Communauté
UNIVERSITÉ Grenoble Alpes

DIGITAL FINGERPRINT ANALYSIS

Hanning Yang
Jiaying Tu
Lynn Farhat
Niloufar Zarghampour

February 22, 2022

Contents

1 Abstract

Fingerprint analysis is a tool of great significance with today's digital devices. In this project, we have composed codes in C++ using OpenCV to simulate certain artefacts that may affect digital fingerprint analysis, and to restore fingerprints that have already been subject to such artefacts. We created fundamental methods which allow to save images,load them, define them, and potentially alter their pixels which was pretty useful when we wanted to define a fingerprint image or make changes to pre-existing ones. As for the artefacts, we focused mainly on those related to finger pressure variation, geometrical warps, and motion blurring.

2 Introduction

Fingerprint recognition has been applied to identify criminals in law enforcement, and currently it is being increasingly used for personal identification in a civilian's daily life, such as an ID card, fingerprints hard disk, and so on. Various fingerprint recognition techniques, including fingerprint acquisition, enhancement, matching,and classification are developed and advanced rapidly.

However, there are still difficult and challenging tasks in this field. One of the main difficulties in matching two fingerprint impressions of the same finger is to deal with the nonlinear distortions, which are caused by the acquisition process.

Fingerprint recognition can be done correctly by locating "minutiae points" which distinguish fingerprints from each other, and could be altered based on several factors which could potentially misrepresent a fingerprint.

In this project, we studied the ways in which fingerprint identification could be distorted or misrepresented. Then, we relied on using mathematical tools to simulate each of these potential scenarios of distortion of the fingerprint. Afterwards, we used similar tools to repair the distorted fingerprint to make it comprehensive and identifiable.

We completed our work with OpenCV in C++ in order to perform operations on fingerprint images to get the desired results and alterations. First, we began by defining a general class Image which contains fundamental methods that were helpful in defining or creating a certain image at several points throughout the project.

We relied on mathematical concepts and models such as linear filtering and convolution, as well as creating geometrical figures in order to complete the simulations and restorations of the fingerprint images.

3 Image Loading, Saving and Pixels Manipulation

3.1 Fundamentals

The intensity range we chose originally for our images is [0,255] as it is for most images. The Image frame is as below:

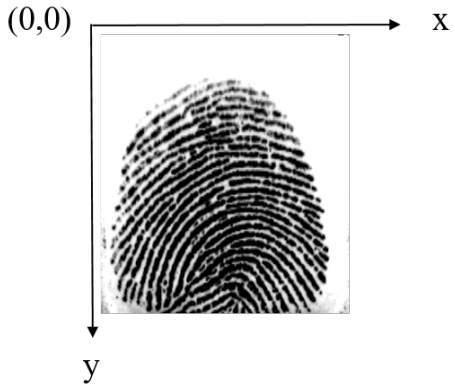


Figure 1

We chose to use OpenCV as our image processing library as we found that it has fundamental operations on images and matrices which will prove to be quite useful in the codes we constructed. Moreover, and more importantly, it very simply allows us to load, output, and use an image, before and after performing our desired alterations on it. There are several features for the OpenCV library, some of the general features are its open-source, fast speed, ease of integration, ease of coding, and fast prototyping.

Below, the diagram of the General class "Image" is shown. This class enables loading, saving and displaying images, alongside multiple methods that we will go into details later on.

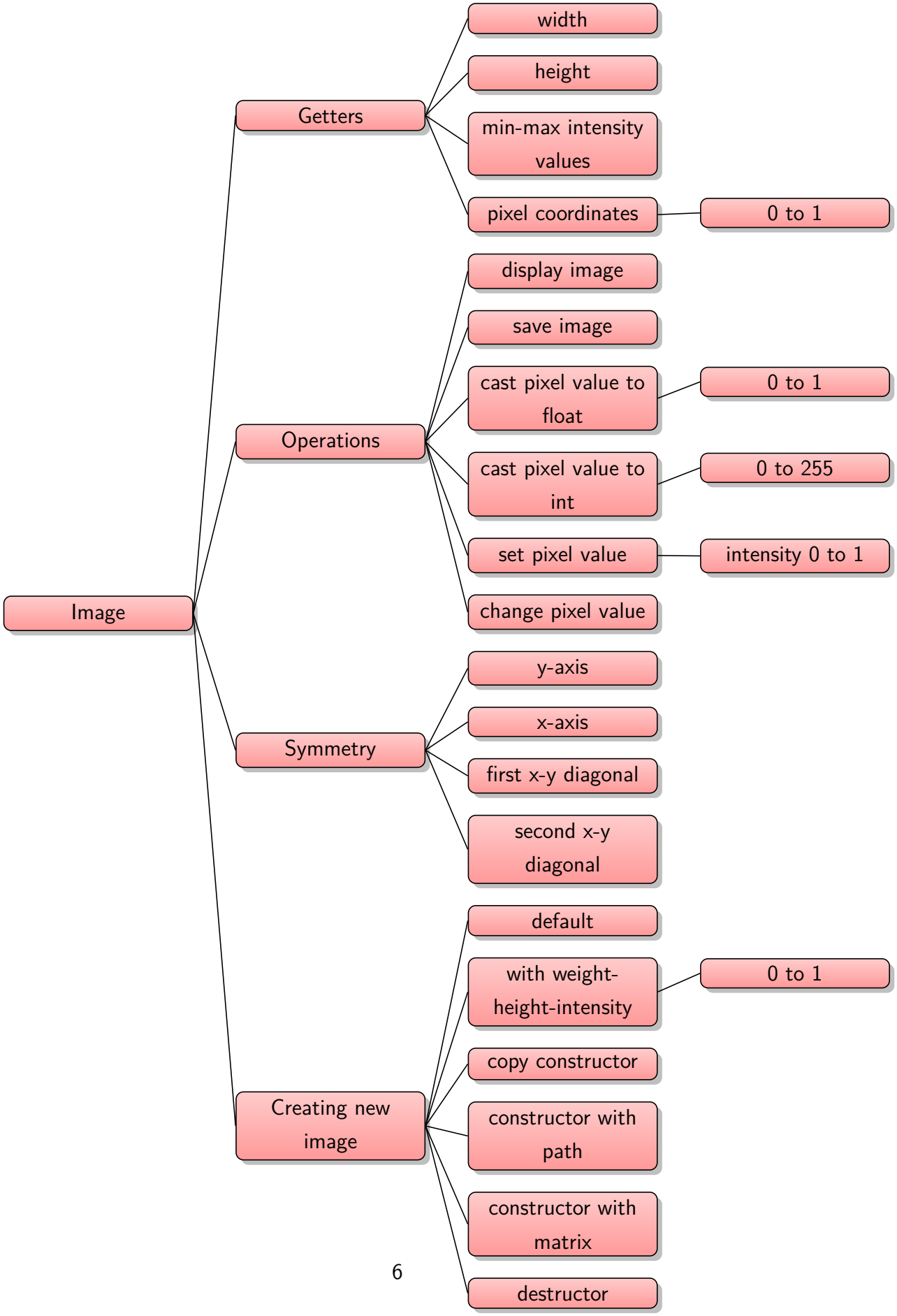


Image
Getters
width
height
min-max intensity values
pixel coordinates
0 to 1
Operations
display image
save image
cast pixel value to float
0 to 1
cast pixel value to int
0 to 255
set pixel value
intensity 0 to 1
change pixel value
Symmetry
y-axis
x-axis
first x-y diagonal
second x-y diagonal
Creating new image
default
with weight-height-intensity
0 to 1
copy constructor
constructor with path
constructor with matrix
destructor

3.2 Some basic methods

Originally, the pixel intensity range is the integer values between 0 and 255. However, since most of the operations that we will be performing are float point operations, we created a method that casts all integer values of intensity between 0 and 255 to float values between 0 and 1. This normalization is also done to avoid high-value numerical computations. Therefore, every integer was divided by 255. Our new minimum is now 0 and the maximum is 1.

Afterwards, we created a method to change the intensity of specific pixels in order to allow us to create black or white boxes on the fingerprint image. An application of this method results in the following, whereby the "clean fingerprint" in Figure 1 is the sample fingerprint that has not been tampered with, and the fingerprint in Figure 2 is the result of the pixel changes:

(a) Clean fingerprint (b) Fingerprint with squares

Figure 2: Adding black and white squares

In order to save the resulting image, we created a method to allow us to save it as a png file. However, in this method, before saving the image, we made sure to cast all the pixel values back to integers in the range [0,255].

3.3 Symmetries

After implementing the aforementioned methods, we moved on to implementing methods that create a new symmetrical image to the original clean fingerprint, with respect to several axes.

To start off, we created symmetry with respect to the y-axis. Assuming $s(x, y)$ is the image of the fingerprint that is symmetric to the original fingerprint image $f(x, y)$ with respect to the y-axis, we get the following mathematical relation between the two images:

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

Thus, the symmetry with respect to the y-axis is implemented by maintaining the vertical coordinates and choosing the negative of the horizontal ones. We then apply this change to our image, which yields the following result:

(a) Clean fingerprint (b) Symmetric w.r.t. y-axis

Figure 3: Symmetry w.r.t. y-axis

We then implemented two methods to create symmetry with respect to the x-y diagonal

axes, one method for each diagonal axis. However, it must be taken into account that the origin (0,0) of our image is considered at the top left corner of the image.

Thus, the symmetry with respect to the first diagonal is obtained by switching the x and y coordinates. So, for a pixel (i,j), when creating the symmetry with respect to the first x-y diagonal, becomes of coordinates (j,i).

Similarly, we apply the same rule for the second diagonal axis, whereby pixel (i,j) becomes (height-j-1,width-i-1) whereby the height and the width are those of the whole image. Thus, we get the following results:

(a) Clean fingerprint (b) Symmetric w.r.t. y=x

Figure 3: Symmetry w.r.t. first diagonal axis

(a) Clean fingerprint (b) Symmetric w.r.t. y=-x

Figure 4: Symmetry w.r.t. second diagonal axis

The pixel swapping of the x-y diagonals may appear like it is a simple rotation, but it is in fact not. That is because the image is somewhat "mirrored". Now, we know that the determinant of a rotation matrix must be 1. So, taking the transformation matrix that we used to complete the first diagonal symmetry, we get the following:

$$\begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} y & x \end{bmatrix}$$

The matrix has a determinant of -1, which makes it not a rotation matrix.

3.4 Weak Pressure Simulation

The goal in this section is to simulate the pressure variation of a fingerprint on a sensor, whereby a point with highest pressure would be one where pixel intensity are high, and that would be our spot center on the fingerprint. In fingerprints of weak pressure, the outer ring of the finger could appear to be faded or completely gone, as the pixel intensity decreases as we move further away from the spot center.
In order to implement the simulation, we use the following equation:

$$g(x, y) = c(x, y)f(x, y)$$

where f is the original image, $c(x, y) \in [0, 1]$ is the scalar coefficient or the weight, and g is the resulting image.
We recall that an isotropic function is a function that is independent of direction. An example of such a function is a radial function, such as the unit circle.
Some examples of a function $c(r)$ that is monotonically decreasing as r tends to infinity, with $c(0) = 1$ and which vanishes as r tends to infinity could be as follows, $\forall \alpha, \beta, n \in \mathbb{R}^+$:

- $c(r) = e^{-\frac{r^n}{\alpha}}$
- $c(r) = \frac{1}{\frac{r^n}{\alpha}+1}$
- $c(r) = \frac{1}{1+e^{\alpha r-\beta}}$ (Logistic function)

Firstly, we created a method that identifies the barycenter of the clean fingerprint image by taking the midpoint, and it is identified by the white square on the image as below:

Figure 4: Barycenter

Then, we tested out the above functions for certain values of α, β, and γ in our code which gave different results:

(a) $e^{\frac{-r^2}{1000}}$ (b) $\frac{1}{\frac{r^2}{1000}+1}$ (c) $\frac{1}{1+e^{0.45r-30}}$

Figure 5: Isotropic Results

Now, we recall that anisotropy is the property of being dependant on direction. This means that for the alterations of the edges of the finger, an anisotropic approach would allow the alterations to go in more than one direction. Because the image of the weak fingerprint that we are supposed to get is elliptical, whereas the images that we got above using an isotropic approach are more circular, then we observe that the pixel intensity transform is in fact anisotropic. Thus, we altered our previous method to make it anisotropic.
From our isotropic results, we can tell that the most satisfactory result we got was from

the third function $c(r)$. Thus, we will reuse the function (with slightly different values) $c(r) = \frac{1}{1+e^{0.25r-50}}$ to apply an anisotropic adaptation to it.
In order to do so, we created a method that identifies a contour of the fingerprint and creates an ellipse that is of similar shape, since that is the simplest and closest geometrical figure to the fingerprint.
Thus, we take the barycenter of the fingerprint as the center of the ellipse and we calculate the distance from that center to the upmost point of the fingerprint ($a \in \mathbb{R}^+$) and the distance from the center to the left-most point ($b \in \mathbb{R}^+$). These distances are what we will define our ellipse with.
Considering that the center of the ellipse is $C = (x_c, y_c)$, the equation of the ellipse is as follows $\forall x, y \in \mathbb{R}$:

$$r^2 = \left(\frac{x - x_c}{a}\right)^2 + \left(\frac{y - y_c}{b}\right)^2$$

and the following is its figure:

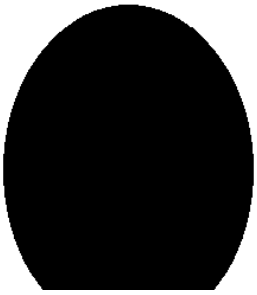


Figure 6: Finger Zone Ellipse

The main idea behind this is implemented through detecting the edge of the fingerprint. The way this works is that we take a point/pixel on the far top of the image and we create a small square surrounding this point. Then, we keep moving the point down towards the barycenter so long as the average of the intensities of the points of the square is greater than 0.9. As soon as this average becomes less than 0.9, then that means that the pixels are no longer pure white, which means we have reached the edge of the fingerprint. That is mainly due to the fact that the color white results in a relatively significant change with black (as we are using greyscale) since they are on the two extremes of the intensity interval [0,1]. We then do the same for the rest of the points until we have found the entire contour of the fingerprint.

However, in order to reach such a figure, we had to find an adequate rotation of our ellipse based on the way we took our coordinate system (in the top left corner of the image).

Thus, originally, our ellipse was shaped as below by default whereby the angle of rotation θ is null. This gave us a result as in Figure 7.

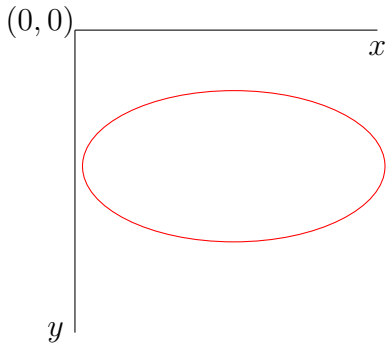


Figure 7: Original Anisotropic Result for $c(r) = \frac{1}{1+e^{0.25r-50}}$

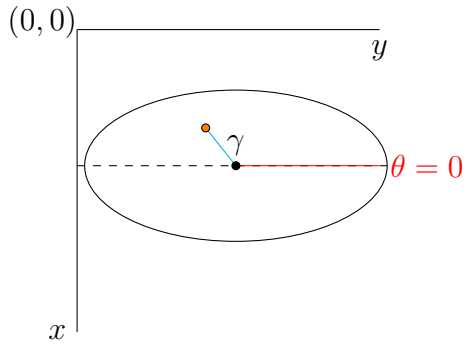
We now consider a point $P = (x_p, y_p)$ distinct from C. Then, there must exist an angle $\gamma \in [0, 2\pi]$ such that we can define the polar coordinates of P by considering C to be the vertex of the angle, and the abscissa to be the one parallel to the y-axis defined. Therefore, the polar coordinates of P could be defined as follows:

$$x_p - x_c = -r_p cos(\gamma)$$

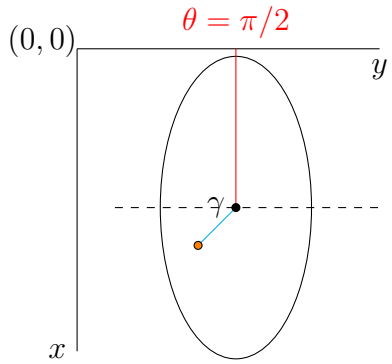
$$y_p - y_c = r_p sin(\gamma)$$

In the graph below, the orange point is some point P, the and between the red line and the black dashed line is θ (initially null), and the angle between the red line and the cyan

line is γ.



Now, if we were to rotate the ellipse by an angle θ, let's say $\theta = \frac{\pi}{2}$ while keeping the P at the same spot with respect to the center of the ellipse, then we get the following diagram:



We can thus see that since γ remains constant and θ varies with rotation (in our case, it increased from 0 to $\frac{\pi}{2}$), then our new angle which defines the polar coordinates of point P_{rot} after rotation becomes $(\gamma + \theta)$.
Therefore, we define the polar coordinates of P_{rot} as follows:

$$x_{rot} - x_c = -r_P sin(\gamma + \theta)$$

$$= -r_P(sin(\gamma)cos(\theta) + cos(\gamma)sin(\theta))$$

$$= (x_P - x_c)cos(\theta) - (y_P - y_c)sin(\theta)$$

$$\implies x_{rot} = (x_P - x_c)cos(\theta) - (y_P - y_c)sin(\theta) + x_c$$

and:

$$y_{rot} - y_c = r_P cos(\gamma + \theta)$$

$$= r_P(cos(\gamma)cos(\theta) - sin(\gamma)sin(\theta))$$

$$= (y_P - y_c)cos(\theta) + (x_P - x_c)sin(\theta)$$

$$\implies y_{rot} = (y_P - y_c)cos(\theta) + (x_P - x_c)sin(\theta) + y_c$$

In the desired result image, the fingerprint seems to be rotated by about 85° which is what we applied in our method.
We then updated each pixel by multiplying the old pixels at that point by the coefficient function $c(r)$ that we have chosen.
The result that we get after applying the above in the anisotropic method is shown in Figure 8.

Figure 8: Anisotropic Result for $c(r) = \frac{1}{1+e^{0.25r-50}}$

In conclusion, our resulting simulated image of the fingerprint seems to be closer to the desired result when an anisotropic approach is applied to it. Moreover, our resulting image has edges that are faded out as required. However, the fingerprint image we are trying to achieve has edges that do not form a proper ellipse like our result as it has some "bends" in its boundaries, which is the main distinction.

3.5 Weak Pressure Restoration

In this section, we intend to restore fingerprints that have been subject to information lost around the edges, mainly as a result of weak pressure of the finger on the sensor. The main idea is to collect some small sections from the fingerprint information we already have and attempt to stick them where information is lost in the most ideal way.
In order to restore the distorted image with missing information, we created three new classes

in our code:

a) Class Mask: generates a mask of the same size as the fingerprint region.

b) Class Patch: generates nxn squares from the original image.

c) Class Dictionary: stores the middle coordinate of the patch.

To begin with, we created patches with random coordinates on the image, after which we created vectors that store the middle coordinates of each one of these patches. Below is an example of a dictionary that we created with patches of size 9x9 pixels each:

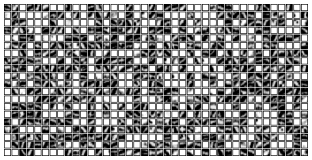


Figure 9: Dictionary

After that, we created a mask with a small square of size 30x30 pixels, for testing purposes, in the middle of the fingerprint image, as shown in Figure 10.

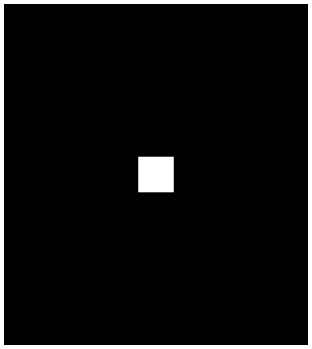


Figure 10: Testing Square Mask

In our code, we consider the black pixels to be False and the white pixels to be True which need to be altered. Thus, we combine the square mask with the original fingerprint image as shown in Figure 11 to get an image with the missing information.

Figure 11: Fingerprint with Square Mask

Then, for every pixel of the mask that gives "True", we need to find its surrounding patch. If the surrounding patch is out of range - meaning the chosen pixel is at an edge - then we choose by default the nearest patch. After that, we calculate the Euclidean distance between the patch and all the other patches in the dictionary. In this case, the distance is not the distance of coordinates but the difference between the intensity values. Our goal here is to find the patch P_s in the dictionary with the minimum distance from our original patch. Once we find this patch, we replace the lost pixel with the intensity of the middle coordinate of P_s. The result is shown below:

Figure 12: Restored Square Mask

Similar to what we have done above, we want to create a rink mask for the fingerprint image. In the simulation part of this section, we have successfully calculated the long and short axes of the ellipse as well as the barycenter of the fingerprint image.
For the reconstruction of the ring masks, we intially attempted to reconstruct them from the top left corner to the bottom right as we did for the square mask. However, we then found that there was a problem using this approach for the ring masks. Thus, we had to reconstruct one pixel at a time based on the distance to the center of the ellipse. So, we created a new method that calculates the distance from the the barycenter to all the pixels

that need to be reconstructed and sorts them from largest to smallest. Then, when restore the image from the outer ellipse inwards.

Below is a ring mask of width 10 and the image that needs to be restored, to which the ring mask is applied:

(a) Ring Mask (b) Fingerprint with Ring Mask

Figure 13: Ring Mask

We then perform the same operations of restoration on this ring mask. We tested out different variables to compare results..
Firstly, we change the value of the size of dictionary to 100, 300, and 1000 respectively, while fixing the patch size to 9, which yields the below results:

(a) Dictionary Size 100 (b) Dictionary Size 300 (c) Dictionary Size 1000

Figure 14: Varying Dictionary Size

We found that as the number of dictionaries gets larger, the clearer and more detailed the reconstructed image becomes. But our program is very slow when the number of dictionaries is 1000. In fact, we needed about 6 minutes to run it, so it is difficult for us to pick a larger

number of dictionaries. This is because for each pixel that needs to be replaced, we have to iterate through the whole dictionary to calculate the distance.

We then changed the patch size while fixing the dictionary size to 300 to see the effect of this change. We tested out sizes 5x5, 9x9, and 15x15 respectively as shown in Figure 15. We thus found that when the patch size is too small, we had difficulties in finding a patch in the dictionary that matched it. Thus, when the size of the patch is larger, the reconstruction is more complete; otherwise there will be a lot of missing information left.

(a) patch size 15 (b) patch size 9 (c) patch size 5

Figure 15: Varying Patch Size

4 Geometrical Warps

4.1 Introduction to Image Warp

Image warping is a transformation which maps all pixel coordinates in one image plane to coordinates in a second plane. This concept is used in many image analysis problems like removing optical distortions introduced by a camera or a particular viewing perspective, registering an image with a map or template, or aligning two or more images.
A warping is a pair of two-dimensional functions, $u(x, y)$ and $v(x, y)$, which map a position (x, y) in one image, where x denotes column number and y denotes row number, to position (u, v) in another image.

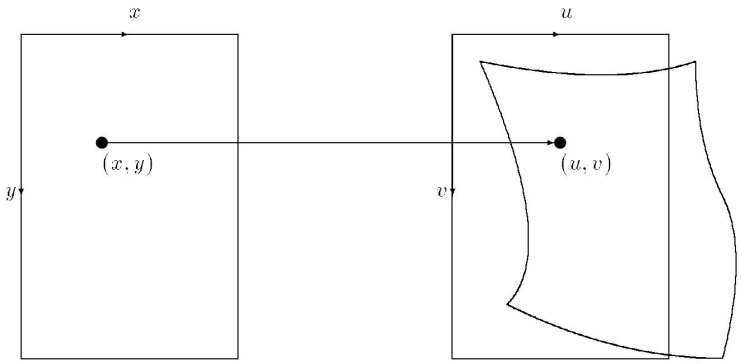


Figure 16: Illustration of Image Warp in the defined Image frame

As seen in Fig 17 and 18, the difference between image warping and image filtering is that in image filtering we alter the range whereas in image warping we change the domain of the image.

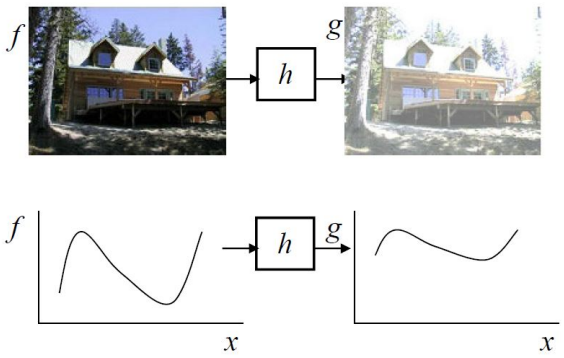


Figure 17: Image Filtering Alters the Image Range

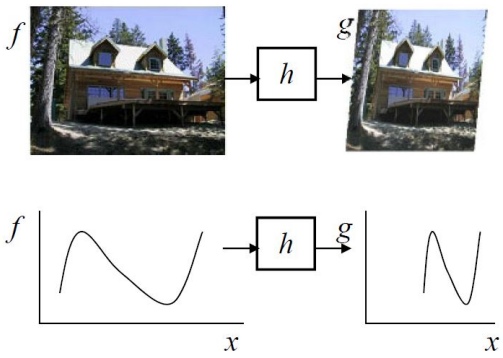


Figure 18: Image Warp Alters the Image Domain

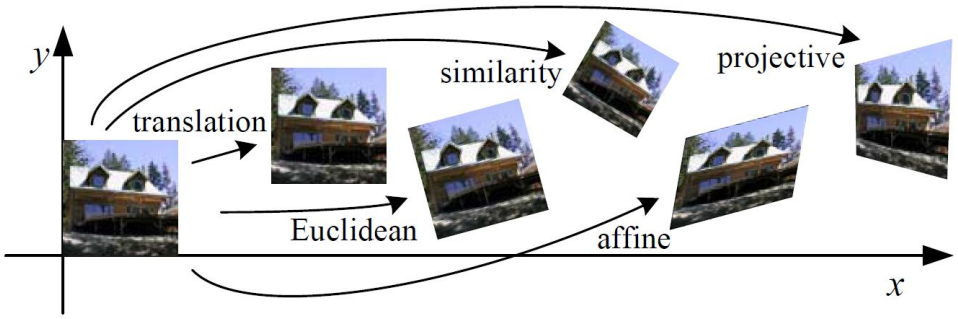


Figure 19: All types of 2D Geometrical Transformation

In general, we have 3 classes of geometric transformations:

1. Rigid transformation : This transformation is distance preserving and consists of translations along with rotation.

2. Similarity transformation : This transformation is angle preserving and consists of translation, rotation and uniform scaling.

3. Affine transformation : This transform preserves the parallelism feature of the original image and consists of translation, rotation, shear, and scale.

For this report, we go into details about the Affine Transformation.

4.2 Affine Transformations

An affine transformation is a function that maps an object from an affine space to another while preserving features such as lines or distance ratios. However, it changes the orientation,

size or position of the object. To transform an image , we use a matrix $T \in M_4(\mathbb{R})$:

$$\mathbf{T} = \left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & T_x \\ a_{21} & a_{22} & a_{23} & T_y \\ a_{31} & a_{32} & a_{33} & T_z \\ \hline P_x & P_y & P_z & 1 \end{array}\right]$$

The vector $[T_x, T_y, T_z]$ represents the translation vector according the canonical vectors. The vector $[P_x, P_y, P_z]$ represents the projection vector on the basis. The square matrix composed by the a_{ij} elements is the affine transformation matrix.In image processing due to the bi dimensional nature of images we will only used a reduced version of the previous matrix :

$$\mathbf{T} = \left[\begin{array}{cc|c} a_{11} & a_{12} & T_x \\ a_{21} & a_{22} & T_y \\ \hline P_x & P_y & 1 \end{array}\right]$$

We will also consider that our projection vector : $[P_x, P_y]$ is the null vector.
The set of affine transformations are :
1. Translation : Modifies the object position in an image.
The corresponding matrix for this transformation is as follows:

$$\mathbf{X}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \left[\begin{array}{cc|c} 1 & 0 & T_x \\ 0 & 1 & T_y \\ \hline 0 & 0 & 1 \end{array}\right] \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

2. Rotation : Allows to rotate an object according to its axis.
The corresponding matrix for this transformation is as follows:

$$\mathbf{X}'_{\mathbf{h}} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \left[\begin{array}{cc|c} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 1 \\ \hline 0 & 0 & 1 \end{array}\right] \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

3. Shear mapping : A linear map that displaces each point in a fixed direction, by an amount proportional to its signed distance from the line that is parallel to that direction and goes through the origin. Shearing can be done in both horizontal and vertical directions. The

corresponding matrix for the horizontal shearing is:

$$\mathbf{X'_h} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \left[\begin{array}{cc|c} 1 & 0 & 0 \\ s_h & 1 & 0 \\ \hline 0 & 0 & 1 \end{array}\right] \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Also, for the vertical shearing we have the following matrix:

$$\mathbf{X'_v} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \left[\begin{array}{cc|c} 1 & s_v & 0 \\ 0 & 1 & 0 \\ \hline 0 & 0 & 1 \end{array}\right] \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

It goes without saying that we can obtain an affine transformation containing a combination of all these three transformations. Note that, all these transformations are implemented using matrix multiplications.
With the use of homogeneous coordinates , we will be able to use the mathematical properties of matrices to perform the mentioned transformations.

4.2.1 Forward Transformation

In this technique, all the pixels of the original image are mapped to their position in the new image. The Algorithm is as follows :

Algorithm 1: ForwardWarp

Input: The original image (f) , Transformation Function (h)
Output: Transformed Image (g)

1 For every pixel x in $f(x)$:
2 Compute the corresponding location $x^{'} = h(x)$
3 Copy the pixel $f(x)$ to $g(x^{'})$

Figure 20: Illustration of Forward Warping

There are several issues that could occur while implementing this approach. First, when a pixel is copied from $f(x)$ to a location $x^{'}$ in the new image,if $x^{'}$ has a non-integer value, then g is not well defined. This issue could be fixed with the help of interpolation, for example, we round the value of $x^{'}$ to the nearest integer coordinate, but that could result in severe aliasing. There is also another method where we can distribute the value among its four nearest neighbors in a weighted fashion and then normalizing the values at the end. This method is called splatting and it is often used for volume rendering. However, in this method, we could experience aliasing and also, a fair amount of blur.
Based on the reasons mentioned, we will investigate another method, called "Inverse Warp".

4.2.2 Inverse Transformation

In this method, each pixel in the destination image $g(x^{'})$ is sampled from the original image. The algorithm is as below:

Algorithm 2: InverseWarp

Input: The original image (f) , Transformation Function (h)
Output: Transformed Image (g)

1 For every pixel $x^{'}$ in $f(x^{'})$:
2 Compute the source location $x = \hat{\boldsymbol{h}}\,(\boldsymbol{x}')$
3 Resample $f(x)$ at location x and copy to $g(x^{'})$

Figure 21: Illustration of Inverse Warping

Basically in this method, since $\hat{\boldsymbol{h}}(\boldsymbol{x}')$ is defined for all pixels in $g(x')$, we no longer have holes. Furthermore, in most cases, $\hat{\boldsymbol{h}}(\boldsymbol{x}')$ is the inverse of $h(x)$

4.3 Implementation

The proposed question in this section was to find the transformation between these two versions of the same fingerprint:

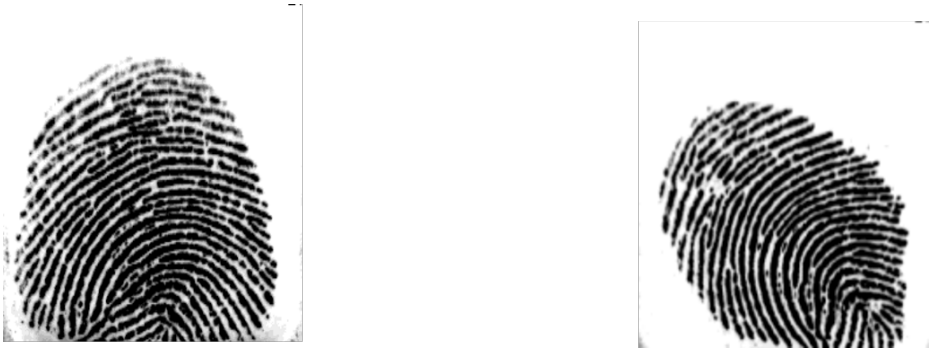


Figure 22: The left image is the original image, the one on the right is the desired result

The first step to realize such transformation would be to look at the minutiae patterns in both of the fingerprints to have a grasp of the difference between these two pictures. Minutiae patterns are made up of the ridges and the valleys that appear on the fingerprint. Basically to understand the differences between fingerprints, it is a really common study to extract the minutiae pattern first. We will not go into details of the minutiae patter extraction methods, as it is not intended for this project.

Here, we used one of the simplest methods, using MATLAB, 2 filters are performed on the images, the binarization filter and then the thinning filter, afterwards , for loops are used

to find the intersections and discontinuity, intersection would be called ridge endings.

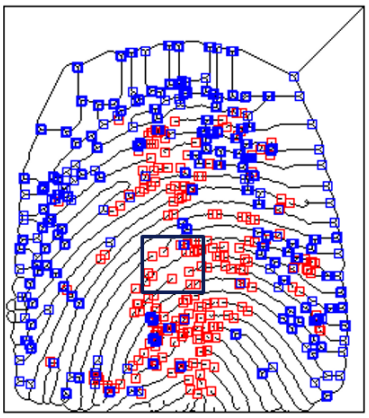


Figure 23: Minutiae patterns for the original image

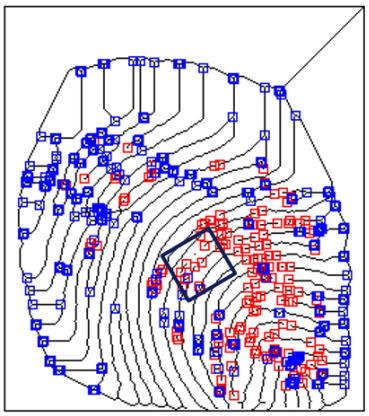


Figure 24: Minutiae patterns for the transformed image

Based on the minutiae patterns, we observed that the right picture might be the result of a "shear" like translation and some degree of rotation.

To begin with, our first intuition was to perform a simple affine transformation where we would alter the positions of the 3 points, shown below:

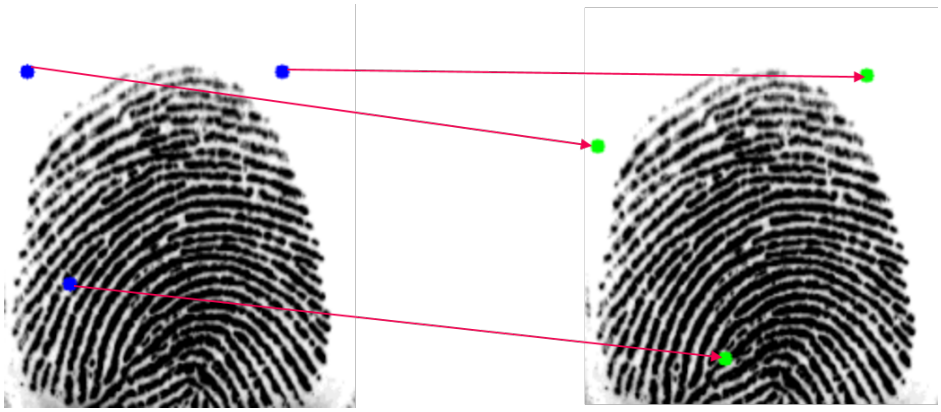


Figure 25: The location of the three points that construct the affine matrix

These three points will create the affine matrix needed for the transformation, and the result was as follow:

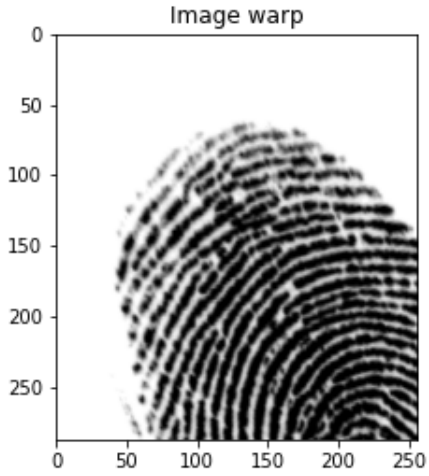


Figure 26: Image warp

As can be seen, the translated picture, does not quite convey the same "shear" like translation that we want. we need the picture to stretch out wards but in a different magnitude in different directions, which is why we performed a perspective affine transformation:

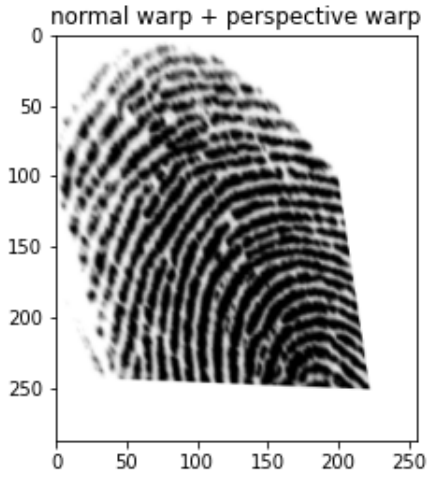


Figure 27: Perspective Image Warp

Now, we have achieved the "shear" like transformation that we desire, however, the image does not match the frame, therefore we need to perform a translation to make the image match the frame:

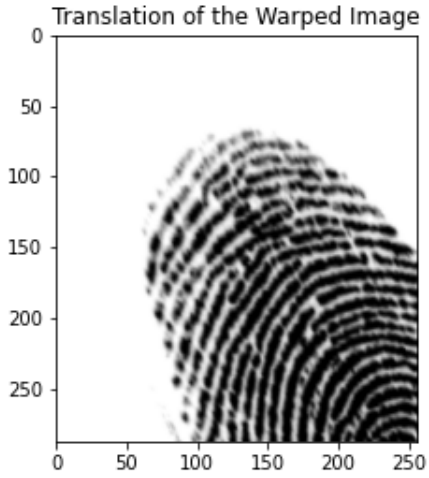


Figure 28: Translation after performing the Image Warp

The problem right now is that the orientation of the tip of the finger is not the same as we would want, which is why we need to rotate the fingerprint about 15 degrees around its center, and we get the following result:

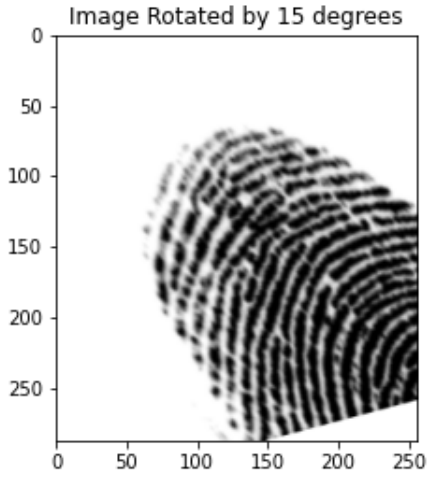


Figure 29: Rotation around the center of the Image

Again, the image does not match the frame, so another translation is added:

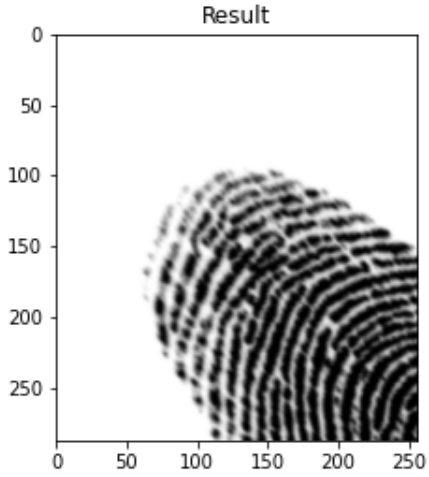


Figure 30: The final output

In all of these affine transformations, we tried all the different interpolations, but at the end we saw no difference with the results. Therefore, to figure out which interpolation actually gives the best result, we down sampled the result with different interpolations and re-sampled it again to visualize which one gave a more clear out put.

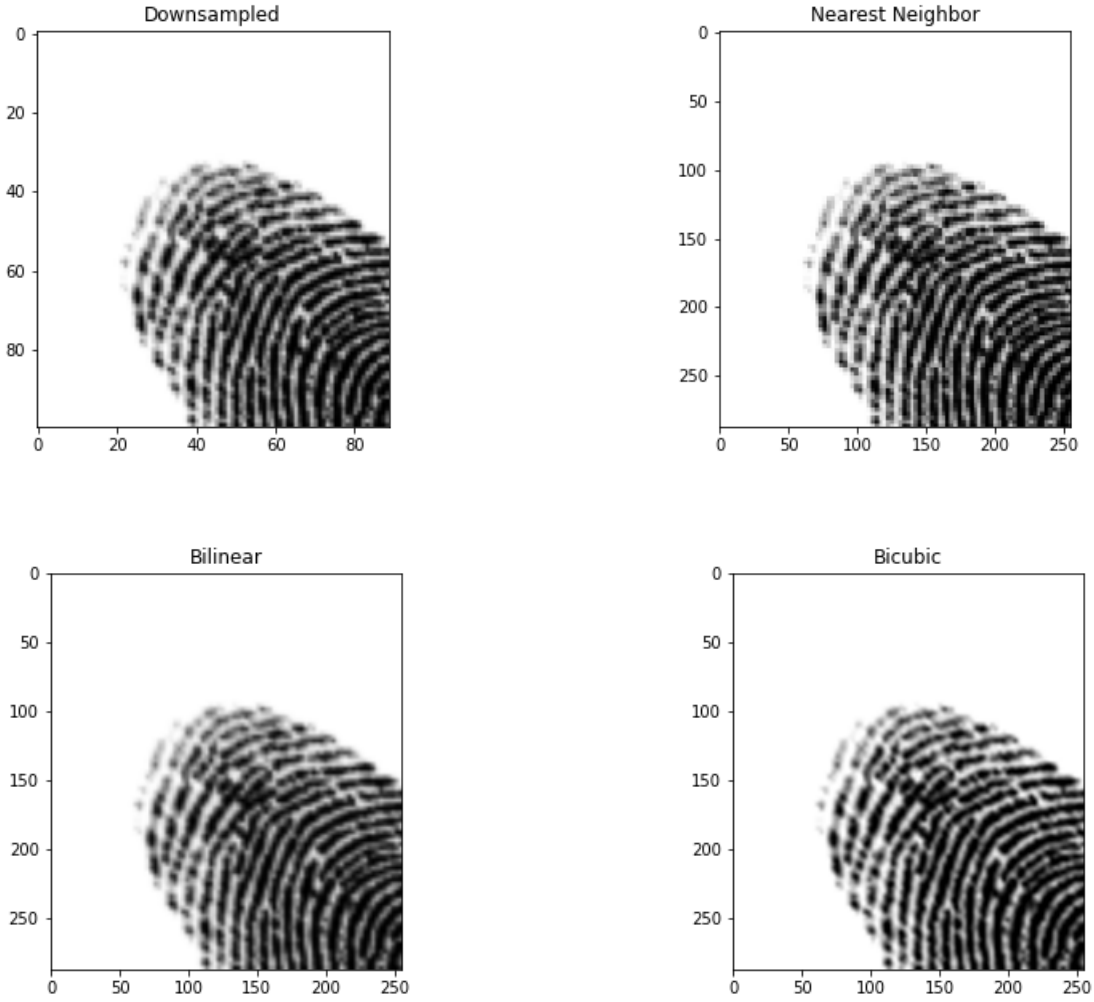


Figure 31: Display of the effect of different interpolations

As can be seen from the pictures above, the bicubic interpolation gives the best result. Then the nearest neighbor would be a valid choice. However, the bilinear interpolation does not perform well as the others.

5 Linear Filtering

5.1 2D Convolution Operation

Linear filtering in image processing "is filtering in which the value of an output pixel is a linear combination of the values of the pixels in the input pixel's neighborhood". Normally,

linear filtering uses the convolution operation.

The discrete 2D convolution operation normally represents the effect of one image on another. More precisely, given a matrix, we can apply a kernel to that matrix through convolution, whereby the kernel is usually much smaller than the original matrix, and then we get a resulting matrix that is the modification of the original one by the kernel.

This operation is usually definite for the center values of a given matrix; however, for the edge values, there are several ways to apply the kernel, depending on our desired result. For instance, one could replace the values of the kernel that spill out when places on top of the edge values with zeros or other values, or one could simply disregard them.

The simplest example of such an operation is the blurring kernel whose entry values are all the average between the number of entries (or simply all ones). For example, if we have a 3x3 kernel, then a blurring kernel of that size would have all entries of $\frac{1}{9}$. This process is mainly the averaging of each pixel with its neighboring pixels which creates the desired blurring effect.

Another very common example of a kernel that could be used is the Gaussian kernel. This kernel is the same idea as the Gaussian normal distribution, whereby the center of the kernel has the highest values, and the values decrease as we move further from the center.
It is important to note that the Gaussian kernel is an isotropic kernel, which means that the behavior of the function is the same in any direction.

5.1.1 Naive algorithm

In the naive algorithm, the problem with the edges will be improved by using a built-in function in OpenCV to extend the borders of the input image by $\frac{KernelSize-1}{2}$ with white pixels. Then, we will go through all the pixels in the newly created image to complete the 2D convolution operation. The result is generated using the following 5x5 mean blur kernel

$$\text{kernel} = \begin{pmatrix} 0.04 & 0.04 & 0.04 & 0.04 & 0.04 \\ 0.04 & 0.04 & 0.04 & 0.04 & 0.04 \\ 0.04 & 0.04 & 0.04 & 0.04 & 0.04 \\ 0.04 & 0.04 & 0.04 & 0.04 & 0.04 \\ 0.04 & 0.04 & 0.04 & 0.04 & 0.04 \end{pmatrix}$$

We also generated results by using the built-in function filter2D in order to compare it to our results using the kernel. As we can see from the below results, the outcome seems to be quite similar.

(a) Original Image (b) 2D Convolution (c) filter2D

Figure 32: Blurring Comparisons

For the complexity, if the image is of size n x m, the kernel is of size k_n x k_m, then the complexity is n x m x k_n x k_m since we have for loop in our code.

5.1.2 DFT and convolution

We have the following expression for 2D circular convolution:

$$f(x,y) * h(x,y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m,n)h(x-m, y-n) \quad (3.1)$$

for $x = 0, 1, 2, ..., M-1$ and $y = 0, 1, 2, ..., N-1$. Eq.(3.1) gives one period of 2D periodic sequence. The 2D convolution theorem is given by the expression:

$$f(x,y) \star h(x,y) \Leftrightarrow F(u,v)H(u,v) \quad (3.2)$$

and,

$$f(x,y)h(x,y) \Leftrightarrow F(u,v) \star H(u,v) \quad (3.3)$$

where F and H are obtained using the Discrete Fourier Transform, and the double arrow is used to indicate that the left and right sides of the expressions constitute a Fourier transform pair.

Eq.(3.2) states that the inverse DFT of the product of F and H yields the 2D spatial convolution of f and h. Similarly, the DFT of the spatial convolution yields the product of the transforms in the frequency domain. This equation is fundamental in linear filtering, which is why we found it necessary to apply it in our approach. Therefore, we apply perform the DFT of f and of h respectively, multiply them in the frequency domain, then perform the Inverse DFT of their product.

We can thus finally get the result of the 2D convolution of f and h.

However, we must note that wraparound error might occur in the process, which could potentially cause distortions near the edges of the image. If we would like to obtain the same convolution result between the "direct" convolution approach and the DFT approach, we must minimize the potential for having this error. Thus, we must pad the functions in the DFT before computing the transform.

Visualizing a similar example in 2D would be difficult, but we would arrive at the same conclusion regarding wraparound error and the need for appending zeros to the functions.

Figure 33 shows the result generated by this approach. The kernel we used here is the same as in the "direct" 2D convolution - a 5x5 mean blur kernel.

Figure 33: 2D Convolution Using DFT

For complexity in DFT, we used a built-in function.

5.2 Simulation

5.2.1 Motion Blur

There are many reasons for fingerprint image degradation or fingerprint image blur. In this section, we are concerned with motion blur, which is the blur caused by the relative motion between the finger sensor and the subject's finger during recognition. The Motion Blur effect is a filter that grasps the effect of the moving state of the object (the finger in our case), so that the image produces a dynamic effect. This filter is what we used to simulate the motion blur of our fingerprint.

Mathematical understanding: Suppose that an image $f(x.y)$ undergoes planar motion and that $x_0(t)$ and $y_0(t)$ are the time-varying components of motion in the x and y directions, respectively. The total exposure at any point of the recognition is obtained by integrating the instantaneous exposure over the time interval during which the sensor is processing the fingerprint. Thus, considering T to be the duration of the exposure, it follows that

$$g(x, y) = \int_0^T f[x - x_0(t), y - y_0(t)]dt$$

where $g(x, y)$ is the blurred image.

Since images are blurred in a specific direction and at a specific intensity, then mathematically, the y-axis is positive upwards. However, in image processing, the y-axis is positive downwards. So, after deciding on the direction angle, we should first rotate it by 180° along the square, and then continue our application in the specified direction.

It is important to note that Motion Blur does not simply imply moving the image up or down, but copying and shifting pixels in certain way within a small distance in order to give the desired effect. Simplified, one can say that this effect appears to have multiple copies of the fingerprint stacked on top of each other in a desired direction, and then averaged. Figure 34 shows some results yielded by our program, whereby, we can see the differences caused by changing the motion angle and distance.

(a) angle $= 90°$, distance $=$ 10

(b) angle $= 180°$, distance $= 30$

(c) angle $= -90°$, distance $= 30$

Figure 34: Motion Blurs with Varying Angle and Distance

5.2.2 Using varying kernel in an image

Our purpose here is to create a varying kernel from the boundary to the barycenter.
The idea is that we first blur the original image with one kernel, then we extract a rectangle area from the original image and extract the same size area from the blurred image. After that, we replace the area from the original image by the area extracted from the blurred image. We keep repeating this process from the boundary to the barycenter. More importantly, the blur step should be done on the original image, and the rectangle area we extract should have the same ratio with the original image. We can see the result generated from the program in Figure 35.

Figure 35: Motion Blurs with Varying Angle and Distance

The base kernel we used here is the same 5x5 mean blur kernel as above. We changed the overall sum of elements in the kernel from 2 to 1 for 20 times linearly. Therefore, we can see the blur effect is gradual and increases near the boundary.

5.2.3 Kernel

For the choice of kernel, we tested our different kernels and finally ended up with Gaussian kernel as it yielded the most ideal results for us. We changed the overall sum of entries in the 5x5 deviation 15 Gaussian kernel from 10 to 1 and generated the results as follows:

(a) sum = 10 (b) sum = 3 (c) sum = 2

(d) sum = 1.6 (e) sum = 1.3 (f) sum = 1

Figure 36: Gaussian Blur with Varying Sum of Elements

After observing the varying effect on the original image as the sum changes, we decided to choose 10 different sums for 10 different areas in the original image. Sum 10 is for the boundary, 1 for the barycenter. However since the blur change is not linear from boundary to barycenter, we will build a function as coefficient to multiply with our base 5x5 deviation 15 Gaussian kernel.

5.2.4 Exponential curve fitting

For the function we mentioned above, we chose the exponential function, so we created two arrays:

$$t = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$$

$$f = (10, 3, 2, 2.2, 1.9, 1.8, 1.6, 1.3, 1.2, 1.0)$$

Then, we applied the exponential curve fitting method in Python using the two arrays t and f as input, which yielded the following output:

$$a = 1.6782992598876423 \quad b = -0.1813496535867255$$

The general function form is as follows:

$$f(t) = e^{a} e^{b \times t}$$

Hence, the final function is:

$$f(t) = e^{1.6782992598876423} e^{-0.1813496535867255 \times t}$$

After checking the values, we found the results to be quite satisfactory. (better add a function plot to compare).

Thus, our final result is as follows (we also added a little bit of motion blur to our final result for an increased effect):

(a) Desired Result

(b) Our Result

(c) Our Result with Motion Blur (angle $= 90°$, distance $= 5$)

Figure 37: Final Result

5.3 Restoration

The purpose of this section is to restore a fingerprint that has been distorted by motion blur back to its original form in for better recognition. As we used 2d convolution DFT

5.3.1 Blurring matrix operator

The blurring kernel is:

$$k = \begin{pmatrix} 0 & 0 & 0 & 0 & \frac{1}{7} \\ 0 & 0 & 0 & \frac{1}{7} & 0 \\ 0 & 0 & \frac{1}{7} & 0 & 0 \\ 0 & \frac{1}{7} & \frac{1}{7} & 0 & 0 \\ \frac{1}{7} & 0 & \frac{1}{7} & 0 & 0 \end{pmatrix}$$

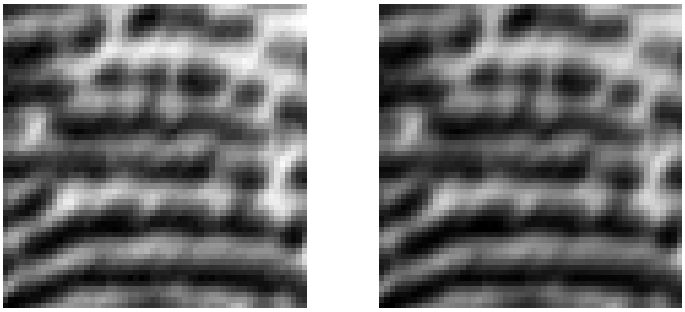
Restoration In this case, we are given a 5x5 kernel k from which a larger matrix K of size 3600X3136 is built. Convolution as expressed in starter 3 is a linear operation, which means that it could be rewritten as a matrix operation of the form:

$$f_b = Kf$$

where f is the unknown vectorized clean acquisition of length 3600, f_b is the vectorized corrupted acquisition of length 3136, and K is the blurring matrix operator of size (3600,3136).

In order to define matrix K from the kernel k, we have to vectorize our original and resulting acquisition, whereby f_b is the corrupted (resulting) on and f is the clean (original) one. We know that the clean fingerprint image is of size 60 * 60, and f is the vectorized clean acquisition of length 3600. So, we transformed the image into a vector of length 3600. For the matrix K, we have 3136 columns, whereby the columns represents the pixel (i,j) from (0,0) to (56,56). Thus, for each pixel, we need the 5*5 grid of pixels near it, which means we need to set the seven pixels to a value of $1/7$. Because we have vectorized the picture into length of 3600, we change the position (i,j) to i*60+j, and we need to set the pixel (i,j+4),(i+1,j+3),(i+2,j+2),(i+3,j+1)(i+3,j+2)(i+4,j)(i+4,j+2) to a value of $1/7$, which is essentially how we created our code for this part.

Eventually, the result we generated is pretty close to the corrupted fingerprint that we desired to reach.



(a) Desired Result (b) Our Result

Figure 38: Sample Restored Result

After comparison, we think we have succeeded to build matrix K.

As can be seen, the big matrix K, is an extremely parsed matrix whose columns are not linearly dependant, hence the problem is ill-posed.

In the next section, we have to find a linear solve, to solve the least square estimate for f, which we chose to use the SVD and QR decomposition to do so.
In general, solving least square problems $Ax = b$, for a given tall matrix $A_{m\times n}$, m>n, is done by the following methods:
(1) Solve the normal equation $A^TAx = A^Tb$
(2) Find QR factorization $A = QR$ and solve $Rx = Q^Tb$
(3) Find SVD factorization $A = U\Sigma V^T$ and solve $x = V\Sigma^{-1}U^Tb$
In terms of figuring out which methods to use, there are several factors to keep in mind:
In terms of speed, (1) is the fastest. However, the condition number is squared and thus

less stale. The QR factorization is more stable but the cost is almost doubled . The SVD approach is more appropriate when A is rank-deficient. In our Question, $A = K$,$b = f_b$, and $x = f$.

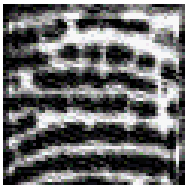


Figure 39: Restoration using the SVD decomposition

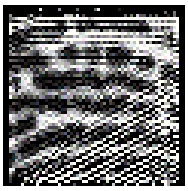


Figure 40: Restoration using the QR decomposition

However, both of these methods took a long time to run, the SVD decomposition gave better results than the QR decomposition. We believe, this is due to the fact that in the SVD approach, the singular values (that contain the most crucial information) that are close to zero are considered as "noise", therefore, this would be a good criteria to restore the image.

In the next step, we will add more than enough linearly independent columns that the matrix K, hence, making the system to be well-posed.
by changing the values of λ and using the SVD decomposition, we get the following results:

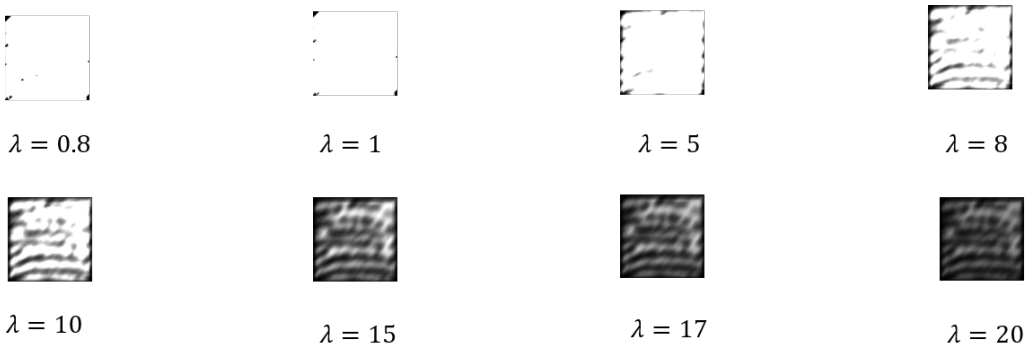


Figure 41: The effect of λ in the restoration

As can be seen, the most suitable value for λ is 15 and as decrease this value, we are basically mapping more pixels to "white" and by increasing the the value from 15 , we are mapping more pixels to "black" which is not suitable, and all of this was due to the new reformation of the system.

6 Conclusion

The aim of this project was to create a fingerprint image processing toolbox which allows us to simulate scenarios in which fingerprint recognition may be distorted, and attempt to solve the issues causing this corruption using mathematical models.

By doing some research on some mathematical notions and assembling concepts from numerous books and projects, we were able to come up with somewhat satisfying results. However, we still believe that there could have been more improvements in the complexities of our codes as some of them did take us quite a while to run. Nonetheless, we believe that our codes have greatly improved since the intermediate ones as we were able to further develop and compare the program.

Eventually, we found that this modelling project has truly increased our interest in the analysis of digital fingerprints as it shows to have far more intricacies than one might assume. Finally, its great dependence on mathematical programming has also allowed us to view the connections that could be made with applied mathematics and day-to-day activities.

7 References

1. Bradski, G., Kaehler, A. (2015). *Learning OpenCV*. O'Reilly Media.

2. Chanklan, Ratiporn Chaiyakhan, Kedkarn Hirunyawanakul, Anusara Kerdprasop, Kittisak Kerdprasop, Nittaya. (2015). *Fingerprint Recognition with Edge Detection and Dimensionality Reduction Techniques*. 569-574.

3. Coste, A. (2012). *Affine Transformation, Landmarks registration, Non linear Warping*. Image Processing.

4. Glasbey, C.A., Mardia, K.V. (1998). A Review of Image Warping Methods. *Journal of Applied Statistics, 25, 155-171*

5. *Image Filtering Using Convolution in OpenCV*. LearnOpenCV. (2021, July 15). Retrieved from https://learnopencv.com/image-filtering-using-convolution-in-opencv/

6. Li, Z., Zhao, S., Wang, L. (2021). *Isotropic and Anisotropic Edge Enhancement with a Superposed-Spiral Phase Filter*. Optics express, 29(20), 32591–32602.

7. Shiffman, D. (2008). *Images and Pixels*. Processing.org. Retrieved from https://processing.org/tutoria

8. Szeliski, R. (2020). *Computer Vision: Algorithms and Applications*. Springer Nature.