1. Graded Exercise: Thresholding and Segmentation

1.1 Instructions

You are allowed 1 hour 45 minutes for this graded exercise and you have to submit your files by 10:00 AM. You may use your notes, books and code from previous exercise sessions, but access to Internet resources is forbidden. Once you are done, make sure that your files are successfully uploaded to the system. There are two exercises, and they do not depend on each other. If you get stuck in one of them, you can proceed to the other. In all, you are asked to implement the following three functions:

- `thresholding.m` for Task 1A
- `adaptive_thresholding.m` for Task 1B
- `superpixel.m` for Task 2.

The files for each task have already been put in separate folders, namely Task 1 and Task 2. In each folder there is a file called `main.m` that sets up necessary data and runs the experiments. You do not have to modify it.

1.2 Task 1: Thresholding (60 pts)

Thresholding is used to segment an image by setting all pixels whose intensity values are above a threshold to a foreground value and all the remaining pixels to a background value.

Whereas the conventional thresholding operator uses a global threshold for all pixels, adaptive thresholding changes the threshold dynamically over the image. In this exercise, you will implement in Task 1A a basic thresholding algorithm and in Task 1B an adaptive thresholding algorithm to threshold the image given in Fig. 1.1.

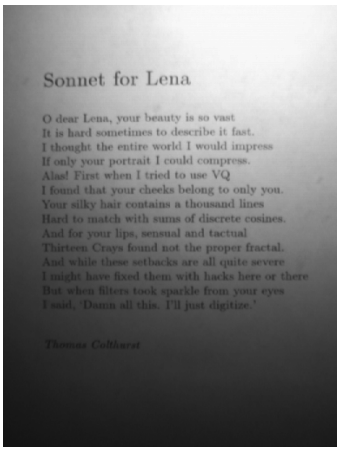

Figure 1.1: The input image which includes a strong illumination gradient.

1.2.1 Task 1A: Basic Thresholding (40 pts)

In `thresholding.m` you are asked to implement the basic thresholding algorithm that finds the best threshold to separate background and the foreground pixels in the image.

Note that, in `main.m`, we have converted the RGB image to grayscale, on which you will run the thresholding algorithm.

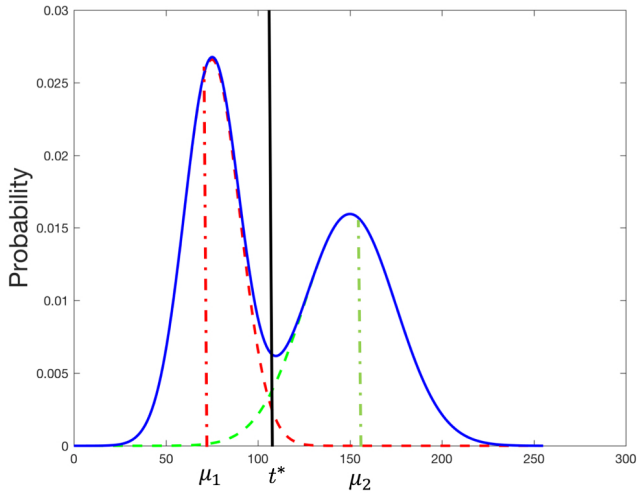

Figure 1.2: An example approximation of image histogram with distributions corresponding to background and foreground. μ_1 and μ_2 are means of the classes and t^* is the threshold returned by Otsu's algorithm.

In particular, you are asked to implement Otsu's algorithm for estimating the optimal threshold. The algorithm assumes that the image contains two classes of pixels following a bi-modal histogram (foreground pixels and background pixels) as shown in Fig. 1.2. It then calculates the optimum threshold separating the two classes, such that their intra-class variance (variance *within* each class) is minimal and their inter-class variance (variance *between* two classes) is maximal. The steps of the algorithm are as follows:

Exercise 1.1 Otsu thresholding implementation (35 pts)

1. Make a histogram of the intensities, h, of the image. The histogram has N number of bins, passed as a parameter. (*Hint:* You can use Matlab's built-in function `imhist`[a].)
2. For every bin i of the histogram, calculate its probability, $p(i)$, which is equal to the value of the bin in the histogram divided by the sum of the values of all bins in the histogram. (*Hint:* To check that you implemented this correctly, the sum of all $p(i)$ should be 1.)
3. For every t from 1 to N compute the cumulative probability of the first class, $\omega_1(t)$, as follows: $\omega_1(t) = \sum_{i=1}^{t} p(i)$
4. Then for every t from 1 to N calculate the class mean: $\mu_1(t) = (\sum_{i=1}^{t} p(i)i)/\omega_1(t)$
5. Use the following formulas to estimate statistics of the second class, e.g. ω_2 and μ_2, for every t from 1 to N:

$$\omega_2(t) = 1 - \omega_1(t) \tag{1.1}$$

$$\mu_2(t) = (\sum_{i=1}^{N} p(i)i - \sum_{i=1}^{t} p(i)i)/\omega_2(t) \tag{1.2}$$

6. Otsu's method exhaustively searches for the threshold that minimizes the variance within the class among all t from 1 to N. Variance within the class is

defined as the weighted sum of variances of the two classes: $\sigma_w^2(t) = \omega_1(t)\sigma_1^2(t) + \omega_2(t)\sigma_2^2(t)$. Otsu[b] shows that minimizing the variance *within* the classes (σ_w^2) is the same as maximizing the variance between the classes (σ_b^2). It was shown that the variance between the classes (σ_b^2) can be expressed as:

$$\sigma_b^2(t) = \omega_1(t)\omega_2(t)(\mu_1(t) - \mu_2(t))^2 \tag{1.3}$$

Therefore, for this step of the algorithm, find the value of t^* that maximizes the variance between the classes given in Eq. (1.3), i.e.,:

$$t^* = \underset{t}{\text{argmax}}\ \sigma_b^2(t) \tag{1.4}$$

7. To find the optimal threshold normalize t^* in the following way:

$$thresh = \frac{t^* - 1}{N - 1} \tag{1.5}$$

8. In the same function, you now need to create a mask, of the same size as the input image, which has 1's for the pixels for which the intensities of the original image are higher than the threshold (*thresh*), and 0's for the others. Apply this mask to the image to threshold the image.

■

[a] If you do not know the specific usage of a function, you can type in Matlab command window "help function_name", e.g. "help imhist".

[b] "A Threshold Selection Method from Gray-level Histograms", N. Otsu, *IEEE Transactions on Systems, Man, and Cybernetics, 1979.*

Once you have implemented these, you should see an output similar to the one given in Fig. 1.3.

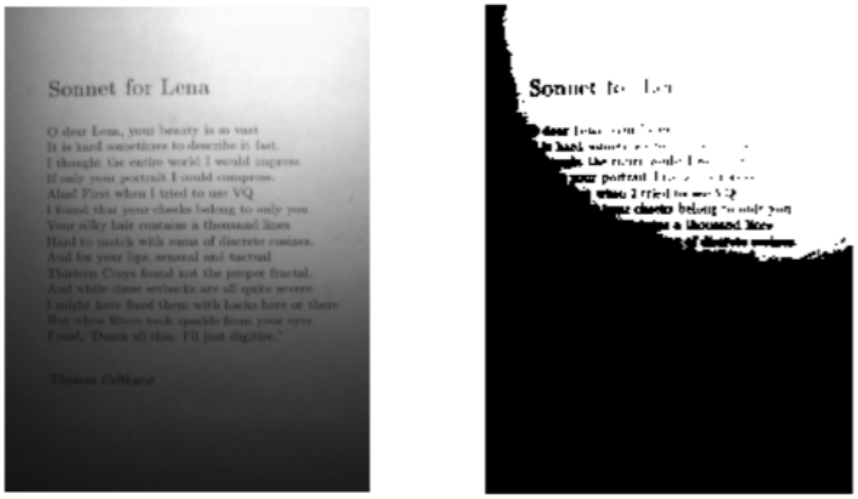

Figure 1.3: Left: Original image, Right: Image obtained after basic thresholding. You may also get a result that is the inverted version of the image on the right, that is correct as well.

Exercise 1.2 Analysis (5 pts)

Why is the performance of basic thresholding so poor? Write your explanation as comments in `thresholding.m`. ■

1.2.2 Task 1B: Adaptive Thresholding (20 pts)

In this task, you will implement an adaptive thresholding algorithm, which will take into account spatial variations in illumination. Similar to the previous exercise, you will work on a grayscale image. You are asked to implement the following steps in `adaptive_thresholding.m` function:

Exercise 1.3 Adaptive thresholding implementation (10 pts)

1. Convolve the image with an averaging filter of size (sz), passed as an argument to the function. Don't forget to correctly process the boundaries. (Hint: you can use Matlab's built-in functions `fspecial` and `imfilter`.)
2. Subtract the original image from the convolved image that you acquired at the previous step.
3. Subtract the threshold (predefined in the code and passed as the third parameter to the function `adaptive_thresholding.m`.) from the resulting image.
4. Create a mask of the same size as the input image, which has the values equal to 1 for those pixels, where the resulting image from the previous step is more than 0 and 0 otherwise. Apply the mask to the image to threshold the image.

■

Once you have implemented these steps, you should see the output in Fig. 1.4.

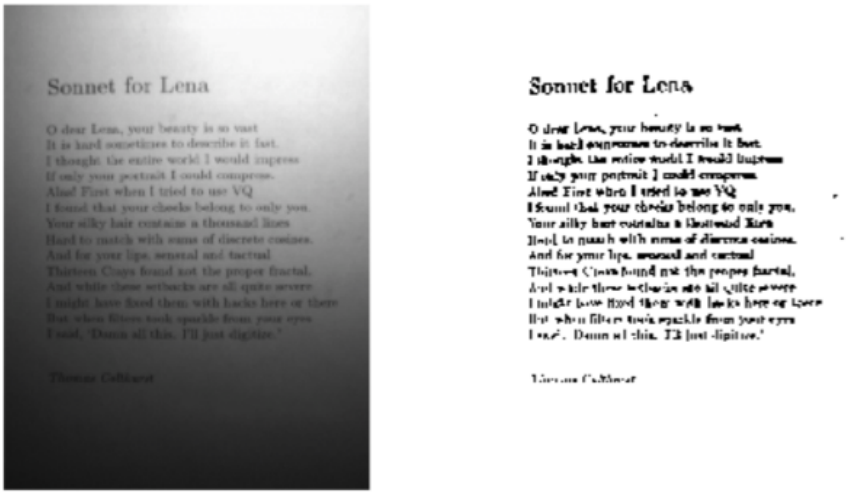

Figure 1.4: Left: Original image, Right: Image obtained after adaptive thresholding. You may also get a result that is the inverted version of the image on the right, that is correct as well.

Exercise 1.4 Analysis (10 pts)

Why is the performance better compared to the basic thresholding algorithm in

Task 1A? How does the adaptive thresholding algorithm in this example account for the spatial variations in the image? Write your explanation as comments in `adaptive_thresholding.m`. ■

1.3 Task 2: Superpixels (40 pts)

In the `superpixel.m` you are now asked to implement the superpixel algorithm, which works on grayscale image. This algorithm is similar to clustering. What you need to do is to combine pixels in the image that are close to each other in terms of distance and intensities. Note that, in the `main.m` we have converted the RGB image to grayscale, on which you run the superpixel algorithm. To do so you are suggested to implement the following steps:

Exercise 1.5 Superpixel segmentation

1. Create a mask of the same size as the 2D grayscale image. Each pixel of the mask contains an ID of the superpixel it belongs to, or zero if it is not assigned to any superpixel yet. Initially, fill all pixels of the mask with zero;
2. Start from the pixel $(1, 1)$ of the image, and set $superPixelCount$ to 1;
3. Find all the pixels (x, y) that are 'close' to it, namely the pixels that fulfill simultaneously the following three conditions:
   - $|im(x, y) - im(i, j)| < diff$ (intensity comparison)
   - $(x - i)^2 + (y - j)^2 < sz^2$ (distance in the image)
   - $mask(x, y) == 0$ (not assigned to any other superpixel)

   The values for *diff* and sz are passed as parameters to the function and they are provided in the code.
4. For each of those pixels, set their value in the mask to $superPixelCount$ to flag that they belong to a new superpixel;
5. Increment $superPixelCount$ by 1;
6. Go to the next pixel (i, j), for which the value of the $mask(i, j)$ is equal to zero;
7. Repeat the steps 3 - 6 until all the pixels in the mask have value different from 0;
8. Return the mask image, which should have all its values different from zero by now.

Hint: you can use `find` and `meshgrid` if necessary.

■

Once you have implemented the steps above, you should see the output similar to the Fig. 1.5, with superpixels of different sizes. The black boundaries represent the boundaries between adjacent superpixels.

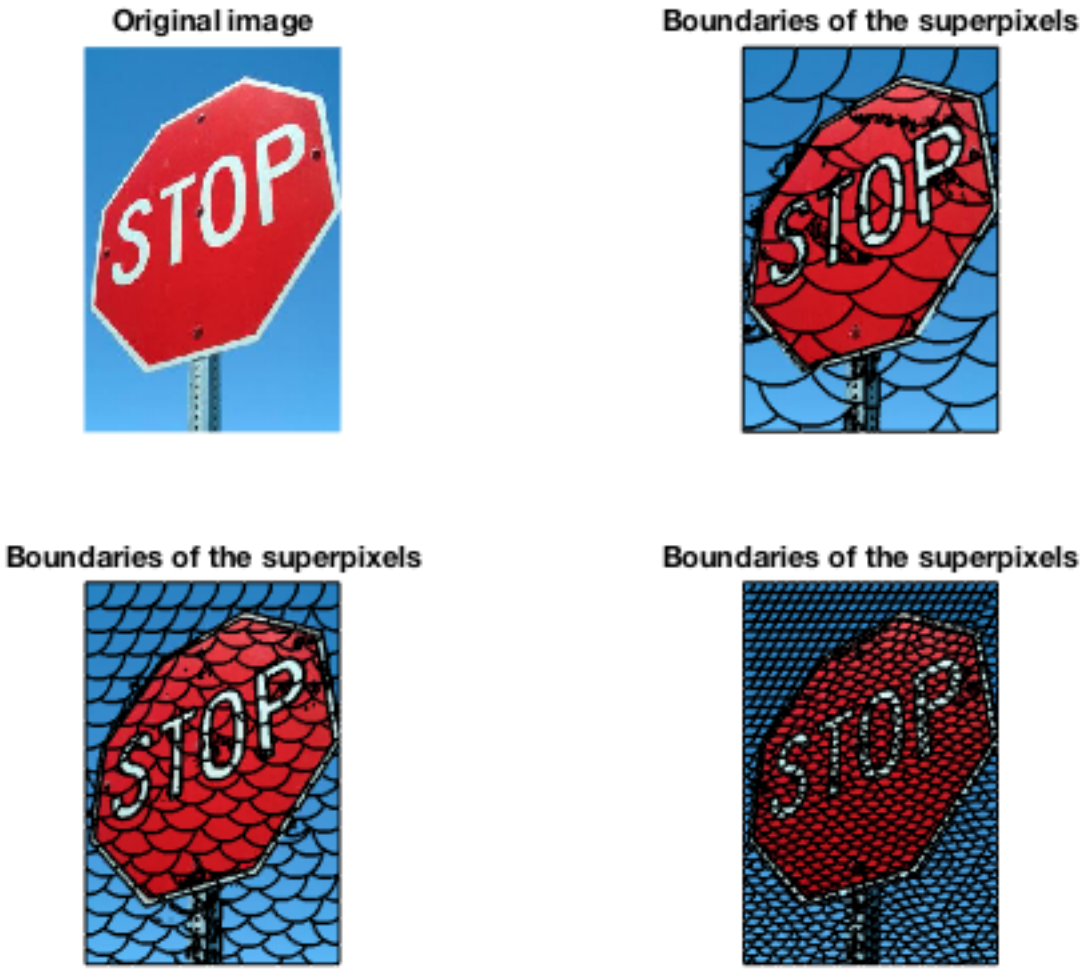

Figure 1.5: Various superpixels obtained from the original image