CS-107: Mini-Project 1
Fingerprint recognition

Yanis De Busschere, Barbara Jobstmann, Jamila Sam

-

version 1.0

Contents

This document uses colors and contains clickable links. It is best to view it in digital format. 1

1 Presentation

Whether it is for a police investigation or to unlock our phones, the software processing fingerprints must be able to recognize and differentiate them. The goal of this project is to implement a program capable of comparing fingerprint images. Concretely, given fingerprint images, it is a question of being able to say if they come from the same finger.

1.1 Structure of a fingerprint

Fingerprints are unique to each individual. However, their general structure varies very little. Indeed, only three main structures are defined for 95% of individuals: loop structure (fig. 1), whorl structure (fig. 2) and arch structure (fig. 3).

Figure 1: Footprint with loop structure

Figure 2: Footprint with whorl structure

Figure 3: Footprint with arch structure

Therefore, it is very difficult to compare two fingerprints by looking at their general structure.

However, there are distinctive points on all fingerprints:

- global singular points:
  - nucleus or center (fig. 4), i.e., the place of convergence of the streaks,
  - delta (fig. 6), place of divergence of the cutmarks;
- the local singular points, called **minutiae**. There are about ten different types, but in this project we will only be interested in the two main ones: Terminations (fig. 5) and Bifurcations (fig. 7). In this project, we will limit ourselves to the use of these two types of minutiae which prove to be sufficient to obtain correct comparison results in a large

number of cases. Using more would not significantly increase the precision of our program while requiring a more complex algorithm than the one we are going to implement.

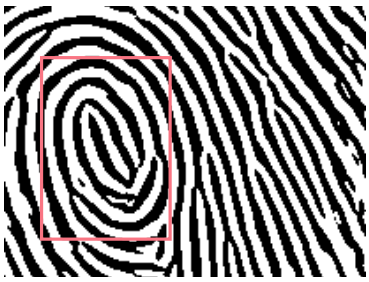

Figure 4: Core

Figure 6: Delta

Figure 5: Termination

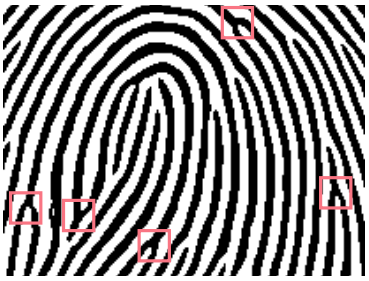

Figure 7: Bifurcation

1.2 How to extract the minutiae?

Extracting the minutiae is one of the most important steps in comparing fingerprints. This treatment is carried out in three stages:

1. the pre-processing of images, which consists of improving the quality of the image and transforming it into binary format (see Binary format and shade of gray). **This part does not fit into the framework of the project**.
2. the Skeletonization (or thinning), which allows easier extraction. This step consists in transforming all the lines of the image so that they have a thickness of one pixel. A pixel is the basic unit of an image, it is the smallest point of color. This transformation maintains the structure of the image while simplifying it for easier processing.
3. and, the localization of the minutiae and the calculation of the orientations.

You can view these steps in fig. 8.

The images will be given to you already pre-processed. You will therefore only need to perform steps 2 and 3.

1.3 How do I compare fingerprints?

There are many methods for comparing fingerprints. We are going to use one of the simplest, but also one of the most recognized methods: comparing the minutiae.

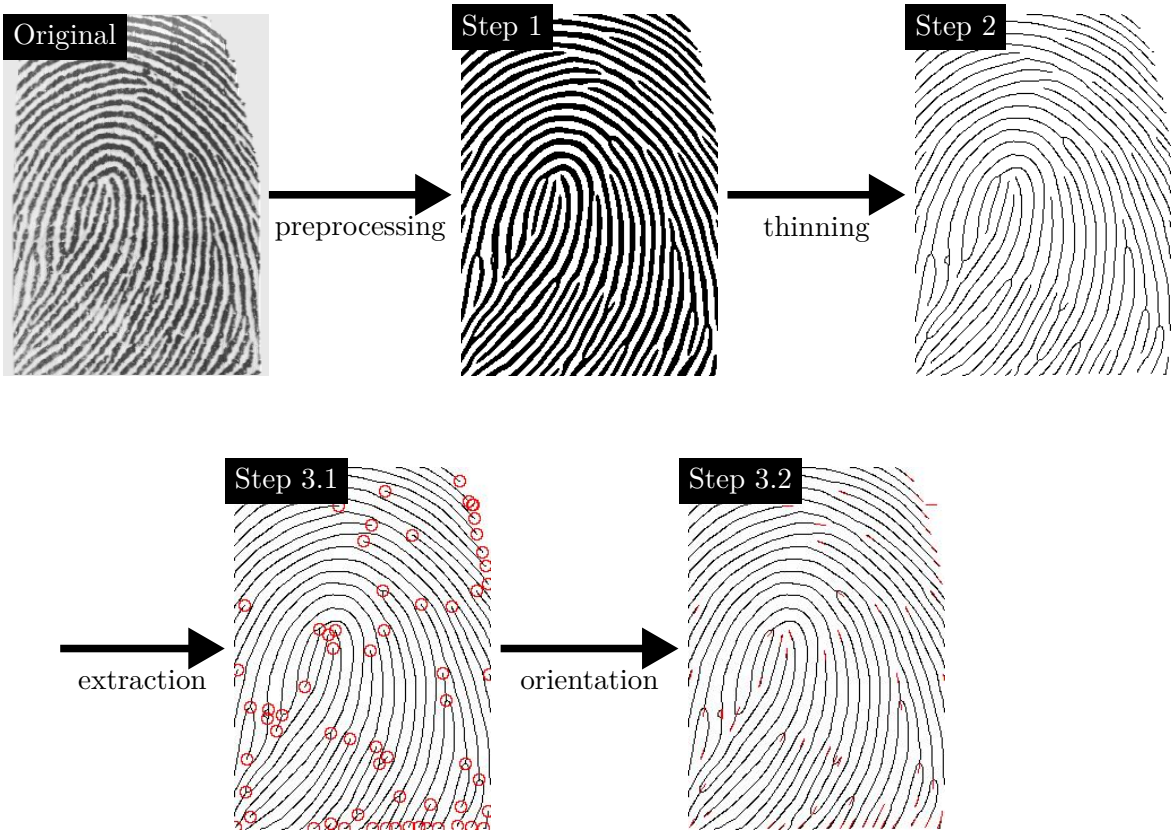

Figure 8: Stages of minutiae extraction

To compare two fingerprints with this method, it is therefore necessary first to find these distinctive points for the two images that we want to compare. Once the distinctive points have been found, their coordinates and their orientation are extracted. It is these coordinates and orientations that we are going to compare. If the two images contain a certain number of points which have the same coordinates and the same orientations we will consider that the fingerprints are identical, that is to say, that they belong to the same finger, if not, we will say they are different.

This method takes into account the possible differences of rotation and translation in an image. Thus, if in an image, the fingerprint is skewed or if it is not centered, the algorithm will still be able to detect which finger it comes from.

2 Structure and code provided

2.1 Structure

The project, in its mandatory part, is divided into three parts:

1. the implementation of the functions necessary for the skeletonization of a binary image,
2. the implementation of the functions allowing to extract the coordinates and the orientation of the minutiae,
3. the implementation of functions allowing to say if two sets of minutiae come from the same finger.

- All the mandatory code must be done in the file `Fingerprint.java`.
- The headers of the methods to be implemented are provided and **must not be changed**.
- The file provided `SignatureChecks.java` gives the set of signatures not to be changed. It serves in particular as a control tool during submissions. It will allow all the required methods to be used **without testing its operation**[a]. Check that this program **compiles** well, before submitting.
- You will find in the folder provided `resources/`, **some files you can use** as data set **to test your methods**.

[a] This is to verify that your signatures are correct and that your project will not be rejected upon submission.

2.2 Code provided

As the manipulation of files/images and windows is tedious and too advanced for this course, part of the code is given to you. The file `Helper.java` simplifies your interaction with images and conversions:

- `boolean[][] readBinary (String)` and `boolean writeBinary (String, boolean[][])` allow to read and write images whose format is an array of pixels in binary format (see Binary format and shade of gray).
- the methods `int[][] readARGB`, and `boolean writeARGB (String, int [][])` allow to read and write images whose format is an array of pixels in ARGB format (see ARGB representation). Using ARGB images is not strictly necessary for the project, but it can be useful to verify that they are working properly. You can for example use the methods described later to draw circles, lines or minutiae with colors,
- the methods `void show (int [][] array, String title)` and `void show (boolean [][] array, String title)` allow to display images in ARGB and binary formats,

- the methods `int [][] fromBinary (boolean [][])` and `boolean [][] toBinary (int [][])` allow switching from ARGB to binary and vice versa,
- methods allowing to debug your code are given to you. These methods allow you to draw lines, circles and minutiae: `void addLine (/*...*/) void addCircle (/*...*/)`, `void drawMinutia (/*...*/)`.

2.3 Tests

Important: It is your responsibility to verify the correct behavior of your program. The file provided `Main.java`, partially written, will be used to test your developments in a simple way. It is up to you to complete `Main.java` by invoking your methods there appropriately to verify that there are no errors in your program.

When correcting your mini-project, we will use automated tests, which will pass randomly generated inputs to different functions in your program. There will therefore also be tests verifying how the *special cases* are treated. Thus, it is important that your program correctly processes any *valid* input data.

Regarding the management of error cases, it is helpful to test the input parameters of functions; eg, check that an array is not zero, and/or is of the correct dimension, etc. These tests generally make debugging easier, and help you reason about the behavior of a function. **We will assume that the arguments of the functions are valid** (except when exceptions are explicitly mentioned).

To guarantee this assumption, we invite you to use the Java assertions[1]. An assertion is written in the form:

```
assert expr;
```

with `expr` a Boolean expression. If the expression is false, then the program throws an error and stops, otherwise it continues normally. For example, to verify that a method parameter `key` is not `null`, you can write `assert tab! = null;` at the start of the method. An example of using an assertion is given in the shell of the method `getNeighbors` in the file `FingerPrint.java` .

> Assertions must be enabled to work. This is done by launching the program with the option "-ea" [Clickable link here].

Finally, to test the general functioning of your program, you have several resources:

- in `resources/fingerprints`, you have a set of preprocessed fingerprints. The name of the images consists of two numbers. The first corresponds to the finger number and the second to the fingerprint number. For example `1_1.png` and `1_2.png` are prints 1 and 2 of the first finger while `2_1.png` is the first image of a different finger. This database is complex. The program we are going to do will not be perfect and may be wrong, that is to say, it may say that two fingerprints come from the same finger when it is wrong, or,

[1] We will have the opportunity to come back to them in detail, but their use is intuitive enough that we can already use them.

conversely, that he will be able to say that they come from different fingers while they belong to the same finger. The program we are going to do will not be perfect and may be wrong, that is to say, it may say that two fingerprints come from the same finger when it is wrong, or, in the other direction , that he will be able to say that they come from different fingers while they belong to the same finger.

- in `resources/original_fingerprints`, you have the same set of fingerprints, but they haven't been preprocessed. So you cannot use them in your project as they are. You are free to implement the preprocessing as a bonus if you wish, but it is not necessary to get the maximum score and it takes a lot of work and research.
- in `resources/test_inputs` and `resources/test_outputs`, you will find images to check how your code works with our results. You will find in particular an image `skeletonTest.png`. This is a fingerprint that has already undergone skeletonization. This will allow you to go to step 2 and 3 without having completed the first one entirely.

Here is a summary of the main instructions/indications to follow for the coding of the project:

- The method parameters will be considered error-free, unless explicitly stated otherwise.
- The headers of the provided methods must remain unchanged: the file `SignatureCheck.java` must therefore not contain any **compilation** errors at submission time.
- Apart from the imposed methods, you are free to define any additional method that seems relevant to you. **Modularize and try to produce a clean code!**
- It is your responsibility to verify the correct behavior of your program. However, we provide the file `Main.java` , showing how to invoke your methods to test them. The examples of tests provided are not exhaustive and you are authorized/encouraged to modify `Main.java` to do more checks. Your efforts in terms of testing will also be rewarded.
- Your code must respect the usual naming conventions.
- The project will be coded without the use of external libraries. If you have any doubts about the use of such or such library, ask us the question and especially pay attention to the alternatives that Eclipse offers you to import on your machine.
- Your project **should not be stored in a public repository** (of type github). For those of you who are familiar with git, this may also be useful: https://gitlab.epfl.ch/.

3 Task 1: Skeletonization

The first task of this project is to program an algorithm capable of tracing the skeleton of an image. That is, a function which, given a binary image, is able to refine all of its strokes to have a thickness of one pixel. The goal of this step is to simplify the image as much as possible while keeping its structure. This will greatly simplify the extraction of minutiae.

You will be working with images that are already pre-processed (i.e. you will start from step 1 of figure 8). Each of these images is stored in a two-dimensional array of booleans. The boolean value `true` represents a black pixel and the value `false`, a white pixel. You will notice that the coordinate system usually used to represent an image is unusual. **The origin is located in the top left corner of the image**. Each line of the image corresponds to an array of booleans. We will therefore proceed as follows to access the pixels of an image that is 100 pixels wide and 100 pixels high:

```
image [0][0] = true; // access to the top left pixel
image [99][0] = true; // access to the bottom left pixel
image [0][99] = true; // access to the top right pixel
image [99][99] = true; // access to the pixel at the bottom right
```

The class provided, `Helper.java` , is encoded according to this convention.

3.1 Preliminary methods

The algorithm operates on all black pixels having eight neighbors, that is, all pixels not lying on the edge of the image. For each of these pixels, the algorithm will determine if it is essential to the integrity of the fingerprint or if it can be deleted.

To determine if a pixel P is essential (relevant), it suffices to look at its eight direct neighbors $(P_0, \ldots, P_7)$. By convention, they are called as follows:

P_7	P_0	P_1
P_6	P	P_2
P_5	P_4	P_3

3.1.1 Method `getNeighbors`

Start by writing the method `getNeighbors` able to extract the neighbors of a pixel in the desired order: P_0, P_1,..., P_7.

```
boolean [] getNeighbors (boolean [][] image, int row, int col)
```

This function takes as parameter the image as well as the coordinates of the pixel from which we want to extract the neighbors. The return type is an array of booleans. The element at index 0 must contain the value of P_0, the element at index 1 must contain the value of P_1 and so on. The array must contain exactly 8 elements. It is your responsibility to test if all neighbors exist in the image. If one of the pixels is not in the image, it will be returned as if it existed

and was white. If the column or row passed in as parameters are not in the image, the function must return `null`. You will find two examples in fig. 9.

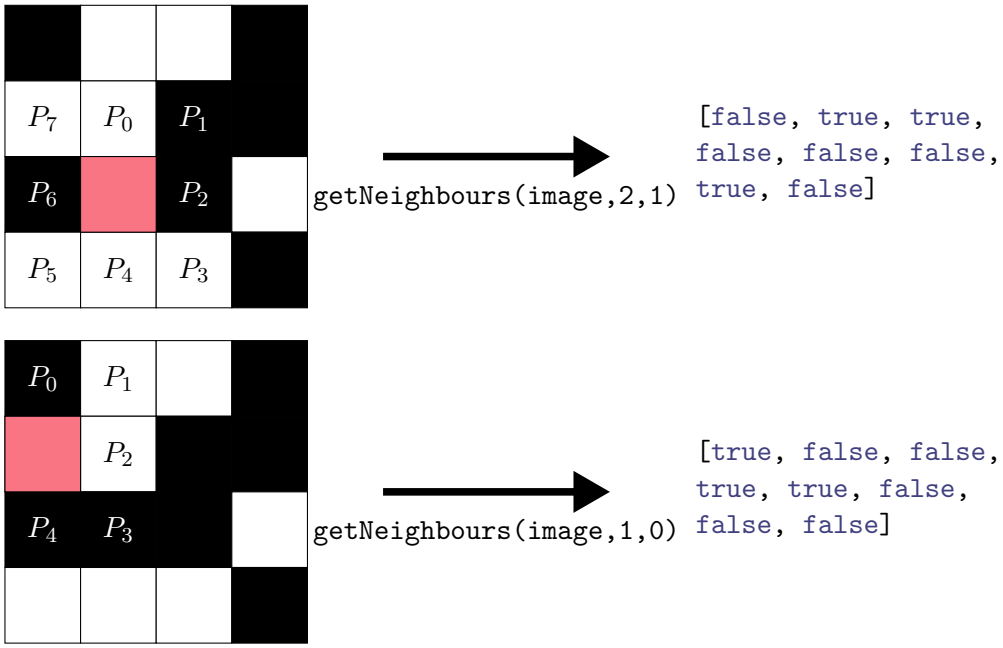

Figure 9: Example result of `getNeighbors`

3.1.2 Method `blackNeighbors`

Now write the method `blackNeighbors`.

```
int blackNeighbors (boolean [] neighbors)
```

This method will be used to count the number of black pixels in the array resulting from a call to the Method `getNeighbors`. It therefore returns an integer between 0 and 8. See fig. 10 for an example. The red pixel has 3 black neighbors.

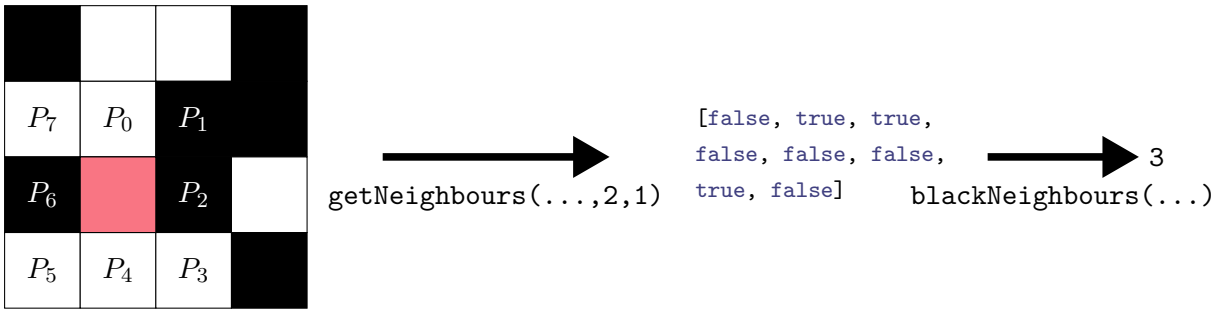

Figure 10: Example result of `blackNeighbors`

3.1.3 Method `transitions`

To determine if a pixel is essential, we calculate the number of transitions from white to black in the sequence P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_0.

For example, in the sequence: `[white, black, black, white, white, white, black, white, white]` the number of transitions is 2. We go once from white to black when we go from P_0 to P_1 and once again when we go from P_5 to P_6.

Visually, this example translates as follows (see fig. 11).

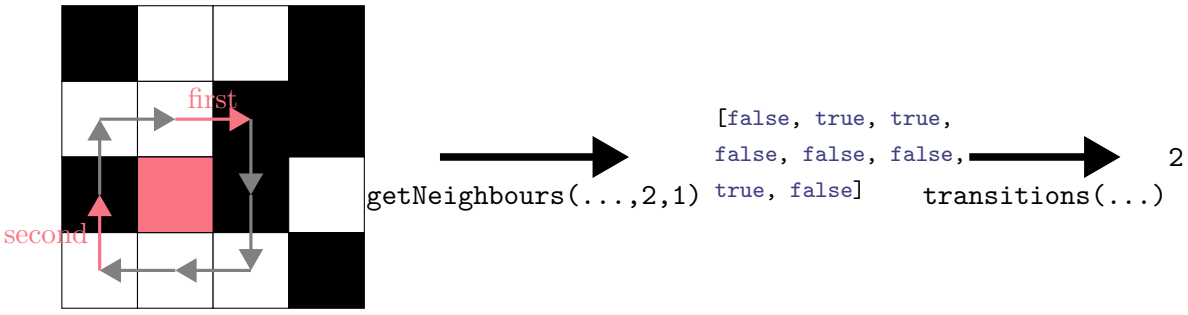

Figure 11: Example number of transitions

Write the method `transition` which, like `blackNeighbors` takes the result of `getNeighbors`. This method will return the number of transitions for the pixel to which the neighbors belong.

```
int transitions(boolean[] neighbours)
```

3.1.4 Method `identical`

The skeletonization algorithm will iteratively remove irrelevant pixels until the image stabilizes (reaches skeletonization). It is therefore useful to have a method capable of telling whether one image is identical to another. This will allow you to know if an image has not changed from one iteration to the next. We will then know that all the pixels are relevant and that the algorithm can stop. Write for this the following method which returns `true` if the images passed in parameter are identical and `false` otherwise.

```
boolean identical (boolean[][] image1, boolean [][] image2)
```

3.2 Main methods

With the help of the functions implemented previously, we are now able to tell if a pixel is important for the integrity of the fingerprint by testing certain conditions. These conditions are grouped in two steps and are tested independently, that is to say, all pixels are first tested with the conditions of the first step which will remove some irrelevant pixels. Then, all the pixels are tested with the conditions of the second step and thus other irrelevant pixels are eliminated. The algorithm performs the two steps one after the other. Both steps are performed exactly

the same number of times. After having carried out the two steps, it is checked if a pixel has been modified. As long as a pixel has been modified, the two steps are carried out again.

The algorithm takes as a parameter an original image, which must remain unchanged, and produces a clean image of the pixels deemed irrelevant. The latter is initially identical to the original image and is then iteratively modified as follows:

> **Step 1**
> All pixels are tested. Pixels meeting all of the following conditions should be considered as irrelevant and removed (set to white) in the resulting image:
>
> 1. The pixel is black,
> 2. The array of its 8 neighbors is not null,
> 3. $2 \leq$ `blackNeighbors ()` ≤ 6,
> 4. `transitions ()` $= 1$,
> 5. P_0 or P_2 or P_4 is white,
> 6. P_2 or P_4 or P_6 is white.
>
> **2nd step**
> the image resulting from de previous step is considered as the initial image. Starting from this image, which must remain unchanged, a new resulting image must be computed where the pixels meeting all of the following conditions are removed (set to white) :
>
> 1. The pixel is black,
> 2. The array of its 8 neighbors is not null,
> 3. $2 \leq$ `blackNeighbors ()` ≤ 6,
> 4. `transitions ()` $= 1$,
> 5. P_0 or P_2 or P_6 is white,
> 6. P_0 or P_4 or P_6 is white.
>
> **Iteration**
> As long as there are changes in step 1 or 2, we repeat the two steps. To check if there has been a change, you can use the Method `identical ()`.

The algorithm will be divided into two methods:

- a main method that will implement the iteration and check if the image has changed after a call to both steps:

```
boolean [][] thin (boolean [][] image)
```

- and, a method that will apply step 1 or 2 and return a stripped image of irrelevant pixels (according to the criteria of each step):

```
boolean [][] thinningStep (boolean [][] image, int step)
```

  Note that only one method is used for both steps, because the treatments are very similar. The parameter `step` indicates which step is implemented (0 for step 1 and 1 for step 2).

Warning : to copy a two-dimensional array, do not use the predefined Method `Arrays.copyOf` unless you know exactly what you are doing. For the moment, prefer an explicit copy, made element by element, by means of two nested loops[2].

3.3 Tests

Use the program `Main.java` to test your code. The provided examples of tests are not exhaustive. **Complete them** as you see fit, to test all the methods developed in this first part of the project.

4 Task 2: Locate and calculate the orientation

In this part, it is now a question of finding the position and orientation of the minutiae using the skeleton of the fingerprint (see fig. 12 for an example of extraction).

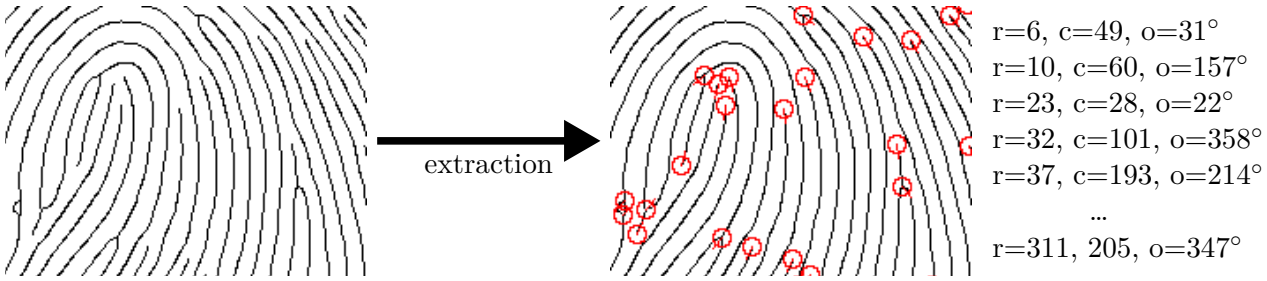

Figure 12: Example of minutiae extraction

To find the coordinates of the minutiae, we iterate over each pixel and determine whether it is a point of interest or not. Once a minutia has been found, its orientation is determined.

To calculate the orientation, you will need to write three utility methods:

- in section 4.1, you will write the method `boolean [][] connectedPixels (boolean [][] image, int row, int col, int distance)`
- in section 4.2, you will write the method `double computeSlope (boolean [][] connectedPixels, int row, int col)`
- in section 4.3, you will write the method `double computeAngle (boolean [][] connectedPixels, int row, int col)`

Thanks to these three methods, in section 4.4, you will have to write the function `int computeOrientation (boolean [][] image, int row, int col, int distance)`.

[2]We will come back a little later in the semester to the concepts of surface copy and deep copy (”deep vs shallow copy”)

Finally, in section 4.5, you will write the function `List<int[]> extract (boolean [][] image)` which iterates over each pixel, determines if it is a minutia and calls `computeOrientation (...)`.

4.1 Function `connectedPixels`

To calculate the orientation, we use pixels that are close to the minutia and are connected to it. For example, in figure13 on the left, the minutiae is in red and the pixels that interest us are in dark green. The remaining pixels, in black, are of no interest to us. They are not connected or are not close enough. A pixel is considered close enough if it is within the size square `2 * distance + 1` centered on the minutiae. `distance` is a parameter of the function. Please note that this region may exceed the image (overflow). Remember to take the necessary precautions.

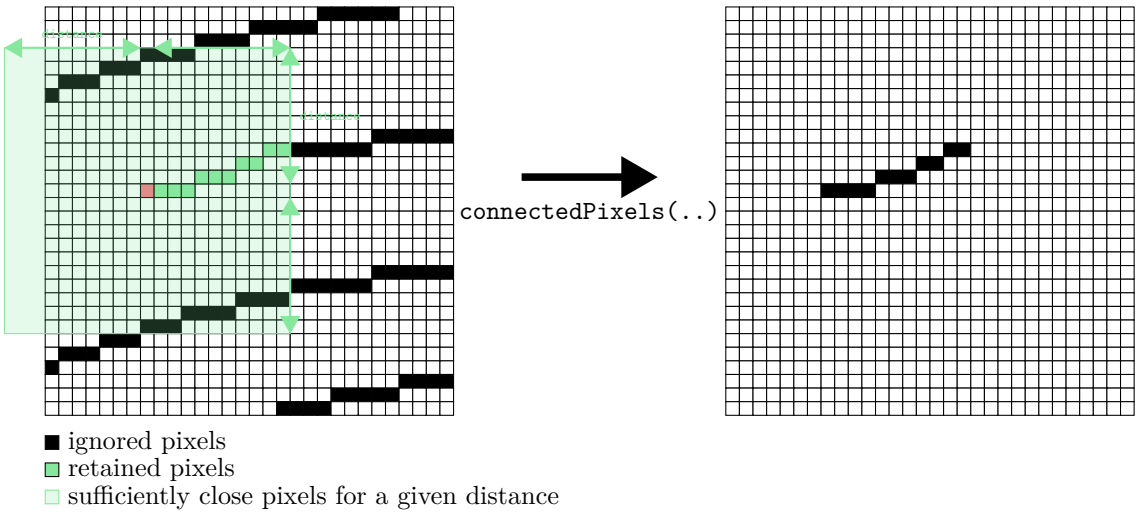

Figure 13: Pixels used to determine the orientation

We are going to write a function capable of finding all these pixels (which are in dark green or in red, see figure 13 on the left and the result on the right).

```
boolean [][] connectedPixels (boolean [][] image, int row, int col,
   int distance)
```

The function takes the image as well as the coordinates of the minutiae and the distance discussed above. It will return a two-dimensional array of Booleans. The table should be the same size as the image. The elements of this table must be `true` if

1. the pixel at these coordinates is black, and
2. the pixel is connected to our minutia, and
3. the pixel is in the square of size `2 * distance + 1` centered on the minutia.

Otherwise, the element should be `false`. See fig. 14 for an example.

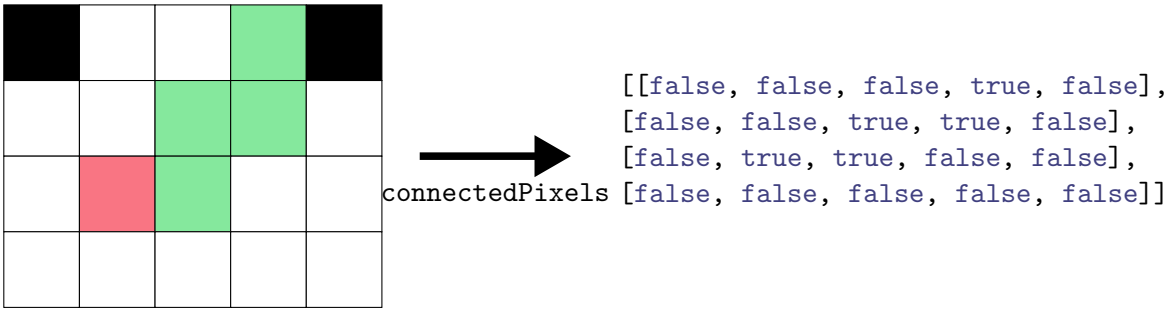

Figure 14: Example result of `connectedPixels` (image, 2, 1, 2): in this example, the image given as parameter is the left one (the result of `connectedPixels` has the same size as `image`)

Suggestion 1 : a simple algorithm for `connectedPixels` consists in iterating over the whole image, an undetermined number of times, using a `while` loop. At each iteration, we check for each pixel if it is black, if it is neighbouring a pixel already present in the array of connected pixels and if it is within the desired distance. If the pixel fulfill all these conditions, it is added to the array of connected pixels. A new iteration must start as long as new pixels are added. If after a full iteration, no pixel is added to the array of connected pixels, the process stops.

Suggestion 2 : if you have coded the MOOC assignment of week 4 (exercise 3, "sense of ownership"), an algorithm close to the one described in page 10 of the handout can be used to implement `connectedPixels` .

4.2 Function `computeSlope`

To calculate the orientation of a minutia m, we try to find the line which passes through m and which is as close as possible to the surrounding connected points. To find this line, we proceed mathematically using a method called linear regression (explanations are provided in the sidebar on the next page).

The line which gives the direction of the minutia m has the equation:

$$y = a \cdot x \text{ with } \begin{cases} a = \frac{\sum x \cdot y}{\sum x^2}, & \text{if } \sum x^2 \geq \sum y^2 \\ a = \frac{\sum y^2}{\sum x \cdot y}, & \text{if } \sum x^2 < \sum y^2 \end{cases}$$

where (x, y) are the coordinates (in the coordinate system taking its origin in m) of the black pixels in the array of connected pixels. These coordinates are defined in the usual mathematical reference (origin at the bottom left and axis y rising up).

To calculate the line it is therefore sufficient to determine the slope a. It is necessary to calculate the sums of the formulas above, but taking into account the fact that in the representation of the images used, the reference takes its origin at the top left (with the axis of y directed downwards). Let $[row_m][col_m]$ be the coordinates of the minutia in the image, by setting row the row with a green pixel and col its column in the image, we can calculate the coordinates x and y as follows:

$$x = col - col_m$$
$$y = -(row - row_m) = row_m - row$$

Note: the negation in the calculation of y comes from the fact that the axis of y does not vary in the same direction for the two marks.

Concrete example: in the fig. 14, the minutia m is on row 2 and column 1 in the image. To find its orientation, we must calculate the three sums used in the previous formula for all the green pixels. That is, the sum of x multiplied by y and the sums of x^2 and y^2.

More precisely, the green pixels in this example have for coordinates (x, y) in the usual frame whose origin is in m: (1, 0), (1, 1), (2, 1) , (2, 2) and therefore the sums are:

$$\sum x \cdot y = 1 \cdot 0 + 1 \cdot 1 + 2 \cdot 1 + 2 \cdot 2 = 7$$
$$\sum x^2 = 1^2 + 1^2 + 2^2 + 2^2 = 10$$
$$\sum y^2 = 0^2 + 1^2 + 1^2 + 2^2 = 6$$

If $\sum x^2 \geq \sum y^2$, the line which gives the orientation in the fig. 14 finally for slope $a = 7/10$.

Write the function that calculates this slope:

```
public static double computeSlope (boolean [][] connectedPixels,
   int row, int col)
```

This function takes parameters similar to those of `connectedPixels` but instead of the full image of the fingerprint, it takes the one resulting from calls to `connectedPixels`. That is, the one containing only the pixels that are close and connected to the minutiae. The distance parameter is no more necessary.

There is a special case where the connected pixels are best approximated by a vertical line ($\sum x^2 = 0$). In this case, the function must return the value `Double.POSITIVE_INFINITY`.

Explanation - Simple linear regression *Understanding the mathematical details is not necessary for the success of the project. You can just apply the previous equation. The explanation is intended for the most curious among you.*

In this method, we try to find a line of equation $y = a \cdot x + b$ which is closest to our points. You can observe in fig. 15 an example of a straight line which passes close to the points and in fig. 16 a straight line which remains far from the points.

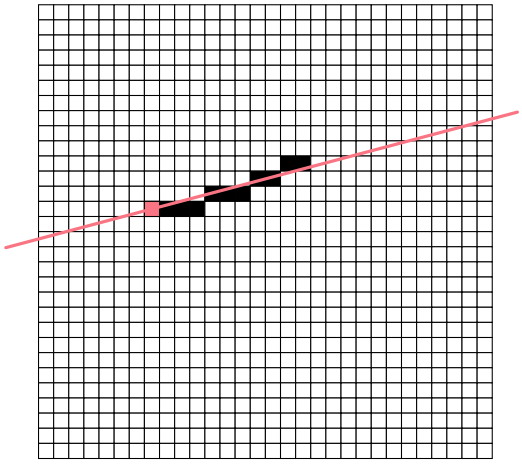

Figure 15: Line close to our points

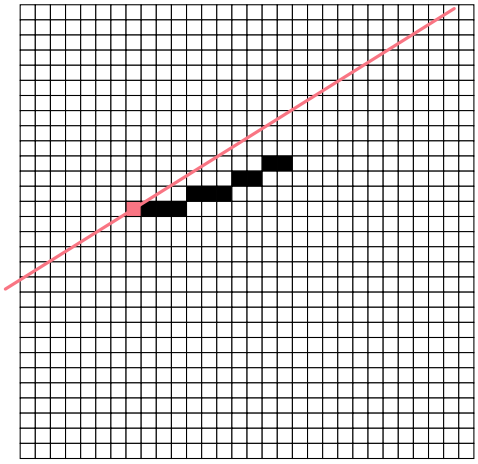

Figure 16: Line far from our points

To decide if a line passes close to our points, we can calculate the sum of the distances between our line and each point and try to minimize this sum. On the fig. 17, you can see the distances in red.

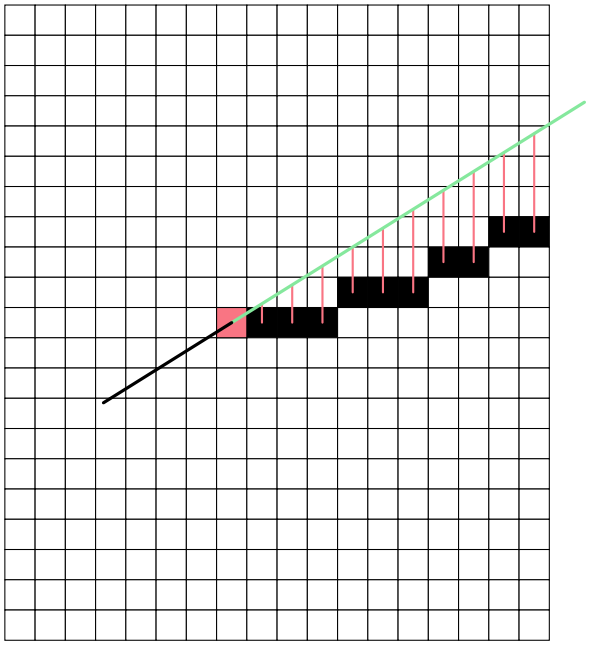

Figure 17: Distance to the line

However, when the line passes below certain points and above other points, negative values appear when calculating the difference. In order not to have a negative value in our calculations, we use the distance squared.

Finally, if we choose the minutia point as the origin of our Cartesian coordinate system, the line takes the form $y = a \cdot x$, because the intercept, b, is equal to 0. It is therefore possible to calculate the total squared distance as

$$\begin{aligned} error &= \sum(\text{line value} - \text{pixel value}) \\ &= \sum(ax - y)^2 \\ &= \sum(a^2x^2 - 2axy + y^2), \text{ for any pair}(x, y) \end{aligned}$$

To find the a which gives the line closest to the points, we want to minimize the distance squared, $error$. For that, we derive $error$ and we look for when it vanishes:

$$\begin{aligned} &\frac{d(error)}{dx} = 0 \\ \iff &\sum(2ax^2 - 2xy) = 0 \\ \iff &a = \frac{\sum xy}{\sum x^2} \end{aligned}$$

This solution poses problems in practice. When the desired line approaches the vertical, it becomes more difficult to find an optimal result.

For this, we imagine that we rotate the image by 90° to fall back to a horizontal line. This is equivalent to reversing the x and y. So, $\sum xy$ remainder $\sum xy$ but $\sum x^2$ becomes $\sum y^2$. You must then reverse the slope to put it back in the original direction of the image. And we get:

$$a = \frac{\sum xy}{\sum y^2}$$

However, you have to take into account that the image had been rotated and flip the slope of 90°. To do this we reverse a:

$$a = \frac{1}{\frac{\sum xy}{\sum y^2}} = \frac{\sum y^2}{\sum xy}$$

We will use this new formula if the connected pixels extend more along the y than along the x. That is, if $\sum y > \sum x$ which is equivalent to saying that the sum of the $\sum y^2 > \sum x^2$:

$$\begin{cases} a = \frac{\sum x \cdot y}{\sum x^2}, & \text{if } \sum x^2 \geq \sum y^2 \\ a = \frac{\sum y^2}{\sum x \cdot y}, & \text{if } \sum x^2 < \sum y^2 \end{cases}$$

4.3 Function `computeAngle`

We know at this stage how to calculate the direction of the minutia. However, we still have two possibilities as to its orientation (see fig. 18), we must therefore now be interested in the calculation of the direction.

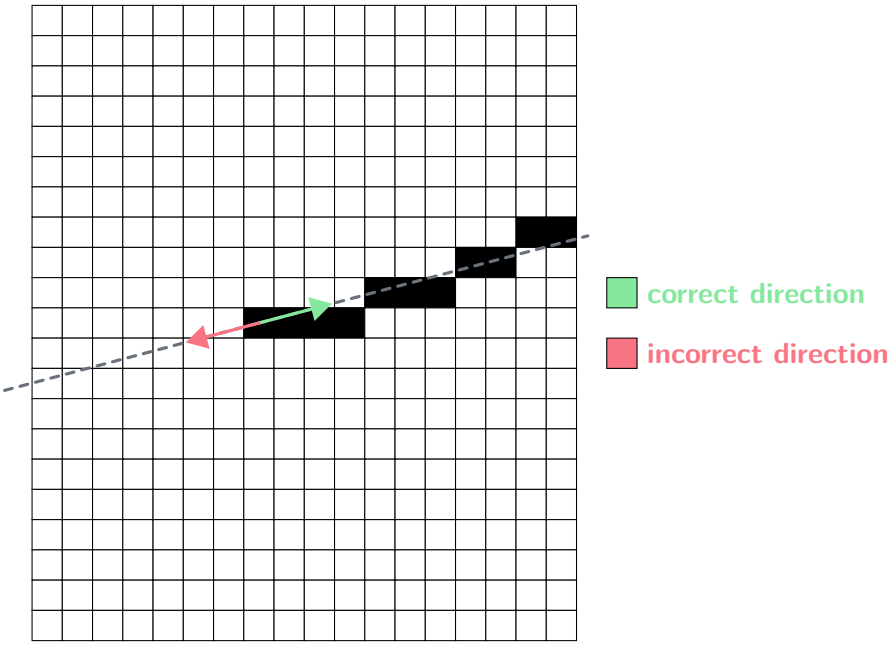

Figure 18: Example direction of a minutie

The direction is determined by the angle formed by the chosen direction vector with the x axis.

The *function* arctan() (`atan` in Java) applied to the slope of the line calculated previously, in principle makes it possible to calculate the desired angle in radians. However, since the angle thus calculated will be between $-\pi/2$ and $\pi/2$. It is impossible *a priori* to distinguish if we are in the 1 st or the 3 rd quadrant (0-90 ° or 180-270 °), or in the 2 nd or the 4 th quadrant (90-180 ° or 270-360 °).

To distinguish these cases, we are therefore going to choose to orient the minutie in the direction that there are the most pixels. To find in which direction there is the most, we imagine the perpendicular line and we count the number of pixels which are *above* this new line and the number of pixels that are below. This new line has the following equation:

$$y = -\frac{1}{a}x, \text{ where } a \text{ is the directing coefficient of the line giving the direction}$$

In the fig. 19, we can observe the perpendicular line in red and the line which gives the direction in grey. The green area contains all the pixels that are *above* from the right. The blue area contains those below.

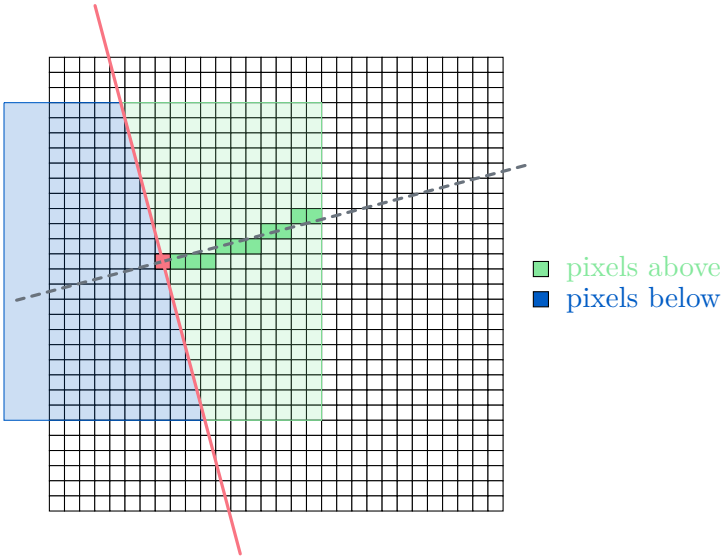

Figure 19: Separation of pixels by the perpendicular line

So a point that is at coordinates (x, y) is above if $y \geq -\frac{1}{a}x$. Otherwise, it is below. The coordinates (x, y) correspond here again to the coordinates of the usual mathematical reference, centered on the minutia.

To calculate the correct angle, it will therefore be necessary to add π to the angle calculated using $arctan()$ if:

- the angle is positive, but the number of pixels below the perpendicular and greater than the number of pixels above,
- or, if the angle is negative, but the number of pixels below the perpendicular and less than the number of pixels above.

You are asked to implement the function that returns the angle, in radians, indicating the orientation of the minutiae according to the previously suggested algorithm:

```
public static double computeAngle (boolean [][] connectedPixels,
   int row, int col, double slope)
```

This function takes the same parameters as `computeSlope` and the result of the latter, `slope`.

Note: Do not forget to pay attention to the special case where the slope is worth `Double.POSITIVE_INFINITY`. In this case, the angle is $\pi/2$ or $-\pi/2$ depending on the direction of the line: straight up or straight down. In the first case the connected pixels are below the minutia and in the second above.

4.4 Function `computeOrientation`

With the previous methods, write the method `computeOrientation` which returns the angle of the orientation in degrees, between 0° and 359°, rounded to the nearest integer.

```
public static int computeOrientation (boolean [][] image, int row,
   int col, int distance) {
```

When you call this method, the distance parameter will be the variable `ORIENTATION_DISTANCE`.

This method should, in order:

1. to call `connectedPixels (...)`,
2. to call `computeSlope (...)`
3. to call `computeAngle (...)`
4. return the angle in degrees, positive and rounded to the nearest integer. You can use standard Java functions `Math.round (...)` is `Math.toDegrees (...)`. if an angle is striclty negative you can add 360 to it to make it positive.

4.5 Function `extract`

Thanks to the previous functions, it is possible for us to calculate the orientation of any minutia. We just have to find them. For this, you are asked to implement the main function `extract`:

```
public static List<int[]> extract (boolean [][] image)
```

This function takes only the image as a parameter and returns a list of minutiae. The list is of the type `List<int[]>` . Each `int []` matches a minutiae. The first element is the row occupied by the minutiae, the second the column and the third the orientation in degrees. To simplify, we will extract (and then compare) only the minutia with 8 neighbours contained in the image (the borders are discarded).

To find the minutiae, we iterate over the entire image starting from pixel 1 (not zéro) and stopping at the butlast column (to guarantee that each pixel has all its 8 neighbours inside the image). We determine for each pixel if it is a minutiae. As we saw in the introduction, we are

going to use two specific points, the endings and the bifurcations. Endings are characterized by a number of transitions (as returned by the method `transitions ()`) in neighboring pixels equal to 1. That is, there is only one stroke that leaves / comes from this pixel. For the bifurcations, the number of transitions is equal to 3, because there are three lines which come / leave from the pixel. Thus, we look for all the pixels having one or three transitions.

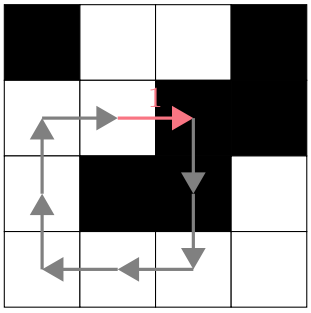

Figure 20: Termination (transitions = 1)

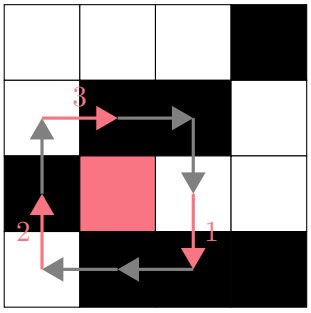

Figure 21: Bifurcation (transitions = 3)

4.6 Tests

Use the program `Main.java` to test your code. The provided examples of tests are not exhaustive. **Complete them** as you see fit, to test all the methods developed in this part of the project.

5 Task 3: Comparison

The two preceding steps make it possible to extract the list of corresponding minutiae from any image. We are now going to write a function that compares two lists of minutiae to say if they come from the same fingerprint.

Note that it is not enough to compare the two lists and check that all the minutiae are at the same positions and with the same orientations. The images of the same fingerprint can be moved horizontally (translation according to the columns), vertically (translation according to the lines) and oriented in any direction. In addition, depending on the pressure or the angle with which the fingerprint is captured, the minutiae may appear slightly closer or further away from each other. In the fig. 22, you can see two fingerprints on the same finger. The second has been translated downwards and is more oriented to the right than the first. Thus, the minutiae will not have the same coordinates or the same orientations although they come from the same finger.

Figure 22: Example of two fingerprints from the same finger having different translation and rotation

5.1 Presentation of the algorithm

To determine if two sets of minutiae come from the same fingerprint, we will try to superimpose them in any way possible and we will see if one of these superimpositions gives sufficient similarity. This is called a "brute force" solution.

In order to test all the possible solutions, we will try to superimpose each minutia m_2 of the second list on each minutia m_1 of the first set. Intuitively, this is like moving the second fingerprint (i.e. applying a translation) so that the coordinates of m_2 are the same as those of m_1. It is also necessary to rotate the second image to align the orientations of m_1 and m_2. In the fig. 23, you can observe the overlay of two fingerprints. The minutiae m_1 and m_2 were chosen at random in this example. It is therefore a question of testing all the pairs of m_1 and m_2 and seeing if one of these pairs gives a good overlap. In this case it will be considered that the two fingerprints come from the same finger. Finally, note that in this example, the entire image of the second fingerprint has been moved, but in practice you will not have to calculate the new coordinates of all the pixels, only the minutiae.

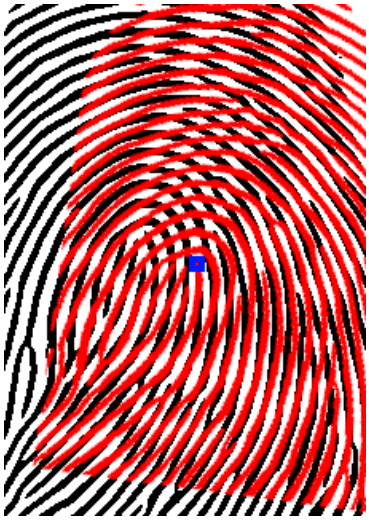

Figure 23: Example of superposition: the fingerprint having undergone a translation and rotation in red; the blue point indicates the position of the two aligned minutiae

You will need to start by writing a method that applies the transformation to all the minutiae in the list. This method is called `applyTransformation`. Then, you will need to write a method that counts the number of overlapping minutiae: `matchingMinutiaeCount`. Eventually you will be able to write the method `match` which returns `true` if two fingerprints come from the same finger and `false` otherwise.

5.2 Method `applyTransformation`

The implementation of `applyTransformation` uses two utility methods, `applyRotation` and `applyTranslation`.

Implement the first utility method that takes a minutiae (`minutia`), the center of transformation (`centerRow` and `centerCol`) and rotation (`rotation`). This method returns the new coordinates and the new orientation of the minutia after applying the rotation.

```
public static int [] applyRotation (int [] minutia, int centerRow,
   int centerCol, int rotation)
```

The new coordinates and orientation are calculated using the following formulas, where `row`, `col` and `orientation` are the parameters of the original thoroughness and `x`, `y`,`newX`, `newY` are temporary variables:

$$
\begin{aligned}
\texttt{x} &= \texttt{col} - \texttt{centerCol} \\
\texttt{y} &= \texttt{centerRow} - \texttt{row} \\
\texttt{newX} &= \texttt{x} \times \cos(\texttt{rotation}) - \texttt{y} \times \sin(\texttt{rotation}) \\
\texttt{newY} &= \texttt{x} \times \sin(\texttt{rotation}) + \texttt{y} \times \cos(\texttt{rotation}) \\
\texttt{newRow} &= \texttt{centerRow} - \texttt{newY} \\
\texttt{newCol} &= \texttt{newX} + \texttt{centerCol} \\
\texttt{newOrientation} &= (\texttt{orientation} + \texttt{rotation}) \mod 360
\end{aligned}
$$

The set returned by `applyRotation` will therefore contain in order: `newRow`, `newCol` and

`newOrientation`. **Note:** remember that cos and sin take as parameter an angle in radians while the rotation is passed to the function in degrees.

Then implement the second utility method taking as parameter a minutia (`minutia`), vertical translation (`rowTranslation`) and horizontal translation (`colTranslation`) to apply to it. This method returns the new minutiae coordinates after applying the translation. The orientation does not change.

```
public static int [] applyTranslation (int [] minutia, int
   rowTranslation, int colTranslation)
```

The new coordinates are calculated using the following formulas:

$$\begin{aligned} \texttt{newRow} &= \texttt{row} - \texttt{rowTranslation} \\ \texttt{newCol} &= \texttt{col} - \texttt{colTranslation} \\ \texttt{newOrientation} &= \texttt{orientation} \end{aligned}$$

Finally write a method `applyTransformation` which applies rotation and translation in order to a given minutia. You will also write an overload of this method which calls it on each of the minutiae of a list:

```
public static int [] applyTransformation (int [] minutia, int
   centerRow, int centerCol, int rowTranslation, int
   colTranslation, int rotation)
public static List<int[]> applyTransformation (List<int[]>
   minutiae, int centerRow, int centerCol, int rowTranslation, int
   colTranslation, int rotation)
```

5.3 Method `matchingMinutiaeCount`

It is now a matter of counting the number of minutiae which overlap in two lists of minutiae given as parameters. Implement the following method to perform this processing:

```
public static int matchingMinutiaeCount (List<int[]> minutiae1,
   List<int[]> minutiae2, int maxDistance, int maxOrientation)
```

It takes as parameter two lists of minutiae and looks for each entry m_1 of the first list if there is an entry m_2 of the second list which is in the same place and with the same orientation. We consider that two minutiae are superimposed if the Euclidean distance between them is less than or equal to `maxDistance` and we consider that they have the same orientation if their orientation differs by at most `maxOrientation` (understood). As a reminder, the Euclidean distance between two pixels is given by the formula $\sqrt{(\texttt{row}_1 - \texttt{row}_2)^2 + (\texttt{col}_1 - \texttt{col}_2)^2}$.

5.4 Method `match`

Finally, write the method:

```
public static boolean match (List<int[]> minutiae1, List<int[]>
   minutiae2)
```

This method should return `true` if the two minutiae lists come from the same fingerprint and `false` otherwise. We consider that two lists of minutiae come from the same fingerprint if we can find at least `FOUND_THRESHOLD` identical minutiae through the application of transformations. You will use for `maxDistance` the constant provided `DISTANCE_THRESHOLD` and for `maxOrientation` the constant `ORIENTATION_THRESHOLD`. The method `applyTransformation` will therefore be invoked on the minutiae m_2 of the second list to try to locate if they can be superimposed on any minutiae m_1 of the first. When calling this method:

- `centerRow` and `centerCol` (center of rotation) correspond to the position of m_1;
- `rowTranslation` and `colTranslation` (vertical and horizontal translations) are calculated as the difference between the row and column coordinates of m_2 and m_1: i.e. row (resp. column) of m_2 minus row (resp. column) of m_1;
- `rotation` is the difference in orientation between m_2 and m_1. However, since the computation of the orientations is not perfectly precise, you will have to test with all the integers included in [`rotation - MATCH_ANGLE_OFFSET`,`rotation + MATCH_ANGLE_OFFSET`].

5.5 Tests

Use the program `Main.java` to test your code. The provided examples of tests are not exhaustive. **Complete them** as you see fit, to test all the methods developed in this part of the project.

6 Theoretical complement

6.1 ARGB representation

The ARGB representation of a pixel using an integer is done in the following way:
The color is broken down into 4 components with a value between 0 and 255:

- **Alpha** is the opacity of the pixel. If it is at 0 then the pixel is "invisible", regardless of its other components. If it is between 1 and 254 then it is transparent and lets the colors of the image pass behind it. If its value is 255, then the pixel is perfectly opaque.
- **Red** is the pixel's red light intensity value.
- **Green** is the pixel's green light intensity value.
- **Blue** is the pixel's blue light intensity value.

Considering that the least significant bit of an integer has index 0 and that the most significant bit has index 31, then these 4 components are placed on the same integer in such a way: bits 31 to 24 define the alpha value, bits 23 to 16 the red value, bits 16 to 8 the green value and bits 7 to 0 the blue value. For example, if we want to represent a red pixel in ARGB we need the alpha and red component to be at most 255. Which gives us :

Alpha	Red	Green	Blue
255	255	0	0
0xFF	0xFF	0x00	0x00
11111111	11111111	00000000	00000000
31 24	23 16	15 8	7 0

The most convenient way to represent a color in Java is to use hexadecimal writing (more concise). For example to define the red pixel we can write:

```
int red = 0xFF_FF_00_00;
```

6.2 Binary format and shade of gray

The representation of a pixel in shade of gray only requires 1 byte. However, in Java bytes are signed and can only represent values between -128 and 127. In this project, we will therefore prefer to use integers.

This is a value between 0 and 255. A value of 0 represents a black pixel while a value of 255 represents a white one. And all values entered represent a shade of gray.

The binary representation only requires a boolean. The value `true` represents a black pixel while the value `false`, a white pixel.