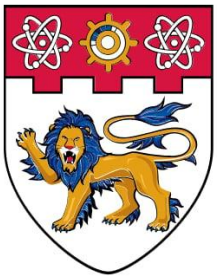


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

College of Computing and Data Science

AY 2025/2026

SC4061 Computer Vision

Lab 2 Report

Submitted By: Isaac Chun Jun Heng

Matriculation No.: U2221389B

Table of Contents

1. Preliminary

This experiment was implemented with MATLAB rather than Python. In the submitted source code, each part is split into its own files for easy running of source code. This report is also part of the submission of Lab 2.

2. Introduction

This laboratory aims to provide further exposure to advanced image processing and computer vision techniques through the exploration of different image segmentation methods, and template matching with pixel sum of squares difference (SSD). The following concepts are introduced in the laboratory:

1. Otsu's Global Thresholding
2. Niblack's Local Thresholding

The following subsections includes the technical details to replicate the results, and the official code release can be found at https://github.com/isaacchunn/computer-vision if any of the files seem to not load.

3. Experiments

3.1 Image Segmentation

3.1.a) Otsu's Global Thresholding

In this section, we apply Otsu's global thresholding algorithm to perform global thresholding for text segmentation. We perform this on the four given images and their associated ground truth **(document01.bmp, document02.bmp, document03.bmp and document04.bmp),** and **(document01-GT.tiff, document02-GT.tiff, document03-GT.tiff and document04-GT.tiff)** respectively. We then quantitatively evaluate the segmentation results by computing the sum of difference image between segmented binary image and corresponding ground truth, and subsequently perform some analysis.

3.1.a.1) Loading of Assets and Preprocessing

In the following Figure, we implement a systematic preprocessing workflow, that is used throughout the sections of 3.1 Image Segmentation. In this workflow, we carry out the following:

1. **Directory Setup**: We define the image folder path and specify filenames for both the document images (document01-04.bmp) and ground truth files (document01-04-GT.tiff) using cell arrays for efficient indexing.

2. **Grayscale Conversion**: With each input images, if they are RGB, we convert them to grayscale using **rgb2gray** to ensure compatibility with the signal channel threshold algorithms we aim to implement as part of the lab, and Otsu's method operates on intensity distributions.
3. **Ground Truth Binarization**: The ground truth images are then explicitly binarized using **imbinarize** to ensure binary values (0,1**),** which will subsequently enable accurate XOR-based pixel difference calculations in our subsequent analysis.

This preprocessing pipeline should ensure data consistency and eliminate any format-related error that could compromise or affect our evaluation, while ensuring our method would apply for any set of input images and ground truth images.

otsu_thresholding.m ×

D:\GitHub\computer-vision\lab2\otsu_thresholding.m

```
%% Part 3.1a) Otsu's Global Thresholding
%% 1. Loading of images and ground truths
fprintf('1. Loading images and ground truths...\n');

% Define image folder that stores the assets from NTULearn
imageFolder = 'assets/';

% Define image files and their relevant ground truths
imageFiles = {'document01.bmp', 'document02.bmp', 'document03.bmp', 'document04.bmp'};
groundTruthFiles = {'document01-GT.tiff', 'document02-GT.tiff', 'document03-GT.tiff', 'document04-GT.tiff'};

% Preallocate cell arrays to store loaded images and ground truths
images = cell(length(imageFiles), 1);
groundTruths = cell(length(imageFiles), 1);

% Load all images and ground truths (each should have its own ground truth
for i = 1:length(imageFiles)
    % Read image
    img = imread(fullfile(imageFolder, imageFiles{i}));

    % Ensure grayscale if in RGB
    if size(img, 3) == 3
        img = rgb2gray(img);
    end
    images{i} = img;

    % Read ground truth
    gt = imread(fullfile(imageFolder, groundTruthFiles{i}));

    % Ensure ground truth is binary
    if ~islogical(gt)
        gt = imbinarize(gt);
    end
    groundTruths{i} = gt;

    fprintf('  Loaded: %s\n', imageFiles{i});
end

```

Figure 1: Otsu Thresholding – Loading of Assets and Preprocessing

3.1.a.2) Storage of Results + Carrying out Otsu Thresholding

Figure 2 below describes the process of carrying out Otsu's thresholding. Otsu's algorithm aims to maximize the inter-class variance between foreground and background pixels, performing optimally when the intensity histogram exhibits a bimodal distribution (histogram has two distinct

peaks) To validate this assumption and try to identify potential failure cases, we also implement a brute-force threshold search (0 to 1in steps of 0.01) as a benchmark.

We use a struct to store Otsu and our best threshold results (which will be computed in later sections). For each image, we carry out Otsu's thresholding. The Otsu's threshold level is computed using inbuilt **graythresh** function, which calculates the optimal threshold to separate the foreground and background pixels. Then, this result is binarized with the threshold acquired from Otsu's method to produce our resultant **binaryImg**. Finally, to evaluate how the result acquired from Otsu's method matches the ground truth, we calculate/compute the difference using **XOR** and use the **sum** method to compute the total number of different pixels when compared to the ground truth. We then store these results in the Otsu results and print it on the console.

otsu_thresholding.m

D:\GitHub\computer-vision\lab2\otsu_thresholding.m

```
%% 2. Store some results structure for Otsu and best results in subsequent experiments.
fprintf('2. Initializing results storage...\n');

% Structure to store Otsu results
otsuResults = struct('imageName', {}, 'threshold', {}, 'binaryImg', {}, ...
                     'diffSum', {}, 'accuracy', {}, 'errorRate', {});

% Structure to store best threshold results
bestResults = struct('imageName', {}, 'threshold', {}, 'binaryImg', {}, ...
                     'diffSum', {}, 'accuracy', {}, 'errorRate', {});

% Define threshold range for testing
thresholdTests = 0:0.01:1; % Test from 0 to 1 with step 0.01 (i.e 0.01, 0.02...)

%% 3. For each image, process and compute metrics while storing our Otsu results.
fprintf('3. Processing images with Otsu and storing results...\n');

for i = 1:length(imageFiles)
    fprintf('Processing %s...\n', imageFiles{i});

    img = images{i};
    gt = groundTruths{i};
    totalPixels = numel(gt);

    % Calculate and apply Otsu's thresholding
    otsuLevel = graythresh(img); % Computation of optimal threshold
    binaryImg = imbinarize(img, otsuLevel); % Apply threshold

    % Compute difference with ground truth
    diffImage = xor(binaryImg, gt);
    diffSum = sum(diffImage(:)); % Total number of different pixels

    % Compute metrics
    correctPixels = totalPixels - diffSum;
    accuracy = (correctPixels / totalPixels) * 100;
    errorRate = (diffSum / totalPixels) * 100;

    % Store Otsu results
    otsuResults(i).imageName = imageFiles{i};
    otsuResults(i).threshold = otsuLevel;
    otsuResults(i).binaryImg = binaryImg;
    otsuResults(i).diffSum = diffSum;
    otsuResults(i).accuracy = accuracy;
    otsuResults(i).errorRate = errorRate;

    fprintf('    Otsu Threshold: %.3f | Accuracy: %.2f%% | Error Rate: %.2f%%\n', ...
            otsuLevel, accuracy, errorRate);
end
```

Figure 2: Storing results and carrying out Otsu's thresholding

3.1.a.3) Calculating other Best Thresholds

To validate Otsu's method, we implement an exhaustive search that tests all possible set thresholds against the ground truth data. The algorithm initializes through tracking variables and initializing **minDiffSum to inf, bestThreshold = 0 and bestBinaryImg = []** and then subsequently iterate through threshold values from 0 to 1 in steps of 0.01. Specifically, for each candidate threshold we:

1. Apply binarization using **imbinarize**(img, t), where t is the threshold value to test
2. Compute the difference from ground truth using XOR: xor(testBinaryImg, gt)
3. Sum the total pixel differences

When a particular threshold manages to produce a smaller difference sum than the previous candidates, we can then update our best result. Our exhaustive approach is meant to serve two purposes: verification of whether Otsu's method truly identifies the optimal threshold, while also providing a baseline for comparison that accounts for image-specific characteristics.

While our method is computationally intensive (testing of 100 thresholds per image) and requires the ground truth, this rather brute-force method has a better chance of guaranteeing in finding the threshold that best matches the ground truth **for each** specific image and is agnostic to different image distributions, which can potentially outperform Otsu's approach when images have unique characteristics that don't align with Otsu's assumptions or strengths (images have bimodal distributions).

otsu_thresholding.m

D:\GitHub\computer-vision\lab2\otsu_thresholding.m

```
88  %% 4. Test different thresholds and find the best one
89  fprintf('4. Testing different thresholds to find optimal values...\n');
90
91  for i = 1:length(imageFiles)
92      fprintf('\n  Testing thresholds for %s...\n', imageFiles{i});
93
94      img = images{i};
95      gt = groundTruths{i};
96      totalPixels = numel(gt);
97
98      % Initialize tracking variables for best threshold
99      minDiffSum = inf;
100     bestThreshold = 0; % we can also assume the best threshold starts from Otsu level, but this is not guaranteed.
101     bestBinaryImg = []; % store empty list
102
103     % Test each threshold
104     for t = thresholdTests
105         % Apply threshold
106         testBinaryImg = imbinarize(img, t);
107
108         % Compute difference
109         testDiffImage = xor(testBinaryImg, gt);
110         testDiffSum = sum(testDiffImage(:));
111
112         % Check if this is the best threshold so far
113         if testDiffSum < minDiffSum
114             minDiffSum = testDiffSum;
115             bestThreshold = t;
116             bestBinaryImg = testBinaryImg;
117         end
118     end
119
120     % Compute metrics for best threshold
121     bestCorrectPixels = totalPixels - minDiffSum;
122     bestAccuracy = (bestCorrectPixels / totalPixels) * 100;
123     bestErrorRate = (minDiffSum / totalPixels) * 100;
124
125     % Store best threshold results
126     bestResults(i).imageName = imageFiles{i};
127     bestResults(i).threshold = bestThreshold;
128     bestResults(i).binaryImg = bestBinaryImg;
129     bestResults(i).diffSum = minDiffSum;
130     bestResults(i).accuracy = bestAccuracy;
131     bestResults(i).errorRate = bestErrorRate;
132
133     fprintf('Best Threshold: %.3f | Accuracy: %.2f%% | Error Rate: %.2f%%\n', ...
134             bestThreshold, bestAccuracy, bestErrorRate);
135     fprintf('Improvement over Otsu: %.2f%% reduction in error\n', ...
136             ((otsuResults(i).errorRate - bestErrorRate) / otsuResults(i).errorRate) * 100);
137 end
```

Figure 3: Testing different thresholds and finding the best one

3.1.a.4) Results – Plotting and Analysis

Finally, after calculating the threshold acquired from Otsu and the best threshold that minimizes the error through our iterative method, we plot out the following results seen in the following Figures and Tables and conduct some analysis.

Image	Threshold	Diff Sum	Accuracy (%)	Error Rate (%)
document01.bmp	0.545	27849	95.78	4.22
document02.bmp	0.439	9476	97.00	3.00
document03.bmp	0.690	179165	81.26	18.74
document04.bmp	0.598	134538	78.77	21.23

Table 1: Otsu Threshold Results

Image	Threshold	Diff Sum	Accuracy (%)	Error Rate (%)
document01.bmp	0.450	24088	96.35	3.65
document02.bmp	0.440	9476	97.00	3.00
document03.bmp	0.400	17038	98.22	1.78
document04.bmp	0.330	19426	96.94	3.06

Table 2: Best Threshold Results

Image	Otsu Error	Best Error	Improvement (%)
document01.bmp	4.22	3.65	0.57
document02.bmp	3.00	3.00	0.00
document03.bmp	18.74	1.78	16.96
document04.bmp	21.23	3.06	18.16

Table 3: Comparison: Otsu vs Best

From the tables listed above, we observe some key findings. Our exhaustive search outperforms Otsu in 75% of cases (3/4 images), with huge improvements of 16.96% and 18.16% for document03 and document04 respectively. We notice that Otsu is optimal in document02, with 0.00% improvement, indicating both our exhaustive search and Otsu found the optimal threshold. To give a better idea and explain the data in the tables, we plot the visualizations of each document against the ground truth, Otsu's threshold and our best threshold, as well as the intensity histogram to acquire more detailed analysis. The following Figures provide this visualization (Figure 4-7).

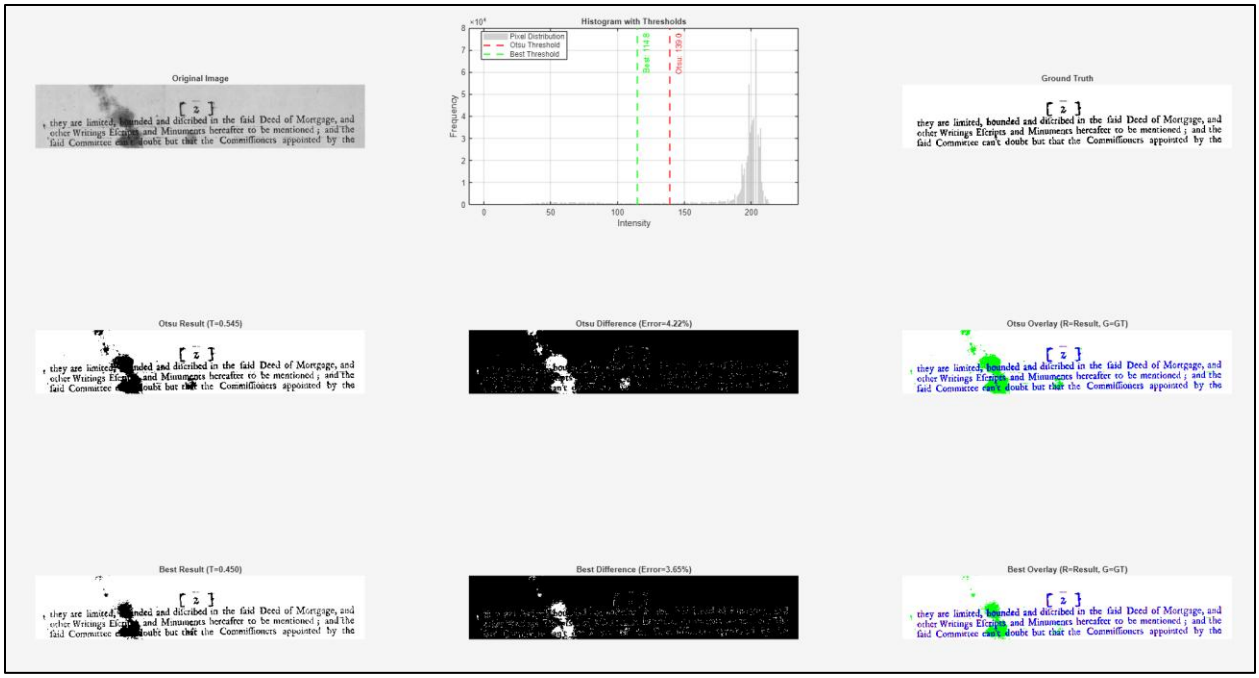


Figure 4: document01.bmp binarization results (Otsu and best)

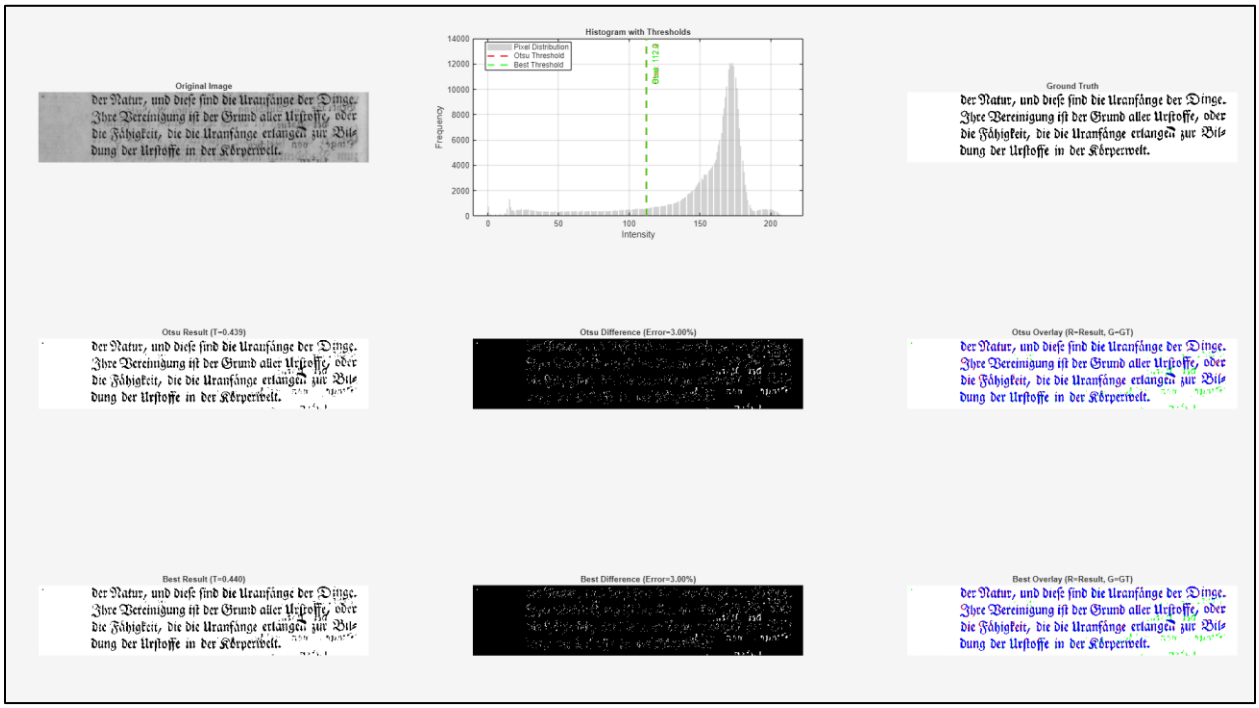


Figure 5: document02.bmp binarization results (Otsu and best)

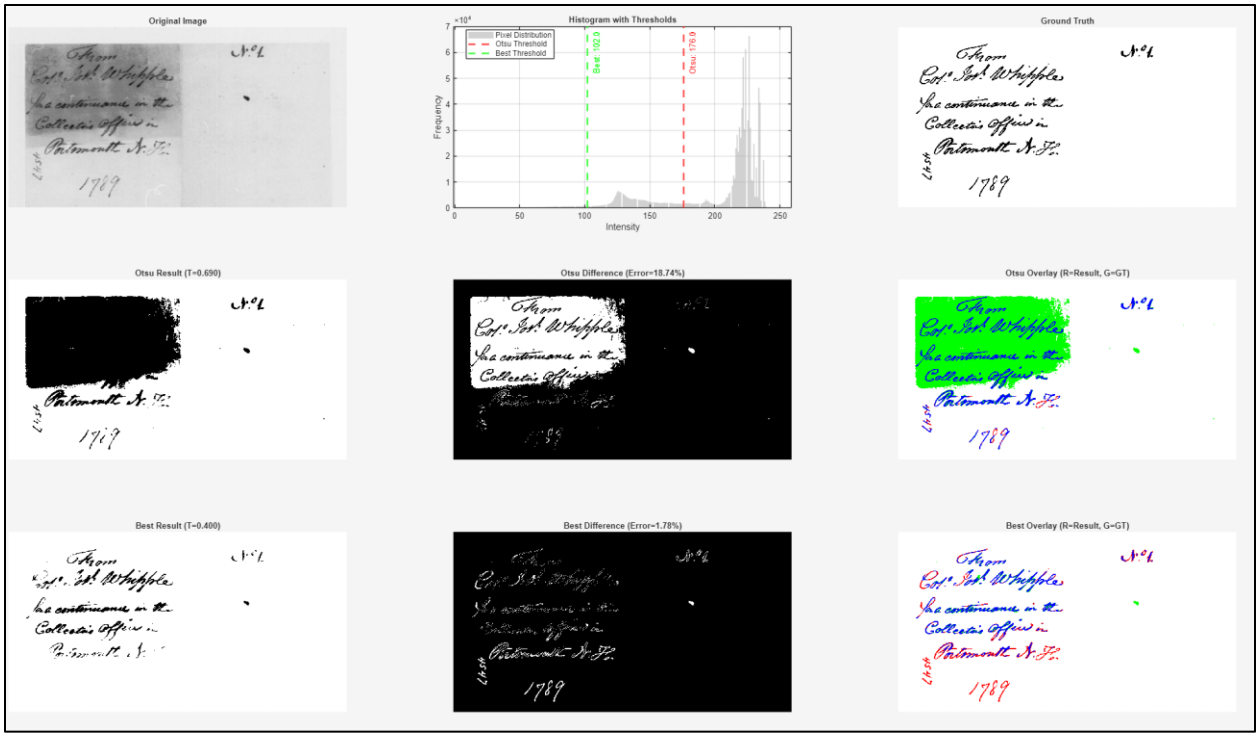


Figure 6: document03.bmp binarization results (Otsu and best)

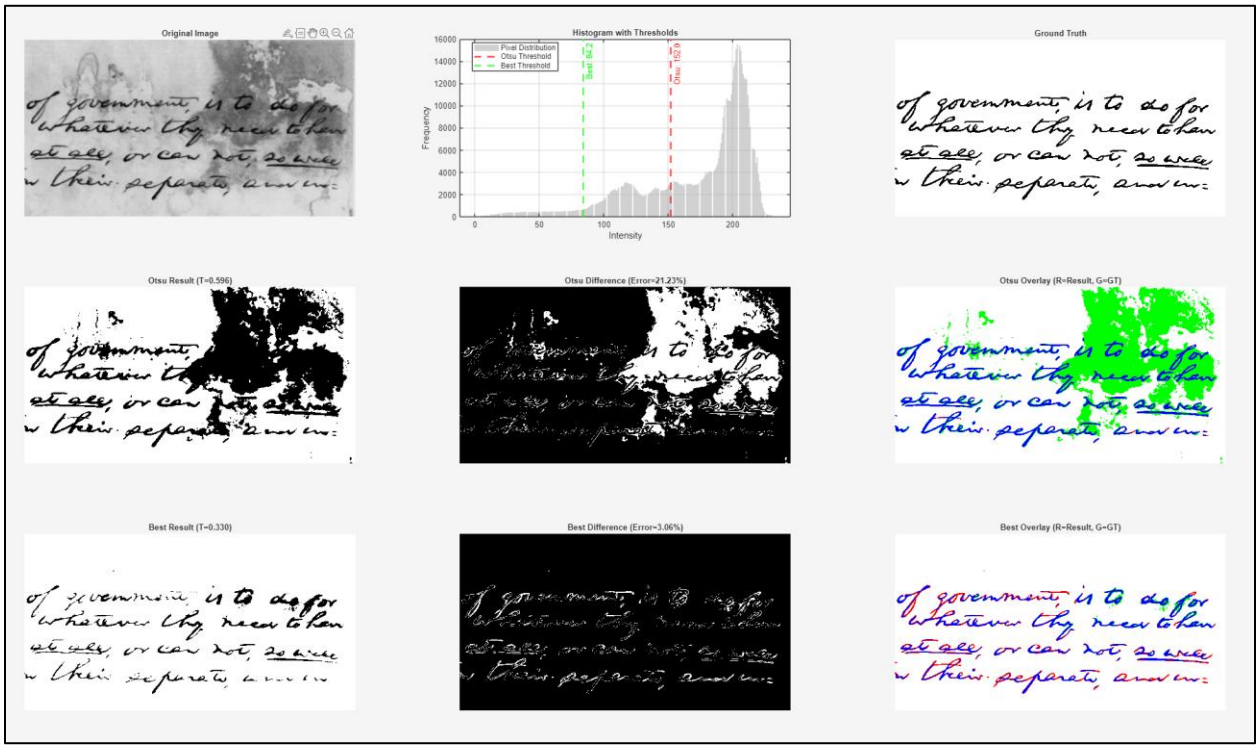


Figure 7: document04.bmp binarization results (Otsu and best)

Across the board, the visualizations reveal significant performance differences between Otsu's method and our exhaustive threshold search, specifically for documents with challenging characteristics (black patches, uneven lighting, etc.)

We can observe that the best improvements can be seen in Figure 6 and Figure 7. In document03.bmp, we observe a bimodal distribution in the histogram plot of document03.bmp which typically indicates two distinct intensity populations – often corresponding to the foreground text and background. This leads Otsu's method into concluding that the best threshold must lie somewhere in there in. However, the histogram does not give any context of spatial information, which results in suboptimal threshold selection using Otsu's method. In reality, the best threshold is determined by our best method. With our best threshold of 0.490 as compared to Otsu's 0.690, we manage to entirely eliminate the black patch and reveal the text as a rather clean result.

Similarly, in Figure 7 (document04), we see the same pattern. Otsu's method concludes that the threshold lies somewhere in between the bimodal distribution. This produces heavy black patches that obscure substantial portions of the document. Our exhaustive search finds the optimal threshold at 0.330, which is notably lower than either of the histogram peak, which reduces the error significantly and producing a significantly cleaner result.

Additionally, in Figure 4 and 5, this problem is not really seen as these documents have relatively uniform backgrounds and clearer separation between the text and background in the histogram, which are conditions where Otsu's assumptions hold well. In conclusion, Otsu's method seems to struggle when histogram-based statistics do not necessarily align with the spatial reality/ground truth. Bimodal distributions that are caused by uneven lighting, low contrast, complex backgrounds or affected by high noise can mislead the algorithm into selecting suboptimal thresholds that maximize inter-class variance statistically but fail to separate the foreground and background correctly in the actual image. Our exhaustive search, when compared against the ground truth identifies thresholds that minimize pixel-level errors regardless of histogram shape, being agnostic and scalable to different images.

3.1.b) Niblack's Local Thresholding

In this section, we apply Niblack's local thresholding algorithm to perform local thresholding for text segmentation. Otsu global thresholding fails when illumination in the image varies spatially across the image. Niblack's method addresses this through computation of adaptive thresholds at each pixel based on local statistics.

$$t = \mu_N + k * \sigma_N$$

We perform Niblack's local thresholding on the four given images and their associated ground truth **(document01.bmp, document02.bp, document03.bmp and document04.bmp),** and **(document01-GT.tiff, document02-GT.tiff, document03-GT.tiff and document04-GT.tiff)**

respectively. We then quantitatively evaluate the segmentation results by computing the sum of difference image between segmented binary image and corresponding ground truth and subsequently perform some analysis. The loading and processing of the image is identical to Part 3.1.a.1. To prevent duplicates, it will be skipped.

3.1.b.1) Storage of Results

As part of Niblack's local thresholding implementation, we can now define a hyperparameter **k** (typically -0.2) and **windowSize**, which determine the threshold at each pixel based on the mean and the standard deviation of the neighborhood surrounding that pixel. In this case, the parameter k controls the threshold sensitivity, with negative values such as -0.2 lowering the threshold in high-contrast regions to improve foreground detection. The **windowSize** determines the spatial extent of the neighborhood. To facilitate systematic evaluation and comparison of results, we also extend our results structure to store not only the binary image, difference sum, accuracy and error rate (as with Otsu), but also the specific k and windowSize values used to acquire each result.

```
%% 2. Initialize results storage
fprintf('\n2. Initializing results storage...\n');

% Structure to store Niblack results for each image
niblackResults = struct('imageName', {}, 'k', {}, 'windowSize', {}, ...
                        'binaryImg', {}, 'diffSum', {}, 'accuracy', {}, 'errorRate', {});
```

Figure 8: Storage of Results for Niblack's Local Thresholding

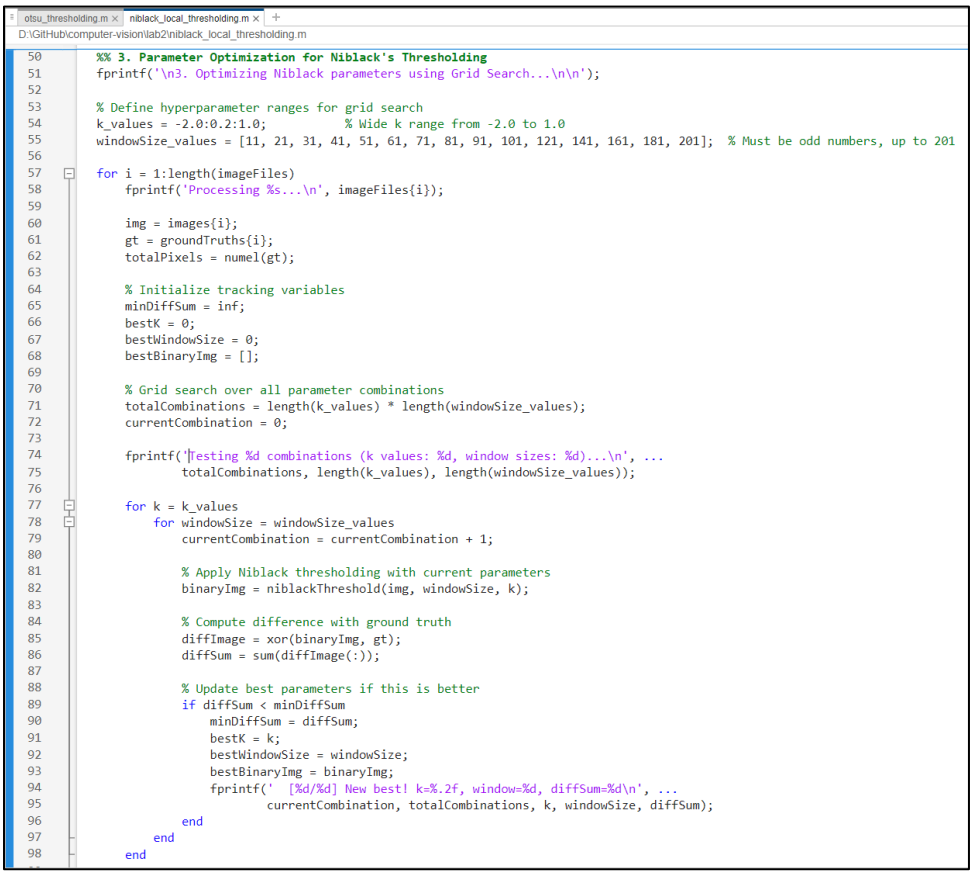
3.1.b.2) Carrying out Niblack's Local Thresholding

In the following Figure 9, in order to study how to obtain the best quantitative segmentation results, we implement an exhaustive grid search over the hyperparameters k and windowSize within carefully defined ranges. The rationale for this grid search is Niblack's high sensitivity to hyperparameter selection, which varies significantly based on different input images. For this example, we define the search space as $\boldsymbol{k} \in [-\mathbf{2.0}, -\mathbf{1.9}, \ldots \mathbf{1.0}]$ with step size 0.1 and **windowSize** $\in \{11,21,31,41,51,61,71,81,91,101,121,141,161,181,201\}$ using odd integers to ensure centered windows. Looping through these parameters gives us 3660 parameter combinations in total.

Then, for each image, we do the following:

1. Initialize tracking variables such as minDiffSum = inf, bestK = 0, bestWindowSize = 0 and bestBinaryImg = []
2. Systematically perform grid search through all parameter combinations
3. Apply niblackThreshold(img, k, windowSize) to generate binary segmentation
4. Employ the same XOR-based evaluation process used in Otsu's analysis
5. Continunously update the best parameters whenever diffSum < minDiffSum, storing the superior k, windowSize and binaryImage.

6. When a new optimum is detected, we print the result to monitor optimization process.
7. Store the optimum in niblackResults structure as defined in the previous section.



```
%% 3. Parameter Optimization for Niblack's Thresholding
fprintf('\n3. Optimizing Niblack parameters using Grid Search...\n\n');

% Define hyperparameter ranges for grid search
k_values = -2.0:0.2:1.0;          % Wide k range from -2.0 to 1.0
windowSize_values = [11, 21, 31, 41, 51, 61, 71, 81, 91, 101, 121, 141, 161, 181, 201];  % Must be odd numbers, up to 201

for i = 1:length(imageFiles)
    fprintf('Processing %s...\n', imageFiles{i});

    img = images{i};
    gt = groundTruths{i};
    totalPixels = numel(gt);

    % Initialize tracking variables
    minDiffSum = inf;
    bestK = 0;
    bestWindowSize = 0;
    bestBinaryImg = [];

    % Grid search over all parameter combinations
    totalCombinations = length(k_values) * length(windowSize_values);
    currentCombination = 0;

    fprintf('Testing %d combinations (k values: %d, window sizes: %d)...\n', ...
           totalCombinations, length(k_values), length(windowSize_values));

    for k = k_values
        for windowSize = windowSize_values
            currentCombination = currentCombination + 1;

            % Apply Niblack thresholding with current parameters
            binaryImg = niblackThreshold(img, windowSize, k);

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

            % Update best parameters if this is better
            if diffSum < minDiffSum
                minDiffSum = diffSum;
                bestK = k;
                bestWindowSize = windowSize;
                bestBinaryImg = binaryImg;
                fprintf('  [%d/%d] New best! k=%.2f, window=%d, diffSum=%d\n', ...
                       currentCombination, totalCombinations, k, windowSize, diffSum);
            end
        end
    end
```

Figure 8: Niblack Local Thresholding – Implementation and Optimization

```
function binary = performNiblackSegmentation(img, k, winSize)
    % Niblack's adaptive thresholding implementation
    % T(x,y) = μ(x,y) + k×σ(x,y)

    img = double(img);
    radius = floor(winSize / 2);

    % Pad image for boundary handling
    paddedImg = padarray(img, [radius, radius], 'symmetric');

    % Compute local statistics
    avgKernel = fspecial('average', winSize);
    localMu = imfilter(paddedImg, avgKernel, 'replicate');
    localMuSq = imfilter(paddedImg.^2, avgKernel, 'replicate');

    % Extract central region
    localMu = localMu(radius+1:end-radius, radius+1:end-radius);
    localMuSq = localMuSq(radius+1:end-radius, radius+1:end-radius);

    % Compute standard deviation
    localVar = max(0, localMuSq - localMu.^2);
    localSigma = sqrt(localVar);

    % Apply Niblack formula
    adaptiveThreshold = localMu + k * localSigma;

    % Threshold to binary
    binary = img > adaptiveThreshold;
end
```

Figure 9: Niblack Local Thresholding Algorithm Implementation

3.1.b.3) Results

Image	Best K	Best Window Size	Diff Sum	Accuracy (%)	Error Rate (%)
document01.bmp	-1.30	201	24716	96.26	3.74
document02.bmp	-0.95	201	16395	94.80	5.20
document03.bmp	-1.85	141	23359	97.56	2.44
document04.bmp	-1.40	201	14905	97.65	2.35

Table 4: Summary of optimal parameters for Niblack's thresholding across all input images with exhaustive grid search

Image	Niblack Diff Sum	Niblack Accuracy (%)	Otsu Diff Sum	Otsu Accuracy (%)
document01.bmp	24716	96.26	27849	95.78
document02.bmp	16395	94.80	9476	97.00
document03.bmp	23359	97.56	179165	81.26
document04.bmp	14905	97.65	134538	78.77

Table 5: Comparative Performance of Niblack's vs Otsu's Thresholding

Analysis:

Table 4 presents the optimal hyperparameters identified as part of our exhaustive grid search approach with Niblack's algorithm across all test images. We also observe several key patterns from our analysis, particularly with Niblack's performance. First, Niblack's local thresholding demonstrates consistently good performance across all the input images, with difference sums ranging from approximately 15,000 to 25,000 pixels and achieving accuracies between 94.80% and 97.65%. Notably, unlike Otsu's method which has major outliers in document03 and document04 in terms of error rates, Niblack's algorithm shows no major outlier performance, suggesting that leveraging spatial context in image segmentation tasks provides more robust and stable results, causing the standard deviation of results to be less from the mean.

Furthermore, the optimal k values identified range from -1.85 to -0.95, which is substantially more negative than the commonly recommended default value of -0.2. This suggests that our test images require more aggressive foreground detection (due to lower thresholds used), likely due to the degraded nature of our images with their inherent contrast.

Interestingly, we also observe a strong preference for larger window sizes, with ¾ images achieving optimal performance at windowSize = 201 (the maximum value in our search values), while document03.bmp achieves its optimum at windowSize = 141. This phenomenon requires deeper investigation. In general, from the definition of Niblack's, larger windows provide several advantages such as being able to capture broader spatial context, which makes Niblack more robust to local noise and small-scale intensity variations. It is also able to better estimate background intensity in regions with uneven illumination. In our test images, the influence of individual text strokes may also play a factor when using our algorithm.

Additionally, Table 5 provides a comparative analysis of Niblack's and Otsu's methods. Niblack algorithm substantially outperforms Otsu in ¾ of images, and most significantly for document03 and document04. However, it is noted that Niblack performs worse on document02, where its diffSum (16395) vs Otsu's 9476. To investigate this phenomenon, we plot the visualizations of Niblack's output on the input images and conduct further analysis.

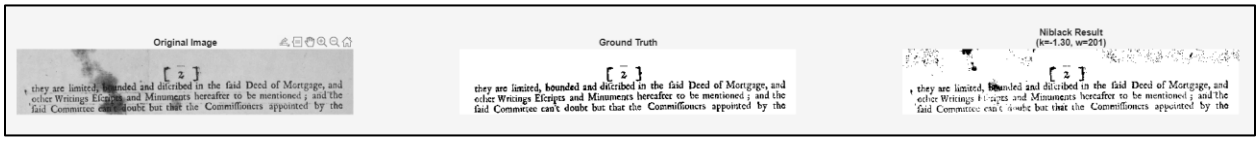


Figure 10: Niblack Local Thresholding – Document01

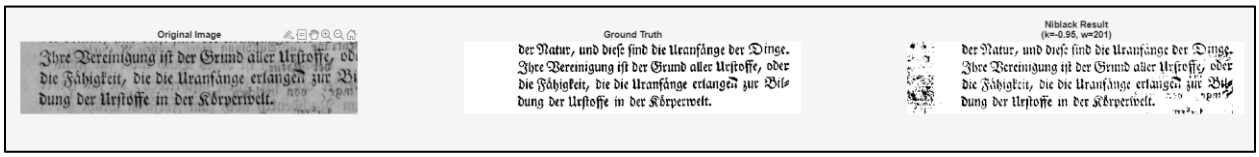


Figure 11: Niblack Local Thresholding – Document02

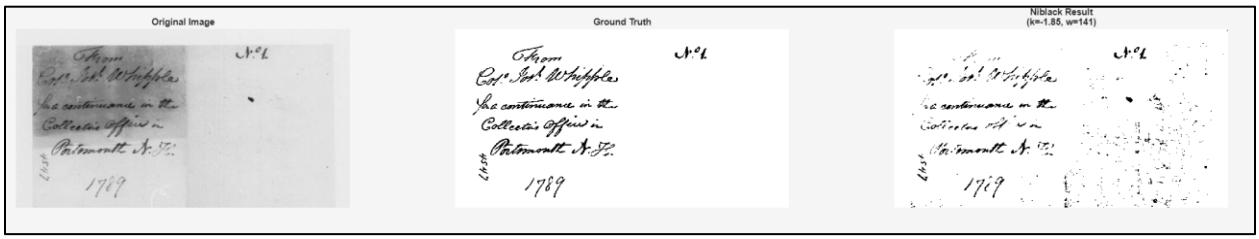


Figure 12: Niblack Local Thresholding – Document03

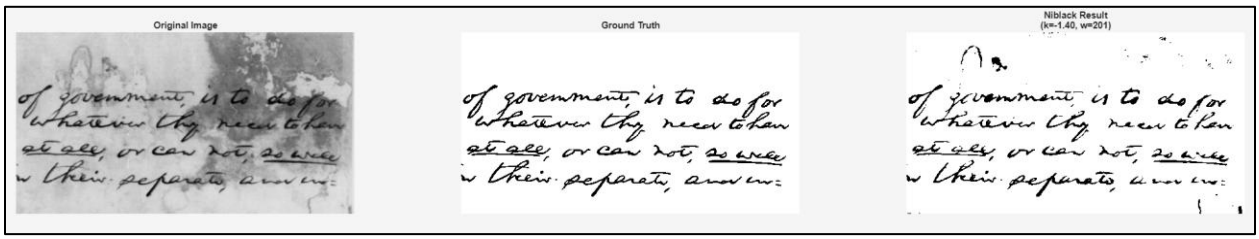


Figure 13: Niblack Local Thresholding – Document04

From visual inspection of Figure 10 to 13, when compared with Otsu's results (Figures 4 to 7), other than metrics, we can confirm visually that Niblack's local thresholding generally produces superior segmentation quality. The extensive black regions that were inherent in Otsu's results (document03 and document04) are almost entirely eliminated when using Niblack's spatially adaptive approach. This also demonstrates that incorporating local spatial context enables the algorithm to make more informed pixel-wise decisions, which allows effective adaptation to varying lighting conditions and background characteristics that global thresholding methods like Otsu is weak in.

However, careful examination of Figure 11 (document02) reveals Niblack's underperformance of that particular image. The segmented result exhibits residual black artifacts along the image edges (left and right of images) which subsequently increases the difference sum relative to the ground truth data. Analyzing the initial document02 image, it features relatively uniform lighting and high contrast between the text and background, a condition that global thresholding methods like Otsu might perform better in. Using adaptive approaches may actually introduce edge artifacts where none had existed before. The large window size which we identified as optimal may even make the issue worse through incorporating distant edge regions into local statistics, for pixels near the document boundaries, making our results worse.

3.1.b.4) Conclusion

From our comparative analysis between Otsu method and Niblack's algorithm, it highlights a rather important principle, where no single thresholding algorithm would perform optimally across all image conditions. Otsu's method excels in well-illuminated, high contrast images with clear bimodal intensity distributions, while Niblack's adaptive approach demonstrates and performs better on 'degraded' documents with spatially varying illumination, low contrast or complex backgrounds. As such, the optimal algorithm choice ultimately depends on the image characteristics of the test image.

3.1.c) Further Improvements of Results

While our exhaustive grid search in the previous section successfully identifies 'optimal' hyperparameters for Niblack's algorithm, the approach suffers the most from several significant limitations that motivate better optimization strategies.

First, the computational cost of the previous method (exhaustive search with grid space) scales linearly with the number of parameter combinations evaluated. Rather, our chosen parameters tests 3660 total configurations per image, requiring substantial processing time that may be hard or unrealistic in real-world applications. For instance, reducing k step size and expanding the window size could double, if not quadruple the computational burden.

Secondly, grid search may require extensive domain knowledge to specify appropriate search ranges for each image. Our choice of k and windowSize space are fine-tuned based on what we thought was a good range and was tuned based on careful consideration of typical parameter ranges reported. However, the optimal parameters (particularly the large window size) suggest that we may have not fully explored the optimal parameter space. It poses several questions: Would a large window size help for these images? Does this apply to all images?

Third, grid search evaluates our parameters independently at discrete intervals. We might potentially miss optimal configurations that lie between these sampled points. This is limiting as the true optimal window or k might fall between our tested values, or perhaps the global optimal is out of our search space. There is no learning or adaptation as it progresses, as each parameter combination is evaluated independently, making it inefficient and ineffective.

3.1.c.1) Utilising Bayesian Optimization

Bayesian optimization addresses each of the mentioned limitations in the previous section. It dramatically reduces computational cost by learning from previous evaluations to intelligently select which parameter combinations to test next, typically requiring only 50-100 evaluations instead of thousands. It also eliminates the need for extensive domain knowledge about appropriate search ranges, possibly allowing us to define a broad and uninformative prior range (e.g. $k \in [-5.0,5.0]$ and $windowSize \in [1,500]$ and letting the algorithm automatically identify promising regions through adaptive exploration. More importantly, it treats parameters as continuous variables, allowing evaluation of any real-valued k or any odd integer window size within the bounds, which eliminates the artificial discretization imposed by grid search and this likewise increases the likelihood of finding true optimal configurations.

3.1.c.2) Hyperparameter Setup

In this setup, we now set $k \in [-2.0,1.0]$ and $windowSize \in [1,201]$, which is the same range as the experiment done in the previous experiments. This is done to ensure fair comparison as we are using the same range. However, in this experiment, using Niblack's with Bayesian optimization would allow us to explore continuous values along this range, in hopes to find a better optimum and overall reduction in diffSum as compared to discrete intervals in the previous method. We also set a modest exploration rate of 0.3 for the optimizer to find a good optimum without too much exploration.

```
% Optimization parameters
PARAM_K_BOUNDS = [-2.0, 1.0];      % k parameter range
PARAM_WIN_BOUNDS = [1, 201];      % Window size range (pixels)
MAX_EVALUATIONS = 50;              % Number of Bayesian iterations
EXPLORATION_RATE = 0.3;            % Exploration vs exploitation ratio
```

Figure 14: Hyperparameter Setup for Niblack with Bayesian Optimization

3.1.c.3) Implementation Details

```
%% 3. Parameter Optimization
fprintf('\n3. Running parameter optimization...\n\n');

for docID = 1:numDocs
    fprintf('Processing %s...\n', documents{docID});

    currentImage = imgData{docID};
    currentTruth = gtData{docID};
    [rows, cols] = size(currentImage);

    % Define objective function (minimize pixel errors)
    objFunc = @(p) computeSegmentationCost(currentImage, currentTruth, p.k, p.window);

    % Setup optimization variables
    varK = optimizableVariable('k', PARAM_K_BOUNDS);
    varWindow = optimizableVariable('window', PARAM_WIN_BOUNDS, 'Type', 'integer');

    % Run Bayesian optimization
    tStart = tic;

    optResult = bayesopt(objFunc, [varK, varWindow], ...
        'MaxObjectiveEvaluations', MAX_EVALUATIONS, ...
        'AcquisitionFunctionName', 'expected-improvement-plus', ...
        'ExplorationRatio', EXPLORATION_RATE, ...
        'IsObjectiveDeterministic', true, ...
        'Verbose', 1, ...
        'PlotFcn', {@plotObjectiveModel, @plotMinObjective});

    timeElapsed = toc(tStart);

    % Extract optimized parameters
    optK = optResult.XAtMinObjective.k;
    optWin = optResult.XAtMinObjective.window;

    % Ensure odd window size
    if mod(optWin, 2) == 0
        optWin = optWin + 1;
    end

    % Apply Niblack with optimal parameters
    finalBinary = performNiblackSegmentation(currentImage, optK, optWin);
```

Figure 15: Niblack with Bayesian Optimization Implementation

In Figure 15, the code for the optimization process of Niblack's thresholding with Bayesian Optimization is shown. First, we define a objFunc (objectiveFunction), which is a function that checks the results of each **k** and **windowSize** through comparing the thresholded image result to the ground truth. We aim to minimize this objective function. The function is shown in the Figure below. We normalize the windowSize so that it is always an odd number before performing any calculations. In this case, we are just performing Niblack's algorithm given some k and windowSize and returning the cost. This calculation is essentially the diffSum calculation.

```
function cost = computeSegmentationCost(image, groundTruth, k, windowSize)
    % Objective function: returns pixel mismatch count
    if mod(windowSize, 2) == 0
        windowSize = windowSize + 1;
    end

    binaryResult = performNiblackSegmentation(image, k, windowSize);
    differenceMap = xor(binaryResult, groundTruth);
    cost = sum(differenceMap(:));
end
```

Figure 16: Segmentation Cost Objective Function

Then, we use the Bayesian optimization (**bayesopt**), which aims to find the optimal values of **k** and **windowSize** that returns the smallest difference sum through our objective function. In our case, we run the optimization only **50** times, a set number that seems fair and performs shorter in time respective to the experiment done in 3.1.b without Bayesian optimization to ensure fair comparison. The acquisition function is set to the recommended 'expected-improvement-plus' to balance between trying new values or optimizing currently found good values. Lastly, the exploration rate as mentioned above, is set to 0.3 to complement the acquisition function, exploring only when our acquired result is poor.

Finally, after the algorithm has finished, the 'best' values of **k** and **windowSize** are stored as the **optK** and **optwindowSize**, and a final pass of Niblack is run to acquire the finalBinary image supposedly with the optimal values, the one that returns the lowest diffSum.

3.1.c.4) Results

As part of the Bayesian optimization process, a plot function can be input into the function to visualize how different values of **k** and **windowSize** affects the segmentation accuracy of the image across the search space. Figure 17 shows four plots, which visualize the objective function model (in this case the Gaussian Process surrogate) that Bayesian optimization constructs during the parameter search for each document.

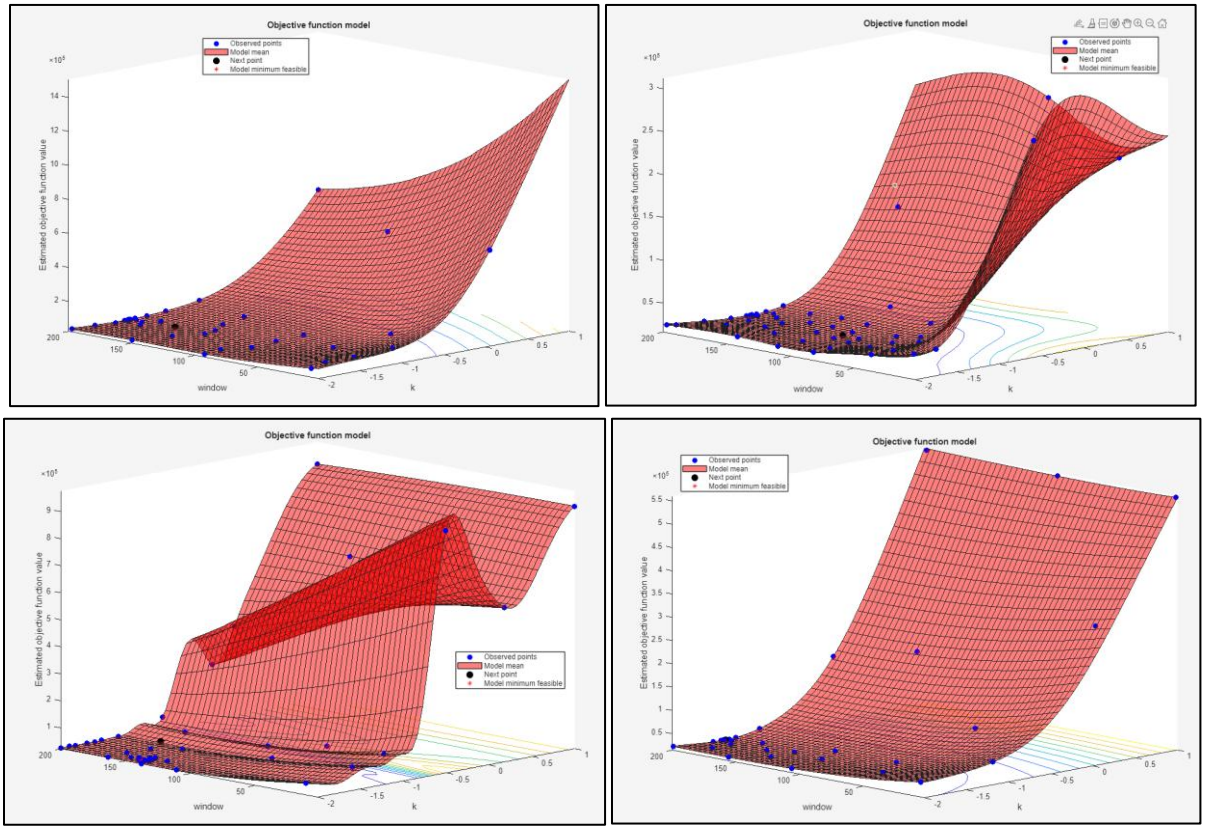


Figure 17: 3D Visualization of Niblack's thresholding parameters with Bayesian optimization

The following information is described in each axes:

1. **X-axis(k):** The Niblack k parameter, which ranges from [-2.0, 1.0]
2. **Y-axis (windowSize):** Local neighborhood window size, ranging from [1, 201] pixels
3. **Z-axis:** Estimated pixel error count (the objective function value)

Each document has different surface topologies, which can be seen by some documents having peaks while others having gentle gradients. While this graphical result can differ between experiments, these visualizations capture how the algorithm explores different k and windowSize combinations while evaluating the objectiveFunction, eventually slowly deciding on the parameters that result in the best text segmentation. The fact that different images have different topologies also highlights that different images have different optimal parameters, depending on the image's characteristics such as illumination, contrast, etc.

Image	Best K	Best Window Size	Diff Sum	Accuracy (%)	Error Rate (%)
document01.bmp	-1.28	201	24124	96.57	3.43
document02.bmp	-0.94	201	15782	94.92	5.08
document03.bmp	-1.87	145	22612	97.68	2.32
document04.bmp	-1.38	201	14274	97.75	2.25

Table 6: Summary of optimal parameters for Niblack's thresholding across all input images with Bayesian optimization

Table 6 presents the optimal parameter configurations obtained through Bayesian optimization for Niblack's adaptive thresholding algorithm across all the test documents. We observe that the Bayesian optimization approach acquires better text segmentation results, while only requiring about 40% of the computational time (on my testbench) as compared to exhaustive grid search. Specifically, the optimized parameters yielded reduced pixel error counts, improved accuracy rates ranging from 94.92% to 97.75% and correspondingly lower error rates between 2.25% and 5.08%.

Notably, the optimal parameters values identified through Bayesian optimization is similar to those discovered via the grid search, suggesting that Bayesian optimization allows finding of optimal parameters even with reduced computation time, confirming that Bayesian optimization was able to successfully navigate the parameter space and identify the global minima despite evaluating significantly fewer parameter combinations.

In conclusion, the results demonstrate that Bayesian optimization provides a viable and efficient alternative to traditional grid search for parameter tuning, particularly in adaptive thresholding applications like this experiment, proving better parameter search when computational resources may be limited or when processing large document vocabularies/database where per-image optimization is required. While using this method may not have reduced the diffSum significantly

like other thresholding methods might achieve, we demonstrate that optimization of parameters is likewise just as important as achieving great results with slower algorithms.

3.1.c.5) Supplementary

In this additional experiment, we run the same experiment of Niblack using Bayesian optimization, but this time using the same **k** but increase the **windowSize** range to further test how windowSize affects the results. We use $k \in [-2.0, 1.0]$, $windowSize \in [1, 501]$ and evaluations = 50 for this experiment. The following table summarizes the results:

Image	Best K	Best Window Size	Diff Sum	Accuracy (%)	Error Rate (%)
document01.bmp	-1.32	439	19201	97.09	2.91
document02.bmp	-0.82	497	10070	96.81	3.19
document03.bmp	-1.37	501	15399	98.39	1.61
document04.bmp	-1.47	385	13463	97.88	2.12

Table 7: Summary of optimal parameters for Niblack's thresholding across all input images with Bayesian optimization and increased window size range

The result in Table 7 demonstrates that the initial window size constraint artificially limited optimal performance. By increasing the window size and expanding the search space, the Bayesian optimizer consistently identifies larger optimal window sizes (385-501 pixels) for all documents, which manages to yield improved text segmentation accuracy. Comparing this result to Table 6, accuracy increased across all images, with document02.bmp showing the most substantial improvement (+1.89%). Similarly, pixel error counts (diff sum) decreased for all documents as well.

These findings and analysis reveal that larger local neighbourhoods are beneficial for the Niblack algorithm **on these particular document images**, likely because it is able to capture better global intensity trends while remaining adaptive to local variations. The consistent convergence to window sizes near the upper limit of the expanded range seems to suggest that even larger windows may warrant investigation or testing. Similar to our previous findings, this experiments shows and highlights the importance of appropriate and well selected parameter selection in optimization tasks, as small changes in hyperparameters may change the result significantly. Seemingly reasonable constraints can also likewise prevent discovery of superior solutions.

3.2 Stereo Vision

In this section, we write codes to compute disparity maps for pairs of rectified stereo images P_l and P_r., where the disparity map is inversely proportional to the depth map which gives the distance of points in the scene from the camera.

3.2.a) Algorithm

This section shows our setup and implementation of the computeDisparityMapSSD function to calculate disparity maps.

3.2.a.1) Parameter Setup

```
7   %% Configuration Parameters
8   TEMPLATE_HEIGHT = 11;
9   TEMPLATE_WIDTH = 11;
10  MAX_DISP = 15;  % Maximum disparity search range
```

Figure 18: Parameter Setup according to question specifications

Figure 18 shows our parameter setup according to the question specifications. We define important parameters such as **maxDisparity** = 15, which limits the search range of matching pixels between both images (left and right), as well as a **templateSize** that sets the dimensions of the template window for matching along the axis.

3.2.a.2) Implementation

The following Figure shows the implementation of our computeDisparityMapSSD function. It takes in the following parameters:

1. leftImg: Left (reference) image, grayscale
2. rightImg: Right image, grayscale
3. templateHeight: Template height (should be odd number)
4. templateWidth: Template width (should be odd number)
5. maxDisparity: Maximum disparity to search (15 as per specifications)

And then outputs the respective disparity map.

```
%% Part (a): Function to Compute Disparity Map using SSD
function dispMap = computeDisparityMapSSD(leftImg, rightImg, templateHeight, templateWidth, maxDisparity)
    % Convert to double for precision
    leftImg = double(leftImg);
    rightImg = double(rightImg);

    [rows, cols] = size(leftImg);

    % Initialize disparity map and cost volume
    dispMap = zeros(rows, cols);
    costVolume = inf(rows, cols, 2*maxDisparity + 1);

    % Create template matching kernel (ones for SSD computation)
    templateKernel = ones(templateHeight, templateWidth);

    fprintf('Computing disparity map');

    % Loop only over disparity values
    disparityIdx = 1;
    for d = -maxDisparity : maxDisparity
        % Shift right image by disparity d
        if d < 0
            % Negative disparity: shift right
            shiftedRight = [rightImg(:, -d+1:end), zeros(rows, -d)];
        elseif d > 0
            % Positive disparity: shift left
            shiftedRight = [zeros(rows, d), rightImg(:, 1:end-d)];
        else
            % Zero disparity: no shift
            shiftedRight = rightImg;
        end

        % Compute squared differences using vectorization
        sqDiff = (leftImg - shiftedRight) .^ 2;

        % Use conv2 to sum over template windows
        ssdMap = conv2(sqDiff, templateKernel, 'same');

        % Store in cost volume
        costVolume(:, :, disparityIdx) = ssdMap;
        disparityIdx = disparityIdx + 1;
    end

    fprintf('Finding minimum cost disparities...\n');

    % Find disparity with minimum cost at each pixel
    [~, minIdx] = min(costVolume, [], 3);

    % Convert indices to actual disparity values
    dispMap = minIdx - (maxDisparity + 1);

    % Handle border regions by setting to zero
    halfH = floor(templateHeight / 2);
    halfW = floor(templateWidth / 2);
    dispMap(1:halfH, :) = 0;
    dispMap(end-halfH+1:end, :) = 0;
    dispMap(:, 1:halfW) = 0;
    dispMap(:, end-halfW+1:end) = 0;
end
```

Figure 19: computeDisparityMapSSD Function

3.2.a.3) Explanation

The function begins by converting both input images to double precision for numerical accuracy during computation. This is standard practice due to higher number of floating points The image dimensions such as **[rows,cols]** are extracted to allow subsequent operations and scalability for images of varying sizes. To prevent unnecessary creation of for loop compared to traditional approaches, our implementation initializes two key data structures:

1. a **dispMap** matrix with the same dim as the input images and initialized to zeros
2. a **costVolume** array initialized with infinity values with having dimensions **[rows, cols, 2*maxDisparity+1]**

In our use case, the costVolume serves as storage for all possible disparity costs at each pixel location. To further optimize our computation, we introduce **templateKernel**, a matrix of ones with dimensions [templateheight, templateWidth]. This kernel serves as a convolution filter that

will aggregate squared differences over the template window, effectively replacing the innermost nested loops found in traditional implementations.

Rather than iterating over each pixel and then searching for matches (which requires nested for loops over rows, columns and disparities), we instead loop over the disparity range (31 iterations for maximum disparity of 15). For each disparity value ***d***, the entire right image is shifted horizontally to simulate the displacement between stereo image pairs. When the disparity becomes negative, the right image is shifted to the right through zero padding on the right side and removing the columns on the left. Conversely, when disparity is positive the image shifts left with zero-padding on the left. This is made efficient through MATLAB's array slicing, processing all pixels simultaneously rather than individually. This idea was taken from the lecture notes, where it is the equivalent of moving the image right or left.

Once the shifted image is created, the function computes squared differences between the left and the shifted right image using vectorizations **sqDiff = (leftImg – shiftedRight) .^ 2**, performing element-wise subtraction and squaring across the entire matrix simultaneously, removing the pixel-by-pixel loops. After processing all the disparity values, we can then use [~, minIdx,] = min(costVolume,[],3) to find the disparity with the minimum cost at each pixel. We then convert these disparity values and store them into the dispMap. Finally, border regions where the template would extend beyond image boundaries are set to zero using the half dimensions computed.

In conclusion, our approach just uses one loop and vectorized operations to speed up the overall processing. This leverages MATLAB's strengths in matrix operations and optimized built in functions rather than primitive and easily readable for loops, making the stereo vision computation significantly more efficient than explicit loop-based implementations.

3.2.b) Convert both image to greyscale

This section shows the simple code to convert both corridor images to grayscale, similar to what we have done before in previous sections.

```
% Load corridor images
imgLeftPath = fullfile(ASSET_PATH, 'corridorl.jpg');
imgRightPath = fullfile(ASSET_PATH, 'corridorr.jpg');

imgLeft = imread(imgLeftPath);
imgRight = imread(imgRightPath);

% Convert to grayscale if needed
if ndims(imgLeft) == 3
    imgLeftGray = rgb2gray(imgLeft);
else
    imgLeftGray = imgLeft;
end

if ndims(imgRight) == 3
    imgRightGray = rgb2gray(imgRight);
else
    imgRightGray = imgRight;
end
```

Figure 20: Converting corridor images to grayscale.

3.2.c) Obtaining disparity map D

In this section, we use the aforementioned function in part a to compute the disparity map based on the corridor images. We apply our function to the grayscale images that were acquired in part b with specified parameter values **maxDisparity**, **templateHeight**, **templateWidth.**

```
tic;
disparityMapSynthetic = computeDisparityMapSSD(imgLeftGray, imgRightGray, ...
                                               TEMPLATE_HEIGHT, TEMPLATE_WIDTH, ...
                                               MAX_DISP);
elapsedTime = toc;

fprintf('Disparity map computed in %.2f seconds.\n', elapsedTime);
fprintf('Disparity range: [%.2f, %.2f]\n', min(disparityMapSynthetic(:)), max(disparityMapSynthetic(:)));

% Display results
figure('Name', 'Synthetic Stereo - Corridor', 'NumberTitle', 'off', 'Position', [100, 100, 1600, 400]);
subplot(1, 4, 1);
imshow(imgLeftGray);
title('Left Image (Reference)');
```

Figure 21: Computing disparity map and plotting visualization code

```
Loading synthetic stereo pair...
Synthetic images loaded successfully.
Image dimensions: 256 x 256

Computing disparity map for synthetic stereo pair...
  Computing disparity map...
  Finding minimum cost disparities...
Disparity map computed in 0.02 seconds.
Disparity range: [-15.00, 15.00]
```

Figure 22: Output of disparity map

From the printing statements, we can see that the computation of our disparity map was very fast, only in 0.02 seconds, and the disparity range matches the parameters we set.

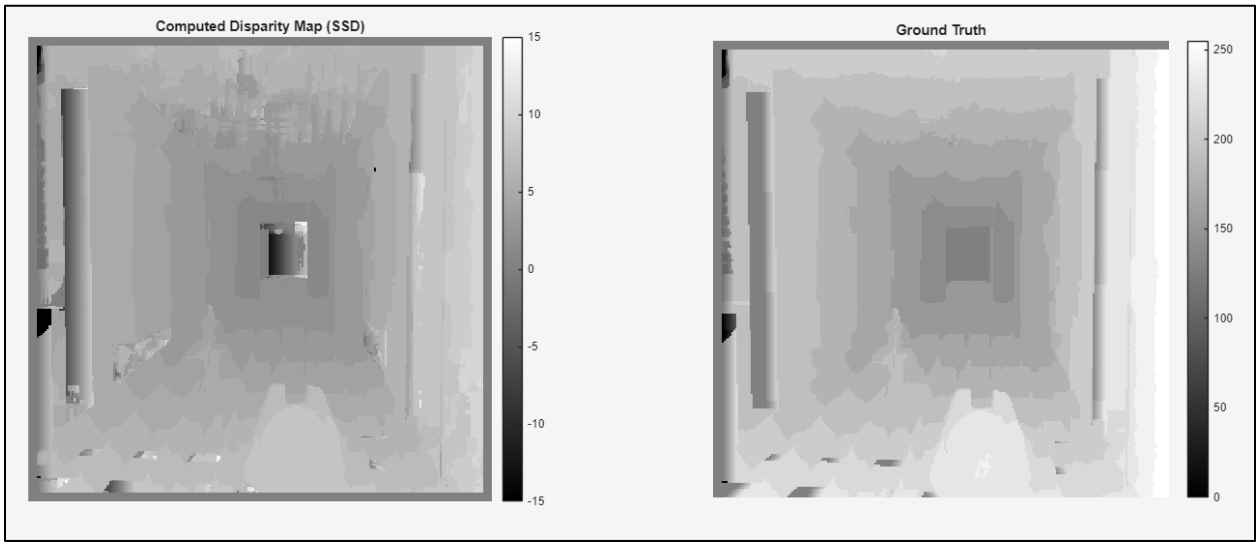


Figure 23: Disparity Map Visualization (Computed vs Ground Truth)

Figure 23 visualizes our computed disparity map (left) generated using Sum of Squared Differences (SSD) alongside the ground truth (right) for comparison. Our computed disparity map successfully captures the overall depth structure of the corridor scene, where nearer objects

appear brighter and farther objects appear darker as expected in the question brief. Our SSD-based implementation is effective in identifying the major depth features, with the central object and corridor being distinguishable.

We can observe some relationships between the computed disparities and the corresponding local image structure. The quality of computed disparities varies significantly with the local image structure. In regions with strong texture and distinctive features (corridor walls with surface details, edges of objects, and areas with visible patterns), the algorithm is able to produce accurate disparity estimates. These textured regions seem to provide unique intensity patterns that allow our SSD block matching to reliably discriminate between correct and incorrect matches between both images, resulting in good results that match the ground truth.

Conversely, in uniform or weakly textured regions such as smooth floor surfaces and the wall sections, the disparity quality seems to degrade noticeably, subsequently exhibiting increased noise and scattered values. For example, in the original synthetic image, the disparity located at the end of the corridor is not able to model the accurate disparity (showing some parts as white and indicates it thinks that the surface is close to the camera). Without distinctive local features to guide the matching process, multiple disparities may yield similar SSD costs, where the computed map shows rougher and less coherent depth estimates as compared to the ground truth's smooth gradient. Additionally, at depth discontinuities and object boundaries, the fixed template window may capture pixels from different depth planes, which produces **less sharp transitions** compared to the ground truth.

3.2.d) Rerunning of algorithm on triclops

In this section, we load the triclops image using the similar approach as all sections. They are provided here just for completeness.

```
%% Part (d): Compute Disparity Map for Real Stereo Images
fprintf('\n--- Processing Real Stereo Images ---\n');

% Load triclops images
imgLeftRealPath = fullfile(ASSET_PATH, 'triclopsi2l.jpg');
imgRightRealPath = fullfile(ASSET_PATH, 'triclopsi2r.jpg');
groundTruthRealPath = fullfile(ASSET_PATH, 'triclopsid.jpg');

imgLeftReal = imread(imgLeftRealPath);
imgRightReal = imread(imgRightRealPath);
groundTruthReal = imread(groundTruthRealPath);

% Convert to grayscale if needed
if ndims(imgLeftReal) == 3
    imgLeftRealGray = rgb2gray(imgLeftReal);
else
    imgLeftRealGray = imgLeftReal;
end

if ndims(imgRightReal) == 3
    imgRightRealGray = rgb2gray(imgRightReal);
else
    imgRightRealGray = imgRightReal;
end

if ndims(groundTruthReal) == 3
    groundTruthReal = rgb2gray(groundTruthReal);
end

% Convert ground truth to double for proper display
groundTruthReal = double(groundTruthReal);

fprintf('Real stereo images loaded successfully.\n');
fprintf('Image dimensions: %d x %d\n', size(imgLeftRealGray, 1), size(imgLeftRealGray, 2));

% Compute disparity map
fprintf('Computing disparity map for real stereo pair...\n');
tic;
disparityMapReal = computeDisparityMapSSD(imgLeftRealGray, imgRightRealGray, ...
                                          TEMPLATE_HEIGHT, TEMPLATE_WIDTH, ...
                                          MAX_DISP);
elapsedTimeReal = toc;
```

Figure 24: Loading of triclops image and converting to grayscale

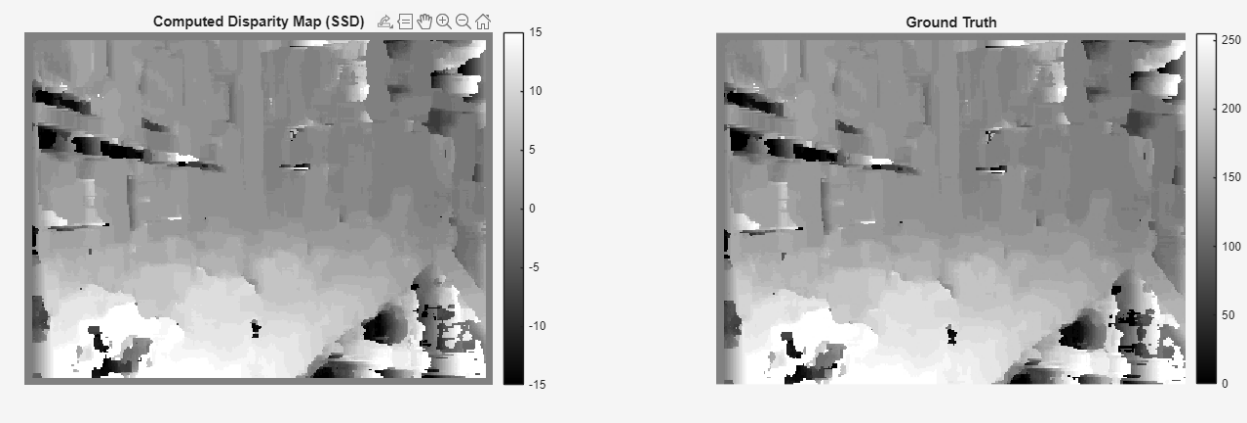


Figure 25: Disparity Map Visualization (Computed vs Ground Truth) - Triclops

As seen in Figure 25, it demonstrates strong disparity estimation in textured regions (vegetation, foliage) where the computed map closely resembles the ground truth. However, just like in previous sections, it struggles in uniform areas such as the pavement, sky and building facades. This performance illustrates the dependence of block-matching algorithms on local texture. The vegetation in the scene provides distinctive intensity patterns within each 11x11 template, which manages to produce clear SSD minima at correct match positions. In contrast, smooth and homogenous surfaces yield ambiguous matching, the template extracted from smooth pavement may look nearly identical at different scanline positions, causing the algorithm to produce noisy, unreliable disparity values where it would not have a confident match.

These results are similar to our corridor findings, where we observe the same issues and limitations in the real-world context with natural lighting and genuinely ambiguous surfaces. The success with the matching of vegetation in the scene despite the outdoor complexity confirms that texture richness determines stereo matching quality and not the scene difficulty, as can be inferred from the slides. This explains why modern stereo systems incorporate additional constraints beyond pure SSD, such as smoothness assumptions, learned priors, etc, to handle uniform regions where classical block matching fundamentally struggles due to insufficient discriminative information.