**The National University of Computer and Emerging Sciences**

**Islamabad Campus**



# **Assignment 1**

# **Digital Image Processing**

**Taha Rasheed 22i-2009 / Hamza Asad 22i-1908 / Shafay Siddiq 22i-2007**

**DIP DS B**

**Question 1**

The Niblack thresholding method is a local adaptive thresholding algorithm that calculates thresholds dynamically based on local mean and standard deviation.

## **Main Features of the Application**

1. **Grayscale Image Processing**.
2. **Adaptive Local Thresholding**: Unlike global thresholding, the Niblack method applies different thresholds to different image regions based on local statistics.
3. **Adjustable Parameters**: Users can modify the window size and the k parameter to control the segmentation effect.
4. **Multiple Configurations Comparison**: The script evaluates different window sizes to observe their impact on the thresholded output.
5. **Visualization of Results**: The processed images are displayed for comparison to determine the best settings.

## **Implementation Details**

1. Load an image in grayscale mode.
2. Convert the image to a floating-point format for numerical stability.
3. Compute the local mean and standard deviation within a given window size.
4. Determine the threshold using Niblack’s formula: where is the mean, is the standard deviation, and is a user-defined parameter.
5. Generate a binary image based on the computed threshold.
6. Process the image for different window sizes and display results for comparison.

## **Results and Comparisons**

The algorithm was tested on an input image (selfie.jpg) using four different window sizes: **15, 25, 35, and 45**. The results of each are presented below:

* **Window Size = 15**: Small window captures finer details but may introduce noise in regions with small variations.
* **Window Size = 25**: Balances fine details and noise suppression, providing a clearer thresholded image.
* **Window Size = 35**: Captures larger structural variations, reducing noise but blurring finer details.
* **Window Size = 45**: Over-smooth small details, making edges less distinct.

**Best Window Size Selection** Based on visual evaluation, a window size of **25** provided the best trade-off between detail preservation and noise reduction.

## **Observations**

1. **Impact of Window Size**: Smaller window sizes highlight details but may introduce noise, whereas larger windows create smoother transitions but may blur important edges.
2. **Effect of k Parameter**: The parameter k influences the threshold sensitivity. A lower k (e.g., -0.2) provides moderate background suppression, while a more negative value (e.g., -0.5) results in aggressive binarization.
3. **Application Suitability**: Niblack thresholding is well-suited for images with varying lighting conditions but may struggle in high-noise scenarios without preprocessing steps like smoothing.

**Question 2**

Sauvola methods dynamically calculate thresholds based on local mean and standard deviation, making them well-suited for images with varying illumination conditions.

## **Main Features of the Application**

1. **Grayscale Image Processing**: The algorithms process grayscale images to generate binary outputs.
2. **Adaptive Local Thresholding**: Unlike global thresholding, these methods apply different thresholds to different image regions based on local statistics.
3. **Adjustable Parameters**: Users can modify the window size and key parameters (k for Niblack, k and R for Sauvola) to control the segmentation effect.
4. **Multiple Configurations Comparison**: The script evaluates different window sizes to observe their impact on the thresholded output.
5. **Visualization of Results**: The processed images are displayed for comparison to determine the best settings.

## **Implementation Details**

1. Load an image in grayscale mode.
2. Convert the image to a floating-point format for numerical stability.
3. Compute the local mean and standard deviation within a given window size.
4. Determine the threshold using the respective formula:

**Sauvola Formula:** where represents the dynamic range of standard deviation, and controls sensitivity.

1. Generate a binary image based on the computed threshold.
2. Process the image for different window sizes and display results for comparison.

## **Results and Comparisons**

The algorithms were tested on an input image (selfie.jpg) using four different window sizes: **15, 25, 35, and 45**. The results of each are presented below:

### **Sauvola Thresholding Results:**

* **Window Size = 15**: Generates sharper edges with better text preservation.
* **Window Size = 25**: Maintains details while reducing background noise.
* **Window Size = 35**: Smoothens the image while preserving important features.
* **Window Size = 45**: Reduces noise but may blur finer elements.

**Best Window Size for Sauvola:** **25**

**Question 3**

**1. Main Features**

* **Image Preprocessing:** Converts input images to grayscale, resizes them to 28x28 pixels, applies Gaussian blur, and performs adaptive thresholding.
* **Template Loading:** Loads preprocessed digit templates (0-9) for comparison.
* **ZNCC Template Matching:** Compares the test image with stored templates and selects the best match based on correlation score.
* **KNN Classification:** Uses Euclidean distance to determine the closest matching digit from the templates.
* **Dual-Method Prediction:** The system provides predictions from both ZNCC and KNN classification for comparison.

**2. Clear Outputs**

For a test image (e.g., test/3.11.png), the system outputs:

Template Matching Prediction: 3

KNN Classification Prediction: 3

Both methods correctly classify the digit as **'3'**.

**3. Comparison of Methods**

| **Method** | **Approach** | **Pros** | **Cons** |
| --- | --- | --- | --- |
| ZNCC Template Matching | Measures correlation similarity | Works well with structured templates | Sensitive to distortions |
| KNN Classification | Uses Euclidean distance | More flexible with variations | Requires more computation |

**4. Observations**

* Both **ZNCC Template Matching** and **KNN Classification** correctly classify digits in the test set.
* ZNCC is effective when the test image closely matches the stored templates, but may struggle with variations.
* KNN is more robust against variations but requires additional computations.
* The combination of both methods ensures a reliable prediction system.

**Question 4**

The project is organized into several key functions:

* **preprocess\_card\_image(image)**  
  Converts the image to grayscale, applies a Gaussian blur, and uses Otsu thresholding to create a binary image that highlights the digits.
* **refine\_roi(roi)**  
  Applies morphological operations (closing followed by opening) to the region of interest (ROI) to better separate the digits.
* **find\_digit\_contours(roi\_thresh)**  
  Detects contours in the thresholded ROI and sorts them from left to right. This function is critical for identifying the potential digit areas.
* **create\_digit\_templates()**  
  Generates templates for digits 0–9 using OpenCV’s text rendering. The templates are normalized and centered for better matching accuracy.
* **match\_digit(roi, templates)**  
  Uses template matching (with normalized cross-correlation) to compare a candidate digit ROI against each template and determine the best match.
* **process\_contour(contour, roi\_thresh, templates)**  
  Processes each detected contour by resizing it to the template size and matching it with the available digit templates. The function also includes a check to filter out noise based on the area of the contour.
* **recognize\_credit\_card\_number(image\_path)**  
  Orchestrates the overall workflow: it loads the image, extracts the ROI where the card number is expected, applies preprocessing and contour detection, processes each contour, and finally assembles the recognized digits into the complete card number.

**Initial Issue: Incomplete Recognition**

During early tests, the code output only “703” instead of the full credit card number. The debugging process revealed several key issues:

1. **Wide Contour Splitting:**  
   The initial implementation attempted to split wide contours, which was not always effective. Many digits were merged into a single contour, and splitting them arbitrarily resulted in multiple false matches. We decided to remove the wide-contour splitting branch entirely. This allowed each contour to be processed as a single digit.
2. **Template Accuracy:**  
   The digit templates created using the default settings did not match well with the actual card digits. We improved the template creation process by centering the text and using a larger font (with a scale of 2.0 and thickness of 3). This adjustment resulted in templates that more closely resembled the digits on the credit card image.
3. **Threshold Tuning:**  
   Both the minimum contour area and the match acceptance threshold were too strict in the original code. We lowered the minimum area threshold and the acceptance score threshold to ensure that valid contours were not inadvertently discarded. The acceptance threshold was decreased to 0.05, allowing more contours to be accepted based on their matching score.
4. **Morphological Operations:**  
   The parameters for the morphological operations were tweaked slightly. By increasing the kernel sizes in the closing and opening operations, we improved the separation of touching digits, resulting in more accurate contour detection.

**Verification and Results**

After making the necessary changes, repeated tests with the provided credit card image showed that the full number (e.g., “512345575901”) was recognized correctly. Debug print statements helped verify that each digit was matched with a reasonable normalized score and in the correct order.

**Conclusion**

The debugging process involved several iterations where we adjusted the template creation, threshold parameters, and morphological operations. By removing the problematic wide-contour splitting logic and fine-tuning the template matching parameters, the code now robustly recognizes the full credit card number from the image.