Project 2 - Thresholding

Thales Mateus Rodrigues Oliveira (RA 148051)

***Abstract*—In this project, it was given the task of applying global and local threshold methods to grayscale images. To fulfill the requirements, the solution implements 8 different algorithms, using approaches listed in the literature. It was able to generate the binarized thresholded images, and the differences of each approach, the pros and cons of the implementation are explained in this report.**

I. Introduction

Thresholding is a technique used for segmentation, and consists on the classification of the pixels within an image according to a(some) specific threshold(s). It is used for basic segmentation, to be able to separate objects from brackground in images and make it easier image processing steps. There are global techiniques, which considers one (or more) general threshold(s) for the entire picture, and local ones, that calculates local threshold(s) for every region in the image. The goal of this work is to implement global and local approaches, and be able to compare their strenghts and weakness. The next sections explains the implemented algorithm, the experiments realized and the output analysis

The code, along with the input files and the report is delivered in the compressed file THALES_MATEUS_RODRIGUES_OLIVEIRA_148051.tar, in the Google Classroom.

II. The Program

The program was implemented with Python 3.7.3. The libraries used and their respective versions are OpenCV 4.1.0, Numpy 1.16.4 and MatPlotLib 3.1.0.

A. How to execute it

The project has a Makefile available to help performing some actions on it. The Makefile has 3 basic instructions: clean, build and exec. Clean instruction removes generated images stored in the **output** folder, the execution code in **bin** folder and the folders itself. The Build instruction creates the **output** and **bin** folders, and moves the source code to **bin**. The Exec instruction executes the code with images in the **input** folder. Listing 1 provides examples of how to execute the three instructions in a terminal.

```
#clean environment, deletes output and bin folders and their content
make clean

#prepare the environment for code execution
make build

#executes code
make exec
```

Listing 1. Makefile usage example

B. Input

The program does not have an input argument by default, the input images are listed in code, and they are expected to be stored in the **input** folder. Listing 2 shows how images are listed to be executed in code. The $images$ tuple is implemented in $src/main.py$.

The images are expected to be in the *.pgm* format.

```
# for inserting other images, add tem to /input folder and list them here
images = (
    'baboon',
    'fiducial',
    'monarch',
    'peppers',
    'retina',
    'sonnet',
    'wedge'
)
```

Listing 2. Input images inside code

C. Output

The output of the program is a series of binarized images based on the input ones, threshold methods and their peculiarities, and their respective histograms. The output images are stored in the **output** folder, and they stored based on global or local threshold methods (**global-thresholding** and **nxn-window** folders) and are labeled by concatenating the image name and the threshold method, for local cases (*e.g.: output/99x99-window/baboon_bernsen.pgm*). The histograms of the images are stored following the same convention (*e.g.: output/99x99-window/fiducial_contrast_histogram.pgm*). The ratio of black (object) pixels in each image is also an output of the program, using the standard output.

D. Implementation

The functions which implement the thresholding operations are defined in the *src/thresholding.py* file. The file has an auxiliary function to build histograms for images (calculate_histogram), and has 8 thresholding functions, 1 for the global method and 7 other to the local approaches. The thresholding methods were implemented according to the project specification. For some local methods, as they deal with neighborhood centered in the analyzed pixel, the image is padded according to the neighborhood size. The convention here applied is that background pixels have maximum value (255) and object pixels have minimum value (0). The following items describe each of the used approaches.

1) Global Thresholding: The value of each image pixel is compared to a threshold. If greater than the threshold, it is an object pixel. Otherwise, it is background. It is the simpler method, and the used threshold is received as an input for

ra148051@students.ic.unicamp.br

the function. Listing 3 shows the most important lines of the implementation.

```
def global_thresholding(img, threshold=128):
  ...
  result = np.where(img < threshold, 255, 0)
  result = np.uint8(result)
  ...
```

Listing 3. Global Thresholding Implementation

2) Bernsen Thresholding: The Bernsen approach for local thresholding calculates the threshold of a specific pixel based on the mean of the maximum and minimum values of its neighborhood centered in the pixel. Then, the comparison to set the value is done. Listing 4 shows the most important lines of the implementation.

```
  def bernsen_local_thresholding(img, window_size=3)
    :
    ...
    padded_img = np.pad(img, (window_size//2,
    window_size//2), 'constant')
    for j in range(img_height):
        for i in range(img_width):
            window = padded_img[j:j + window_size, i
    :i + window_size]
            local_threshold = (int(np.min(window)) +
     int(np.max(window)))//2
            if img[j][i] < local_threshold:
                result[j][i] = 255
    ...
```

Listing 4. Bernsen Local Thresholding Implementation

3) Niblack Thresholding: The Niblack approach for local thresholding is based on the pixel's neighborhood mean and standard deviation values. The threshold is calculated as:

$$T(x,y) = \mu(x,y) + k\sigma(x,y) \tag{1}$$

where μ,σ are the mean and standard deviation in the neighborhood, respectively, and k an adjustment factor. The literature calculates a good value of k as 0.2. Listing 5 shows the main lines of the implementation.

```
  def niblack_local_thresholding(img, window_size
    =15, k=-0.2):
    ...
    for j in range(img_height):
        for i in range(img_width):
        window = img[window_size*(j//window_size):
    window_size*(j//window_size) + window_size,
        window_size*(i//window_size):window_size*(i
    //window_size) + window_size]
          mean = np.mean(window)
          std_dev = np.std(window)
          local_threshold = int(mean + k*std_dev)
          if img[j][i] < local_threshold:
              result[j][i] = 255
    ...
```

Listing 5. Niblack Local Thresholding Implementation

4) Sauvola and Pietaksinen Thresholding: This solution tries to improve Niblack one, specially for documents with bad lighting. The threshold is calculated as:

$$T(x,y) = \mu(x,y) * (1 + k(\frac{\sigma(x,y)}{R} - 1)) \tag{2}$$

The standard deviation and mean are calculated based on neighborhood centered in the analyzed pixel. Listing 6 shows main parts of implementation. The default k and R values are suggestions from the creators.

```
def sauvola_pietaksinen_local_thresholding(img,
  window_size=3, k=0.5, r=128):
  ...
  padded_img = np.pad(img, (window_size//2,
  window_size//2), 'constant')
  img_height, img_width = img.shape
  for j in range(img_height):
      for i in range(img_width):
          window = padded_img[j:j + window_size, i
  :i + window_size]
          mean = np.mean(window)
          std_dev = np.std(window)
          local_threshold = int(mean*(1 + k*((
  std_dev/r) - 1)))
          if img[j][i] < local_threshold:
              result[j][i] = 255
  ...
```

Listing 6. Sauvola and Pietaksinen Local Thresholding Implementation

5) Phansalskar, More and Sabale Thresholding: This approach is a variation of Sauvola and Pietaksinen, to deal with low cotnrast images. The calculation of threshold for local neighborhood is done as:

$$T(x,y) = \mu(x,y) * (1 + pexp(-q\mu(x,y)) + k * (\frac{\sigma(x,y)}{R} - 1)) \tag{3}$$

The p, q, k, R default parameters are suggested from the authors as $2, 10, 0.25 and 0.5$, respectively. The calculation is done for normalized image. Listing 7 shows the main aspects of the implementation.

```
def phansalskar_more_sabale_local_thresholding(img
  , window_size=3, k=0.25, r=0.5, p=2, q=10):
  ...
  for j in range(img_height):
      for i in range(img_width):
          window = img[window_size*(j//window_size
  ):window_size*(j//window_size) + window_size,
                       window_size*(i//window_size
  ):window_size*(i//window_size) + window_size]
          # image is normalized
          window = window/255
          mean = np.mean(window)
          std_dev = np.std(window)
          local_threshold = (mean*(1 + (p*math.exp
  (-q*mean)) + (k*(((std_dev)/r) - 1))))
          if img[j][i]/255 < local_threshold:
              result[j][i] = 255
  ...
```

Listing 7. Phansalskar, More and Sabale Local Thresholding Implementation

6) Contrast Thresholding: This solution analyzes de distance of the pixel value to the min and max values of the local neighborhood. If it is closer to the max, it is part of the object. It is background, otherwise. Listing 8 clarifies the implementation.

```
def contrast_local_thresholding(img, window_size
  =3):
  ...
  for j in range(img_height):
      for i in range(img_width):
          window = img[window_size*(j//window_size
  ):window_size*(j//window_size) + window_size,
                       window_size*(i//window_size
  ):window_size*(i//window_size) + window_size]
          min_v = np.min(window)
          max_v = np.max(window)
          dist_min = abs(min_v - int(img[j][i]))
          dist_max = abs(max_v - int(img[j][i]))
          if dist_min < dist_max:
              result[j][i] = 255
```

```
    ...
```

Listing 8. Contrast Thresholding Implementation

7) Mean Thresholding: This approach sets the threshold as the mean of the local neighborhood of the analyzed pixel. Listing 9 shows the main aspects of the implementation.

```
def mean_local_thresholding(img, window_size=3):
  ...
  for j in range(img_height):
      for i in range(img_width):
          window = img[window_size*(j//window_size
):window_size*(j//window_size) + window_size,
                       window_size*(i//window_size
):window_size*(i//window_size) + window_size]
          mean = np.mean(window)
          if img[j][i] < mean:
              result[j][i] = 255
  ...
```

Listing 9. Mean Thresholding Implementation

8) Median Thresholding: The solution sets the threshold as the median of the local neighborhood of the analyzed pixel. Listing 10 shows main parts of implementation.

```
def median_local_thresholding(img, window_size=3):
  ...
  for j in range(img_height):
      for i in range(img_width):
          window = img[window_size*(j//window_size
):window_size*(j//window_size) + window_size,
                       window_size*(i//window_size
):window_size*(i//window_size) + window_size]
          median = np.median(window)
          if img[j][i] < median:
              result[j][i] = 255
  ...
```

Listing 10. Median Thresholding Implementation

III. Experiments

The *src/main.py* file executes the test pipeline. The idea is the following: for each input image, for each global threshold, execute the global thresholding method. For each window size, executes all local thresholding methods listed prior. For each global and local executions, save the output images and their respective histograms. Also, print in the standard output the ratio of black (object) pixels. The input images are stored in **input** folder as mentioned before, and their names and dimensions are listed in table I. The input images are shown in figure 1. The generated output images for the report are saved in *png* extension in order to make it easier the report manipulation, and are stored in the report folder.

Image Names	Dimensions (width x height)
baboon.pgm	512 x 512
fiducial.pgm	640 x 480
monarch.pgm	768 x 512
peppers.pgm	623 x 594
retina.pgm	256 x 256
sonnet.pgm	384 x 510
wedge.pgm	507 x 384

TABLE I
Input images used in experiments

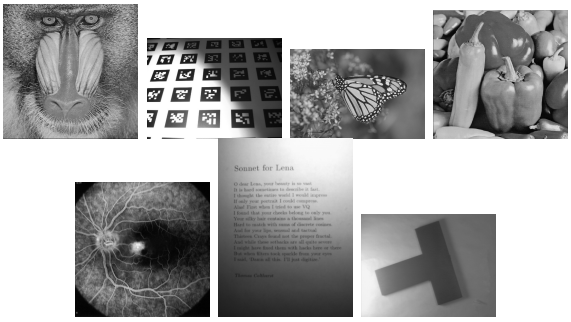

Fig. 1. Input images used in experiments. a) Baboon. b) Fiducial c) Monarch d) Peppers e) Retina f) Sonnet g) Wedge

The images original histogram are also shown in Figure 2. They provide a good description of the input images. The Baboon image, which focus on the face of the animal, has a good number of details. The Fiducial image has a "almost binarized" appearance, with values concentrated in the extremes (either black or white, with high concentration of white), and also has some shading (as seen in intermediate values of the histogram). The monarch image has a concentration of intensities in the first half of the spectrum ($[0, 128]$), and it might be difficult to distinguish the butterfly from the background. The peppers image has good, and it might be difficult to separate objects from background. The retina image also has a high concentration of intensities in the first half of the spectrum. The sonnet one has variety of intensities, and some bad lighting in the text. The wedge image has high occurence of middle intensities.

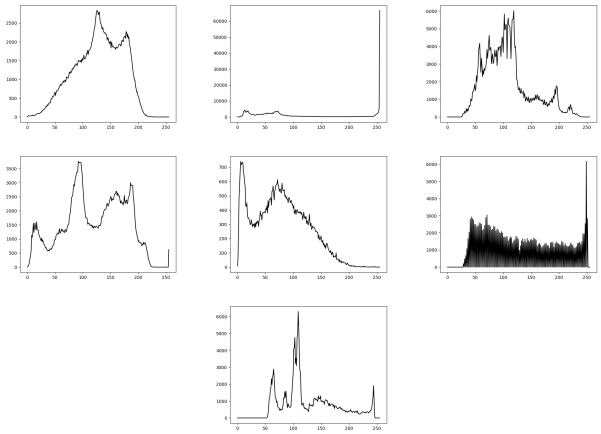

Fig. 2. Histogram of input images used in experiments. a) Baboon. b) Fiducial c) Monarch d) Peppers e) Retina f) Sonnet g) Wedge

For the global thresholding method, values in the $[0, 255]$ scale were chosen. Table II lists the global threshold valus used.

Global Threshold values
50
128
200

TABLE II
Global threshold values used in experiments

For the window sizes, a variety of values were picked. The neighborhood is chosen to be always a squared nxn region, where n is the given size. Table III lists the picked values.

Window sizes (pixels)
3x3
9x9
15x15
33x33
99x99

TABLE III
WINDOW SIZES FOR SELECTING PIXEL'S NEIGHBORHOOD

As we have 7 input images, 3 global threshold methods, 7 local threshold methods and 5 window sizes, we have 266 images of output, and their respective histograms. The output images are stored and well organized in the **output** folder.

IV. DISCUSSION

This section is organized in four parts. The first part analyses global thresholding methods and their effectiveness. The second part takes into consideration each local thresholding method, varying the window size. The third one does the comparisons between the local thresholding methods. To sum up, the comparison between global and local methods.

A. Global threshold techniques

In order to analyze the output images for the global threshold techniques, the following images are shown, comparing the thresholded images with different threshold values. The figures 3, 4, 5 shows the output images for the threshold values (50, 128, 200).

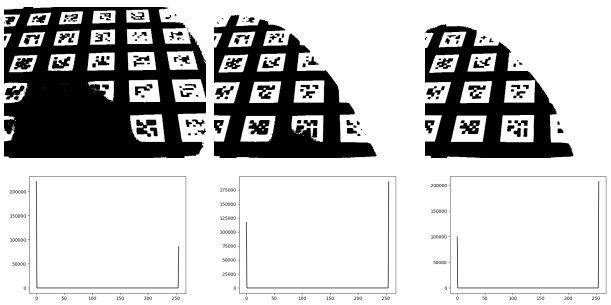

Fig. 3. Fiducial images with global threshold applying a) Global threshold equals 50. b) Global threshold equals 128. c) Global threshold equals 200. d)Histogram for value 50 e)Histogram for value 128 f)Histogram for value 200

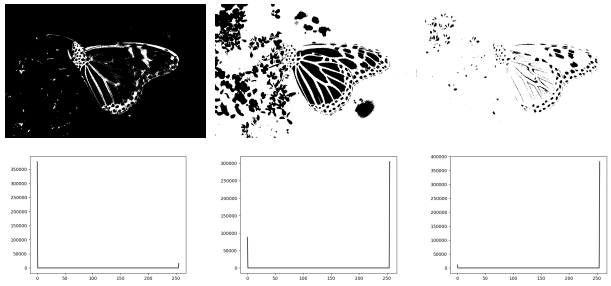

Fig. 4. Monarch images with global threshold applying a) Global threshold equals 50. b) Global threshold equals 128. c) Global threshold equals 200. d)Histogram for value 50 e)Histogram for value 128 f)Histogram for value 200

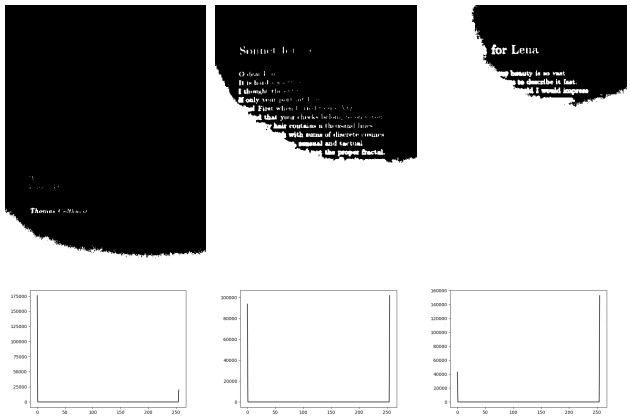

Fig. 5. Sonnet images with global threshold applying a) Global threshold equals 50. b) Global threshold equals 128. c) Global threshold equals 200. d)Histogram for value 50 e)Histogram for value 128 f)Histogram for value 200

As expected, even though the global method is able to distinguish background from objects, it does not perform well for any of the thresholds and input images (except for 128 threshold with monarch images). It is does not take into consideration local variations of details and lightning, it cannot identify situations such as shading (for fiducial and sonnet) and any of the threshold values is good enough to separate the text from the background for the sonnet one. As it can be seen from the respective histograms, the number of pixels classified as objects increases as the threshold value is increased. This might not be adequate for images in general.