First we take the transpose of the image X^T and we apply row-wise 1-D convolution using $h_1(i)$ on it to obtain $g(x,y)$

(6).

$$\sum_i h_1(i) X^T(x, y-i) = g(x, y) \tag{6}$$

Then we define $U(x, y) = g(y, x)$ and perform row-wise 1D convolution using $h_2(j)$ this time (7).

$$\sum_j h_2(j) U(x, y-j) = \sum_j h_2(j) \sum_i h_1(i) X^T(y-j, x-i) = \sum_i \sum_j h_1(i) h_2(j) X(x-i, y-j) \tag{7}$$

We finally recover our 2D convolution formula.

3.2.3 Fast Fourier Transform Algorithm

Let f, g two functions. Let $h = f * g$, and let's apply the Fourier transform on it. We have $\hat{h}(\xi) = \widehat{f * g}(\xi) = \hat{f}(\xi) \times \hat{g}(\xi)$ using the convolution theorem. Finally, if we apply the inverse Fourier transform we have:

$$h(x) = (f * g)(x) = \mathcal{F}^{-1}\left(\hat{f} \times \hat{g}\right)(x)$$

To implement the convolution operation using the Fast Fourier Transform (FFT), we then need to apply the Fourier transform to the image and the filter, before multiplying them and applying the inverse FFT to the product. Using the method `dft` of OpenCV, we can implement this.

However we have to make some changes on the filter to properly apply this algorithm on matrices. Indeed, in general the filter is smaller than the image, so we have to correct that. We expand the filter size to the image size by putting the center of the filter to the top left of the new sized filter. This way, the FFT algorithm implemented in OpenCV can use efficiently this matrix and compute the result. To do that, we have to translate all the values of the matrix to the top left until the center is in place. Some values will get out of the matrix. We have to take them back to the filter using a circular vision. Finally, as shown in **Figure ??**:

- the top left square will be in the bottom right square of the new sized filter,
- the bottom left square which top left value is the center of the filter will be in the top left of the new sized filter,
- the bottom left rectangle and the top right rectangle permute.

Figure 38: 5x5 Laplacian filter applied with the convolution algorithm

Figure 39: 5x5 Laplacian filter applied with the FFT convolution algorithm

3.3 Spatially variant kernel

We would like to be able to blur only the surroundings of our fingerprint in order to simulate a noisy acquisition far away from the center of pressure. To do that, we define a spatially variant kernel:

$$\text{Kernel} = \frac{d}{d_{max}}\text{BlurKernel} + (1 - \frac{d}{d_{max}})\text{IdKernel}$$

with d being the distance from the center of the blur. We have:

$$\begin{cases} \text{Kernel} \approx \text{Identity Kernel when } d \approx 0 \\ \text{Kernel} \approx \text{Blur kernel when } d \gg 0 \end{cases}$$

This enable us to recover the results shown in Figure 40. The center of the image is very sharp while the borders of the fingerprints become quite blurry but not like in the target. In our image the blur is ”uniform”, we will have the same blur intensity at every pixel which is at a given distance from the center. Thus, we are unable to recover the random shaped noise of our test image.

Figure 40: Target blurred image and our result with a spatially variant kernel

4 Morphological filtering

Morphological filtering consists in the scan of the image with a mask of a certain shape. For each pixel, the morphological filter takes the neighbours into account but does not take their values into account. That's why it is best suited for binary images.

Basic operations with morphological filters are dilation and erosion, which lead to high level operations as closing and opening. They are then essentially set operations. The dilation consists in the dilation of the image according to a certain shape. Let's take a cross shape as in **Figure 41**. When we apply the dilation on the left image **Figure 42** according to this shape, it means we extend all the pixel to its neighbours. The result is as in **Figure 42**.

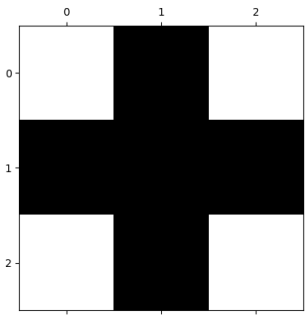

Figure 41: Cross shape of size 3x3

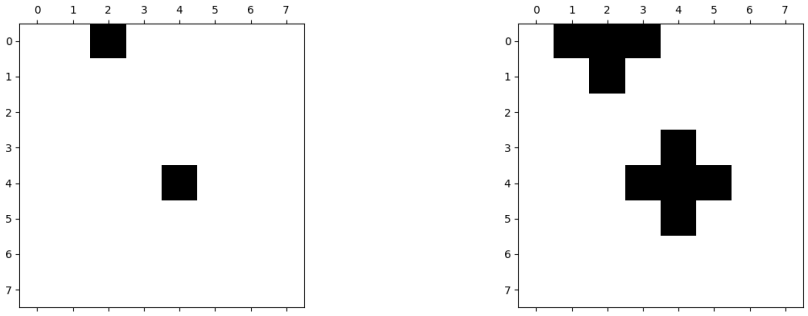

Figure 42: Dilation applied with a cross shape

For the erosion, the principle is the same, but instead of putting 1 to every neighbour of a black pixel, we check if one of the neighbour is white. In that case, the current pixel is put at 0. Examples are in **Figure 43**

Using these algorithms, we can obtain a simulated moist fingerprint using the dilation because stripes are more spread with humidity, and a simulated dry fingerprint using the erosion because the matter don't stick to the acquisition machine when our fingers are dry.

Thus, the first step to apply these algorithms on our fingerprints is to transform the grayscale images into binary images. To do this, we define a threshold below which the values are put to 0, and above which the values are put to 1. The images become then binary. To find the right threshold, we use the Otsu method. It consists in computing the different binary classes possible, and to find the one which its intra-class variance is minimum.

Once the image is binarized, we can apply the erosion and the dilation. The results are not the same if we apply the dilation before the erosion, or if we apply the erosion before the dilation. Indeed, as shown in **Figure 45**, isolated black pixels disappear if we apply the erosion and do not appear again if we apply the dilation then. The same way, isolated white pixel disappear if we apply the dilation and can not appear again if we apply the dilation. That is how is defined the opening and the closing of an image.

The opening is the operation "erosion followed by a dilation", which means that black pixels details are erased, and so only the global dark shape is kept. The closing operation on the contrary is the operation "dilation followed by the

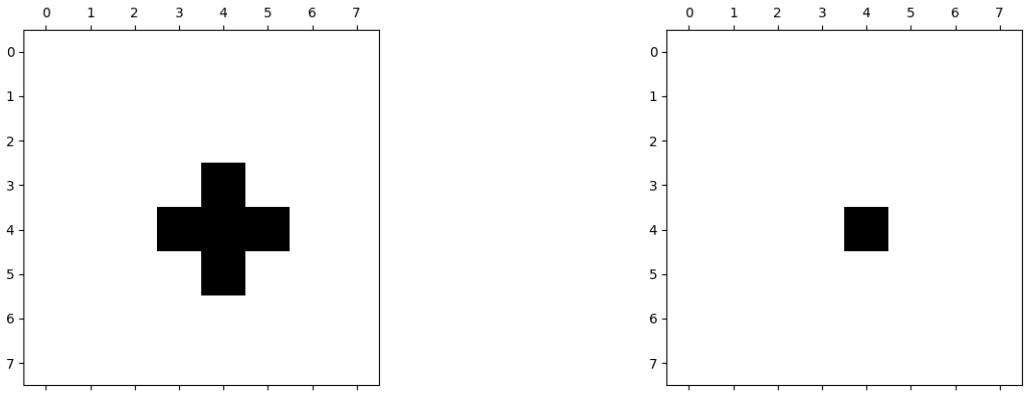

Figure 43: Erosion applied with a cross shape

erosion", which means that black pixels details are kept but white pixels details are erased. The black shape is the same. Examples are in **Figure 44** and **Figure 45**.

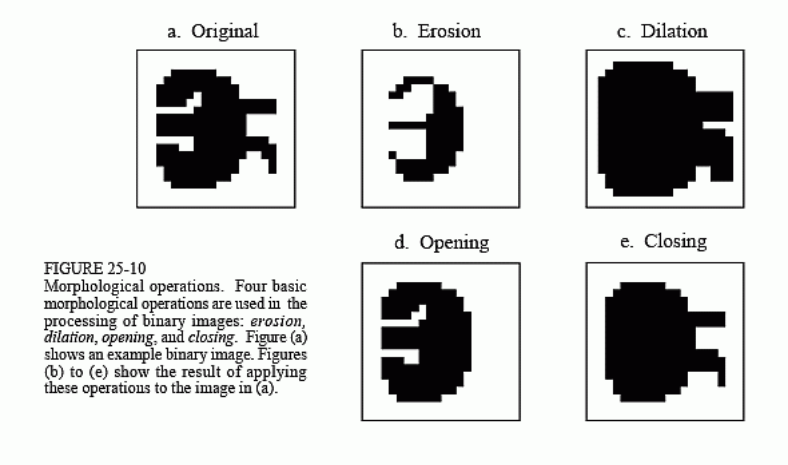

Figure 44: Application of erosion and dilation filter on binary image

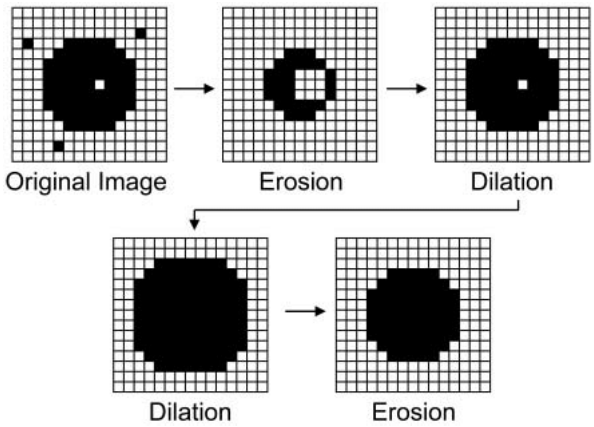

Figure 45: Application of erosion and dilation filter on binary image

These operations applied on the clean fingerprints give the images on **Figure 46** with the kernel of size 3x3 full with 1. We see that the dilation looks like the moist fingerprints and the erosion looks like the dry fingerprints.

Figure 46: Binary dilation and erosion applied with a square shape of size 3x3

4.1 Grayscale morphological filtering

We want now to apply these kind of operation on grayscale images. We decide to modify the definitions as the following:

Erosion: According to the kernel, for each pixel, we keep the minimum of the neighbours

Dilation: According to the kernel, for each pixel, we keep the maximum of the neighbours

When we apply this algorithm with the square kernel full of 1 of size 3x3, we obtain the results as in **Figure 47**

Figure 47: Grayscale dilation and erosion applied with a square shape of size 3x3

5 Optimization for Image Registration

Let's assume that we have an original picture g and the same image f but that had been subjected to a warp of unknown parameters.
The goal is to find the warp to go from g to f.
In other words, which translation and rotation we must applied to the original Image g to obtain Image f.

Figure 48: g (left) and f (right)

The main idea here is to measure the difference between the two pictures(f and warped g). When the difference is minimal, we can consider the two images to be the same; and then the parameters of the warp are the right ones.

In fact, our images f and g can be seen as vectors in $\mathbb{R}^{width+height}$. Therefore, we can compute the difference between those two images with the euclidean distance for the canonical dot product as following :
$d(f, g_p) = ||f - g_p||$.We denote g_p the test image associated to a warp of parameters p. In other words, $g_p(x, y) = g(w(x, y; p))$, with w the function representing the warp. Of course, as d is a distance, the warped image that minimizes it is f.

To simplify notations, we call f the Image of unknown warp, g our original Image, g_p is g warped with the parameter p and g_p* the optimal warped Image.
We also introduce a $lossfunction$:

$$l(p) = d(f, g_p)^2 = \sum_{x,y} (f(x, y) - g_p(x, y))^2 \tag{8}$$

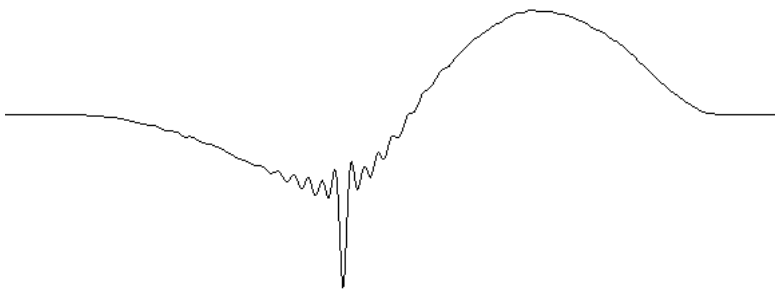

Figure 49: example of the loss function between [-width, + width] with only a translation

Here we distinguish clearly a minimum, therefore this point corresponds to the translation to apply to g in order to get f.

5.1 Translation along the y axis

First, we deal with simple cases.
Let's suppose that the warp function is a translation along the y axis. There is only one parameter, $p_y \in \mathbb{Z}$, the y value of the translation.
In order to find the smallest difference, the easiest way is to compute the distance for all the parameter's possible values and save the one that gives the minimum distance. The main problem is that this method takes a lot of useless computations.

To fix that, we use a greedy algorithm. Here are the different steps :

- We choose randomly an initial parameter p_{init} and calculate its $lossfunction, l(p_{init})$
- The first time, we either jump to $p_{init} + 1$ or to $p_{init} - 1$ and check which one lowers the $lossfunction$
- We continue to jump in the same direction, only choosing the parameter that lowers the $lossfunction$
- Once the $lossfunction$ increases, we stop : we found a minimum

Unfortunately, without any other conditions, this algorithm only could give a local minimum. It doesn't say anything about the global minimum. We can avoid this problem in some ways by checking the area around a local minimum and find out if it's in fact the global minimum or only a local one.

Thus, our algorithm is clearly not perfect, it is indeed easy to build a counter example that will indicate a local minimum as a result. Nonetheless, in practice, it seems to work most of the time.

Let's take the image txfinger.png would be **Figure 50** with p_y = -36.

Figure 50: The image of unknown translation : txfinger.png (left) and the image warped with a translation : -36 along y axis (right)

5.2 Translation along the x and y axis

Here, we experiment with a more complicated scheme by making a translation along the x and y axis. We therefore now have two parameters : p_x and p_y.
To do so, we needed to extend our algorithm.

- We choose randomly an initial parameter p_{yinit} and apply the previous algorithm to find its optimal p_x and calculate its $lossfunction, l(p_{yinit})$
- The first time, we either jump to $p_{yinit}+1$ or to $p_{yinit}-1$ (with their respective optimal p_x translation) and check which one lowers the $lossfunction$
- We continue to jump in the same direction, only choosing the parameter that lowers the $lossfunction$
- Once the $lossfunction$ increases, we stop : we found a minimum

Let's take the example of txyfinger.png would be **Figure 51** with p_x = 22 and p_y = -16.

Figure 51: The image of unknown translation : txyfinger.png (left) and the image warped with a translation : 22 and -16 along x and y axis (right)

In order to visually validate our results, we display the absolute error image, which is the per pixel absolute difference between f and g_p* where $p*$ is the optimal parameter that minimizes the distance. In the best case scenario, we should obtain a white image (the distance between two pixels is always 0).

Figure 52: Absolute error Image between f and g_p* where $p*$ = (22,-16)

The **Figure 52** shows that the two images are almost the same. Why are they not exactly the same :
-either the real value of the warp are real and not integer
-either the image f is not exactly a translation of g but another acquisition.

5.3 Going further in precision

Until now, We found an estimation (with a precision of 1) of the integer portion of p_x and p_y. We want now to find their real value (in R).
Algorithm :

- We take a step s (= 0.9 for instance) and we compute the value of the loss function on the new points : $(px \pm s, py)$, $px, py \pm s)$, $(px \pm s, py \pm s)$.
- We jump on the point that lowers the most the lossfunction.
- We take a smaller step (for instance $s = s/2$) and we repeat the process.

We should rapidly converge to the optimal real values.

Let's take the example of the image txyfinger.png. We previously founded a translation of 22 and -16 along x and y axis.
Using this new method, we find : $p_x = 21.5219$ and $p_y = -15.4938$

Figure 53: Absolute error with interger parameters: (22,-16) (left) and absolute error with real values (right)

The new Absolute error image is whiter than the previous one. It indicates that with this algorithm we obtain best results.
However this is still not perfect (not white). This suggests that txyfinger.png is not a translation of cleanfinger.png but rather another acquisition.

5.4 Validation of the algorithm

However : is our algorithm legit? or does it depends on the lossfunction? Maybe, the euclidian distance is a particular case that makes it work. We therefore chose to use another lossfunction to validate our results .

To create a new way of measuring the difference between two pictures, we took inspiration in the correlation coefficient and got the following formula, knowing that $\bar{f}$ is the mean of all the pixel's value:

$$l(p) = \frac{\sum_{x,y}(f(x,y) - \bar{f})(g_p(x,y) - \bar{g_p})}{\sqrt{\sum_{x,y}(f(x,y) - \bar{f})^2 \sum_{x,y}(g_p(x,y) - \bar{g_p})^2}} \tag{9}$$

Since f and g are in $\mathbb{R}^{width+height}$, using Cauchy-Schwarz inequality, we have :

$$l(p) \leq 1 \tag{10}$$

It is then important to notice that $l(p) = 1$ when $g_p = f$. Therefore, we here need to find the maximum of the function instead of the minimum.

As expected, using the same example, the algorithm returns the same parameters as before, p_x = 22 and p_x = -16.

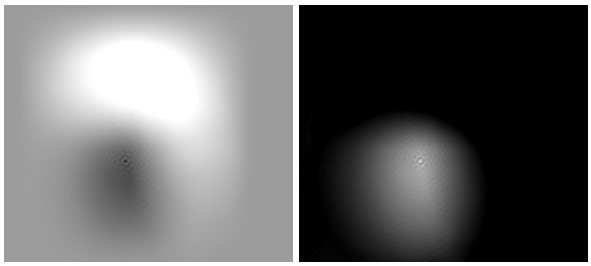

Figure 54: Euclidean distance (left) and correlation coefficient (right) between txyfinger.png and cleanfinger.png in 2D : every pixel (i,j) represents $l(p = (i, j))$, white is high and black is small: proof that the two lossfunction indicate the same optimal parameter

The computation's time is slightly slower with this new lossfunction because this function is more complex.

5.5 General Warp: Rotation and Translation

Now that our algorithms work with the easiest warp (only translation), we can deal with complex cases : a combination of a rotation and a translation.
Since a rotation and a translation are not commutative, we decided that the warp would be defined as first a rotation then a translation.
We adapt our previous greedy algorithm adding a new parameter : θ the rotation.
Algorithm :

- We take an angular step ϵ
- We choose randomly an initial parameter θ_{init} and apply the previous algorithm to find its optimal translation (p_x, p_y) and calculate its $lossfunction, l(p_{\theta init})$
- The first time, we either jump to $\theta_{init} + \epsilon$ or to $\theta_{init} - \epsilon$ (with their respective optimal (p_x, p_y) translation) and check which one lowers (or maximizes depending on the lossfunction chosen) the $lossfunction$
- We continue to jump in the same direction, only choosing the parameter that lowers (or maximizes) the $lossfunction$
- Once the $lossfunction$ increases, we stop : we found a minimum/maximum

Let's try it on the image f = rtxyfinger.png. We find $\theta = -0.26, p_x = 18, p_y = -27$. The computation time is quite good (12 seconds for rtxyfinger.png with initial values set to (0,0,0)).

Figure 55: Absolute error Image between f and g_p* where $p*$ = (-0.26,18,-27)

However, there is a black point in our algorithm : we don't know where to initialize our parameters. If we are not lucky, we will start very far from the optimal parameters and it will take many computations. The images we use are quite small, that explains the reasonable computing time for rtxyfinger.png. We can suppose that for high quality pictures, the computing time will be enormous.

To solve that, we want to give to the algorithm an idea where the minimal is (a rough approximation) to avoid useless computations.
The main idea is to find a point $A = (x_a, y_a)$ on g (cleanfinger.png) and its equivalent $B = (x_b, y_b)$ on f (rtxyfinger.png). For every rotation,we will try to match these 2 points, giving us the translation ($p_x = x_b - x_a$ and $p_y = y_b - y_a$).
Here we are going to use the pressure point (represented with the barycenter) as A and B. Of course, since we lost some data in rtxyfinger.png, B will not be exactly the equivalent of A but will be close if we don't loose to much data (i.e the translation is not too high).

Let's try it on rtxyfinger.png(**Figure 56**),we find as a rough approximation of the translation : $p_x = 15$ and $p_y = -36$ instead of (18,-27) (**Figure 55**) . We can now apply our previous algorithm starting with this translation.
On rtxyfinger.png, we have gained 2 seconds.

Figure 56: A :Pressure Point on original image (left) and B: pressure point on unknown warped image (right)

However, the barycenter is not a good point to use since it gives only an approximation and only works with the translation. Reading an article (Recalage d'images d'empreintes digitales en biométrie sans contact), we found a new way more efficient.
The goal is to solve:

$$\underbrace{\begin{pmatrix} x' - x_0 \\ y' - y_0 \end{pmatrix}}_{f} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \underbrace{\begin{pmatrix} x \\ y \end{pmatrix}}_{g} + \underbrace{\begin{pmatrix} t_x \\ t_y \end{pmatrix}}_{T}$$

How to do that:

- Select a set of points of g and their corresponding points in f using moments of Zernike
- Estimate the transformation and statistically verify it

We didn't try to apply this method because it forces us to admit a lot of mathematical results (moments of Zernike, Ransac algorithm) and to copy an idea that wasn't ours.

5.6 Main Course 5 : Gradient descent strategies

A new method to find the minimum is the Gradient descent strategies.
How does it work?
Let f be a function from $R^n -> R$ smooth enough and differentiable. The sequence $x_{k+1} = x_k - \alpha \nabla f(x_k)$ converges to a local minimum $x*$ in the neighborhood of x_0.
However, this algorithm only find a local minimum.
Let's try it on txysmall where there is no rotation and the translation is close to (0,0).
Let f be txysmall and g cleanfinger.
Let ω be the warp (in the case of the translation $\omega_x = x + p_x, \omega_y = y + p_y$)
Let's compute the derivative of g :

$$\frac{\partial g(\omega(x,y;p))}{\partial p} = \begin{cases} \frac{\partial g_\omega}{\partial \omega_x}\frac{\partial \omega_x}{\partial p_x} \\ \\ \frac{\partial g_\omega}{\partial \omega_y}\frac{\partial \omega_y}{\partial p_y} \end{cases}$$

Here, we will have $x_0 = (p_x, p_y) = (0, 0)$.
$\nabla l(p) = (\frac{\partial l(p)}{\partial p_x}, \frac{\partial l(p)}{\partial p_y})$ We will use the first loss function : the euclidean distance.
Let p be a vector (here $p = (p_x, p_y)$).

The sum is finite (only few thousands pixels), therefore we can permute the operators.

$$\begin{aligned} \nabla_{p_x} = \frac{\partial l(p)}{\partial p_x} &= \sum_{x,y} \frac{\partial (f(x,y) - g_p(\omega_x, \omega_y))^2}{\partial p_x} \\ &= \sum_{x,y} -2(f(x,y) - g_p(\omega_x, \omega_y)) \frac{\partial g_p(x,y)}{\partial \omega_x} \\ since \frac{\partial \omega_x}{\partial p_x} &= 1 \end{aligned} \tag{11}$$

We have the same results for p_y.
We can now proceed to the algorithm (we took $\alpha = 10^{-5}$).
We find, with this method, $p_x = 2.10791 p_y = -1.67856$.

Figure 57: Absolute error Image between f and g_p* where $p*$ = (2.10791,-1.67856)

The **Figure 57** shows that the gradient method works if we are close to the global minimum.
But how to avoid falling in a local minimum ?
We could to adapt the α. However, the Frank–Wolfe algorithm (built to choose the step α at every iteration) works mainly for convex function, which it's not the case of our lossfunction.
The only solution we found is to combine our greedy algorithm to approach the global minimum and then us the gradient descent.

Now let's deal with the general warp : a rotation and translation: $p = (p_x, p_y, \theta)$

We proceed the same way, adding a new parameter (the rotation).

$$\begin{aligned}\nabla_\theta &= \frac{\partial l}{\partial \theta}(p) \\ &= \sum_{x,y} -2(f(x,y) - g_p(\omega_x, \omega_y))\frac{\partial g_p}{\partial \theta}(\omega_x, \omega_y) \\ &= \sum_{x,y} -2(f(x,y) - g_p(\omega_x, \omega_y))(\frac{\partial g_p}{\partial \omega_x}(\omega_x, \omega_y)\frac{\partial \omega_x}{\partial \theta}(x,y) + \frac{\partial g_p}{\partial \omega_y}(\omega_x, \omega_y)\frac{\partial \omega_y}{\partial \theta}(x,y))\end{aligned} \tag{12}$$

Since, we didn't find a nice and elegant way to avoid the local minimum, On the Image rtxyfinger.png, the algorithm doesn't work and do not find the optimal parameters.

6 Conclusion

The aim of this project was to simulate several natural artifacts thanks to mathematical filters applied on fingerprint images : rotation, strong and weak pressure, moist and dry skin, skin warp and motion blur.

We tried to do so by collecting different verified works from the computing community and assembling them together. We consider that our results are satisfactory at a certain extend. Of course, we can always improve our different algorithms in many ways in order to reduce their complexity but we are pleased by their performance, especially compared to what we first got at the beginning of the project when using naive and simple approaches.

The tools that really helped us to make sure that we kept the same valid results despite changing some methods or even approaches are Google Tests. We created several functions based on this particular library that allowed us to easily test if the expected image was always matching our result. This saved us a lot of time because since we were using them each time we changed the code a bit, we were not carrying errors elsewhere.

Moreover, if there had been more time to work on the project, we certainly would have focused more on the ”Main course 5” and the gradient descent method because we came across several problems that we just did not solve in time. In fact, this technique is highly interesting to understand, and compute, thanks to its high efficiency compared to other similar algorithms. It is also used a lot in Machine Learning for evaluating and reducing errors computed by Neural Networks.

References

[1] Vincent Dumoulin and Fransesco Visin. *"A guide to convolution arithmetic for deep learning"*. In *ArXiv e-prints* (2016)

[2] Dan Bloomberg. *Image Rotation*. URL: https://tpgit.github.io/UnOfficialLeptDocs/leptonica/rotation.html

[3] Tahirou Djara, Amine Nait-ali, Antoine Cokou Vianou, Marc Kokou Assogba. *Recalage d'images d'empreintes digitales en biométrie sans contact* (2013). URL: https://www.researchgate.net/publication/258960130

[4] Yao Wang. *"Median Filtering and Morphological Filtering"*. URL: http://eeweb.poly.edu/~yao/EL5123/lecture7_median_morph.pdf

[5] Juan Pablo Balarini, Sergio Nesmachnow. *"A C++ Implementation of Otsu's Image Segmentation Method"*. URL: https://www.ipol.im/pub/art/2016/158/

[6] Michael A. Wirth. *"Grayscale Morphological Analysis*. URL: http://www.cyto.purdue.edu/cdroms/micro2/content/education/wirth08.pdf