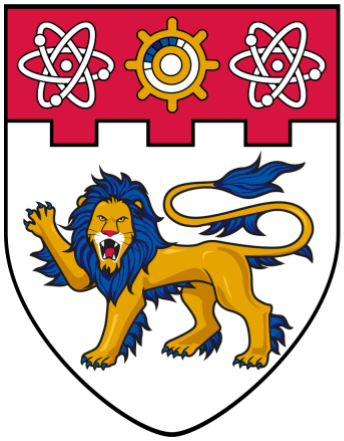

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

SC4061 – Computer Vision
Laboratory 2

Cholakov Kristiyan Kamenov
U2123543B

College of Computing and Data Science

1 Table of Contents

2 Introduction

The objective of the laboratory is to introduce the following fundamental concepts in the Computer Vision field via a series of practical experiments:

- Image Segmentation: Various segmentation techniques, including Otsu's global thresholding, Niblack's local thresholding with Bayesian optimization, Sauvola's thresholding, and enhanced segmentation using Gabor filter banks combined with optimized K-Means clustering. These methods were used to isolate text from degraded document images and to optimize segmentation results based on quantitative evaluation metrics.
- 3D Stereo Vission: Techniques to calculate disparity maps for synthetic and real stereo images. Using algorithms such as Sum of Squared Differences (SSD), Zero-Mean Normalized Cross-Correlation (ZNCC), and Sum of Absolute Differences (SAD), depth perception from stereo image pairs was explored by computing disparity maps that show pixel-by-pixel depth information.

The experiments were conducted using MATLAB and its Image Processing Toolbox, together with the Statistics and Machine Learning Toolbox. MATLAB's robust environment allowed for hands-on experience applying these techniques and understanding their applications. The images used for the experiments were stored in a separate directory **img**, inside the working directory of the laboratory MATLAB file.

3 Experiments

In this section, I will examine each experiment in detail, starting with an overview of the key concepts involved, followed by a review of the MATLAB code used to implement the experiment, and concluding with an analysis of the results and insights drawn from the experiment.

3.1 Image Segmentation

3.1.1 Concept

Image segmentation is the process of partitioning an image into meaningful regions to simplify analysis or isolate areas of interest, such as text in degraded document images. In the context of the assignment, image segmentation was used to isolate the black font text from the background of document images affected by various types of degradation. Different segmentation techniques work differently depending on the image conditions. In this lab, I tried out several methods to handle the challenges posed by degraded document images.

Otsu's Global Thresholding

Otsu's method is a global thresholding technique designed to determine an optimal threshold T that separates the foreground and background based on the intensity histogram of an image. It works particularly well when the histogram has a bimodal distribution (two distinct peaks), with each peak representing one of the two classes.

The goal is to find T that minimizes the intra-class variance (the variance within each class) or, equivalently, maximizes the inter-class variance (the variance between the two classes). Here's a breakdown of the key mathematical concepts involved:

The total variance σ^2_{total} of the image, which represents the overall spread of intensity values, is defined as:

$$\sigma^2_{\text{total}} = \omega_0(T)\sigma_0^2(T) + \omega_1(T)\sigma_1^2(T)$$

where:

- $\omega_0(T)$ and $\omega_1(T)$ are the probabilities of the foreground and background classes, respectively, as defined by the threshold T.
- $\sigma_0^2(T)$ and $\sigma_1^2(T)$ are the variances within the foreground and background classes.

To calculate these probabilities, Otsu's method uses the image histogram. The class probabilities up to a given threshold T are:

$$\omega_0(T) = \sum_{i=0}^{T-1} p(i) \quad \text{and} \quad \omega_1(T) = \sum_{i=T+1}^{L-1} p(i)$$

where:

- $p(i)$ represents the probability of each intensity level i, calculated as $\frac{\text{number of pixels at level } i}{\text{total number of pixels}}$.
- L is the total number of intensity levels in the image.

The mean intensities of each class, $\mu_0(T)$ and $\mu_1(T)$, are calculated as:

$$\mu_0(T) = \frac{\sum_{i=0}^{T-1} i \cdot p(i)}{\omega_0(T)} \quad \text{and} \quad \mu_1(T) = \frac{\sum_{i=T+1}^{L-1} i \cdot p(i)}{\omega_1(T)}$$

Otsu's algorithm then focuses on maximizing the inter-class variance σ_B^2, which represents the separation between the foreground and background. It is defined as:

$$\sigma_B^2(T) = \omega_0(T)\omega_1(T)\big(\mu_0(T) - \mu_1(T)\big)^2$$

Maximizing $\sigma_B^2(T)$ is equivalent to finding the threshold T that creates the largest possible separation between the two classes, yielding an optimal distinction between foreground and background.

To find this optimal threshold T, Otsu's method iterates over all possible values of T and selects the one that maximizes $\sigma_B^2(T)$:

$$T = \arg\max_T \sigma_B^2(T)$$

Once the optimal threshold T is determined, it is applied to the image, classifying each pixel as foreground or background based on whether its intensity is above or below T.

Otsu's method is effective when there's a clear intensity difference between text and background; however, it may perform less effectively in cases with non-uniform illumination or significant noise, as these conditions can obscure the bimodal nature of the histogram.

Niblack's Local Thresholding

Unlike the global approach in Otsu's algorithm, Niblack's method is a local thresholding technique designed to handle images with varying distortion and noise, making it suitable for segmenting text from degraded document backgrounds. Unlike global thresholding, which applies a single threshold across the entire image, Niblack's method calculates a unique threshold T for each pixel based on the local characteristics within a specified window. The threshold T at each pixel location (x, y) is computed as:

$$T(x, y) = \mu(x, y) + k \cdot \sigma(x, y)$$

where $\mu(x,y)$ is the mean intensity within the window centered at (x,y), $\sigma(x,y)$ is the standard deviation of intensities within the same window, and k is a user-defined parameter that adjusts the sensitivity of the thresholding. Positive values of k make the threshold higher, favoring background pixels, while negative values lower the threshold, favoring foreground pixels (such as text). The local mean $\mu(x,y)$ and standard deviation $\sigma(x,y)$ within the window around each pixel (x,y) are computed as follows:

$$\mu(x,y) = \frac{1}{W \times H} \sum_{i,j \in W(x,y)} I(i,j)$$

where $I(i,j)$ represents the intensity at pixel (i,j), $W \times H$ is the size of the local window, and $W(x,y)$ is the set of pixels within the window centered at (x,y). The standard deviation is given by:

$$\sigma(x,y) = \sqrt{\frac{1}{W \times H} \sum_{i,j \in W(x,y)} \left(I(i,j) - \mu(x,y)\right)^2}$$

This measures the intensity spread within the window, providing insight into local contrast. Higher values of σ indicate areas with more variation, which is useful for distinguishing text from background.

In Niblack's thresholding, the choice of window size and k significantly impacts performance, especially in degraded or unevenly illuminated regions. Bayesian Optimization can be used to automate the tuning of these parameters by defining an objective function that minimizes segmentation error and finding the optimal values for k and window size. The optimization process iteratively tests various combinations of k and window size, evaluating performance to identify the optimal values that minimize segmentation error.

This method adjusts the threshold locally, which, along with Bayesian optimization, helps extract text from unevenly lit or degraded documents. Niblack's method is good at handling local contrast and intensity changes, making it useful for difficult segmentation tasks.

Sauvola's Local Thresholding

Sauvola's method builds upon Niblack's local thresholding approach by introducing an additional parameter R to better handle areas with high variance, such as noisy or textured backgrounds. This enhancement makes Sauvola's method more adaptable to challenging image conditions, where there may be a significant contrast between text and background, or uneven lighting.

The threshold T at each pixel location (x,y) is calculated as:

$$T(x,y) = \mu(x,y)\left(1 + k \cdot \left(\frac{\sigma(x,y)}{R} - 1\right)\right)$$

where:

- $\mu(x,y)$ is the local mean intensity within a window centered at (x, y),
- $\sigma(x,y)$ is the local standard deviation within the same window,
- k is a sensitivity parameter similar to that in Niblack's method,
- R is a dynamic range parameter, usually set to the maximum standard deviation expected in the image (e.g., 128 for an 8-bit grayscale image).

In Sauvola's formula, the term $\frac{\sigma(x,y)}{R}$ provides a normalization factor that adjusts the threshold based on local contrast. When the local standard deviation $\sigma(x,y)$ is high, the threshold T is lowered, making it more likely to classify high-contrast regions (such as text) as foreground. Conversely, in areas with low variance, the threshold remains close to the local mean, helping to suppress background noise.

This method works effectively for document images, especially where text may be on a noisy or uneven background. By tuning k and R, the thresholding can be adjusted to emphasize or de-emphasize regions of high contrast, allowing for more robust text extraction.

As with Niblack's method, Bayesian Optimization was used here to automatically select optimal values for k, R, and the window size, aiming to minimize segmentation error. Bayesian Optimization iteratively explores various parameter combinations and evaluates segmentation quality, converging on values that yield the best performance for each image. This optimization ensures that Sauvola's thresholding adapts well to diverse document conditions, maximizing clarity in text extraction.

Filter Bank with K-Mean Clustering

This technique uses Gabor filters to extract texture-based features that emphasize text regions. A Gabor filter $G(x, y; \lambda, \theta)$ is defined as:

$$G(x, y; \lambda, \theta) = \exp\left(-\frac{x'^2 + \gamma^2 y'^2}{2\sigma^2}\right)\cos\left(2\pi\frac{x'}{\lambda} + \psi\right)$$

where:

- $x' = x\cos\theta + y\sin\theta$ and $y' = -x\sin\theta + y\cos\theta$ are the rotated coordinates,
- λ is the wavelength, θ is the orientation, σ is the scale, γ is the aspect ratio, and ψ is the phase offset.

For each orientation θ and wavelength λ , the filtered image captures directional texture details. By pooling and smoothing these Gabor responses, we create a down-sampled representation, denoted $P\big(G(x, y; \lambda, \theta)\big)$, where pooling is applied using Gaussian smoothing with a pooling scale s.

The pooled Gabor features are concatenated with the original intensity $I(x, y)$ and gradient magnitude $|\nabla I|$ to form the complete feature set:

$$FeatureSet = [P(G(x, y;\ \lambda, \theta)), I(x, y), |\ \nabla I\ |]$$

This feature set is normalized by standardizing each feature dimension:

$$\text{NormalizedFeature} = \frac{\text{FeatureSet} - \mu}{\sigma}$$

where μ and σ are the mean and standard deviation of each feature dimension.

For clustering, we apply K-Means to group pixels into k clusters by minimizing the sum of squared distances to the nearest cluster center μ_i :

$$\min \sum_{i=1}^{k} \sum_{x \in C_i} |\ x - \mu_i|^2$$

where C_i are clusters and μ_i are their centroids. Each cluster's mean intensity is analyzed, and the cluster with the lowest mean intensity is typically identified as the text region. The final binary image $B(x, y)$ is created by mapping this cluster to a binary output.

3.1.2 MATLAB Code Analysis and Results

3.1.2.1 Part a)

MATLAB Code:

As seen in Figure 1, I am starting the code for Otsu's Thresholding by defining the image folder and the document files with their corresponding groundtruths. the difference between the thresholded image and the groundtruth. This range, spanning from 0 to 1 in steps of 0.05, provides a detailed set of candidate thresholds for testing.

Next, I am entering a for loop where I process each image individually. For each image, I read both the image and its groundtruth file using the **imread** function. The code checks if the images are in color (RGB), and if so, it converts them to grayscale to ensure consistency in the analysis. Similarly, the groundtruth images are verified to be binary, converting them with **imbinarize** if necessary. This preprocessing ensures that each image is uniformly prepared for the thresholding steps that follow, facilitating accurate comparisons against the groundtruth.

```
%% a) Otsu's Global Thresholding with Multiple Thresholds

% Path to the images folder
imgFolder = 'img/';

% List of images and their ground truths
imageFiles = {'document01.bmp', 'document02.bmp', 'document03.bmp', 'document04.bmp'};
gtFiles = {'document01-GT.tiff', 'document02-GT.tiff', 'document03-GT.tiff', 'document04-GT.tiff'};

% Range of thresholds to test
thresholdValues = 0:0.05:1;  % From 0 to 1 in steps of 0.05

% Process each image
for i = 1:length(imageFiles)
    % Read the image and its ground truth
    imgPath = fullfile(imgFolder, imageFiles{i});
    gtPath = fullfile(imgFolder, gtFiles{i});
    img = imread(imgPath);
    gt = imread(gtPath);

    % Convert to grayscale if necessary
    if size(img, 3) == 3
        img = rgb2gray(img);
    end
    if size(gt, 3) == 3
        gt = rgb2gray(gt);
    end

    % Ensure ground truth is binary
    if ~islogical(gt)
        gt = imbinarize(gt);
    end
```

Figure 1. Image Segmentation – the code for Otsu's Global Thresholding (1/3)

In Figure 2, I continue with the code for the for loop, iterating through the document images. First, I apply Otsu's global thresholding and then initiate testing for multiple threshold values on each image. The Otsu threshold level (**otsuLevel**) is computed using the **graythresh** function, which calculates the optimal threshold to separate foreground and background pixels based on Otsu's method. The image is binarized using this threshold to

produce **binaryImgOtsu**, which represents the Otsu-thresholded binary version of the image. To evaluate how closely this result matches the groundtruth, I compute the difference using the **xor** function, which identifies discrepancies between the Otsu-thresholded image and the groundtruth, summing them as **otsuDiffSum**. This difference sum serves as a benchmark for comparing the Otsu threshold with other tested thresholds.

Following this, I initialize variables (**minDiffSum**, **bestThreshold**, and **bestBinaryImg**) to keep track of the lowest difference sum and corresponding threshold, allowing the code to identify the most accurate threshold for each image. I then enter a nested loop where I test each threshold in **thresholdValues**. For each threshold value, I binarize the image, compute the difference with the groundtruth using **xor**, and calculate the difference sum (**diffSum**). I store each threshold and its difference sum in **thresholdDiffs**, which allows me to analyze and plot the performance of each threshold later. If a threshold achieves a lower difference sum than the current minimum, I update **bestThreshold** and **bestBinaryImg** to reflect this improved result. This iterative testing process enables me to find the threshold value that best aligns with the groundtruth for each image, potentially yielding a more accurate result than the Otsu threshold alone.

```
% Apply Otsu's thresholding
otsuLevel = graythresh(img);
binaryImgOtsu = imbinarize(img, otsuLevel);
otsuDiff = xor(binaryImgOtsu, gt);
otsuDiffSum = sum(otsuDiff(:));

% Initialize variables to find the best threshold
minDiffSum = inf;
bestThreshold = otsuLevel;
bestBinaryImg = binaryImgOtsu;

% Store sum differences for each threshold
thresholdDiffs = zeros(length(thresholdValues), 2);

% Test each threshold value
for j = 1:length(thresholdValues)
    t = thresholdValues(j);

    % Apply thresholding
    binaryImg = imbinarize(img, t);

    % Compute difference with ground truth
    diffImg = xor(binaryImg, gt);
    diffSum = sum(diffImg(:));

    % Store results
    thresholdDiffs(j, :) = [t, diffSum];

    % Update best threshold if necessary
    if diffSum < minDiffSum
        minDiffSum = diffSum;
        bestThreshold = t;
        bestBinaryImg = binaryImg;
    end
end
```

Figure 2. Image Segmentation – the code for Otsu's Global Thresholding (2/3)

Finally, Figure 3 presents the code for visualizing the results. For each image, I am printing the Otsu's threshold and the best threshold retrieved from the test trials. This provides a quick

reference to see how each image performs under Otsu's threshold versus the threshold resulting the lowest difference sum with the groundtruth.

In addition, I create a table **thresholdTable** of the difference sums across all tested thresholds for further analysis. Following this, I plot a graph of the difference sums for each threshold value, marking the Otsu threshold on this plot to visually compare it with the other tested thresholds. The plot shows how close each threshold's difference sum is to the Otsu threshold, helping to identify if Otsu's threshold was the optimal choice or if another threshold offers a better match to the groundtruth.

Lastly, I create figures to display side-by-side visual comparisons. The first figure shows the original image, the Otsu thresholded image, the best thresholded image, and the groundtruth image, enabling a direct comparison of each thresholding result with the original and groundtruth images. In a separate figure, I display the difference images: one showing the discrepancies between the Otsu-thresholded image and the groundtruth, and the other showing the discrepancies between the best thresholded image and the groundtruth.

```
    % Display results
    fprintf('Image: %s\n', imageFiles{i});
    fprintf('Otsu Threshold: %.2f, Difference Sum (Otsu): %d\n', otsuLevel, otsuDiffSum);
    fprintf('Best Threshold: %.2f, Difference Sum (Best): %d\n\n', bestThreshold, minDiffSum);

    % Show threshold differences as a table
    thresholdTable = array2table(thresholdDiffs, 'VariableNames', {'Threshold', 'DifferenceSum'});
    disp(thresholdTable);

    % Plot sum differences for each threshold
    figure('Name', sprintf('Threshold Differences for %s', imageFiles{i}), 'NumberTitle', 'off');
    hold on;
    plot(thresholdDiffs(:, 1), thresholdDiffs(:, 2), 'b-o');
    plot(otsuLevel, otsuDiffSum, 'ro', 'MarkerSize', 8, 'LineWidth', 1.5);
    xlabel('Threshold');
    ylabel('Sum of Differences');
    title(sprintf('Difference Sums for Thresholds - %s', imageFiles{i}));
    legend('Thresholds', 'Otsu Threshold', 'Location', 'best');
    hold off;

    % Show images
    figure('Name', sprintf('Global Thresholding Results for %s', imageFiles{i}), 'NumberTitle', 'off');
    subplot(2, 2, 1); imshow(img); title('Original');
    subplot(2, 2, 2); imshow(binaryImgOtsu); title(sprintf('Otsu Threshold (%.2f)', otsuLevel));
    subplot(2, 2, 3); imshow(bestBinaryImg); title(sprintf('Best Threshold (%.2f)', bestThreshold));
    subplot(2, 2, 4); imshow(gt); title('Ground Truth');

    % Show difference images
    figure('Name', sprintf('Difference Images for %s', imageFiles{i}), 'NumberTitle', 'off');
    subplot(1, 2, 1); imshow(otsuDiff); title('Difference (Otsu)');
    subplot(1, 2, 2); imshow(xor(bestBinaryImg, gt)); title('Difference (Best)');
end
```

Figure 3. Image Segmentation – the code for Otsu's Global Thresholding (3/3)

Results:

Table 1 presents the results from the experiment with Otsu's Thresholding. At first glance, the best threshold from the test trials is achieving lower sum for most of the images, apart from **document02.bmp**. In this case, Otsu's one outperforms it because the step for searching the best threshold is set to 0.05 and it cannot achieve the result of 0.44 from Otsu. Another interesting insight is that the difference between the threshold values is not so big for the first two document images, but the difference sum of Otsu for the 3rd and 4th documents is several times higher than the one from the found best threshold.

Table 1. Image Segmentation – Comparison of Otsu Threshold and Best Threshold Values for Document Images

Image	Otsu Threshold	Best Threshold	Difference Sum (Otsu)	Difference Sum (Best)
document01.bmp	0.55	0.45	27 849	24 088
document02.bmp	0.44	0.45	9 476	9 496
document03.bmp	0.69	0.40	179 165	17 038
document04.bmp	0.60	0.35	134 548	19 869

Figure 4 shows the differences between the binarized images with the different thresholds (Otsu vs Best). As seen in the figure, the first two document images are not so severely degraded which results in both approaches achieving similar results. For **document01.bmp** both methods successfully segment the text from the background but struggle to handle the dark mark (stain). **document02.bmp** is not as degraded as the other images and thus both strategies achieve good results (difference sum around 10 000).

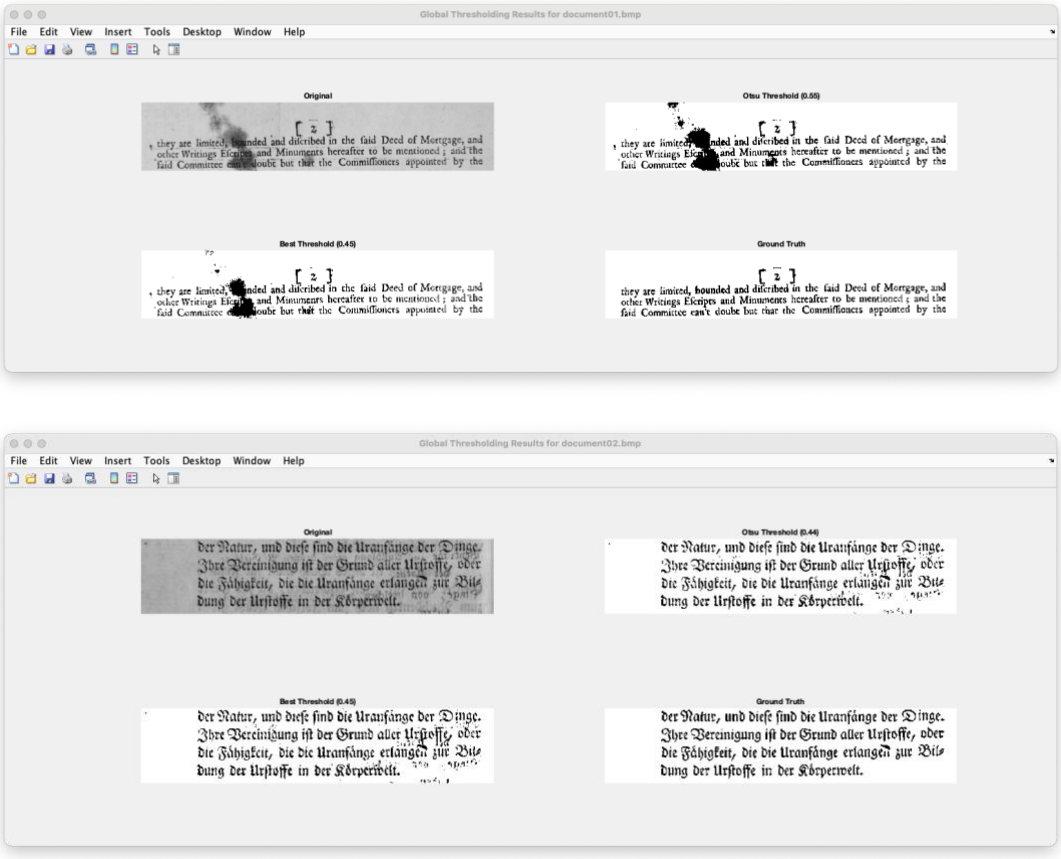

Figure 4. Image Segmentation – document01 vs document02 binarization results

The observations are further proved by the visualizations, seen in Figure 5, of the difference for Otsu and the best threshold from the test trials. The white pixels in the visualization show the difference between the binarized image and groundtruth image.

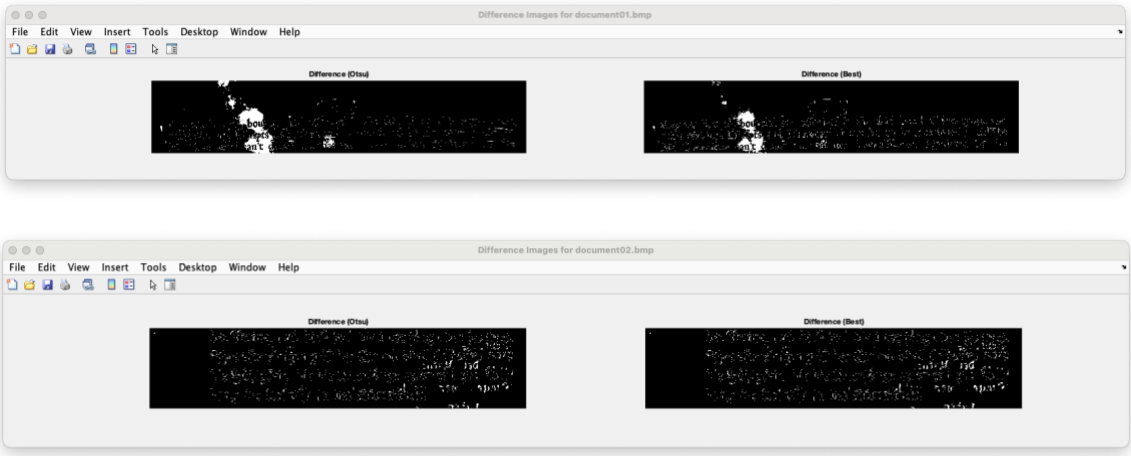

Figure 5. Image Segmentation – document01 vs document02 difference visualizations

The similar performance of Otsu's and the best threshold is clearly seen in Figure 6. The line charts for the first two document images both show that Otsu's algorithm gives a threshold which is close to the global optimum.

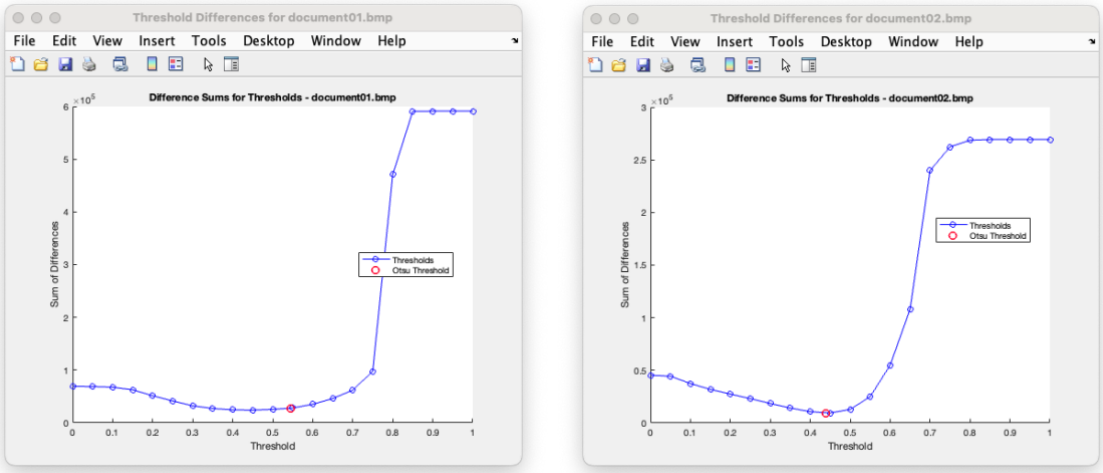

Figure 6. Image Segmentation – document01 vs document02 sum of differences over threshold values

On the other hand, as observed in Figure 7, **document03.bmp** and **document04.bmp** both include more sophisticated types of noise and degrading. The top right images show the results of binarization with the Otsu threshold, both images contain big black regions which hide the text behind it. However, this is not the case for the binary images, resulted from the best threshold. Thus, it can be deduced that Otsu's algorithm does not perform optimally in the presence of complex noise or uneven degradation, as it tends to over-segment or misrepresent areas with significant intensity variation.

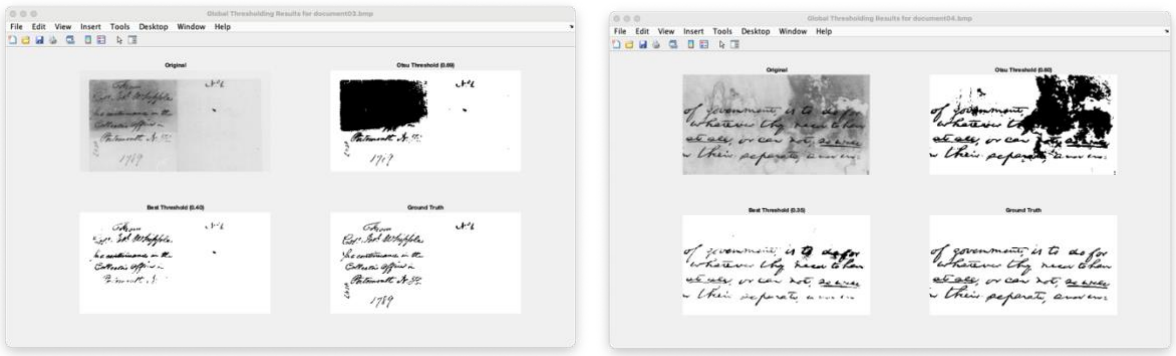

Figure 7. Image Segmentation – document03 vs document04 binarization results

The big results from the table are confirmed once again by the visualization (Figure 8) of the differences between Otsu and the retrieved best threshold. It is obvious that Otsu performs worse because there is a big white region in both comparisons with the groundtruths, while the threshold retrieved from the range search cannot capture the outer pixels of the text.

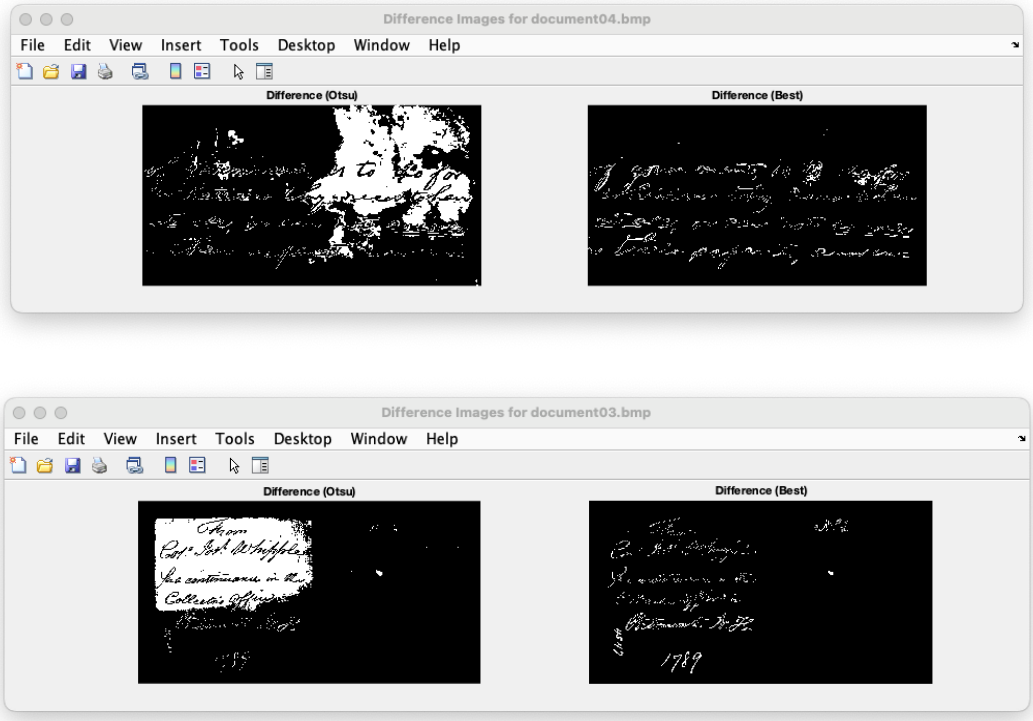

Figure 8. Image Segmentation – document03 vs document04 difference visualizations

Figure 9 also supports this statement that Otsu underperforms compared to the global optimal threshold. As highlighted in the figure, the best threshold is around 0.4 for both images, while Otsu's algorithm outputs a value higher on the increasing slope on the right.

In conclusion, Otsu's algorithm often struggles with text segmentation in images with uneven lighting, low contrast, complex backgrounds, or high noise. These factors distort the histogram, making it difficult for a single global threshold to accurately separate text from the background. In such cases, adaptive thresholding or preprocessing steps like noise reduction offer more reliable results.

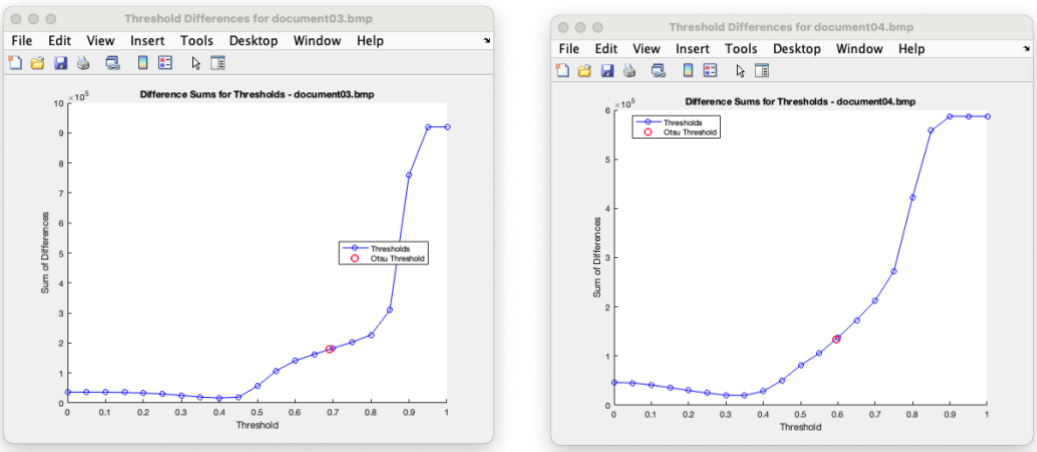

Figure 9. Image Segmentation – document03 vs document04 sum of differences over threshold values

3.1.2.2 Part b)

MATLAB Code:

In Figure 10, I begin implementing Niblack's Local Thresholding with Bayesian Optimization by defining the parameter ranges for **k** and **window size**. The **k** parameter, controlling threshold sensitivity, ranges from -3.5 to 3.5, while **window size** varies from 3 to 300 pixels, allowing flexible adaptation across images.

I initialize **resultsSummary** to store the best thresholding results for each image, making analysis and comparison straightforward. Similar to Otsu's process, I then loop through each image, loading both the image and its groundtruth in grayscale for consistency. If the groundtruth is not binary, I convert it using **imbinarize** to ensure a reliable reference.

```
%% b) Niblack's Local Thresholding with Bayesian Optimization

% Search ranges for k and window size
kRange = [-3.5, 3.5];
windowSizeRange = [3, 300];

% Initialize results summary
resultsSummary = struct([]);

% Process each image
for i = 1:length(imageFiles)
    % Read the image and its ground truth
    imgPath = fullfile(imgFolder, imageFiles{i});
    gtPath = fullfile(imgFolder, gtFiles{i});
    img = imread(imgPath);
    gt = imread(gtPath);

    % Convert to grayscale if necessary
    if size(img, 3) == 3
        img = rgb2gray(img);
    end
    if size(gt, 3) == 3
        gt = rgb2gray(gt);
    end

    % Ensure ground truth is binary
    if ~islogical(gt)
        gt = imbinarize(gt);
    end
```

Figure 10. Image Segmentation – the code for Niblack's Local Thresholding (1/4)

The next segment of code (Figure 11) sets up the main optimization process for Niblack's thresholding. I start by defining **objectiveFunction**, which checks each combination of **k** and **window size** by comparing the thresholded image to the groundtruth to minimize segmentation errors. The optimization variables, **kVar** and **windowSizeVar**, are then set within the given ranges.

To find the best parameters, I use Bayesian optimization with **bayesopt**, which looks for the values of **k** and **window size**that give the smallest difference sum. The optimization is set to run up to 250 times, and an acquisition function helps the algorithm balance between trying new values and refining promising ones. During this process, the **plotObjectiveModel**function shows how the search is progressing, so I can see how the algorithm tests different values and moves toward the best settings.

When the optimization finishes, the best values of **k** and **window size** are stored as **bestK** and **bestWindowSize**, and **minDifference** keeps track of the smallest difference sum achieved. Using these optimal values, I generate **bestBinaryImg** as the final binary segmentation image. This output gives the clearest and most accurate segmentation result achieved with the Niblack method for each image.

```
% Define the objective function for optimization
    objectiveFunction = @(params) niblackObjective(img, gt, params.k, params.windowSize);

    % Define optimization variables
    kVar = optimizableVariable('k', kRange, 'Transform', 'none');
    windowSizeVar = optimizableVariable('windowSize', windowSizeRange, 'Type', 'integer');

    % Run Bayesian optimization
    results = bayesopt(objectiveFunction, [windowSizeVar, kVar], ...
        'AcquisitionFunctionName', 'expected-improvement-plus', ...
        'MaxObjectiveEvaluations', 250, ...
        'IsObjectiveDeterministic', true, ...
        'ExplorationRatio', 0.4, ...
        'PlotFcn', {@plotObjectiveModel});

    % Extract best parameters
    bestK = results.XAtMinObjective.k;
    bestWindowSize = results.XAtMinObjective.windowSize;
    minDifference = results.MinObjective;

    % Generate best binary image
    bestBinaryImg = niblackThreshold(img, bestWindowSize, bestK);
```

Figure 11. Image Segmentation – the code for Niblack's Local Thresholding (2/4)

In the code, presented in Figure 12, I display and store the results from the Niblack thresholding optimization. First, I print key values for each image, including the image name, best **k** value, best **window size**, and the minimum difference sum achieved. This provides a quick summary of the optimal parameters and their performance for each image.

To visualize the results, I create a figure that includes four subplots: the original image, the binary image generated with the best **k** and **window size**, the groundtruth, and a difference image that highlights discrepancies between the optimized binary image and the groundtruth. This layout allows for a direct comparison of the original, segmented, and groundtruth images, showing the effectiveness of the optimized parameters.

Lastly, I store the results in **resultsSummary** for each image, which includes the image name, best **k**, best **window size**, and the corresponding minimum difference sum. This summary

provides a structured record of the best parameters for each image, and I print this consolidated information as a final summary for easy reference and analysis.

```
    % Display results
    fprintf('Image: %s\n', imageFiles{i});
    fprintf('Best k (Niblack): %0.2f\n', bestK);
    fprintf('Best Window Size: %d\n', bestWindowSize);
    fprintf('Difference Sum (Best): %d\n\n', minDifference);

    % Show images
    figure('Name', sprintf('Best Niblack Segmentation for %s', imageFiles{i}), 'NumberTitle', 'off');
    subplot(2, 2, 1); imshow(img); title('Original');
    subplot(2, 2, 2); imshow(bestBinaryImg); title(sprintf('Best Binary (k = %0.2f, Window = %d)', bestK, bestWindowSize));
    subplot(2, 2, 3); imshow(gt); title('Ground Truth');
    subplot(2, 2, 4); imshow(xor(bestBinaryImg, gt)); title('Difference Image');

    % Store results
    resultsSummary(i).imageName = imageFiles{i};
    resultsSummary(i).bestK = bestK;
    resultsSummary(i).bestWindowSize = bestWindowSize;
    resultsSummary(i).minDifference = minDifference;
end

% Summary of best results
fprintf('\n=== Summary of Best Results ===\n');
for i = 1:length(resultsSummary)
    fprintf('Image: %s\n', resultsSummary(i).imageName);
    fprintf('Best k (Niblack): %0.2f\n', resultsSummary(i).bestK);
    fprintf('Best Window Size: %d\n', resultsSummary(i).bestWindowSize);
    fprintf('Difference Sum (Best): %d\n\n', resultsSummary(i).minDifference);
end
```

Figure 12. Image Segmentation – the code for Niblack's Local Thresholding (3/4)

Figure 13 shows the final part of the code, which defines the objective function and the main thresholding function needed for Niblack's local thresholding. In **niblackObjective**, the code calculates how well a given set of parameters (**k** and **window size**) performs by generating a binary image and comparing it to the groundtruth. This function calls **niblackThreshold** to create a binary version of the image based on the current parameters. It then calculates the mismatch between this binary image and the groundtruth using **xor**, with the total number of mismatched pixels saved as **diffSum**. This value acts as the "score" for the optimization process, which tries to minimize it.

The **niblackThreshold** function is where the Niblack local thresholding algorithm is actually applied. To handle calculations accurately, it first converts the image to **double** precision. It then pads the image edges to ensure that pixels near the borders are processed correctly. Using an averaging filter defined with **fspecial**, it calculates the local mean intensity within each **window size** region. Additionally, it calculates the local variance and standard deviation to capture the spread of intensities.

Niblack’s thresholding formula (defined in the concept section) is then applied, setting a local threshold at **localMean + k * localStdDev** for each pixel. This local adaptation allows the threshold to adjust based on the intensity variations within the window, making the method particularly effective for images with uneven lighting or high local contrast. Finally, the function produces a binary image by comparing each pixel’s intensity to its calculated threshold, creating a result tailored to the specific characteristics of the image.

```
% Objective function for Bayesian optimization (Niblack)
function diffSum = niblackObjective(img, gt, k, windowSize)
    binaryImg = niblackThreshold(img, windowSize, k);
    diffImg = xor(binaryImg, gt);
    diffSum = sum(diffImg(:));
end

% Niblack's thresholding function
function binaryImg = niblackThreshold(img, windowSize, k)
    img = double(img);
    halfWin = floor(windowSize / 2);
    padImg = padarray(img, [halfWin, halfWin], 'symmetric');

    meanFilter = fspecial('average', windowSize);
    localMean = imfilter(padImg, meanFilter, 'replicate');
    localMean = localMean(halfWin+1:end-halfWin, halfWin+1:end-halfWin);

    localSqMean = imfilter(padImg.^2, meanFilter, 'replicate');
    localSqMean = localSqMean(halfWin+1:end-halfWin, halfWin+1:end-halfWin);

    localVariance = localSqMean - localMean.^2;
    localStdDev = sqrt(localVariance);

    threshold = localMean + k * localStdDev;
    binaryImg = img > threshold;
end
```

Figure 13. Image Segmentation – the code for Niblack's Local Thresholding (4/4)

Results:

The 3D surface plots in Figure 14 show how **k** and **window size** impact segmentation accuracy for each image, with the **z-axis** representing the difference sum from the groundtruth. Most of the blue points concentrate in the lower areas of each plot, as they represent parameter settings that achieve better results, highlighting optimal values. These visualizations capture how Bayesian optimization explores different **k** and **window size** combinations, homing in on regions that yield the best segmentation. Each image's unique surface landscape emphasizes that the optimal parameters vary depending on image characteristics.

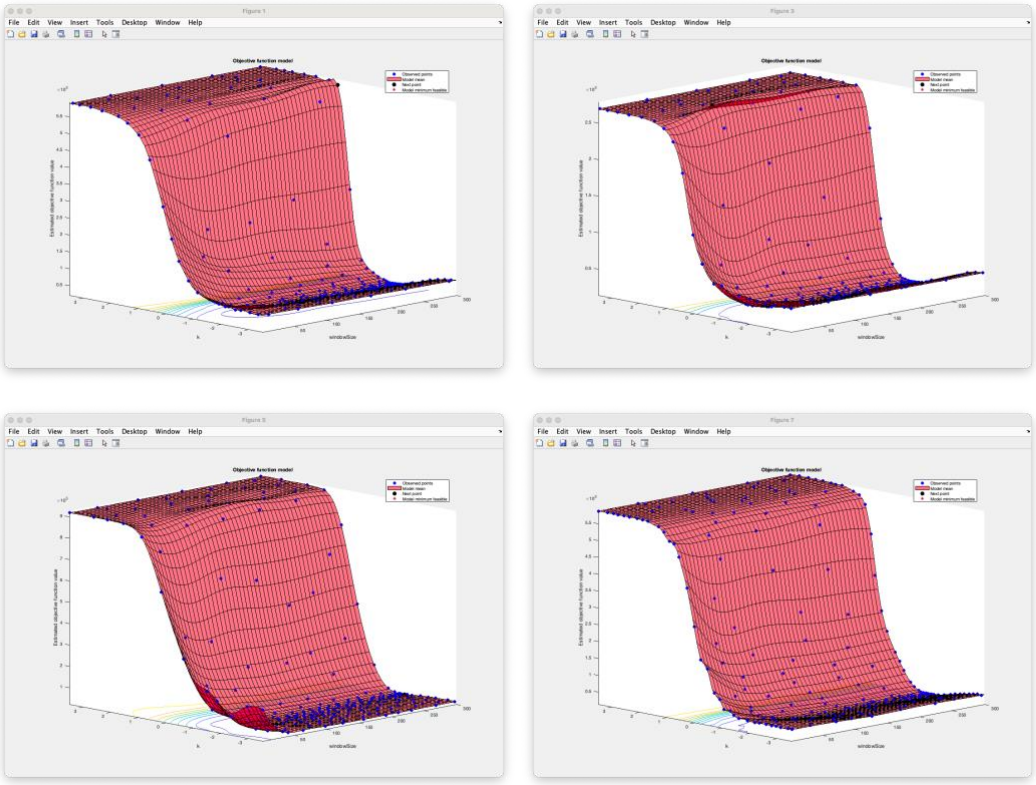

Figure 14. Image Segmentation – 3D visualization of Bayesian optimization for Niblack's thresholding parameters

Table 2. Image Segmentation – Summary of the optimal parameters for Niblack's thresholding across all document images

Image	Best k	Best Window Size	Difference Sum
document01.bmp	-1.29	300	21 092
document02.bmp	-0.87	300	14 242
document03.bmp	-1.47	299	21 922
document04.bmp	-1.44	300	13 575

The surface plots in Figure 14 give a global view of the Bayesian optimization, but spotting the most optimal values on these plots is extremely difficult because of the many closely located points. Thus, I have presented the final optimal results after the optimization in Table 2.

As shown in Table 2, the optimized results generally achieve lower difference sums compared to Otsu's method, with the exception of **document02.bmp**, where the difference is minimal. Notably, for **document03.bmp** and **document04.bmp**, the optimized Niblack thresholding reduces the difference sum by several times compared to Otsu's, highlighting a significant improvement in segmentation accuracy for these images.

Figure 15. Image Segmentation – Niblack's thresholding results (document01.bmp and document02.bmp)

After seeing the visualizations (Figure 15 and Figure 16) of the results from the Niblack's local threshold algorithm and its Bayesian parameter optimization, it is clearly visible that the results are better than Otsu's algorithm, not only in terms of the sum of differences but also visually. The large black regions from Otsu's approach are eliminated by Niblack's one. However, the result for **document02.bmp** is worse, as the algorithm performs less effectively on images with low noise.

In cases like **document02.bmp**, where noise is minimal and lighting is uniform, Otsu's global thresholding can outperform Niblack's approach. Otsu's single global threshold works well here, creating a clear separation between foreground and background without local adjustments.

On the other hand, Niblack's method calculates local thresholds within a defined window size. A small window can introduce artifacts by making rapid, unnecessary adjustments in uniform areas, creating patchy regions that reduce segmentation quality. Although increasing the window size can help, it may result in a loss of detail. For cleaner images, Otsu's simplicity often yields better results, while Niblack's approach is better suited for noisy, complex images.

Figure 16. Image Segmentation – Niblack's thresholding results (document03.bmp and document04.bmp)

3.1.2.3 Part c.1)

MATLAB Code:

As seen in Figure 17, the code for the Sauvola's Local Thresholding starts with the same image preprocessing as the previous parts. First, the images are converted to grayscale, if they are not in grayscale already. Then, we ensure that the groundtruths are in binary.

```
%% c.1) Bayesian Optimization for Sauvola's Local Thresholding

% Process each image
for i = 1:length(imageFiles)
    % Read the image and its ground truth
    imgPath = fullfile(imgFolder, imageFiles{i});
    gtPath = fullfile(imgFolder, gtFiles{i});
    img = imread(imgPath);
    gt = imread(gtPath);

    % Convert to grayscale if necessary
    if size(img, 3) == 3
        img = rgb2gray(img);
    end
    if size(gt, 3) == 3
        gt = rgb2gray(gt);
    end

    % Ensure ground truth is binary
    if ~islogical(gt)
        gt = imbinarize(gt);
    end
```

Figure 17. Image Segmentation – the code for Sauvola's Local Thresholding (1/4)

As we are using Bayesian optimization for finding the best Sauvola parameters, the code segment (Figure 18) again starts with initializing the objective function, **objectiveFunction**, which evaluates the quality of different parameter sets (window size, k, and R) by comparing the resulting binary image to the groundtruth. The optimization variables are then defined, with windowSizeVar, kVar, and RVar specifying the parameter ranges.

Using Bayesian optimization, the code then iteratively searches for parameter values that minimize the difference between the binary image and the groundtruth, guided by the "expected-improvement-plus" acquisition function. After 250 evaluations, the algorithm outputs the optimal parameter values for **windowSize**, **k**, and **R**, which are then used in **sauvolaThreshold** to generate a binary image, **bestBinaryImg**, closely aligned with the groundtruth. This approach efficiently automates the tuning of Sauvola's parameters for optimal segmentation accuracy.

The code segment in Figure 19 is used to display the final results of the Sauvola parameter optimization. It begins by printing out key information for each processed image, including the image file name, the **bestWindowSize**, **bestK**, and **bestR** parameters found by the optimization, as well as the minimum difference sum (**minDifference**) between the generated binary image and the groundtruth. This output provides a quick summary of the best parameter values identified for each image, aiding in evaluating the performance of the thresholding.

Following the printed summary, the code visualizes the results in a figure with four subplots. The first subplot shows the original image, while the second displays the **bestBinaryImg** generated using the optimized parameters (**bestWindowSize**, **bestK**, **bestR**). The third subplot shows the groundtruth image, which serves as a reference for the thresholding accuracy. Lastly, the fourth subplot presents the difference image, highlighting discrepancies between the

thresholded result and the groundtruth using an **xor** operation. This layout offers a clear comparison, allowing quick visual assessment of the segmentation accuracy achieved with the optimized Sauvola parameters.

```
    % Define the objective function
    objectiveFunction = @(params) sauvolaObjective(img, gt, params.windowSize, params.k, params.R);

    % Define optimization variables
    windowSizeVar = optimizableVariable('windowSize', [3, 41], 'Type', 'integer');
    kVar = optimizableVariable('k', [0.1, 0.7]);
    RVar = optimizableVariable('R', [32, 192], 'Type', 'integer');

    % Run Bayesian optimization
    results = bayesopt(objectiveFunction, [windowSizeVar, kVar, RVar], ...
        'AcquisitionFunctionName', 'expected-improvement-plus', ...
        'MaxObjectiveEvaluations', 250, ...
        'IsObjectiveDeterministic', true, ...
        'PlotFcn', {@plotObjectiveModel, @plotMinObjective});

    % Extract best parameters
    bestWindowSize = results.XAtMinObjective.windowSize;
    bestK = results.XAtMinObjective.k;
    bestR = results.XAtMinObjective.R;
    minDifference = results.MinObjective;

    % Generate best binary image
    bestBinaryImg = sauvolaThreshold(img, bestWindowSize, bestK, bestR);
```

Figure 18. Image Segmentation – the code for Sauvola's Local Thresholding (2/4)

```
    % Display results
    fprintf('Image: %s\n', imageFiles{i});
    fprintf('Best Window Size (Sauvola): %d\n', bestWindowSize);
    fprintf('Best k (Sauvola): %.2f\n', bestK);
    fprintf('Best R (Sauvola): %d\n', bestR);
    fprintf('Difference Sum (Best): %d\n\n', minDifference);

    % Show images
    figure('Name', sprintf('Best Sauvola Segmentation for %s', imageFiles{i}), 'NumberTitle', 'off');
    subplot(2, 2, 1); imshow(img); title('Original');
    subplot(2, 2, 2); imshow(bestBinaryImg); title(sprintf('Best Binary (w=%d, k=%.2f, R=%d)', bestWindowSize, bestK, bestR));
    subplot(2, 2, 3); imshow(gt); title('Ground Truth');
    subplot(2, 2, 4); imshow(xor(bestBinaryImg, gt)); title('Difference Image');
end
```

Figure 19. Image Segmentation – the code for Sauvola's Local Thresholding (3/4)

In Figure 20, the code defines two key functions: **sauvolaObjective** and **sauvolaThreshold**. These functions work together to perform Sauvola's local thresholding and evaluate its effectiveness against a groundtruth image.

The **sauvolaObjective** function calculates a difference score that quantifies the mismatch between the binary segmentation result and the groundtruth. It first generates a binary image, **binaryImg**, by calling **sauvolaThreshold** with the given parameters (**windowSize**, **k**, and **R**). It then computes a **diffImg** by applying an **xor** operation between **binaryImg** and the groundtruth (**gt**), which highlights differences. The total discrepancy, **diffSum**, is calculated by summing all pixel differences, providing an objective score that the optimization process seeks to minimize. This function is essential for the Bayesian optimization process, as it provides the objective score that guides the optimization toward the parameter set that minimizes segmentation errors.

The **sauvolaThreshold** function applies Sauvola's thresholding algorithm to create a binary image based on the intensity characteristics of local neighborhoods within the image. It starts by converting the image to double precision and padding it to ensure accurate processing near

edges. A mean filter (created with **fspecial**) calculates the **localMean** within each **windowSize** region, while a squared mean filter helps compute the **localVariance** and standard deviation, **localStdDev**. These values capture local intensity patterns, allowing the threshold to adapt dynamically to variations.

Using Sauvola's formula (defined in the concept section), the code computes the threshold as **localMean * (1 + k * ((localStdDev / (R + eps)) - 1))**, ensuring pixel values fall within a bounded range of 0 to 255. The binary image, **binaryImg**, is finally generated by comparing each pixel's intensity to its local threshold, creating a segmented output tailored to the image's local intensity features.

```
% Objective function for Bayesian optimization (Sauvola)
function diffSum = sauvolaObjective(img, gt, windowSize, k, R)
    binaryImg = sauvolaThreshold(img, windowSize, k, R);
    diffImg = xor(binaryImg, gt);
    diffSum = sum(diffImg(:));
end

% Sauvola's thresholding function
function binaryImg = sauvolaThreshold(img, windowSize, k, R)
    img = double(img);
    halfWin = floor(windowSize / 2);
    padImg = padarray(img, [halfWin, halfWin], 'symmetric');

    meanFilter = fspecial('average', windowSize);
    localMean = imfilter(padImg, meanFilter, 'replicate');
    localMean = localMean(halfWin+1:end-halfWin, halfWin+1:end-halfWin);

    localSqMean = imfilter(padImg.^2, meanFilter, 'replicate');
    localSqMean = localSqMean(halfWin+1:end-halfWin, halfWin+1:end-halfWin);

    localVariance = max(0, localSqMean - localMean.^2);
    localStdDev = sqrt(localVariance);
    localStdDev(localStdDev == 0) = eps;

    threshold = localMean .* (1 + k * ((localStdDev / (R + eps)) - 1));
    threshold = min(max(threshold, 0), 255);

    binaryImg = img > threshold;
end
```

Figure 20. Image Segmentation – the code for Sauvola's Local Thresholding (4/4)

Results:

Table 3. Image Segmentation – Summary of the optimal parameters for Sauvola's thresholding across all document images

Image	Best Window Size	Best k	Best R	Difference Sum
document01.bmp	40	0.52	73	8 821
document02.bmp	37	0.70	57	7 729
document03.bmp	16	0.13	87	9 645
document04.bmp	17	0.32	74	9 270

The results in Table 3 summarize the optimal parameters for Sauvola's thresholding across the document images, providing the best Window Size, k, R, and Difference Sum values for each image. Notably, Sauvola's thresholding achieves relatively low difference sums for all images, indicating that this method effectively aligns with the groundtruth segmentation. The Best Window Size varies significantly across images, ranging from 16 to 40, suggesting that Sauvola's adaptability enables the use of different window sizes based on the complexity of each image. In **document01.bmp** and **document02.bmp**, larger windows (40 and 37, respectively) are optimal, likely because these images have more uniform regions with less

intricate detail or noise. A larger window size captures a broader context, which works well in these cases, smoothing out minor variations and achieving effective segmentation with fewer local adjustments. In contrast, **document03.bmp** and **document04.bmp** benefit from smaller window sizes (16 and 17). These images likely contain more detailed structures and complex variations, where smaller windows allow Sauvola's method to adapt more precisely to local intensity changes. The smaller window sizes help capture finer details and handle the increased complexity in these images, leading to improved segmentation accuracy in noisier or more degraded regions.

When compared to Niblack's thresholding results in Table 2, Sauvola's method generally achieves significantly better results, highlighting its potential for better segmentation accuracy in these document images.

Sauvola's thresholding offers key improvements over Niblack's, making it more effective for document processing, especially in noisy or complex backgrounds. A major benefit is its ability to reduce noise, producing cleaner outputs that enhance the readability of degraded or unevenly lit documents. Sauvola's method adapts dynamically to local image variations, preserving details while minimizing artifacts, which is especially useful in documents with stains or faded text. Additionally, Sauvola calculates a unique threshold for each pixel, enabling it to handle local contrast fluctuations more precisely than Niblack.

While both Sauvola and Niblack are effective for non-uniform backgrounds, Sauvola’s approach achieves lower difference sums, showing its advantage in accuracy and noise reduction. In contrast, Otsu’s global thresholding, though suitable for uniform backgrounds, lacks the localized adaptability required for complex documents. This makes Sauvola the superior choice for challenging conditions, as it consistently produces cleaner, more accurate segmentations that closely match the groundtruth.

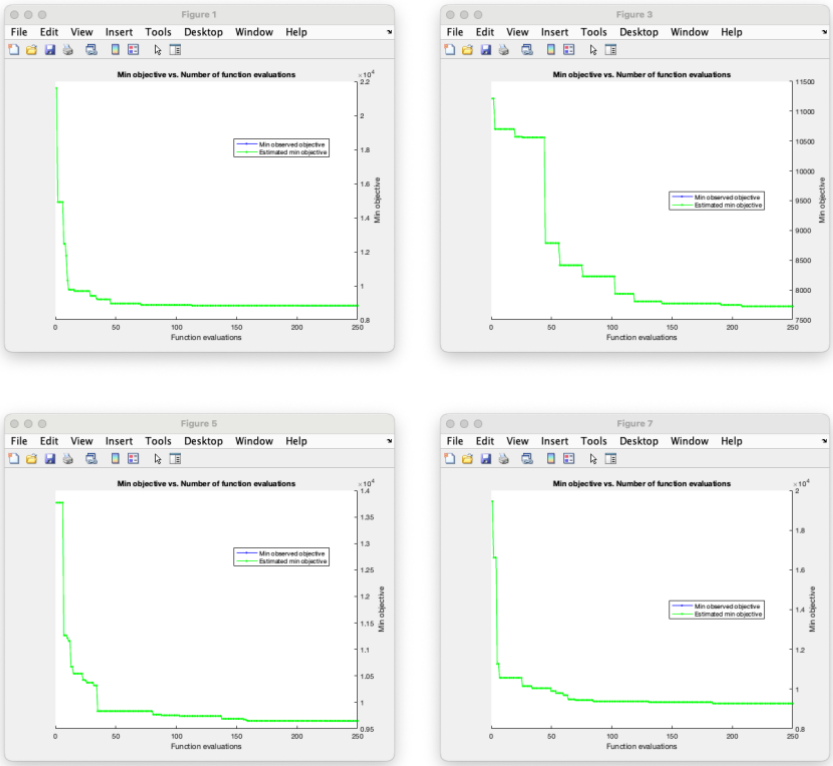

Figure 21. Image Segmentation – optimization results for Sauvola's thresholding parameters ccross the cocument images

Figure 21 shows the optimization progress for Sauvola's thresholding parameters across the different document images. Each plot demonstrates a sharp drop in the objective function (difference sum) within the first few evaluations, indicating rapid improvements as optimal parameter values are identified. This is followed by a gradual stabilization, where adjustments yield minimal additional gains. The convergence patterns vary slightly across images, reflecting different levels of complexity in segmentation. Overall, the plots confirm that Bayesian optimization effectively tunes Sauvola's parameters, achieving low difference sums and accurate segmentation for each document image. At the end, all optimizations achieve the best results so far (< 10 000 for the difference sum).

If we increase the search space or allow more iterations in the Bayesian optimization, we could potentially find slightly better results for both Sauvola and Niblack; however, the improvement would likely be minimal, as the algorithm has already converged. Additionally, extending the search would significantly increase the computation time, making it less efficient for practical use.

Figures 22 and 23 indeed show the improvements of the Sauvola's approach on the given dataset of four images. The outputs of the Sauvola function look very similar to the actuall groundtruths with the difference of little noise around the text and around the bigger and stronger degrading regions on the images.

The segmentation results are quite promising, with Sauvola's approach capturing text regions closely aligned with the groundtruth and only minor noise in degraded areas. For even higher quality, additional filtering can be applied both in preprocessing and post-processing. Preprocessing filters could help reduce noise before segmentation, while post-processing techniques, such as reconnecting split text and removing non-text regions, would enhance readability and accuracy. These combined steps would result in cleaner, more precise segmentation, especially in images with challenging noise or heavy degradation.

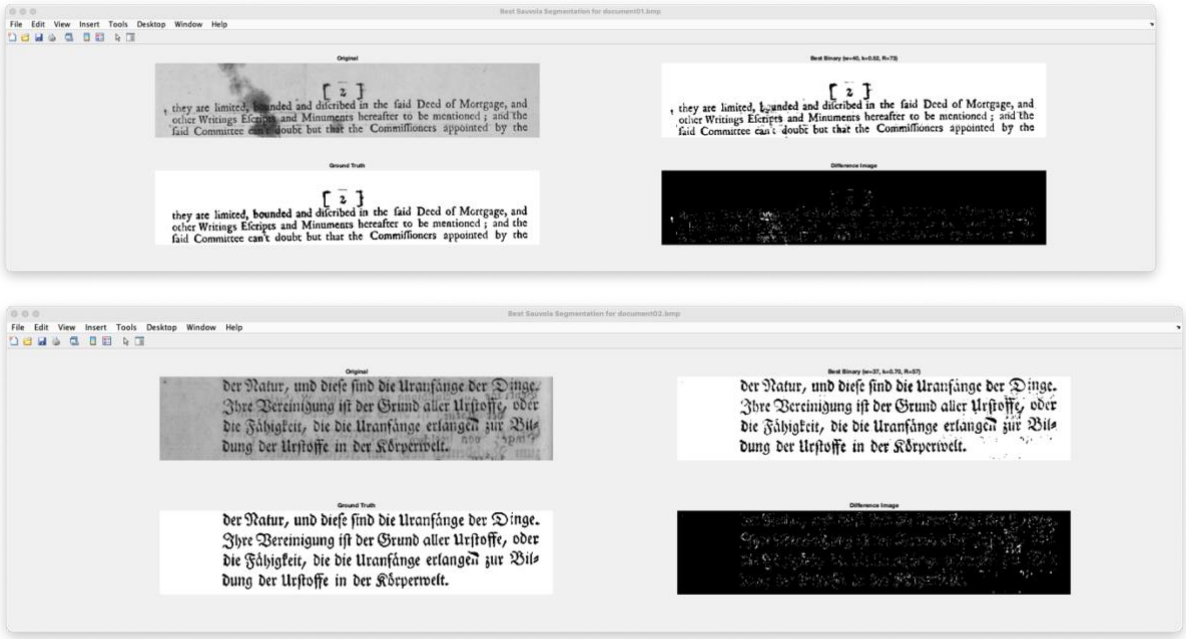

Figure 22. Image Segmentation – Sauvola's thresholding results (document01.bmp and document02.bmp)

Figure 23. Image Segmentation – Sauvola's thresholding results (document03.bmp and document04.bmp)

3.1.2.4 Part c.2)

MATLAB Code:

Figure 24 presents the code for the Filter Bank and K-Means Clustering approach. The code begins by setting parameters for Gabor filters and K-Means clustering: **numOrientationsList** (orientations for texture capture), **kClusterValues** (number of clusters), **gaborWavelengths** (scales for texture analysis), and **poolingScales** (smoothing levels). These ranges provide flexibility in capturing different texture patterns and segmentation granularity, though they also increase computation time due to the extensive parameter combinations. The wide parameter

ranges enable thorough exploration but make this method time-consuming, as more configurations need to be tested to determine the most optimal settings.

Following parameter initialization, the code processes each image by loading it along with its groundtruth and performing traditional preprocessing, similar to previous parts.

```
%% c.2) Filter Bank with Smoothed Features and K-Means Clustering

% Parameters for Gabor filters and clustering
numOrientationsList = [4, 6, 8, 10];
kClusterValues = [2, 3, 4, 5, 6];
gaborWavelengths = [2, 4, 8, 12];
poolingScales = 0.1:0.1:1;

% Process each image
for i = 1:length(imageFiles)
    % Read the image and its ground truth
    imgPath = fullfile(imgFolder, imageFiles{i});
    gtPath = fullfile(imgFolder, gtFiles{i});
    img = imread(imgPath);
    gt = imread(gtPath);

    % Convert to grayscale if necessary
    if size(img, 3) == 3
        img = rgb2gray(img);
    end
    if size(gt, 3) == 3
        gt = rgb2gray(gt);
    end

    % Ensure ground truth is binary
    if ~islogical(gt)
        gt = imbinarize(gt);
    end
```

Figure 24. Image Segmentation – the code for Filter Bank and K-Means clustering (1/4)

The code in Figure 25 defines a loop to explore various combinations of Gabor filter and pooling scale parameters, aiming to find the optimal settings for the Filter Bank and K-Means Clustering approach. It begins by initializing **minDifference**(the lowest observed difference sum) and **bestParams** (the best parameter configuration found), which will be updated as the loop iterates over parameter sets.

For each combination of **numOrientations**, **wavelength**, and **poolingScale**, a message displays the current settings, tracking progress. The code creates an array of Gabor filters based on the specified **wavelength** and **numOrientations**, with orientations evenly distributed across 180 degrees. This filter array, **gaborArray**, is applied to the image to obtain a set of Gabor magnitude features (**gaborMag**), capturing multi-directional texture information.

Each Gabor magnitude feature is then resized according to **poolingScale** and smoothed with Gaussian filtering. These pooled features form a stacked **featureSet** representing multi-scale texture data. Additionally, the grayscale image's intensity and gradient magnitude (**imgResized** and **Gmag**) are included in the feature set, enhancing texture and edge information.

Finally, the **featureSet** is normalized by subtracting the mean and dividing by the standard deviation for each feature dimension. This normalization step ensures consistency across features, allowing K-Means clustering to more effectively utilize the extracted data for segmentation. This thorough parameter testing approach enables detailed exploration of different feature extraction settings, optimizing segmentation performance for each image.

```
% Variables to store best results
minDifference = inf;
bestParams = struct('numOrientations', 0, 'kClusters', 0, 'wavelength', 0, 'poolingScale', 0);
bestSegmentedImg = [];

% Loop over parameters
for numOrientations = numOrientationsList
    for wavelength = gaborWavelengths
        for poolingScale = poolingScales
            fprintf('Processing %s with %d orientations, wavelength %d, pooling %.1f\n', ...
                    imageFiles{i}, numOrientations, wavelength, poolingScale);

            % Create Gabor filters
            angles = linspace(0, 180, numOrientations + 1);
            angles = angles(1:end-1);
            gaborArray = gabor(wavelength, angles);
            gaborMag = imgaborfilt(img, gaborArray);

            % Pool features with smoothing
            pooledFeatures = arrayfun(@(j) imgaussfilt(imresize(gaborMag(:,:,j), poolingScale), 1), ...
                                      1:numel(gaborArray), 'UniformOutput', false);
            pooledSize = size(pooledFeatures{1});
            featureSet = reshape(cat(3, pooledFeatures{:}), [], numel(gaborArray));

            % Add intensity and gradient magnitude
            imgResized = imresize(img, poolingScale);
            imgResized = double(imgResized);
            [Gmag, ~] = imgradient(imgResized);
            featureSet = [featureSet, imgResized(:), Gmag(:)];

            % Normalize features
            featureSet = (featureSet - mean(featureSet, 1)) ./ (std(featureSet, [], 1) + eps);
```

Figure 25. Image Segmentation – the code for Filter Bank and K-Means clustering (2/4)

The code in Figure 26 performs K-Means clustering on the feature set generated from the Gabor filters and pooling process, iterating over the possible values for **kClusters** to find the optimal segmentation.

For each specified **kClusters** value, a message indicates the clustering progress. The K-Means algorithm is configured with a maximum of 500 iterations, parallel computation enabled, and a tolerance setting for convergence. Using the **featureSet** as input, the code clusters the features into **kClusters** groups, initializing clusters with the 'plus' method and running each configuration three times to improve reliability. The resulting **clusterIdx** is then reshaped to match the image dimensions, creating a **segmentedImg** where each cluster label corresponds to a region in the image.

Next, the code identifies the text region by calculating the mean intensity of each cluster in the resized image. The cluster with the lowest mean intensity (assumed to be the text) is designated as **textCluster**, and **binaryImg** is generated by isolating this cluster, converting it into a binary format where text regions are foreground. The binary image is then resized to match the original image size.

To evaluate segmentation accuracy, the code computes the **diffImg** (difference between **binaryImg** and the groundtruth **gt**), and calculates the **diffSum**. If this **diffSum** is lower than the current **minDifference**, the code updates **minDifference** and records the current parameters (**numOrientations**, **kClusters**, **wavelength**, and **poolingScale**) as the best configuration. The best segmented image (**bestSegmentedImg**) is also stored, indicating the configuration that produced the closest match to the groundtruth. This looping structure ensures that the optimal clustering parameters are selected for minimal segmentation error.

```
                % K-means clustering
                for kClusters = kClusterValues
                    fprintf('  Running K-means with %d clusters...\n', kClusters);

                    opts = statset('MaxIter', 500, 'UseParallel', true, 'TolFun', 1e-4);
                    clusterIdx = kmeans(featureSet, kClusters, ...
                                        'Replicates', 3, 'Options', opts, 'Start', 'plus');

                    % Reshape to image size
                    segmentedImg = reshape(clusterIdx, pooledSize);

                    % Map clusters to binary based on mean intensity
                    clusterMeans = arrayfun(@(c) mean(imgResized(segmentedImg == c)), 1:kClusters);
                    [~, textCluster] = min(clusterMeans);

                    binaryImg = segmentedImg == textCluster;
                    binaryImg = ~binaryImg;
                    binaryImg = imresize(binaryImg, size(img), 'nearest');

                    % Compute difference with ground truth
                    diffImg = xor(binaryImg, gt);
                    diffSum = sum(diffImg(:));

                    fprintf('    Difference Sum for orientations=%d, k=%d, λ=%d, pooling=%.1f: %d\n', ...
                            numOrientations, kClusters, wavelength, poolingScale, diffSum);

                    if diffSum < minDifference
                        minDifference = diffSum;
                        bestParams.numOrientations = numOrientations;
                        bestParams.kClusters = kClusters;
                        bestParams.wavelength = wavelength;
                        bestParams.poolingScale = poolingScale;
                        bestSegmentedImg = binaryImg;
                        fprintf('    New best result with difference sum: %d\n', minDifference);
                    end
                end
            end
        end
    end
```

Figure 26. Image Segmentation – the code for Filter Bank and K-Means clustering (3/4)

The code in Figure 27 displays the best segmentation results for each image. It prints the optimal parameters (**numOrientations**, **kClusters**, **wavelength**, and **poolingScale**) and the **minDifference** score. The segmented results are visualized in a figure with four subplots: the **Original** image, the **Best Segmentation** with optimal parameters, the **Ground Truth**, and the **Difference Image**, which highlights discrepancies between the segmentation and ground truth. This straightforward display format is consistent with previous sections, providing a clear comparison of results.

```
    % Display best result
    fprintf('Image: %s\n', imageFiles{i});
    fprintf('Best orientations: %d\n', bestParams.numOrientations);
    fprintf('Best k clusters: %d\n', bestParams.kClusters);
    fprintf('Best wavelength: %d\n', bestParams.wavelength);
    fprintf('Best pooling scale: %.1f\n', bestParams.poolingScale);
    fprintf('Difference Sum: %d\n', minDifference);

    % Show images
    figure('Name', sprintf('Best Segmentation for %s', imageFiles{i}), 'NumberTitle', 'off');
    subplot(2, 2, 1); imshow(img); title('Original');
    subplot(2, 2, 2); imshow(bestSegmentedImg); title(sprintf('Best Segmentation\nOrients=%d, k=%d, λ=%d, Pool=%.1f', ...
        bestParams.numOrientations, bestParams.kClusters, bestParams.wavelength, bestParams.poolingScale));
    subplot(2, 2, 3); imshow(gt); title('Ground Truth');
    subplot(2, 2, 4); imshow(xor(bestSegmentedImg, gt)); title('Difference Image');
end
```

Figure 27. Image Segmentation – the code for Filter Bank and K-Means clustering (4/4)

Results:

Table 4. Image Segmentation – Summary of the optimal parameters for K-Means Clustering across all document images

Image	Best Orientations	Best # Clusters	Best Wavelength	Best Pooling Scale	Difference Sum
document01.bmp	6	3	2	1.0	19554
document02.bmp	4	4	12	1.0	12606
document03.bmp	6	6	4	1.0	30789
document04.bmp	4	3	12	1.0	34628

Table 4 summarizes the best segmentation results for the Filter Bank and K-Means Clustering approach across the document images. Overall, while this method performs better than Otsu's global thresholding, achieving lower difference sums for most images, it does not reach the accuracy of Sauvola or Niblack's methods. Sauvola and Niblack, with their local thresholding approaches, are more effective in handling complex textures and noise, leading to lower difference sums.

Notably, for **document02.bmp**, Otsu's method outperforms the Filter Bank and K-Means approach, suggesting that a global threshold was more suitable for this particular image's characteristics. However, for the other images, the Filter Bank and K-Means approach provides a reasonable middle ground between global and local thresholding, demonstrating improved segmentation over Otsu, though with limitations in precision compared to Sauvola and Niblack.

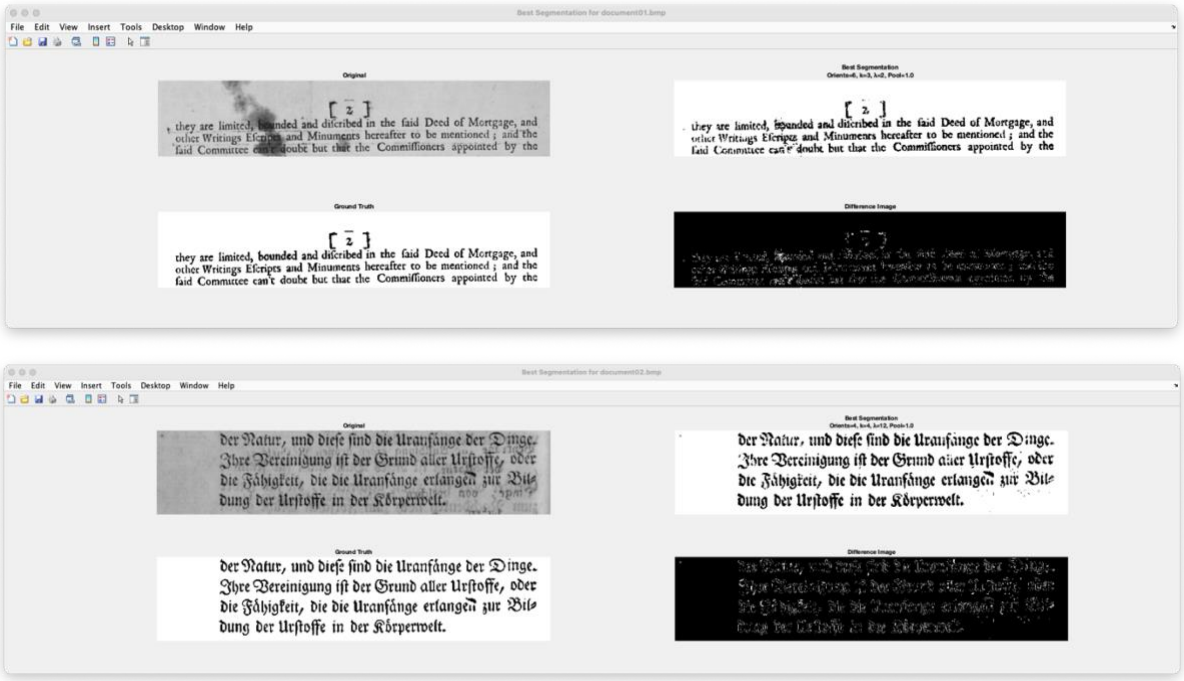

Figure 28. Image Segmentation – K-Means clustering results (document03.bmp and document04.bmp)

Figures 28 and 29 present the K-Means clustering results for the document images, showcasing how this approach performs in segmentation. While the K-cluster results demonstrate resilience against noise, effectively removing background variations and artifacts, they struggle to capture the finer details of the text. This limitation is reflected in the higher difference sums, as the clustering approach often misses thinner strokes and finer elements of the characters. As a result, although the method performs well in filtering out noise, it falls short in accurately segmenting text regions, especially in areas where details are essential for clarity and readability.

Figure 29. Image Segmentation – K-Means clustering results (document03.bmp and document04.bmp)

In conclusion, the K-means clustering approach for segmentation can be useful in cases where noise reduction is a priority and the text or objects of interest are bold, clearly defined, or have high contrast against the background. Its strength lies in effectively grouping similar pixel intensities, which helps in isolating prominent features and filtering out background noise. However, K-means clustering is not ideal for images with intricate details or fine text, as it lacks the adaptability to capture subtle variations within small regions. Without a localized thresholding mechanism, K-means struggles with precision in segmenting thin strokes or complex textures, making it less suitable for detailed document analysis or images with high variability in texture and contrast.

3.2 3D Stereo Vision

3.2.1 Concept

3D Stereo Vision aims to compute depth information from two aligned, rectified images captured from slightly different perspectives, such as a left and a right camera. The primary output, a disparity map, represents depth by encoding the shift, or disparity, between matching points in the two images. This disparity is inversely proportional to the distance of objects from the camera. Thus, objects closer to the camera appear with higher disparity, while distant objects have lower disparity.

To calculate disparity, a template matching approach is used where a small patch (template) centered around each pixel in the left image is matched within a search range along the corresponding scanline in the right image. The goal is to find the best match based on various similarity measures, leading to different methods:

Sum of Squared Differences (SSD)

SSD measures the sum of squared intensity differences between the template T and the matching region in the right image I. For each pixel location (x, y) in the left image, SSD is computed as:

$$S(x,y) = \sum_{j=0}^{M}\sum_{k=0}^{N}\big(I(x+j,y+k) - T(j,k)\big)^2$$

This method finds the location in I where the SSD value is minimized, indicating the best match. SSD can be optimized by decomposing it into three parts:

$$S(x,y) = \sum_{j=0}^{M}\sum_{k=0}^{N} I(x+j,y+k)^2 + \sum_{j=0}^{M}\sum_{k=0}^{N} T(j,k)^2 - 2\sum_{j=0}^{M}\sum_{k=0}^{N} I(x+j,y+k)\cdot T(j,k)$$

The matching disparity d for each pixel in P_l is given by:

$$d(x_l, y_l) = x_l - \widehat{x_r}$$

where $\widehat{x_r}$ is the x-coordinate in P_r where the SSD is minimized.

Normalized Cross-Correlation (NCC) and Zero-Mean NCC (ZNCC)

Both NCC and ZNCC are methods used to measure similarity between a template T and a region in an image I. NCC directly compares the intensity values of the template and image region, while ZNCC additionally normalizes these values by subtracting their mean, making it more robust to variations in lighting and contrast.

The NCC score for a template T in the left image P_l and a matching region in the right image P_r is given by:

$$NCC(x,y) = \frac{\sum_{j,k} I(x+j, y+k) \cdot T(j,k)}{\sqrt{\sum_{j,k} I(x+j, y+k)^2 \sum_{j,k} T(j,k)^2}}$$

In this formula, $I(x+j, y+k)$ and $T(j,k)$ represent the intensity values of the image region and the template, respectively. NCC normalizes the result by the magnitudes of I and T, making it invariant to scale but sensitive to lighting variations, as it does not involve mean subtraction.

The ZNCC score, on the other hand, is calculated by first subtracting the mean intensity from both the template and the image region. It is defined as:

$$ZNCC(x,y) = \frac{\sum_{j,k}(I(x+j, y+k) - \mu_I)(T(j,k) - \mu_T)}{\sqrt{\sum_{j,k}(I(x+j, y+k) - \mu_I)^2 \sum_{j,k}(T(j,k) - \mu_T)^2}}$$

where μ_I and μ_T are the mean intensities of the region in I and the template T, respectively. By centering both the template and the image region around zero mean, ZNCC emphasizes structural similarity over absolute intensity, making it resilient to variations in illumination.

For both methods, a higher score indicates a closer match, with the highest score representing the best match between the template and the region in the image.

Sum of Absolute Differences (SAD)

SAD is a simpler alternative to SSD, measuring the absolute intensity differences between the template and the matching region:

$$S(x,y) = \sum_{j=0}^{M} \sum_{k=0}^{N} |I(x+j, y+k) - T(j,k)|$$

SAD finds the disparity with the smallest sum of absolute differences, minimizing computational complexity while providing effective results.

Disparity Map and Depth Interpretation

The computed disparity $d(x,y)$ values for each pixel form the disparity map, which is visualized to interpret depth. Closer objects exhibit larger disparities, appearing brighter in the disparity map, while farther objects have smaller disparities and appear darker. This disparity-to-depth relationship can be expressed as:

$$\text{Depth} \propto \frac{1}{d(x,y)}$$

By using SSD, ZNCC, and SAD metrics for template matching, different aspects of the images are emphasized, providing flexibility in handling variations in lighting and textures across synthetic and real stereo images pairs.

3.2.2 MATLAB Code Analysis and Results

3.2.2.1 Part a)

MATLAB Code:

The code in Figure 30 presents the initial setup for the 3D Stereo Vision experiment, including parameter initialization, image preprocessing, and disparity map computation using Sum of Squared Differences (SSD). This segment begins by defining essential parameters: **maxDisparity**, which limits the search range for matching pixels between the left

and right images, and **templateSize**, which sets the dimensions of the template window for matching.

The function **computeDisparityMapSSD** is defined at the end of the code section, as is conventional in MATLAB, where functions are typically placed at the end of scripts. This function is responsible for calculating the disparity map by comparing patches from the left and right images and finding the best match for each pixel.

In the **computeDisparityMapSSD** function, the left and right images, **leftImg** and **rightImg**, are first converted to double precision to ensure calculation accuracy and compatibility. The dimensions of the images, **rows** and **cols**, are determined, allowing the function to handle images of various sizes. The code then calculates half of the template dimensions, **halfTempH** and **halfTempW**, to simplify indexing during the matching process. A zero matrix, **disparityMap**, is initialized to store the resulting disparity values, with its dimensions adjusted for template borders.

```
%% 3.2 3D Stereo Vision

% Parameters
maxDisparity = 15;
templateSize = [11, 11];

%% b) Load and Preprocess Synthetic Stereo Images

%% c) Compute Disparity Map for Synthetic Images

%% d) Compute Disparity Map for Real Stereo Images

%% a) Function to Compute Disparity Map using SSD
function disparityMap = computeDisparityMapSSD(leftImg, rightImg, templateSize, maxDisparity)
    % Convert images to double for calculation precision
    leftImg = double(leftImg);
    rightImg = double(rightImg);
    [rows, cols] = size(leftImg);

    % Calculate half of template dimensions for easier indexing
    halfTempH = floor(templateSize(1) / 2);
    halfTempW = floor(templateSize(2) / 2);

    % Initialize disparity map with size adjusted for template borders
    disparityMap = zeros(rows - templateSize(1) + 1, cols - templateSize(2) + 1);

    % Loop through each pixel within the valid template area
    for i = 1 + halfTempH : rows - halfTempH
        for j = 1 + halfTempW : cols - halfTempW
            minCost = Inf;  % Initialize min cost for SSD
            bestDisparity = 0;  % Initialize best disparity to zero

            % Extract the template from the left image
            templateLeft = leftImg(i - halfTempH : i + halfTempH, j - halfTempW : j + halfTempW);

            % Search for matching template in the right image within bidirectional maxDisparity
            for d = max(1, j - maxDisparity) : min(cols, j + maxDisparity)
                % Ensure template does not go out of right image bounds
                if (d - halfTempW >= 1) && (d + halfTempW <= cols)
                    % Extract the template from the right image
                    templateRight = rightImg(i - halfTempH : i + halfTempH, d - halfTempW : d + halfTempW);

                    % Calculate SSD (Sum of Squared Differences) between templates
                    cost = sum(sum((templateLeft - templateRight) .^ 2));

                    % Update best disparity if current cost is lower
                    if cost < minCost
                        minCost = cost;
                        bestDisparity = j - d;  % Calculate disparity
                    end
                end
            end

            % Store the best disparity for the current pixel
            disparityMap(i - halfTempH, j - halfTempW) = bestDisparity;
        end
    end
end
```

Figure 30. 3D Stereo Vision – the code for Sum of Squared Differences (SSD) method

The core of the function is a nested loop that iterates over each pixel within the valid area of the template. For each pixel location, a patch or template is extracted from the left image, **templateLeft**, based on the specified **templateSize**. The function then searches for a matching template in the corresponding region of the right image, constrained by **maxDisparity** in both directions. This ensures that the search remains within a reasonable range, reducing computational load.

For each potential match in the right image, a similar-sized patch, **templateRight**, is extracted. The function then calculates the SSD (Sum of Squared Differences) between **templateLeft** and **templateRight** to measure their similarity. The **cost** variable stores the SSD value, with lower values indicating better matches. If the computed **cost** is lower than the current minimum, **minCost**, the function updates **minCost** and records the current disparity, **bestDisparity**, which is the difference in column position between the matched pixels in the left and right images.

Once the optimal disparity is determined for the current pixel, it is stored in **disparityMap**. This process repeats for all valid pixels in the left image, ultimately building a complete disparity map that represents the depth information based on differences between the left and right images.

Results:

Overall, this function provides a straightforward approach to calculating disparity maps using SSD, a common technique in stereo vision for finding correspondences based on pixel similarity within a defined search window. There are particular results for this part as it is only a function definition.

3.2.2.2 Part b)

MATLAB Code:

Figure 31 shows the code to load and preprocess the synthetic stereo images, **leftImg** and **rightImg**, by converting them to grayscale if necessary. This preprocessing step is consistent with previous experiments, ensuring images are in the correct format for disparity calculation.

```
%% b) Load and Preprocess Synthetic Stereo Images

leftImgPath = 'img/corridorl.jpg';
rightImgPath = 'img/corridorr.jpg';

leftImg = imread(leftImgPath);
rightImg = imread(rightImgPath);

if size(leftImg, 3) == 3
    leftImgGray = rgb2gray(leftImg);
else
    leftImgGray = leftImg;
end

if size(rightImg, 3) == 3
    rightImgGray = rgb2gray(rightImg);
else
    rightImgGray = rightImg;
end
```

Figure 31. 3D Stereo Vision – the code for loading and preprocessing the synthetic images

Results:

The results of this code segment are straightforward, as it simply converts the synthetic stereo images to grayscale, ensuring they are in a uniform format for further processing. This preprocessing step prepares the images for accurate disparity calculations without altering their original content.

3.2.2.3 Part c)

MATLAB Code:

In this code snippet (Figure 32), the disparity map for the synthetic stereo images is computed and displayed. The function **computeDisparityMapSSD** is applied to the grayscale images (**leftImgGray** and **rightImgGray**) with the specified **templateSize** and **maxDisparity** values, resulting in a disparity map, **D**, that represents depth information.

The code then visualizes **D** using **imshow**, setting the disparity range from -15 to 15 to capture meaningful depth variations. A grayscale **colormap** and **colorbar** are added to enhance readability, providing a clear visual representation of depth differences within the synthetic image pair.

```
%% c) Compute Disparity Map for Synthetic Images

D = computeDisparityMapSSD(leftImgGray, rightImgGray, templateSize, maxDisparity);

figure('Name', 'Disparity Map - Synthetic Images', 'NumberTitle', 'off');
imshow(D, [-15 15]);
title('Disparity Map (Synthetic Images)');
colormap("gray");
colorbar;
```

Figure 32. 3D Stereo Vision – the code for computing the disparity for the synthetic images

Results:

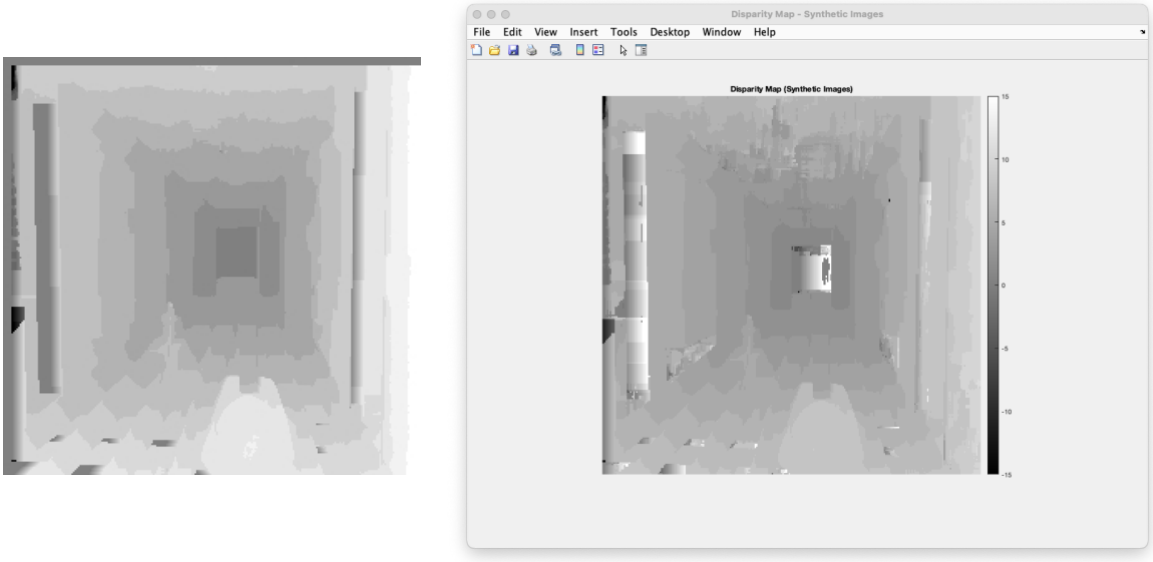

Figure 33. 3D Stereo Vision – Disparity map visualization for Synthetic stereo images using SSD

Figure 33 presents the difference between the ground truth image on the left and the result from the SSD-based disparity map computation on the right. The SSD-based map captures the

general depth structure, with closer objects represented by lighter values and farther objects by darker ones. The central corridor and main features are visible, indicating that the SSD method can approximate depth. However, there are several significant differences with the groundtruth.

First, the SSD-based approach fails to capture the depth at the end of the corridor as its right section is colored in bright white, falsely indicating that the area is closer, while the groundtruth image says it is further away from the camera position. This issue occurs primarily because the SSD algorithm struggles with areas that lack distinct texture or have repetitive patterns (the end of the corridor is all white), which are common at the end of a corridor. In such regions, there are insufficient unique features for the algorithm to accurately match corresponding points between the left and right images. As a result, the SSD method may incorrectly assign disparity values, leading to a misrepresentation of depth. Specifically, it might match points from closer objects to those farther away, falsely indicating that distant areas are nearer than they actually are. Additionally, factors like lighting variations and noise can further degrade the performance of the SSD approach in these challenging areas, contributing to the inaccurate depth perception.

The second main issue that SSD struggles with is the overall noise across the whole disparity map, which leads to unreliable depth estimations. This occurs because the SSD algorithm is highly sensitive to small intensity differences between corresponding pixels in the stereo images. Factors like image noise and illumination changes can introduce slight variations that the SSD method misinterprets as disparities. Moreover, in regions with low texture or homogeneous surfaces, there aren't enough distinctive features for accurate matching, causing the algorithm to rely on noise rather than actual image content. The fixed window size used in SSD comparisons can make this problem worse, as it may enclose areas with varying depths or textures, further distorting the disparity calculations. As a result, the disparity map becomes speckled with random errors, diminishing the quality of the depth perception.

3.2.2.4 Part d)

MATLAB Code:

The code for part d) (Figure 34) is using the same approach of loading the images and converting them to the desired requirements. Then, it calls the **computeDisparityMapSSD** for the real images, and finally, plots the disparity map like in the previous part.

```
%% d) Compute Disparity Map for Real Stereo Images

leftImgRealPath = 'img/triclopsi2l.jpg';
rightImgRealPath = 'img/triclopsi2r.jpg';

leftImgReal = imread(leftImgRealPath);
rightImgReal = imread(rightImgRealPath);

if size(leftImgReal, 3) == 3
    leftImgRealGray = rgb2gray(leftImgReal);
else
    leftImgRealGray = leftImgReal;
end

if size(rightImgReal, 3) == 3
    rightImgRealGray = rgb2gray(rightImgReal);
else
    rightImgRealGray = rightImgReal;
end

D_real = computeDisparityMapSSD(leftImgRealGray, rightImgRealGray, templateSize, maxDisparity);

figure('Name', 'Disparity Map - Real Images', 'NumberTitle', 'off');
imshow(D_real, [-15 15]);
title('Disparity Map (Real Images)');
colormap("gray");
```

Figure 34. 3D Stereo Vision – the code for preprocessing and computing the disparity for the real images

Results:

As seen in the comparison between the groundtruth for the real image and the computed disparity map in Figure 35, the SSD-based computation achieves very similar results to the ones from the target image. The computed disparity map successfully maps the bushes/grass area of the real images. Both the groundtruth and the computed map struggle with mapping the pavement, the sky, and the parts of the building with repetitive color.

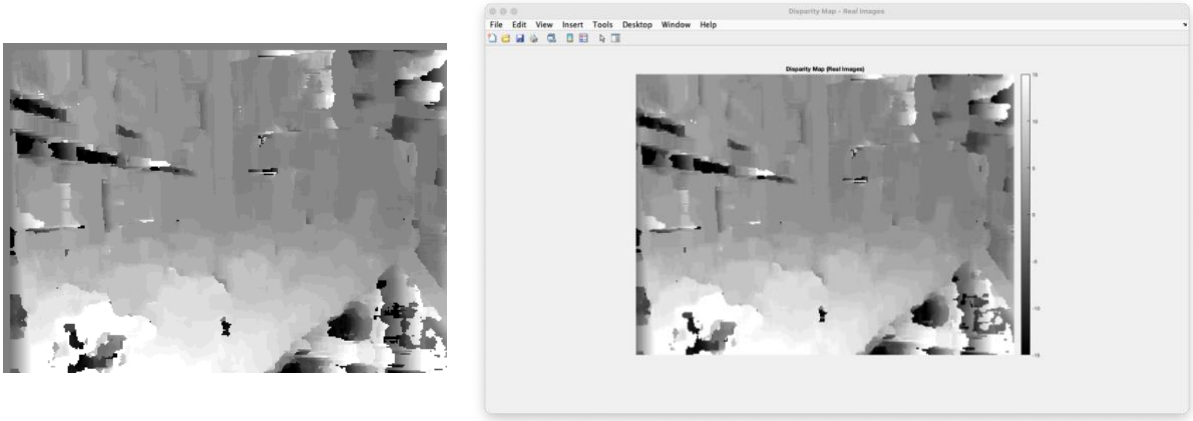

Figure 35. 3D Stereo Vision – Disparity map visualization for Real stereo images using SSD

As mentioned in the previous section, regions with low texture or repetitive patterns pose significant challenges for stereo matching algorithms like SSD. The pavement, sky, and uniformly colored parts of the building lack distinct features that are necessary for accurate correspondence between the left and right images. Without these unique features, the SSD algorithm struggles to find reliable matches, leading to ambiguous or incorrect disparity values in those areas. Additionally, repetitive colors can confuse the matching process by creating multiple potential correspondences, which the algorithm cannot easily resolve. This results in both the ground truth and the computed disparity map failing to accurately represent depth in these regions. However, in textured areas like the bushes and grass, where there are plenty of unique features, the SSD-based computation excels by producing accurate disparity values that closely match the ground truth.

3.2.2.5 Part e) (Additional)

MATLAB Code:

To further experiment with depth estimation, this additional part focuses on Zero-mean Normalized Cross-Correlation (ZNCC) and Normalized Cross-Correlation (NCC) for computing disparity maps. In Figure 36, the code first smooths the left and right grayscale images using Gaussian filtering to reduce noise, following a familiar preprocessing approach used in earlier sections.

Disparity maps are then generated for the synthetic images using both ZNCC and NCC methods, with the results displayed in separate figures. The same process is repeated for the real images, producing disparity maps based on the two correlation techniques. This segment maintains a consistent structure with previous parts of the stereo vision experiments, emphasizing the application of these correlation methods in depth estimation.

```
%% e) Compute Disparity Map using ZNCC and NCC

% Smooth images
leftImgGraySmooth = imgaussfilt(leftImgGray, 1);
rightImgGraySmooth = imgaussfilt(rightImgGray, 1);

% Disparity Map using ZNCC for Synthetic Images
D_ZNCC = computeDisparityMapNCC(leftImgGraySmooth, rightImgGraySmooth, templateSize, maxDisparity, true);

figure('Name', 'Disparity Map using ZNCC - Synthetic Images', 'NumberTitle', 'off');
imshow(D_ZNCC, [-15 15]);
title('Disparity Map using ZNCC (Synthetic Images)');
colormap("gray");
colorbar;

% Disparity Map using NCC for Synthetic Images
D_NCC = computeDisparityMapNCC(leftImgGraySmooth, rightImgGraySmooth, templateSize, maxDisparity, false);

figure('Name', 'Disparity Map using NCC - Synthetic Images', 'NumberTitle', 'off');
imshow(D_NCC, [-15 15]);
title('Disparity Map using NCC (Synthetic Images)');
colormap("gray");
colorbar;

% Smooth real images
leftImgRealGraySmooth = imgaussfilt(leftImgRealGray, 1);
rightImgRealGraySmooth = imgaussfilt(rightImgRealGray, 1);

% Disparity Map using ZNCC for Real Images
D_real_ZNCC = computeDisparityMapNCC(leftImgRealGraySmooth, rightImgRealGraySmooth, templateSize, maxDisparity, true);

figure('Name', 'Disparity Map using ZNCC - Real Images', 'NumberTitle', 'off');
imshow(D_real_ZNCC, [-15 15]);
title('Disparity Map using ZNCC (Real Images)');
colormap("gray");
colorbar;

% Disparity Map using NCC for Real Images
D_real_NCC = computeDisparityMapNCC(leftImgRealGraySmooth, rightImgRealGraySmooth, templateSize, maxDisparity, false);

figure('Name', 'Disparity Map using NCC - Real Images', 'NumberTitle', 'off');
imshow(D_real_NCC, [-15 15]);
title('Disparity Map using NCC (Real Images)');
colormap("gray");
colorbar;
```

Figure 36. 3D Stereo Vision – the code for the ZNCC and NCC approach (1/2)

The function in Figure 37, **computeDisparityMapNCC**, is designed to compute the disparity map using Normalized Cross-Correlation (NCC) or Zero-mean Normalized Cross-Correlation (ZNCC) with a bidirectional disparity search. It accepts the left and right images, template size, maximum disparity, and a boolean flag for zero-mean normalization as inputs.

Initially, the left and right images are converted to double precision to ensure accuracy in calculations. The function determines the image dimensions and computes half of the template height and width for indexing purposes. A zero matrix, **disparityMap**, is initialized to hold the computed disparity values.

```
% Function to Compute Disparity Map using NCC/ZNCC with Bidirectional Disparity
function disparityMap = computeDisparityMapNCC(leftImg, rightImg, templateSize, maxDisparity, zeroMean)
    leftImg = double(leftImg);
    rightImg = double(rightImg);
    [rows, cols] = size(leftImg);
    halfTempH = floor(templateSize(1) / 2);
    halfTempW = floor(templateSize(2) / 2);
    disparityMap = zeros(rows - 2 * halfTempH, cols - 2 * halfTempW);

    for i = 1 + halfTempH : rows - halfTempH
        for j = 1 + halfTempW : cols - halfTempW
            maxScore = -Inf;
            bestDisparity = 0;

            % Extract template from the left image
            templateLeft = leftImg(i - halfTempH : i + halfTempH, j - halfTempW : j + halfTempW);
            if zeroMean
                templateLeft = templateLeft - mean(templateLeft(:));
            end
            denomLeft = sqrt(sum(templateLeft(:) .^ 2));

            % Bidirectional search for matching template in the right image
            for d = -maxDisparity : maxDisparity
                % Ensure the template does not go out of bounds in the right image
                if (j - d - halfTempW >= 1) && (j - d + halfTempW <= cols)
                    % Extract the corresponding template from the right image
                    templateRight = rightImg(i - halfTempH : i + halfTempH, j - d - halfTempW : j - d + halfTempW);
                    if zeroMean
                        templateRight = templateRight - mean(templateRight(:));
                    end
                    denomRight = sqrt(sum(templateRight(:) .^ 2));

                    % Compute NCC/ZNCC score
                    if denomLeft == 0 || denomRight == 0
                        score = 0;
                    else
                        score = sum(sum(templateLeft .* templateRight)) / (denomLeft * denomRight);
                    end

                    % Update the best disparity if the current score is higher
                    if score > maxScore
                        maxScore = score;
                        bestDisparity = d;
                    end
                end
            end

            % Store the best disparity for the current pixel
            disparityMap(i - halfTempH, j - halfTempW) = bestDisparity;
        end
    end
end
```

Figure 37. 3D Stereo Vision – the code for the ZNCC and NCC approach (2/2)

The main processing occurs within nested loops that iterate through each valid pixel location. For each pixel, a template is extracted from the left image. If the **zeroMean** flag is enabled, the mean value of the template is subtracted to perform zero-mean normalization. The normalization denominator is calculated by taking the square root of the sum of squared pixel values in the template.

Next, the function performs a bidirectional search for the best matching template in the right image by iterating over potential disparity values from **-maxDisparity** to **maxDisparity**. It extracts the corresponding template from the right image, applying zero-mean normalization if specified. The NCC or ZNCC score is computed using the dot product of the two templates, normalized by their respective denominators. If either denominator is zero, the score is set to zero to avoid division errors.

The function then checks if the newly calculated score exceeds the current maximum score and updates the best disparity accordingly. Once all potential disparities are evaluated for a given pixel, the optimal disparity is recorded in the **disparityMap**.

Overall, this implementation provides an effective means of calculating disparity maps using NCC/ZNCC, incorporating robust normalization and search techniques to enhance depth estimation in stereo vision tasks.

Results:

The results in Figures 38 and 39 display disparity maps generated using ZNCC and NCC for both synthetic and real stereo images.

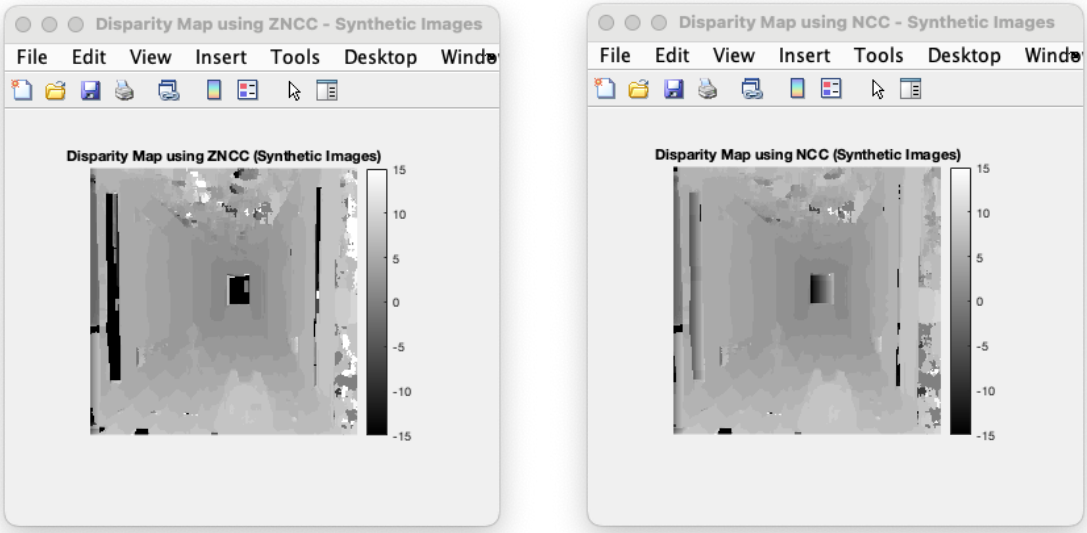

Figure 38. 3D Stereo Vision – Disparity map visualization for Synthetic stereo images using ZNCC and NCC

In Figure 38, showing the synthetic images, both ZNCC and NCC capture the overall depth structure, but ZNCC appears to suffer from increased noise and artifacts, especially around the edges of different regions. The zero-mean normalization in ZNCC, while helping with illumination consistency, can lead to higher sensitivity to minor intensity changes, causing more edge-related noise and inaccuracies. NCC, although somewhat less precise in capturing fine details, exhibits smoother and cleaner edges, as it is less affected by small intensity variations within regions.

In Figure 39, which shows the results for real images, both ZNCC and NCC produce disparity maps that are quite similar overall. Both methods effectively address some of the issues seen in the SSD-based disparity maps, particularly by reducing the building artifacts and handling regions like the pavement and sky more accurately. The extreme black regions that previously represented erroneous depth values in the SSD maps are largely removed, resulting in a more realistic depth representation.

While ZNCC introduces some noise around the edges of different regions, particularly along boundaries, it still improves depth consistency across most areas. NCC, though slightly smoother and less affected by edge noise, similarly provides a stable depth map without the extreme discrepancies seen in SSD. Both ZNCC and NCC perform well in removing extreme

depth errors and improving clarity in large, flat regions like the pavement and sky, making them better suited for depth estimation in these scenes compared to SSD.

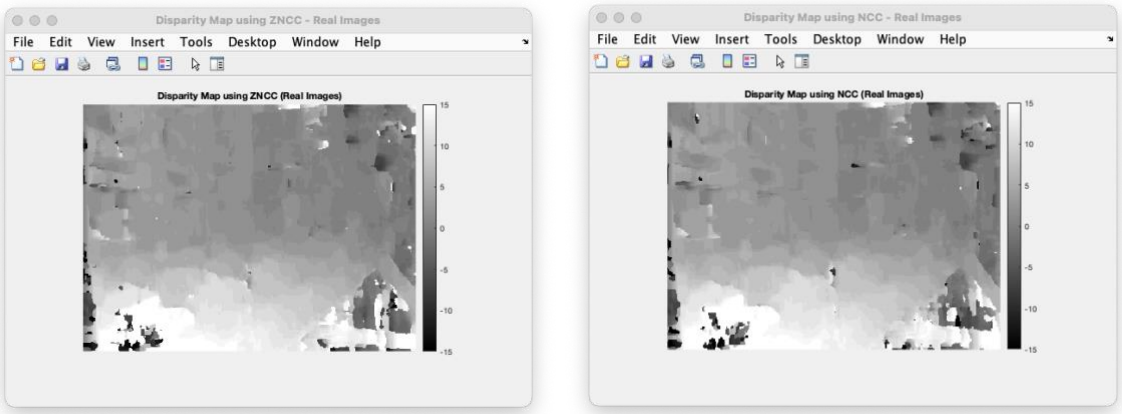

Figure 39. 3D Stereo Vision – Disparity map visualization for Real stereo images using ZNCC and NCC

3.2.2.6 Part f) (Additional)

MATLAB Code:

Figure 40 presents the code for the additional experiments with the Sum of Absolute Differences (SAD) method. In this segment, the code computes disparity maps for both synthetic and real images and visualizes the results in grayscale. The **computeDisparityMapSAD** function is applied to the smoothed grayscale images, producing **D_SAD** for synthetic images and **D_real_SAD** for real images. These disparity maps are displayed with grayscale colormaps, and colorbars are added to indicate depth values, similar to previous visualizations.

The **computeDisparityMapSAD** function performs the core disparity calculation by matching patches between the left and right images based on SAD. It begins by converting the images to double precision to ensure accurate numerical operations, then sets up the necessary template dimensions and initializes a zero-filled **disparityMap** matrix.

For each valid pixel location in the left image, the function extracts a template (**templateLeft**) centered around that pixel. It then performs a bidirectional search across the defined disparity range (from **-maxDisparity** to **maxDisparity**) in the right image. For each disparity value **d**, a matching template (**templateRight**) is extracted from the right image, ensuring that the patch does not exceed image boundaries.

The SAD score is calculated as the sum of absolute pixel differences between **templateLeft** and **templateRight**. SAD is particularly useful for handling outliers, as it minimizes their influence compared to squared differences used in SSD. The function keeps track of the minimum SAD score (**minSAD**) for each pixel, updating **bestDisparity** whenever a lower SAD is encountered.

Once all disparities are evaluated for a given pixel, the best disparity (the one with the lowest SAD score) is recorded in **disparityMap**. This approach provides a balance between simplicity and effectiveness in matching image regions, especially in structured environments. However, like SSD, SAD can struggle with textureless regions or repetitive patterns, where subtle details are insufficient for accurate matching

```
%% f) Compute Disparity Map using SAD

% Disparity Map using SAD for Synthetic Images
D_SAD = computeDisparityMapSAD(leftImgGraySmooth, rightImgGraySmooth, templateSize, maxDisparity);

figure('Name', 'Disparity Map using SAD - Synthetic Images', 'NumberTitle', 'off');
imshow(D_SAD, [-15 15]);
title('Disparity Map using SAD (Synthetic Images)');
colormap("gray");
colorbar;

% Disparity Map using SAD for Real Images
D_real_SAD = computeDisparityMapSAD(leftImgRealGraySmooth, rightImgRealGraySmooth, templateSize, maxDisparity);

figure('Name', 'Disparity Map using SAD - Real Images', 'NumberTitle', 'off');
imshow(D_real_SAD, [-15 15]);
title('Disparity Map using SAD (Real Images)');
colormap("gray");
colorbar;

% Function to Compute Disparity Map using SAD with Bidirectional Disparity
function disparityMap = computeDisparityMapSAD(leftImg, rightImg, templateSize, maxDisparity)
    leftImg = double(leftImg);
    rightImg = double(rightImg);
    [rows, cols] = size(leftImg);
    halfTempH = floor(templateSize(1) / 2);
    halfTempW = floor(templateSize(2) / 2);
    disparityMap = zeros(rows - 2 * halfTempH, cols - 2 * halfTempW);

    for i = 1 + halfTempH : rows - halfTempH
        for j = 1 + halfTempW : cols - halfTempW
            minSAD = Inf;
            bestDisparity = 0;

            % Extract the template from the left image
            templateLeft = leftImg(i - halfTempH : i + halfTempH, j - halfTempW : j + halfTempW);

            % Bidirectional search for matching template in the right image
            for d = -maxDisparity : maxDisparity
                % Ensure the template does not go out of bounds in the right image
                if (j - d - halfTempW >= 1) && (j - d + halfTempW <= cols)
                    % Extract the corresponding template from the right image
                    templateRight = rightImg(i - halfTempH : i + halfTempH, j - d - halfTempW : j - d + halfTempW);

                    % Calculate the Sum of Absolute Differences (SAD)
                    sad = sum(sum(abs(templateLeft - templateRight)));

                    % Update the best disparity if the current SAD is lower
                    if sad < minSAD
                        minSAD = sad;
                        bestDisparity = d;
                    end
                end
            end

            % Store the best disparity for the current pixel
            disparityMap(i - halfTempH, j - halfTempW) = bestDisparity;
        end
    end
end

```

Figure 40. 3D Stereo Vision – the code for the SAD approach

Results:

Figure 41 presents disparity maps computed using the Sum of Absolute Differences method for both synthetic and real images. The results exhibit smoother transitions compared to those obtained with the Sum of Squared Differences (SSD) approach. This increased smoothness is primarily due to SAD's use of absolute differences, which are less sensitive to outliers than squared differences. Consequently, SAD reduces the impact of noise and extreme pixel values, leading to more uniform disparity estimations. However, like SSD, SAD may still encounter challenges in accurately matching regions with repetitive patterns or low texture, as these areas lack distinctive features necessary for precise correspondence.

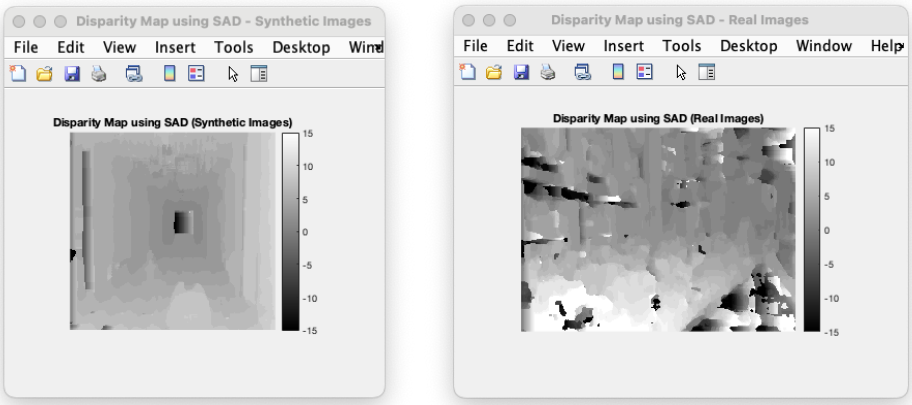

Figure 41. 3D Stereo Vision – Disparity map visualization for both Real and Synthetic stereo images using SAD

4 References

[1] J. Sauvola and M. Pietikäinen, “Adaptive document image binarization,” *Pattern Recognition*, vol. 33, no. 2, pp. 225–236, Feb. 2000, doi: 10.1016/S0031-3203(99)00055-2.

[2] “Cross-correlation,” *Wikipedia*. Apr. 24, 2024. Accessed: Nov. 07, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Cross-correlation&oldid=1220518127#Normalized_cross-correlation_(NCC)

[3] A. Premana, A. Wijaya, and M. Soeleman, *Image segmentation using Gabor filter and K-means clustering method*. 2017, p. 99. doi: 10.1109/ISEMANTIC.2017.8251850.

[4] “Sauvola’s Method - (Computer Vision and Image Processing) - Vocab, Definition, Explanations | Fiveable.” Accessed: Nov. 07, 2024. [Online]. Available: https://library.fiveable.me/key-terms/computer-vision-and-image-processing/sauvolas-method

[5] A. Garg, “Stereo Vision: Depth Estimation between object and camera,” Analytics Vidhya. Accessed: Nov. 07, 2024. [Online]. Available: https://medium.com/analytics-vidhya/distance-estimation-cf2f2fd709d8

[6] M. Thoma, “Zero Mean Normalized Cross-Correlation,” Martin Thoma. Accessed: Nov. 07, 2024. [Online]. Available: https://martin-thoma.com/zero-mean-normalized-cross-correlation/zero-mean-normalized-cross-correlation/