Student: Aditya Sandhu

University: Colorado State University Global

Course: 25FC - CSC515 - 1 [Module 6 – Segmentation]

Critical Thinking Assignment [OpenCV Application of Adaptive Thresholding Scheme Accounting for Changes in Illumination]

Instructor: Dr Dong Nguyen

Date – 10/16/2025

GIT LINKS

Document Link -

Python File –

Image segmentation is a fundamental process in computer vision and digital image analysis that involves dividing an image into distinct regions or segments that share similar characteristics such as intensity, color, texture, or spatial relationship. The primary objective of segmentation is to simplify or transform an image into a representation that is more meaningful and easier to analyze for subsequent tasks like object recognition, measurement, or classification.

In practical terms, segmentation distinguishes objects of interest (foreground) from their background, enabling algorithms to identify boundaries, shapes, and structural relationships within a scene. For example, in medical imaging, segmentation isolates tissues or organs; in autonomous vehicles, it separates road surfaces from pedestrians and obstacles; and in this experiment, it differentiates light and dark objects under variable illumination.

Segmentation techniques can be broadly categorized into threshold-based, edge-based, and region-based methods. Among these, thresholding is one of the simplest and most widely used approaches—it classifies pixels according to their intensity values. However, global thresholding often fails under uneven lighting conditions. To overcome this limitation, adaptive thresholding dynamically computes local thresholds across different regions of an image, achieving robust segmentation even when illumination varies.

Figure 1 - *Adaptive Thresholding Program Structure in Python.*

A screenshot of a computer program

AI-generated content may be incorrect.

cv2 as vision\_lib that imports OpenCV, which provides functions for reading images, processing them, and performing transformations such as thresholding and filtering. matplotlib.pyplot as plot\_lib – imports Matplotlib’s plotting library to visualize and save images or intermediate processing stages. os and sys – enable file path management and allow the program to handle system-level operations like exiting safely when errors occur.

*# Importing required modules for handling images and visualizations  
import* cv2 *as* vision\_lib *# Handles image reading and transformations  
import* matplotlib.pyplot *as* plot\_lib *# Enables creating and saving plots  
import* os *# Assists with file path management  
import* sys *# Allows for program exit on errors*

The function load\_source\_image(file\_name) constructs a full path to the target image based on the script’s directory to prevent file location errors. It then uses OpenCV’s imread() function to load the image into memory for processing. If the image cannot be found or opened, the function prints a clear error message and safely terminates the program using sys.exit(1) to avoid further execution with invalid data. This ensures reliability and portability by allowing the program to locate and load source images correctly on any system.

*# Function to load an image using a full path to avoid directory issues  
def* load\_source\_image(file\_name):  
 *# Build the complete path relative to this script's location* full\_path = os.path.join(os.path.dirname(\_\_file\_\_), file\_name)  
 loaded\_img = vision\_lib.imread(full\_path)  
 *# Check if loading failed and exit with a message if so  
 if* loaded\_img *is None*:  
 print(f"Error: Could not load the image at '{full\_path}'. "  
 f"Verify the file exists and path is correct.")  
 sys.exit(1) *# Terminate the script on failure  
 return* loaded\_img

The function convert\_to\_monochrome(source\_img) converts a color image into a single-channel grayscale image using OpenCV’s cvtColor() method. This transformation simplifies the data by reducing three color channels (blue, green, and red) into one intensity channel, making subsequent image analysis more efficient. Grayscale conversion is an essential preprocessing step in segmentation tasks because it focuses on brightness variations, which are critical for thresholding and other intensity-based operations.

*# Function to convert image to single-channel format for easier handling  
def* convert\_to\_monochrome(source\_img):  
 *# Reduce color channels to one for simplified analysis  
 return* vision\_lib.cvtColor(source\_img, vision\_lib.COLOR\_BGR2GRAY)

The function apply\_noise\_reduction(mono\_img) applies a Gaussian blur filter to the grayscale image to minimize noise and small artifacts that may interfere with segmentation. By using a 7×7 kernel, the filter smooths intensity variations across neighboring pixels, producing a more uniform image. This preprocessing step enhances the reliability of the adaptive thresholding process by ensuring that noise or fine texture details do not cause false object boundaries.

*# Function to reduce image artifacts with a filter  
def* apply\_noise\_reduction(mono\_img):  
 *# Use a larger kernel for broader smoothing effect  
 return* vision\_lib.GaussianBlur(mono\_img, (7, 7), 0)

The function apply\_local\_binarization(filtered\_img) performs adaptive thresholding to convert the smoothed grayscale image into a binary (black-and-white) format. Using OpenCV’s adaptiveThreshold() method, it calculates a local threshold for each small region (11×11 pixels) based on neighborhood intensity values, rather than relying on a single global threshold. The Gaussian-weighted mean is used to account for local illumination differences, and the constant value 2 is subtracted to fine-tune the segmentation boundary. This approach ensures that objects are accurately separated from the background even under uneven lighting conditions.

*# Function to binarize the image based on local variations  
def* apply\_local\_binarization(filtered\_img):  
 *# Adjust parameters for custom threshold behavior  
 return* vision\_lib.adaptiveThreshold(  
 filtered\_img, *# Filtered input* 255, *# Peak intensity value* vision\_lib.ADAPTIVE\_THRESH\_GAUSSIAN\_C, *# Weighted local method* vision\_lib.THRESH\_BINARY, *# Output as black/white* 11, *# Local area size* 2 *# Adjustment factor* )

Figure 2 - *Main Execution Workflow and Output Generation of the Adaptive Thresholding Program.* This figure presents the primary execution logic of the adaptive thresholding system, illustrating how two images—one with a dark object and one with a light object—are processed through sequential stages: loading, grayscale conversion, noise reduction, and adaptive binarization. The workflow generates visual summaries and final binary outputs for each scene, demonstrating consistent segmentation performance under varying illumination conditions.

A screenshot of a computer program

AI-generated content may be incorrect.

*The function generate\_and\_store\_visual\_summary(stages\_list, labels\_list, output\_file) creates a horizontally arranged visualization that displays each major image processing stage—original, grayscale, filtered, and binarized—side by side. The figure size is increased to provide more height for titles, and each subplot is labeled for clarity. Using Matplotlib, the function automatically adjusts layout spacing and saves the figure as a high-resolution image file. This enhancement improves readability and presentation quality, making it easier to analyze the progressive effects of adaptive thresholding across different processing stages.*

*# Function to generate and store a combined view of processing stages  
def generate\_and\_store\_visual\_summary(stages\_list, labels\_list, output\_file):  
 # Initialize a wide figure for horizontal arrangement with more height for titles  
 plot\_lib.figure(figsize=(18, 8))  
 for pos in range(len(stages\_list)):  
 plot\_lib.subplot(1, len(stages\_list), pos + 1)  
 if pos == 0:  
 # Adjust color order for accurate rendering  
 plot\_lib.imshow(vision\_lib.cvtColor(stages\_list[pos], vision\_lib.COLOR\_BGR2RGB))  
 else:  
 # Render in monochrome for processed stages  
 plot\_lib.imshow(stages\_list[pos], cmap='gray')  
 plot\_lib.title(labels\_list[pos], fontsize=10)  
 plot\_lib.axis('off')  
 # Optimize spacing and save with high resolution  
 plot\_lib.subplots\_adjust(top=0.95)  
 plot\_lib.tight\_layout()  
 plot\_lib.savefig(output\_file, dpi=400)  
 # No on-screen display to keep script lightweight  
 plot\_lib.close()*

The function store\_final\_binary(result\_img, output\_file) saves the final binarized image produced by the adaptive thresholding process to a specified file using OpenCV’s imwrite() function. This step preserves the segmentation output as a standalone image that can be used for documentation, further analysis, or comparison across different lighting conditions. By exporting the processed result, the program ensures that the key outcome of the adaptive thresholding scheme is permanently stored and easily accessible for evaluation.

*# Function to store the binarized result separately  
def* store\_final\_binary(result\_img, output\_file):  
 vision\_lib.imwrite(output\_file, result\_img)

The main execution block serves as the central workflow that coordinates all the defined functions to perform adaptive thresholding on two separate images—one featuring a dark object (device) and the other a light object (peach). For each image, the script sequentially loads the source file, converts it to grayscale, applies Gaussian noise reduction, and performs adaptive binarization. It then compiles all intermediate stages into a visual summary for documentation and saves both the composite and final binarized results as separate image files. This structured approach demonstrates the algorithm’s robustness under varying illumination conditions, highlighting how adaptive thresholding can effectively segment both dark and light objects in scenes with uneven lighting.

*# Main execution block  
if* \_\_name\_\_ == "\_\_main\_\_":  
 *# Process first image (dark object - device)* source\_file1 = 'device\_image.jpeg'  
 original1 = load\_source\_image(source\_file1)  
 mono1 = convert\_to\_monochrome(original1)  
 filtered1 = apply\_noise\_reduction(mono1)  
 binarized1 = apply\_local\_binarization(filtered1)  
 processing\_stages1 = [original1, mono1, filtered1, binarized1]  
 stage\_labels1 = ['Original Scene Capture (Device)', 'Monochrome Transformation', 'Filtered Image', 'Binarized Outcome']  
 generate\_and\_store\_visual\_summary(processing\_stages1, stage\_labels1, 'stages\_summary\_device.jpg')  
 store\_final\_binary(binarized1, 'final\_binary\_device.jpg')  
  
 *# Process second image (light object - peach)* source\_file2 = 'peach\_image.jpeg'  
 original2 = load\_source\_image(source\_file2)  
 mono2 = convert\_to\_monochrome(original2)  
 filtered2 = apply\_noise\_reduction(mono2)  
 binarized2 = apply\_local\_binarization(filtered2)  
 processing\_stages2 = [original2, mono2, filtered2, binarized2]  
 stage\_labels2 = ['Original Scene Capture (Peach)', 'Monochrome Transformation', 'Filtered Image', 'Binarized Outcome']  
 generate\_and\_store\_visual\_summary(processing\_stages2, stage\_labels2, 'stages\_summary\_peach.jpg')  
 store\_final\_binary(binarized2, 'final\_binary\_peach.jpg')  
  
 *# Custom note: This approach focuses on handling uneven lighting in scenes, ideal for variable conditions.  
 # Ensure vision\_lib (OpenCV) and plot\_lib (Matplotlib) are available in your setup.  
 # No external dependencies beyond these; runs self-contained.*

Input images Figure 3 and Figure 4 – peach and device respectively

A close up of a fruit

AI-generated content may be incorrect. A black rectangular object on a white surface

AI-generated content may be incorrect.

Figure 5 - *Adaptive Thresholding Results for Dark Object (Device).*

A close-up of a device

AI-generated content may be incorrect.

Figure 6 - *Adaptive Thresholding Results for Dark Object (Device).*

A close-up of a fruit

AI-generated content may be incorrect.

The output images clearly illustrate how adaptive thresholding effectively separates objects from their backgrounds under different illumination conditions. In both cases, for the dark device and the bright peach, the algorithm successfully compensates for lighting variations by using locally computed thresholds. The Gaussian filtering step enhances this performance by smoothing intensity transitions and minimizing noise before segmentation. As a result, the binarized outputs display clean object boundaries with minimal background interference. These visual outcomes confirm the robustness of adaptive thresholding in maintaining segmentation accuracy across uneven lighting environments.

This experiment successfully demonstrated the implementation and effectiveness of an adaptive thresholding scheme for image segmentation under variable illumination conditions. By dividing the image into smaller regions and calculating a local threshold for each, the algorithm was able to dynamically adjust to changes in lighting direction and intensity. The use of Gaussian-weighted adaptive thresholding, combined with noise reduction preprocessing, resulted in accurate and consistent segmentation of both dark and light objects. This approach clearly outperformed traditional global thresholding methods, which tend to fail when exposed to uneven brightness or shadows across the image surface.

Overall, the results confirm that adaptive thresholding provides a robust, data-driven solution for image segmentation tasks where lighting cannot be controlled or predicted. The modular structure of the program ensures reusability and clarity, making it adaptable for various applications such as medical imaging, object recognition, and automated inspection. The generated visual summaries provide clear documentation of each processing stage originally, grayscale, filtered, and binarized, allowing for transparent evaluation of algorithmic performance. This experiment validates adaptive thresholding as an essential method in modern computer vision pipelines for achieving reliable segmentation across diverse illumination environments.

References

GeeksforGeeks. (n.d.). *Python thresholding techniques using OpenCV (Set-2: Adaptive Thresholding).* Retrieved from <https://www.geeksforgeeks.org/python/python-thresholding-techniques-using-opencv-set-2-adaptive-thresholding/>

OpenCV. (n.d.). *Image thresholding using OpenCV.* Retrieved from <https://opencv.org/blog/image-thresholding-using-opencv/>

OpenCV Documentation. (n.d.). *Image thresholding tutorial (Python).* Retrieved from <https://docs.opencv.org/3.4/d7/d4d/tutorial_py_thresholding.html>

Stack Abuse. (n.d.). *OpenCV adaptive thresholding in Python with cv2.adaptiveThreshold().* Retrieved from <https://stackabuse.com/opencv-adaptive-thresholding-in-python-with-cv2adaptivethreshold/>