Homework 1

CS484 - Introduction to Computer Vision

Efe Acer
21602217

Bilkent University, CS

Contents

1 Question 1

A *histogram* in the context of *Image Processing* and *Computer Vision* refers to an *image histogram*. It is a graphical representation, a bar chart, of occurrence counts of each tone in an image. For a grayscale image, the horizontal axis of the histogram goes through the gray tones, which typically range from 0 to 255; and the vertical axis contains the occurrence information. Each bar in the histogram denotes how many pixels in the image correspond to a specific tone of gray. Mathematically, a histogram is defined as:

$$h(i) = |\{(r, c) \mid I(r, c) = i\}| \tag{1}$$

Above i denotes a certain gray intensity (tone) between 0 to 255, whereas r and c correspond to certain rows and columns of the image I. $I(r, c) = i$ evaluates true if the pixel in the rth row and cth column of I has a gray intensity of i.

A simple algorithm that iterates over the image in row-major order and counts the occurrence of each gray intensity is necessary to obtain an array needed to draw the histogram. This algorithm is implemented as a separate function named `count_occurrences`, since obtaining the histogram array is a requisite part of the solutions to the upcoming questions. This method is provided below:

```
% count_occurrences: Given a grayscale image I, this function returns a
% histogram h, which is an array storing the occurrence count of each
% gray tone (from 0 to 255) in the image.
function h = count_occurrences(I)
    h = zeros(1, 256, 'uint32');
    [num_rows, num_cols] = size(I);
    for i = 1:num_rows
        for j = 1:num_cols
            intensity = I(i, j) + 1; % +1 is for proper indexing
            h(intensity) = h(intensity) + 1;
        end
    end
end
```

After obtaining the histogram array `h`, we can use the `bar` method and other `MATLAB` commands to obtain a nice-looking histogram. The `histogram` method is implemented for this purpose:

```
% histogram: Given a grayscale source image, displays the corresponding
% image histogram.
function histogram(source_image)
    % Store the number of pixels corresponding to each gray intensity in h
    h = count_occurrences(source_image);
```

```
6       % Plot the bar chart displaying the occurrence counts (i.e. histogram)
7       figure;
8       bar(0:255, h);
9       format_histogram(); % Draws the GUI details of the histogram
10  end
```

The last line of the `histogram` function calls `format_histogram` method which encapsulates the code needed to draw the GUI details such as titles, labels, and the gray colorbar at the bottom. The implementation of `format_histogram` is available inside the `MATLAB` script submitted together with the report.

The histograms of image 1(a) and image 1(b) obtained using the `histogram` method are shown below:

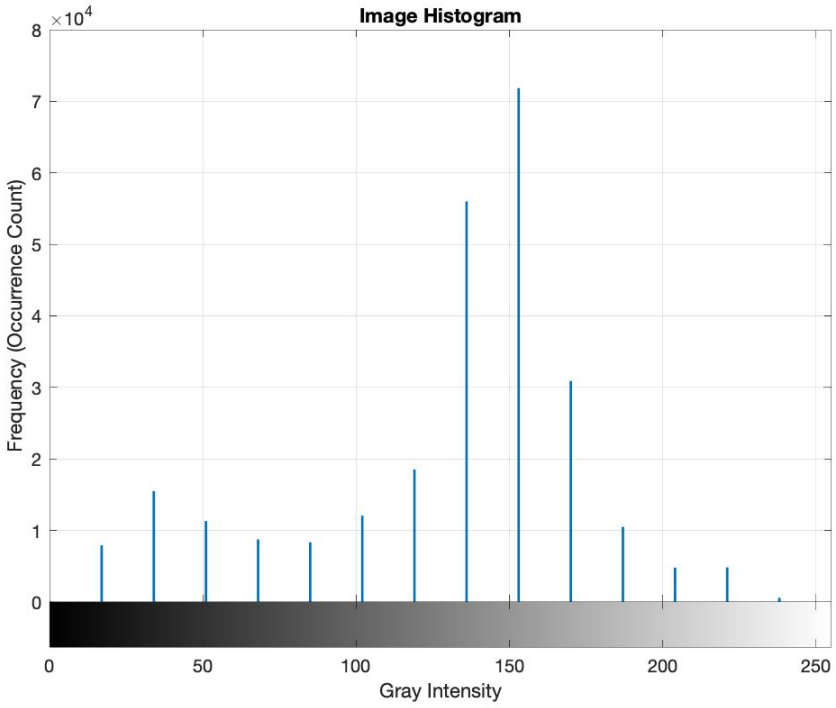

Figure 1: Histogram 1(a)

As it can be seen above, image 1(a) only uses a small fraction of the gray intensity space. There seems to be two peaks corresponding to the peak-points of the distributions of dark and light pixels. The peak of the dark pixel distribution is around the intensity 40, and the peak of the light pixel distribution is around the intensity 150. The distributions overlap and form a valley around the intensity 75, and the dark pixel distribution covers a much smaller number of pixels than the light pixel distribution covers. There are also intensity levels with

nonzero occurrence counts which are not visible due to their occurrence counts being much more smaller than the others.

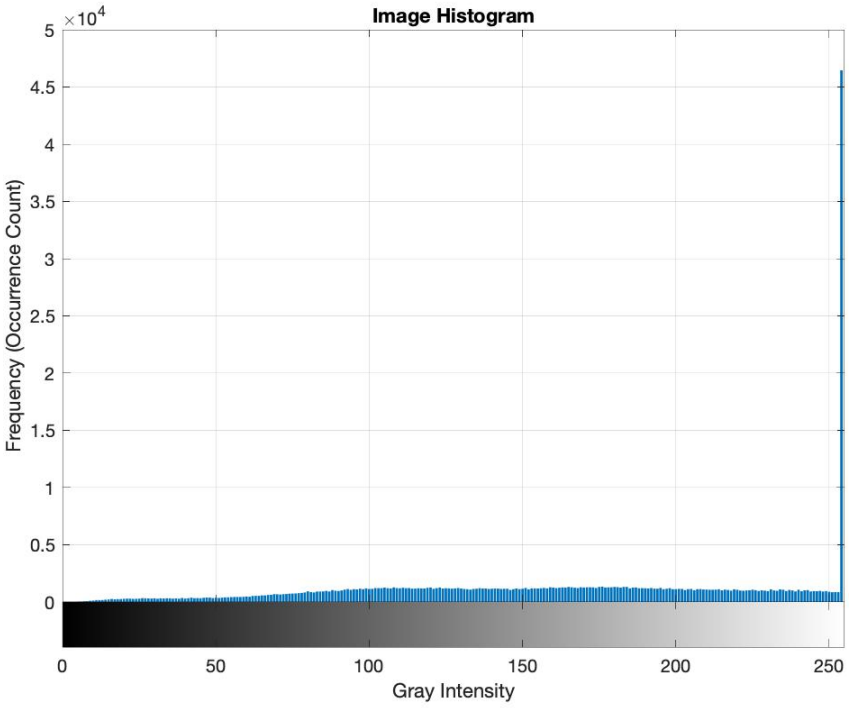

Figure 2: Histogram 1(b)

Looking at histogram 1(b), we can see that there is a roughly uniform distribution of intensity levels. However there is a huge peak at the lightest intensities. We can also confirm this by looking at the actual image 1(b), since the background contains a variety of gray tones but the circular shape in the center mostly includes white pixels.

Note that histograms do not contain any information about the spatial distribution of the gray intensities, this information is already inherent in the actual images.

2 Question 2

Thresholding is a key operation in image processing, which helps to obtain binary images from grayscale images. This way, foreground pixels and background pixels can be separated. The most common form of thresholding is the *threshold above* operation where pixels with gray intensity greater than or equal to a threshold value t are assigned as foreground pixels and others are assigned as background pixels.

Choosing the correct value for t is crucial in thresholding, where $t \in \{0, ..., 255\}$. The value of t directly affects the outcome of thresholding. Assuming that a grayscale image is bimodal, meaning the image consists of two modes corresponding to dark and light pixels, t should be chosen such that the two modes are separated the best. Choosing t by trial and error is oftentimes troublesome.

The *Otsu Method* is an efficient way to determine a good value for t automatically under the bimodal image assumption. The method is discussed below in detail.

The first step of Otsu Method is to obtain a probability distribution over the gray intensities, this is simply done by normalizing the histogram array. In mathematical terms:

$$P(i) = \frac{|\{(r,c) \mid I(r,c) = i\}|}{R \times C} = \frac{h(i)}{R \times C} \text{ ,where } \; i \in \{0, ..., 255\} \tag{2}$$

Then the Otsu Method searches for the best t which is defined as the t value that maximizes the weighted sum of within-group variances; where the groups are $\{(r,c) \mid I(r,c) \geq t\}$ and $\{(r,c) \mid I(r,c) < t\}$, and the weights are the corresponding cumulative probabilities of the groups. The weighted sum of within-group variances as a function of t is denoted as $\sigma_W^2(t)$. Then the problem of finding the best threshold t^* can be formulated as:

$$t^* = \underset{t}{\operatorname{argmin}} \; \sigma_W^2(t) \tag{3}$$

$$\sigma_W^2(t) = w_1(t) \; \sigma_1^2(t) + w_2(t) \; \sigma_2^2(t) \tag{4}$$

$$w_1(t) = \sum_{i=0}^{t} P(i) \qquad w_2(t) = \sum_{i=t+1}^{255} P(i) \tag{5}$$

$$\mu_1(t) = \frac{1}{w_1(t)} \sum_{i=0}^{t} iP(i) \qquad \mu_2(t) = \frac{1}{w_2(t)} \sum_{i=t+1}^{255} iP(i) \tag{6}$$

$$\sigma_1^2(t) = \frac{1}{w_1(t)} \sum_{i=0}^{t} (i - \mu_1(t))^2 P(i) \qquad \sigma_2^2(t) = \frac{1}{w_2(t)} \sum_{i=t+1}^{255} (i - \mu_2(t))^2 P(i) \tag{7}$$

In the equations above $\mu_j(t)$ denotes the mean of group j as a function of t, and $\sigma_j^2(t)$ denotes the variance of group j as a function of t. The formulations of means and variances follow

directly from probability theory where mean is the expected intensity level and variance is the expected squared distance to the mean intensity level.

One can compute t^* by minimizing the within-group variance $\sigma_W^2(t)$ using the formulation above, however there exists a computationally faster approach. Minimizing within-group variance turns out to be equivalent to maximizing the between-group variance, this is proved below.

We begin with writing the formula for the total variance, then we modify it to obtain an expression which justifies the fact that maximizing between-group variance is equivalent to minimizing within-group variance.

$$\sigma^2 = \sum_{i=0}^{255}(i-\mu)^2 P(i) \text{ ,where } \mu = \sum_{i=0}^{255} iP(i)$$

$$\sigma^2 = \sum_{i=0}^{t}(i-\mu_1(t)+\mu_1(t)-\mu)^2 P(i) + \sum_{i=t+1}^{255}(i-\mu_2(t)+\mu_2(t)-\mu)^2 P(i)$$

$$\sigma^2 = \sum_{i=0}^{t}[(i-\mu_1(t))^2 + 2(i-\mu_1(t))(\mu_1(t)-\mu) + (\mu_1(t)-\mu)^2]P(i) + \sum_{i=t+1}^{255}[(i-\mu_2(t))^2 + 2(i-\mu_2(t))(\mu_1(t)-\mu) + (\mu_2(t)-\mu)^2]P(i)$$

We know that $\sum_{i=0}^{t}(i-\mu_1(t))(\mu_1(t)-\mu)P(i) = 0$ due to the fact that $\sum_{i=0}^{t} iP(i) = \mu_1(t)$. Similarly $\sum_{i=t+1}^{255}(i-\mu_2(t))(\mu_2(t)-\mu)P(i) = 0$:

$$\sigma^2 = \sum_{i=0}^{t}[i-\mu_1(t)]^2 P(i) + \sum_{i=0}^{t}[\mu_1(t)-\mu]^2 P(i) + \sum_{i=t+1}^{255}[i-\mu_2(t)]^2 P(i) + \sum_{i=t+1}^{255}[\mu_2(t)-\mu]^2 P(i)$$

Using (4) and (5) we can write:

$$\sigma^2 = w_1(t)\ \sigma_1^2(t) + w_1(t)\ (\mu_1(t)-\mu)^2 + w_2(t)\sigma_2^2(t) + w_2(t)\ (\mu_2(t)-\mu)^2$$

$$\sigma^2 = \sigma_W^2(t) + w_1(t)\ (\mu_1(t)-\mu)^2 + w_2(t)\ (\mu_2(t)-\mu)^2$$

Now using the obvious facts; $\mu = w_1(t)\ \mu_1(t) + w_2(t)\ \mu_2(t)$ and $w_1(t) = 1 - w_2(t)$, and a few simplification steps, we can rewrite the above line as:

$$\sigma^2 = \sigma_W^2(t) + w_1(t)(1-w_1(t))(\mu_1(t)-\mu_2(t))^2 = \sigma_W^2(t) + \sigma_B^2(t) \tag{8}$$

In (8), $\sigma_B^2(t)$ denotes the between-group variance. Since σ^2 is independent of t, minimizing $\sigma_W^2(t)$ is equivalent to maximizing $\sigma_B^2(t)$:

$$t^* = \underset{t}{\text{argmin}}\ \sigma_W^2(t) = \underset{t}{\text{argmax}}\ \sigma_B^2(t) \tag{9}$$

Thus, another valid formulation of Otsu Method is:

$$t^* = \underset{t}{\operatorname{argmax}}\ \sigma_B^2(t) \tag{10}$$

$$\sigma_B^2(t) = w_1(t)\ w_2(t)\ (\mu_1(t) - \mu_2(t))^2 = w_1(t)\ (1 - w_1(t))\ (\mu_1(t) - \mu_2(t))^2 \tag{11}$$

$$\sigma_B^2(t) \propto h_1(t)\ (1 - h_1(t))\ (\mu_1(t) - \mu_2(t))^2 \tag{12}$$

(12) uses $h_1(t)$, frequency distribution of the first group, instead of $p_1(t)$ since normalization does not change the result of the maximization.

$t'*$ can be searched iteratively using the recursive relations ($h(i)$ is used instead of $P(i)$, since it does not change the outcome):

$$w_1(i+1) = w_1(i) + h(i+1) \tag{13}$$

$$\mu_1(i+1) = \frac{\mu_1(i)\ w_1(i) + (i+1)\ h(i+1)}{w_1(i+1)} \qquad \mu_2(i+1) = \frac{\mu - \mu_1(i+1)w_1(i+1)}{1 - w_1(i+1)} \tag{14}$$

(13) follows from (5), and (14) follows from: $\mu = w_1(t)\ \mu_1(t) + w_2(t)\ \mu_2(t)$.

The `MATLAB` implementation of the formulated maximization procedure is provided below:

```
% histogram: Given a grayscale source image, applies automatic otsu
% thresholding to generate a binary output
function binary_image = otsu_threshold(source_image)
    % Obtain the frequency distribution
    h = double(count_occurrences(source_image));
    total_sum = dot(0:255, h); % total sum of gray intensities
    num_pixels = sum(h);

    % Initialization
    w1 = 0; % weight (number of pixels) of the first cluster
    sum1 = 0; % sum of gray intensities in the first cluster

    % Iterate all values to search for the maximizing t
    t = 0;
    max_var = -inf;
    for i = 1:255
        % Recursively update the weights and the centers
        w1 = w1 + h(i);
        sum1 = sum1 + (i - 1) * h(i);
        mu1 = sum1 / w1;
        mu2 = (total_sum - sum1) / (num_pixels - w1);
        % Calculate the between-group variance
        var = w1 * (num_pixels - w1) * (mu1 - mu2)^2;
```

```
        % Update maximizing t if necessary
        if var > max_var
            t = i - 1;
            max_var = var;
        end
    end
    fprintf('Maximing t = %d\n', t); % To see the optimal threshold

    % Apply threshold above operation using the maximizer t
    binary_image = source_image >= t;
end
```

The resulting binary images for figures 2(a) and 2(b) are shown below after Otsu Method was applied:

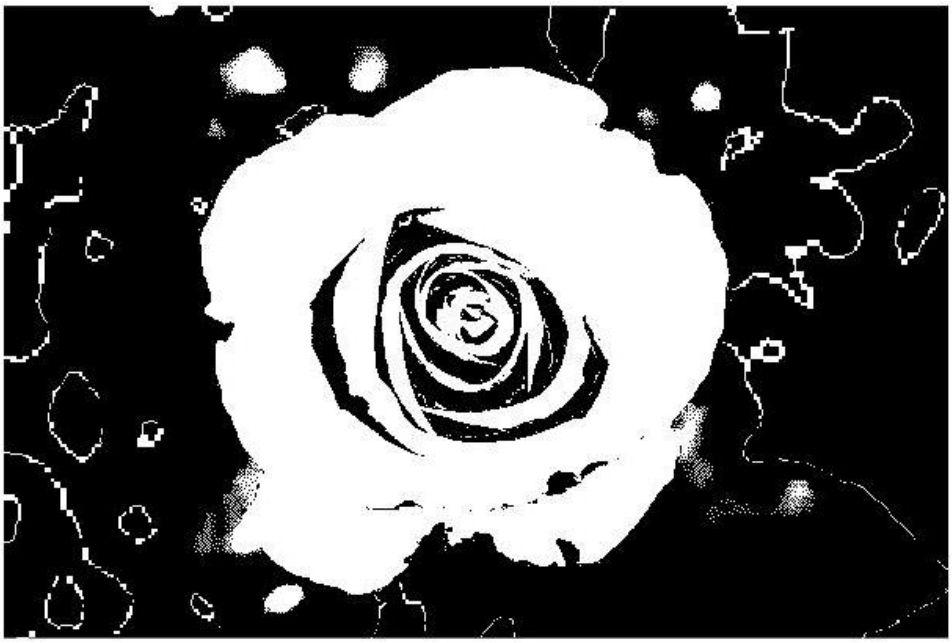

Figure 3: Binary Image 2(a) ($t^* = 15$)

Looking at Figure 3, we can see that Otsu Method performed very well in separating the background from the foreground when applied to figure 2(a). This implies that the original image indeed conforms to the initial bimodal image assumption. The distribution of dark pixels and the distribution of light pixels do not overlap much in figure2(a), since darker pixels are located around the flower while lighter ones establish the flower itself. In other words, the distributions are spatially separated which makes the image a suitable input for Otsu Method.

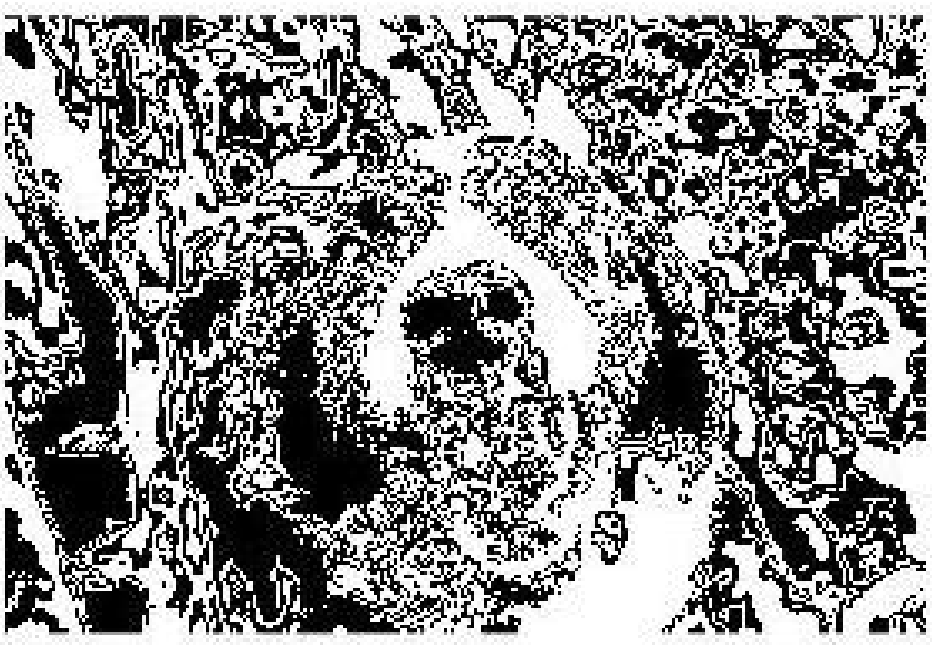

Figure 4: Binary Image 2(b) ($t^* = 19$)

Inspecting Figure 4, we can say that Otsu Method did not perform well for figure 2(b). When we look at the original image, we see that both the dog and the background has a large variance in terms of gray intensity. To be clearer, we can observe both dark and light pixels when we look at a small neighborhood of dog pixels, and also background pixels. This violates the assumption that the image is bimodal, since the distributions of dark pixels and light pixels have been spatially merged together.

3 Question 3

Two fundamental operations of binary *morphology* are *dilation* and *erosion*. Dilation and erosion can be mathematically defined as set operations over the inputs B and S, where B is a binary image and S is a structuring element. S can represent any arbitrary shape as a bitmap.

Dilation, as the name implies, enlarges the boundaries of an object. The definition of dilation is:

$$B \oplus S = \bigcup_{b \in B} S_b \tag{15}$$

In words, the structuring element S is slid over the binary image B and whenever the center of S overlaps with a 1-pixel in B, S is ORed to the output image, which was initially initialized with zeros.

The `MATLAB` code for dilation is given below:

```
% dilation: Given a binary source image and a binary structuring element,
% applies dilation to the image to enlarge the object boundaries.
function dilated_image = dilation(source_image, struct_el)
    % Get the necessary dimension information and compute padding values
    [height_s, width_s] = size(struct_el);
    pad_x = floor(width_s / 2);
    pad_y = floor(height_s / 2);

    % Apply 0 padding to the original image
    I = padarray(source_image, [pad_y, pad_x]);

    % Find indices of 1's
    [row, col] = find(I == 1);

    dilated_image = false(size(I));

    for i = 1:length(row)
        x_begin = col(i) - pad_x;
        x_end = x_begin + width_s - 1;
        y_begin = row(i) - pad_y;
        y_end = y_begin + height_s - 1;
        % There must be at least one 1 in the overlapping section
        if nnz(I(y_begin:y_end, x_begin:x_end) & struct_el) > 0
            % Fill the neighborhood by or'ing with struct_el
            dilated_image(y_begin:y_end, x_begin:x_end) = ...
                dilated_image(y_begin:y_end, x_begin:x_end) | struct_el;
        end
```

```
    end

    % Remove padding
    dilated_image = dilated_image((pad_y + 1):(end - pad_y), ...
        (pad_x + 1):(end - pad_x));
end
```

The above code first finds the pixel indices with 1's and goes over these indices only, this decreases the run-time significantly for sparse B. The code inside the for loop simply checks whether or not there is at least one 1-pixel in the overlapping part of B and S, then performs ORing if necessary.

Erosion, on the contrary, shrinks the object boundaries and thus makes the objects smaller. The definition of erosion is given below:

$$B \ominus S = \{b \mid b + s \in B \ \forall s \in S\} \tag{16}$$

Again, in words, the structuring element S is slid over the binary image B and whenever every 1-pixel of S overlap with a 1-pixel in B, the pixel corresponding to the origin of S is ORed to the output image.

The MATLAB code for erosion is given below:

```
% erosion: Given a binary source image and a binary structuring element,
% applies erosion to the image to shrink the object boundaries.
function eroded_image = erosion(source_image, struct_el)
    % Get the necessary dimension information and compute padding values
    [height_i, width_i] = size(source_image);
    [height_s, width_s] = size(struct_el);
    pad_x = floor(width_s / 2);
    pad_y = floor(height_s / 2);

    % Apply 0 padding to the original image
    I = padarray(source_image, [pad_y, pad_x]);

    % Find indices of 1's
    [row, col] = find(I == 1);

    % Find number of 1's in the structuring element
    nnz_s = nnz(struct_el);

    eroded_image = false(height_i, width_i);

    for i = 1:length(row)
        x_begin = col(i) - pad_x;
        x_end = x_begin + width_s - 1;
```

```
        y_begin = row(i) - pad_y;
        y_end = y_begin + height_s - 1;
        % Number of 1's in the and result must equal that of the struct_el
        if nnz(I(y_begin:y_end, x_begin:x_end) & struct_el) == nnz_s
            eroded_image(y_begin, x_begin) = ...
                eroded_image(y_begin, x_begin) | ...
                struct_el(1 + pad_y, 1 + pad_x);
        end
    end
end
```

The procedure is more or less the same with dilation, but now the code inside the for loop checks whether or not the 1-pixels in S and B overlap entirely and then ORs the pixel corresponding to the origin of S to the output image if necessary.

4 Question 4

In this section, we aim to label and count the objects in two aerial images while preserving the structure of the objects. One of the given aerial images is a grayscale image of parking trucks (image 3(a)) and the other is a grayscale image of parking planes (image 3(b)). Our general processing pipeline to do the labeling tasks is as follows: The specific pipelines for

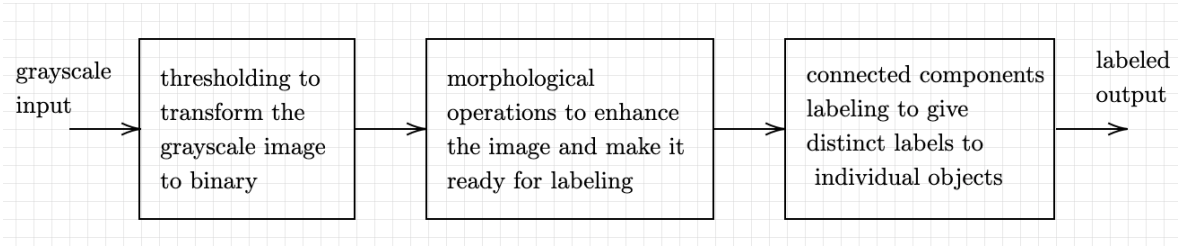

Figure 5: General pipeline for object labeling

the images together with the results of each step are provided below:

4.1 Pipeline for image 3(a)

4.1.1 Thresholding

Otsu Method is used to determine an appropriate threshold t and apply the threshold above operation. As an alternative way, t can be chosen by trial and error; however that would be troublesome and time-consuming. The result of thresholding is given below:

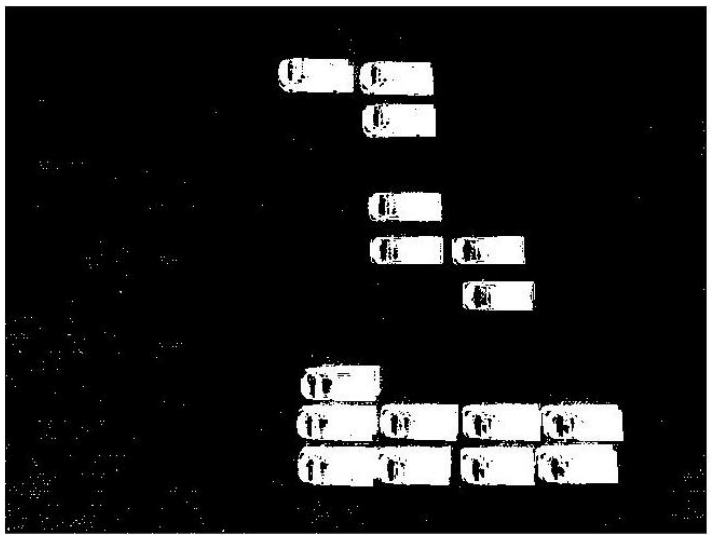

Figure 6: Image 3(a): Result of Thresholding ($t^* = 88$)

As a side-note reversal is applied to the image pixels after the thresholding just to confrom to the convention that the pixels representing the objects are 1's (whites). The code to achieve this step is given below:

```
% Apply thresholding and reversal
binary_image_3a = ~otsu_threshold(image_3a);
```

4.1.2 Morphological Operations

The result is given below, before the explanations to make the argument more meaningful:

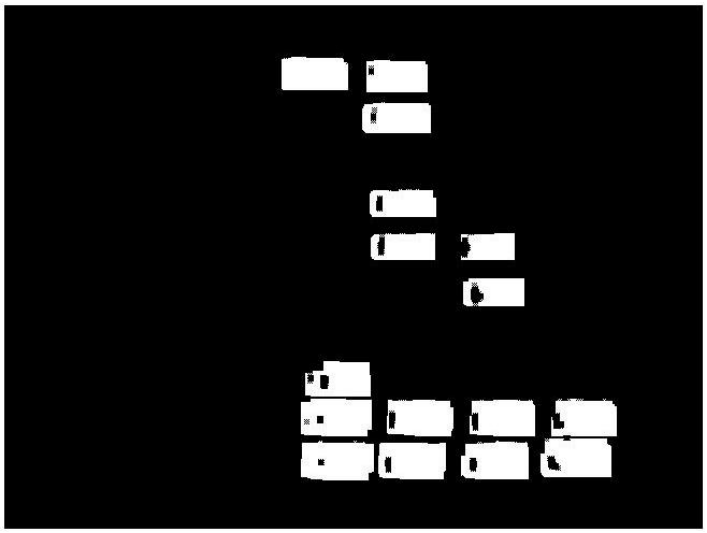

Figure 7: Image 3(a): Result of Morphological Operations

The code to achieve this is given below, the code also specifies the order of the operations performed to get the result above:

```
s = logical([
    1 0 1;
    0 1 0;
    1 0 1]);

s_ = logical([
    1 0 0 0 1;
    0 1 0 1 0;
    0 0 1 0 0;
    0 1 0 1 0;
    1 0 0 0 1]);

```

```
13 left_s = logical([
14     1 1 1 1 1 0 0 0 0 0;
15     1 1 1 1 1 0 0 0 0 0]);
16
17 left_s1 = logical([
18     1 1 1 1 1 0 0 0 0 0]);
19
20 % Apply morphological operations
21 result_3a = binary_image_3a;
22 result_3a = erosion(result_3a, ones(5, 3));
23 result_3a = erosion(result_3a, ones(5, 3));
24 result_3a = erosion(result_3a, ones(5, 3));
25 result_3a = erosion(result_3a, ones(5, 3));
26 result_3a = dilation(result_3a, ones(7, 7));
27 result_3a = dilation(result_3a, ones(5, 5));
28 result_3a = dilation(result_3a, ones(5, 5));
29 result_3a = dilation(result_3a, left_s);
30 result_3a = dilation(result_3a, left_s);
31 result_3a = dilation(result_3a, left_s);
32 result_3a = dilation(result_3a, left_s1);
33 result_3a = dilation(result_3a, left_s1);
34 result_3a = result_3a & opening(closing(binary_image_3a, s_), s);
```

The lines 21-25 apply erosion to the image with a rectangular structuring element to remove the unwanted connections in between the objects. After these four lines, the objects become separate but they also become a lot smaller than their actual size, for this reason the lines 26-33 apply dilation to the image.

First three dilation operations are carried out with square structuring elements because their purpose is just to enlarge the object boundaries, however the last five dilation operations are carried out with `left_s` and `left_s1` structuring elements which contain ones to the halfpoint and zeros to the end. This helps widen the image in the left direction.

At the end, line 34 applies *opening* and *closing* operations to the binary image, and ANDs the output with the result after line 33. Opening is just an erosion followed by a dilation (using the same structuring element), and closing is the other way around. The structuring elements used in the opening and closing operations were X-shaped because X-shaped structuring elements perform well on connecting object parts that are diagonally separated. Finally, the AND operation is performed to capture the small details present in the original image.

Opening and closing codes are also given below:

```
1 function closed_image = closing(source_image, struct_el)
2     closed_image = erosion(dilation(source_image, struct_el), struct_el);
3 end
```

```
4
5 function opened_image = opening(source_image, struct_el)
6     opened_image = dilation(erosion(source_image, struct_el), struct_el);
7 end
```

4.1.3 Connected Components Labeling

For connected components labeling `MATLAB`'s built-in `bwlabel` function is used. The result of the labeling is as follows:

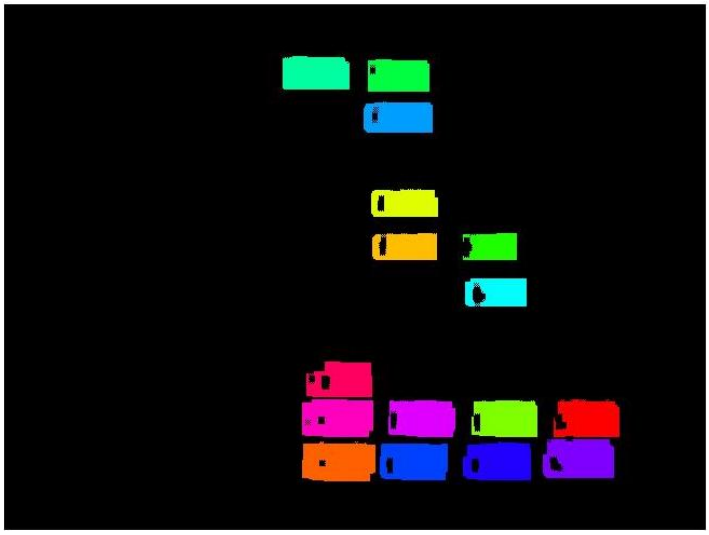

Figure 8: Image 3(a): Result of Connected Components Labeling

The code to achieve this is:

```
1 % Perform connected component labeling
2 [L, n] = bwlabel(result_3a);
3 fprintf("Number of labeled components (image 3(a)): %d\n", n);
4 colored_3a = label2rgb(L, 'hsv', 'k', 'shuffle');
```

The above code outputs: `Number of labeled components (image 3(a)):  16`.

Thus, the pipeline for image 3(a) succeeded to label each object individually and count them with full precision. However, the structures of the objects are a little disturbed. This is mainly because of the inevitable object-counting object-filling trade-off.

4.2 Pipeline for image 3(b)

4.2.1 Thresholding

Again Otsu Method is used to obtain the binary version of image 3(b), but this time no reversal was applied to the thresholded image. The result and the code are given below:

Figure 9: Image 3(b): Result of Thresholding ($t^* = 14$)

```
% Apply thresholding
binary_image_3b = otsu_threshold(image_3b);
```

4.2.2 Morphological Operations

As image 3(b) is far more complicated than image 3(a) in terms of the geometry and alignment of the objects, it was relatively harder to think of a meaningful sequence of morphological operations. First the code to achieve the output is given, and then the output itself:

```
diag1 = logical([
    1 0 0 0 0;
    0 1 0 0 0;
    0 0 1 0 0;
    0 0 0 1 0;
    0 0 0 0 1]);
```

```
7
8  diag2 = logical([
9      0 0 0 0 0 0 1;
10     0 0 0 0 0 1 0;
11     0 0 0 0 1 0 0;
12     0 0 0 1 0 0 0;
13     0 0 1 0 0 0 0;
14     0 1 0 0 0 0 0;
15     1 0 0 0 0 0 0]);
16
17 % Apply morphological operations
18 result_3b = binary_image_3b;
19 result_3b = erosion(result_3b, diag2);
20 result_3b = closing(result_3b, ones(5, 5));
21 result_3b = erosion(result_3b, diag1);
22 result_3b = dilation(result_3b, diag2);
23 result_3b = dilation(result_3b, ones(5, 5));
24 result_3b = dilation(result_3b, ones(5, 5));
25 result_3b = closing(result_3b, ones(5, 5));
26 result_3b = dilation(result_3b, diag2);
```

Figure 10: Image 3(b): Result of Morphological Operations

In the given code, the structuring elements `diag1` and `diag2` are basically diagonally placed lines of different directions. These structuring elements are used specifically in erosion and dilation since the runway lines and plane positions are also diagonal. The other structuring element used was a regular 5-by-5 box, which helps to enlarge and shrink the object boundaries. The order of the operations was selected in a trial-and-error fashion.

4.2.3 Connected Components Labeling

The labeling code and its output are as follows:

```
% Perform connected component labeling
[L, n] = bwlabel(result_3b);
fprintf("Number of labeled components (image 3(b)): %d\n", n);
colored_3b = label2rgb(L, 'hsv', 'k', 'shuffle');
```

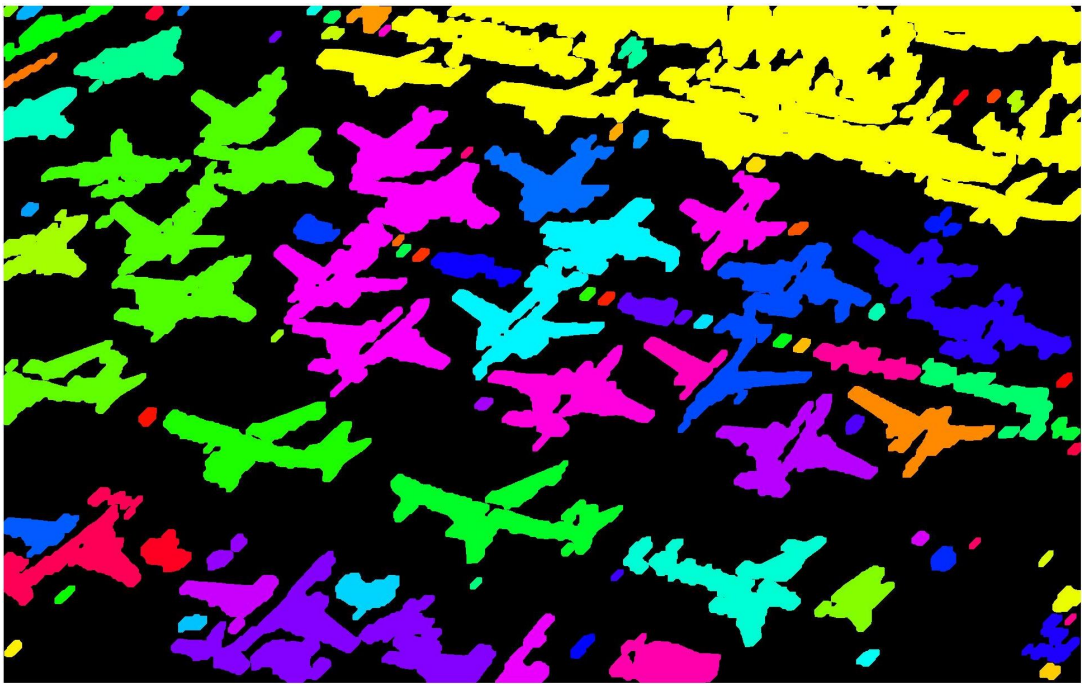

Figure 11: Image 3(b): Result of Connected Components Labeling

The code above outputs: `Number of labeled components (image 3(b)):  89`.

Thus, the pipeline for image 3(b) did not give a high precision. The actual count of planes was 34. However, considering that there were many other objects (such as vehicles, runway lines, trees, etc.) in the background; the performance of the pipeline was not that poor.