Laboratory Experiment #4

Fingerprint Biometrics
Part I: Preprocessing and Feature Extraction

1 Introduction

This Lab is Part I of the exercise on the fingerprint image processing and recognition (matching). The input data for this lab is your fingerprints collected using the provided DigitalPersona USB device. You can also use the limited sample data provided on D2L.

Part I exercise will be executed using the Jupyter notebook `Lab04-Fingerprint1.ipynb`, which covers the fingerprint processing and feature extraction:

- Segmentation, Orientation estimation and Ridge orientation,
- Applying Gabor filters on the fingerprint image to enhance the ridges.

The focus of Part II (Lab 5) is to investigate two fingerprint matching algorithms. We will use the notebooks `Lab05-Fingerprint2-Minutiae.ipynb` and `Lab05-Fingerprint2-GaborFeatures.ipynb`, each one corresponding to a specific type of features used for the matching.

All the code implemented to work with fingerprints in this Lab uses several Python libraries already used in this course PLUS a new one called OpenCV[1]. OpenCV is a well-known library for computer vision tasks. It is available on Anaconda repositories, but it is not installed by default, so you will need to install it manually. To proceed with the OpenCV installation, you should follow the same steps we used to install the PyAgrum library:

1. Go to the main menu > Anaconda3 (64-bit) > Anaconda Powershell Prompt.
2. Type the command: `conda install opencv -c conda-forge`.
3. Accept all the dependencies typing `y` for "yes".

We are also going to use auxiliary files included in the file `utils.zip`, available on D2L. Download this file, unpack it in the same folder as this Lab's notebook.

NOTE: We are using the Scikit-Image version 0.18.3.
If you have a version different from that, please open your Anaconda prompt and type:
`conda install -c anaconda scikit-image=0.18.3`

2 The laboratory procedure

This is Part I of the fingerprint biometric exercise.

2.1 Part I: Fingerprint Image Processing

The focus of this part is imaging processing, required to properly extract image features for further matching process, described in part II (Lab 5).

2.1.1 Fingerprint acquisition

The acquisition of the fingerprints will be performed on the USB *Digital Persona UareU 4500* fingerprint sensors (reader), and the software developed in the Biometric Technologies Laboratory at UofC using the Digital persona SDK. The software creates "raw" data in the `.bmp` format.

To collect the data, connect the *UareU 4500* to the computer using a USB port. The device is ready once the red light flashes.

[1] https://opencv.org/

Download the zip-file `DigitalPersona.zip` located on D2L folder of this Lab, unpack it and run the executable `digitalPersona.exe`. It will prompt the interface opening.

Apply your finger (thumb) to the sensor. To save the image, choose option "File" and then "Save" (save in the folder where your Jupyter notebook is located). To get more images, clean the reader with a Scotch tape (yes, press the scotch tape on the top of the sensor and lift it, – as good as new).

The quality of data is affected by the sensor quality, the skin condition (dryness, cuts), the quality of the procedure (such as correct placement of the finger), humidity etc. You will need few images of good quality and few of poor quality for analysis of fingerprint processing procedures, and later for the matching.

Alternatively, you can use the data collected in the BT Lab and provided in the zip-file `BTLab_Fingerprints.zip`, available on D2L in this lab directory. We recommend that you put your dataset images in the same folder as your Lab's notebook. This way will be easier to access the images when necessary. In total, you have to collect 10 fingerprints of your right thumb (or other finger), and 10 of your left one. Try to have both the 'good' and the 'bad' quality prints. You will use some of those as the 'gallery' prints, and some as the 'probe' prints to compare against each other. Alternatively, collect 10 of your prints and 10 of your partner's. This will be your set of prints of 2 different fingers ('subjects').

2.1.2 Image reading

To read images using Python, we will use the library Scikit-Image[2]:

```
# reading an image with imread(...)
# some images, when loaded, might to return as float [0-1]
# here we convert it to uint8 [0-255] using the function img_as_ubyte(...)
img = img_as_ubyte(imread('FPsamples/1.bmp', as_gray=True))
```

2.2 Fingerprint image processing

The fingerprint processing is implemented using several Python files included on D2L section of this Lab. The main file is the Jupyter notebook `Lab04-Fingerprint1.ipynb`, which calls other functions. Fig. 1 shows the results of each step of a fingerprint image processing.

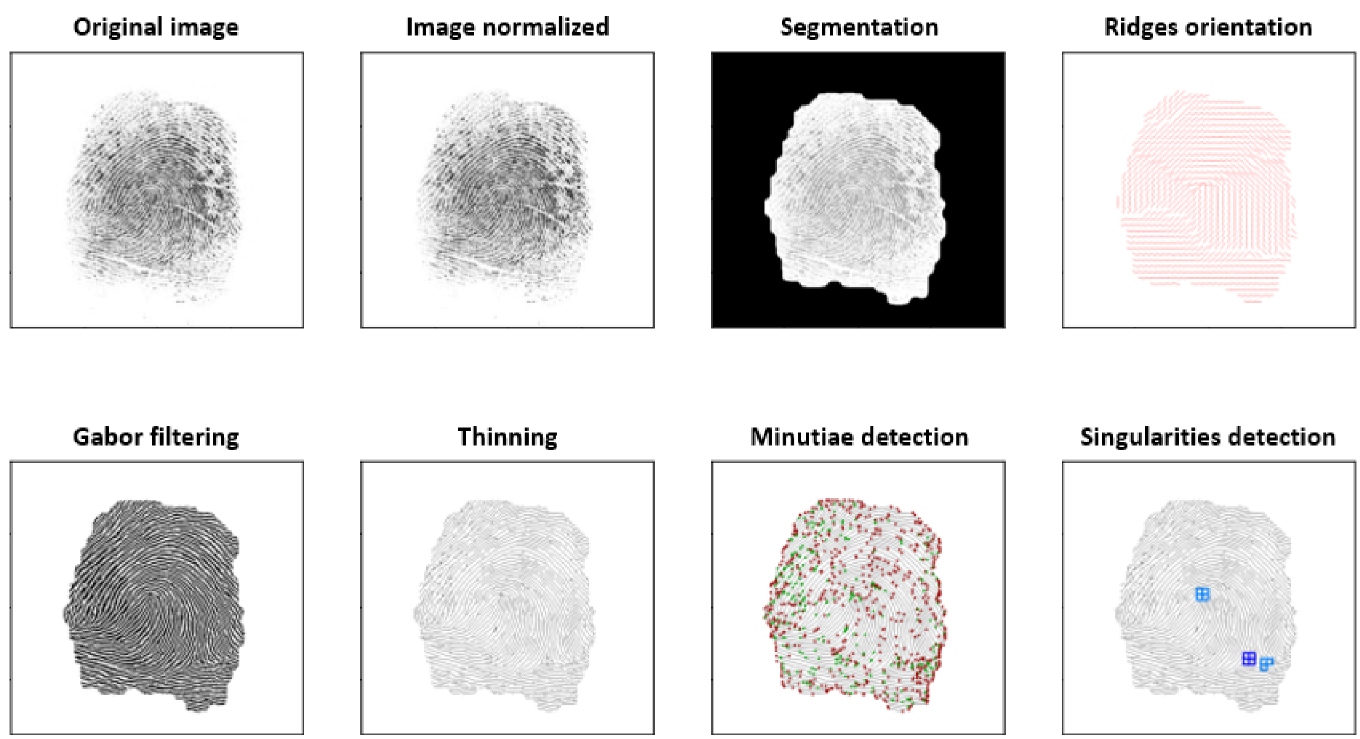

Figure 1: Step-by-step results in the fingerprint image processing.

[2] https://scikit-image.org/

The fingerprint image processing steps are briefly described below:

1. **Normalization** (`normalize`): the image has its pixel intensity normalized within a pre-determined range.
2. **Segmentation** (`segmentation`): extracts a fingerprint area from the background and finds its contour.
3. **Ridges orientation estimation** (`calculate_angles`): computes an orientation array at every pixel.
4. **Ridge frequency estimation** (`ridge_freq`): computes local ridge frequencies (congestion).
5. **Gabor filtering** (`gabor_filter`): implements directional filters to enhance the ridges.
6. **Thinning** and skeleton cleaning (`skeletonize`): removes spurious pixels from the skeleton keeping the ridge lines as thin as 1 pixel wide.
7. **Minutiae detection** (`calculate_minutiaes`): determines the coordinates of the minutiae points such as *ridge engings* and *bifurcations*.
8. **Singularities detection** (`calculate_singularities`): finds the points of cores, deltas, etc.

The notebook `Lab04-Fingerprint1.ipynb` executes these steps and display the output images. The complete execution is performed by the function `fingerprint_processing(...)` :

```python
def fingerprint_processing(img, block_size=16, threshold=0.4):
    output = {}

    # normalization - removes the effects of sensor noise and finger pressure differences.
    normalized_img = normalize(img.copy(), float(100), float(100))
    output['normalized_img'] = normalized_img

    # segmentation
    (segmented_img, normim, mask) = segmentation(normalized_img,
                                                 block_size,
                                                 threshold)
    output['segmented_img'] = segmented_img
    output['normim'] = normim
    output['mask'] = mask

    # orientation estimation
    angles = calculate_angles(normalized_img,
                              W=block_size, smooth=True)
    output['angles'] = angles

    # find the overall frequency of ridges
    freq = ridge_freq(normim, mask, angles,
                      block_size, kernel_size=5,
                      minWaveLength=5, maxWaveLength=15)
    output['freq'] = freq

    # create gabor filter and do the actual filtering
    gabor_img = gabor_filter(normim, angles, freq, block_size)
    output['gabor_img'] = gabor_img

    # create the skeleton
    thin_image = skeletonize(gabor_img)
    output['thin_image'] = thin_image
```

```
    # find the minutiae
    minutiae_lst, minutiae_img, minutiae_arr = calculate_minutiae(thin_image, mask)
    output['minutiae_lst'] = minutiae_lst
    output['minutiae_img'] = minutiae_img
    output['minutiae_array'] = minutiae_arr

    # singularities
    singularities_lst, singularities_img = calculate_singularities(thin_image,
                                                                    angles,
                                                                    block_size, mask)
    output['singularities_lst'] = singularities_lst
    output['singularities_img'] = singularities_img

    return output
```

The main parameters of the `fingerprint_processing(...)` function include:

1. the image (parameter `img`) to be precessed;
2. the size of the filter window used when processing (parameter `block_size`) and;
3. `threshold`, the value used during the segmentation process.

The results of each step is allocated in Python dictionary structure (variable `output`).
To compare two fingerprints, each shall be processed separately. For example, two images must be loaded:

```
# loading two images for Minutiae extraction and matching
im1 = img_as_ubyte(imread('FPsamples/1.bmp', as_gray=True))
im2 = img_as_ubyte(imread('FPsamples/2.bmp', as_gray=True))

# parameters for processing
seg_threshold = 0.2
block_size = 16

# processing
Fp1 = fingerprint_processing(im1, block_size=block_size, threshold=seg_threshold)
Fp2 = fingerprint_processing(im2, block_size=block_size, threshold=seg_threshold)
```

2.3 Image pre-processing

The pre-processing tasks are described below.

2.3.1 Contrast enhancement

The acquired fingerprint images may have poor contrast. Thus, an enhancement technique shall be applied in order to have an image with better contrast before extracting the minutiae. In the notebook `Lab04-Fingerprint1.ipynb`, we present two contrast enhancement techniques based on the *histogram equalization*:

- `equalize_hist(...)`: regular image equalization.
- `equalize_adapthist(...)`: adaptive equalization.

Each technique has its advantages and disadvantages, and the results depend on the original image quality. Fig. 2 shows the original images and the result of two technique applied to the original image. To plot the corresponding pixel intensity histogram for visual inspection, use the command `hist(...)` from Matplotlib. Fig. 2 shows the corresponding histograms. Note that in this case, the *adaptive equalization* worked better than the regular one.

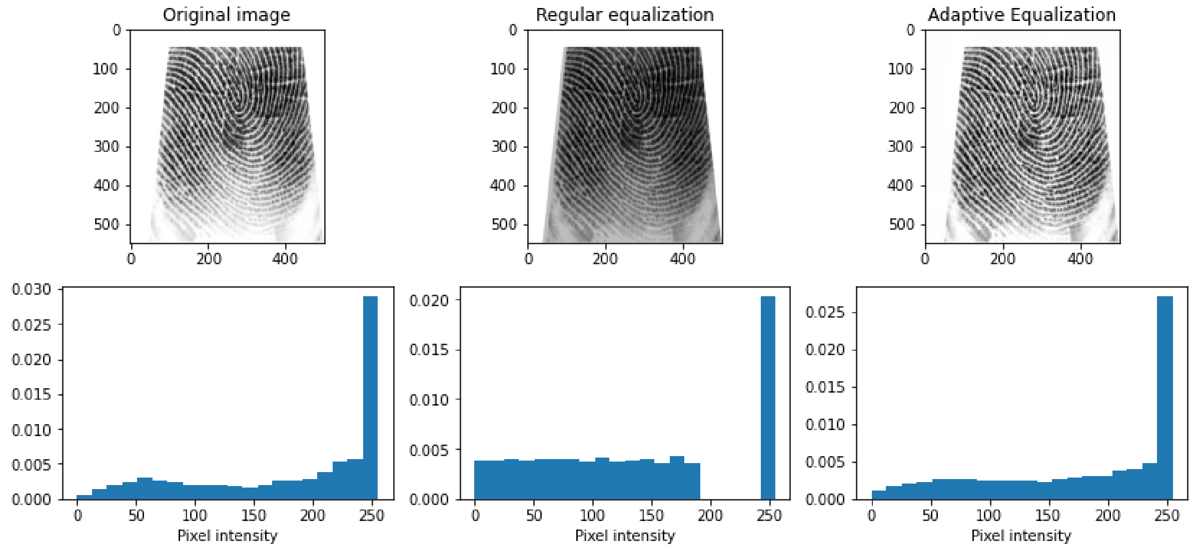

Figure 2: The original image and the results of the histogram equalization

2.3.2 De-noising

De-noising aims to remove noises such as spurious pixels intensities. The two well-known de-noising filters include:

- `wiener(...)`: for Wiener filter,
- `median(...)`: for median filter.

Fig. 3 shows the results after the application of each filter. Visual inspection does not show much differences. However, the filter choice may impacts the outcome of the matching.

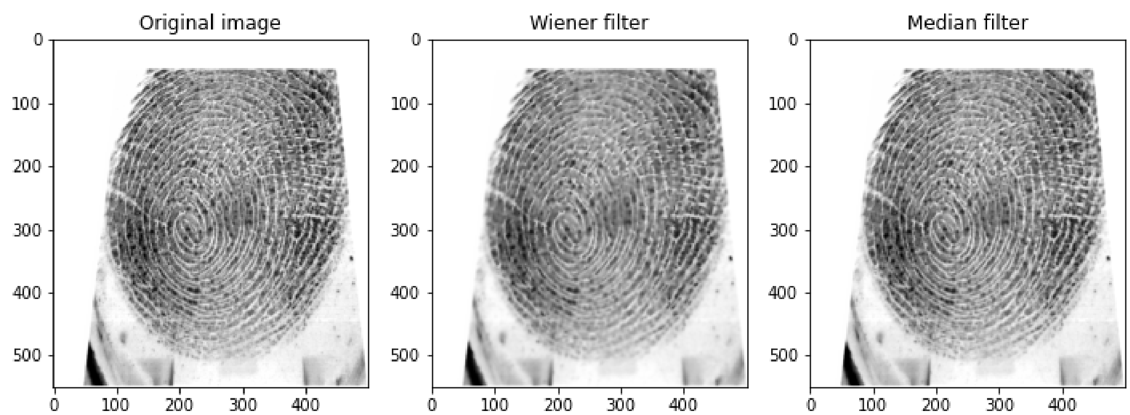

Figure 3: The de-noising results.

The process of removing the noise involves several steps. Note that both the intensity enhancement and de-noising should be done before the segmentation step.

2.3.3 Normalization

The demo notebook (`Lab04-Fingerprint1.ipynb`) includes the normalization step. Normalization is aimed at 'equalizing' the pixels intensities, using the mean and variance equal to 100.

```
normalized_img = normalize(input_img.copy(), float(100), float(100))
```

2.3.4 Segmentation

Segmentation is applied to extract the image from the background. Fig. 4 shows the result of such segmentation, performed using the following code:

```
(segmented_img, normim, mask) = segmentation(normalized_img,
                                              block_size,
                                              threshold)
```

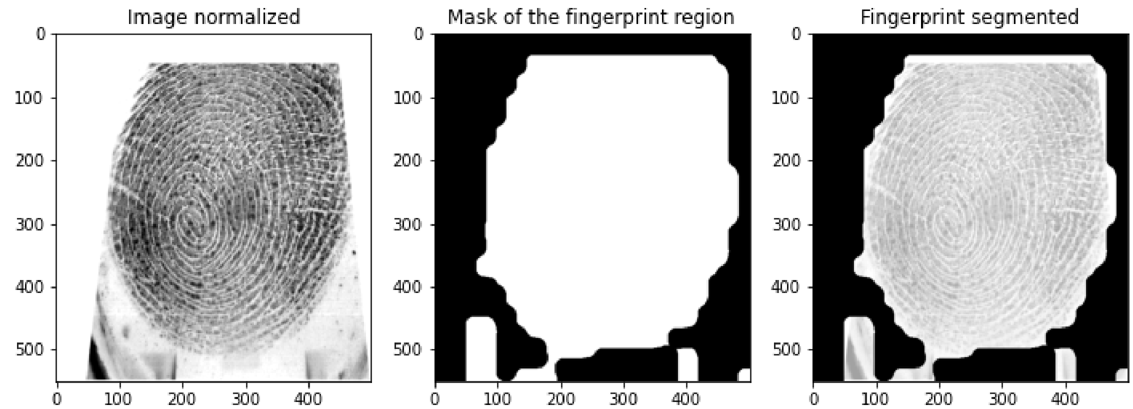

Figure 4: Segmentation process: the mask found and the image after the mask is applied.

Note that the function `segmentation(...)` uses the morphological operations to smooth and segment the fingerprint image using the following parameters:

- `block_size = 16`,
- `threshold = 0.2`.

2.3.5 Ridge orientation

The ridge orientation is defined by the angles and directions, estimated using the function `calculate_angles(...)` included in the notebook `Lab04-Fingerprint1.ipynb`. This function uses the block size as a parameter, `W = 16`. Fig. 5 shows the two outcomes of this process.

```
angles = calculate_angles(normalized_img, W=block_size, smooth=True)
orientation_img = visualize_angles(segmented_img, mask, angles, W=block_size)
```

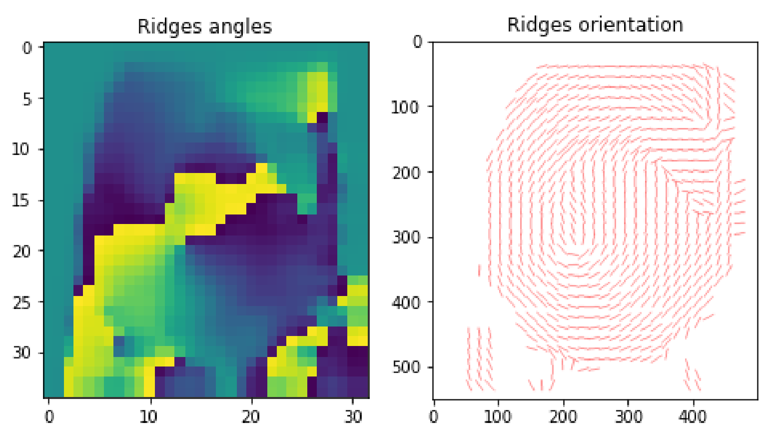

Figure 5: The results of the detection of the ridges' orientation.

2.3.6 Frequency of ridges

The demo `Lab04-Fingerprint1.ipynb` also calls the function `ridge_freq`. It uses the parameters `block_size = 16` and `kernel_size` that determine the region to be processed and control how much the ridges shall be dilated before processing, respectively. The expected length of the ridges are controlled by the parameters `minWaveLength` and `maxWaveLength`.

```
freq = ridge_freq(normim, mask, angles, block_size,
                  kernel_size=5, minWaveLength=5, maxWaveLength=15)
```

2.3.7 Ridge enhancement using Gabor filter and skeleton building

The demo `Lab04-Fingerprint1.ipynb` calls the function `gabor_filter(...)` that executes the fingerprint filtering using the Gabor filter to enhance the ridges candidates. The output is used for the next step, the creation of the fingerprint skeleton using the function `skeletonize(...)`:

```
gabor_img = gabor_filter(normim, angles, freq,
                         block_size2=block_size,
                         kx=0.65, ky=0.65)
thin_image = skeletonize(gabor_img)
```

Figure 6 shows the results of each processing. The skeleton created (left image of Fig. 6), is used later in this exercise for the detection of minutiae and singularities. Note that the same Gabor filter is also used in Part II for the feature extraction in Lab 5.

The function `skeletonize(...)` uses the Scikit-Image library. The result of the Gabor filtering is used for the next step, to create the skeleton. This process involves several morphological operations that detect disconnections and spurious pixels that may become the part of the final skeleton. The final skeleton consists of the "ridges" that are one pixel wide.

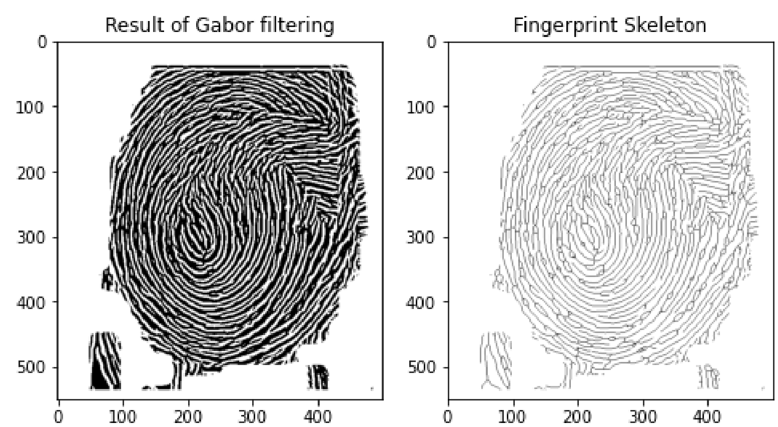

Figure 6: The result of finding the ridges' orientation.

2.3.8 Detection of minutiae and singularity points

The minutiae and singularities points are used for the final step, the matching. Finding the minutia and the singularities is executed by calling two distinct functions as shown below:

```
# minutiae_lst is a list with all minutiae coordinates and its type:
# bifurcation or ending.
# minutiae_img is an image of the minutiae already plot on top of the skeleton
minutiae_lst, minutiae_img, _ = calculate_minutiae(thin_image, mask)

# singularities output are similar to the minutiae.
# However, the singularities can be: whorls, loops and deltas.
singularities_lst, singularities_img = calculate_singularities(thin_image,
                                                               angles,
                                                               block_size, mask)
```

Figure 7 shows the results of minutiae and singularities detection on the 'skeletonized' images. There are two types of minutiae: ridge ending and valley ending (bifurcation). The singularity points are found at the ridges that shape whorls, loops or deltas.

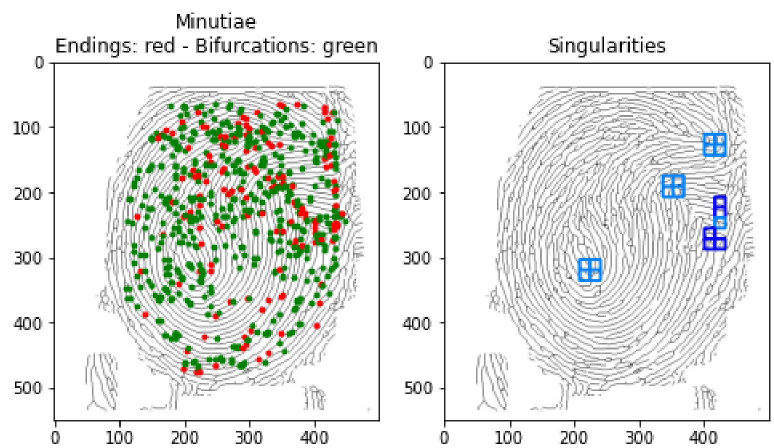

Figure 7: Calculation of the minutiae and singularity points.

Note that all the steps described previously have the parameters that might produce different result. Thus, the entire outcome is sensitive to the image quality and the chosen parameters.

3 Lab Report

Your report in the form of a Jupyter Notebook/Python (file extension `.ipynb`) shall include the description of each exercise with illustrations/graphs and analysis of the results. Save your Notebook using menu "Download As" as `.ipynb`, and submit to the D2L dropbox for Lab 4, by the next lab session (next Friday by 9am).

4 Lab Exercise in Jupyter Notebook with Python

For the following exercises, use the fingerprints provided on D2L directory of this Lab, or your own prints taken using the reader UareU 4500. A detailed description of each exercise to be included in your report (10 marks total) is given below.

- **Exercise 1** (3 marks): Select two fingerprints of different quality (one good and one bad one) from two different fingers each (4 in total). Modify the Lab notebook (`Lab04-Fingerprint1.ipynb`) by adding the histogram equalization step. For each fingerprint, perform all the pre-processing, and then add the histogram equalization. Compare the quantity of minutiae and singularities detected WITHOUT and WITH the histogram equalization step. Record the number of minutiae, the number of singularities for each case. Draw the conclusions from this comparison.

- **Exercise 2** (3 marks): In this exercise, perform the same steps as in Exercise 1, but instead of histogram equalization, choose one de-noising method such as *Wiener* or *Median* filter.
  Modify the Lab notebook (`Lab04-Fingerprint1.ipynb`) accordingly, record the outcomes and draw the conclusions.

- **Exercise 3** (4 marks): In this exercise, perform the same steps as in Exercise 1, but consider the parameters `block_size` and `threshold`. Evaluate the impact of these parameters by changing each of them to another value one at a time, for example: change the `block_size` to 10 (`block_size = 10`) and keep the `threshold = 0.4` (default). Second, change the threshold only to `threshold = 0.5` while keeping the `block_size = 16` (default). Evaluate the number of the detected minutiae and singularities. Compare the results and draw the conclusions.

5 Acknowledgments

The acquisition executable `digitalPersona.exe` was created in the BT Lab based on the Digital Persona SDK. The Python implementation of minutiae extraction using OpenCV is credited to Manuel Cuevas.

Dr. S. Yanushkevich, Dr. H. C. R. Oliveira.
February 2, 2022.