MARMARA
UNIVERSITY

Embedded Digital Image Processing

EE4065

Homework 3

Mustafa Öztürk

150722016

Emirhan Şener

150721069

Faculty of Engineering
Electrical Electronic Engineering

Fall 2025

ACRONYMS

EE4065 Embedded Digital Image Processing

LPF Low-pass filter

UART Universal Asynchronous Receiver Transmitter

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

This report is for the implementation and results of the 3^{rd} homework for the Embedded Digital Image Processing (EE4065) course. The primary objective of this study was to implement image segmentation and morphological processing algorithms directly on the NUCLEO-F446RE microcontroller. Unlike the previous assignment, which focused on linear spatial filtering, this work concentrates on non-linear techniques used for separating objects from their background and manipulating binary shapes. As same as the previous assignment we utilized a dynamic real-time data transfer system via Universal Asynchronous Receiver Transmitter (UART). Instead of relying on static header files, raw image data is transmitted from a PC-based Python script to the microcontroller's RAM, allowing for the efficient testing of algorithms by enabling the rapid exchange of grayscale, color, and binary image data.

The scope of this assignment encompasses three distinct image processing tasks defined in the homework requirements. First, we implemented Otsu's Thresholding Method [1] to automatically determine the optimal threshold value for converting grayscale images into binary formats by minimizing the variance. Second, this segmentation technique was extended to color images by applying the thresholding algorithm independently to the Red, Green, and Blue channels of an RGB888 image. Finally, we developed C functions for fundamental Morphological Operations: Dilation, Erosion, Opening, and Closing to manipulate object shapes and reduce noise within the generated binary images. All implemented algorithms were verified by transmitting the processed data back to the PC for visual inspection. This report details the theoretical background, the specific C implementation strategies used to manage the microcontroller's limited memory, and the resulting images obtained from the STM32 [2] platform.

2. PROBLEMS

2.1. Q-1) OTSU'S THRESHOLDING METHOD

(20 points) This question is on Otsu's thresholding method.

(a) Form a C function on the microcontroller to calculate Otsu's thresholding method on a given grayscale image.

(b) Form a grayscale image of your choice with appropriate size on PC. Transfer it to the STM32 microcontroller. Apply Otsu's thresholding method on it. Return the thresholded image back to PC. Display your results in Python there.

Theory

Otsu's method is a widely used algorithm for automatic image thresholding. It allows us to convert a grayscale image (pixel values 0-255) into a binary image (0 or 255) by finding a single optimal threshold value T^*.

The core idea behind Otsu's method is to separate the image pixels into two classes: the foreground (objects) and the background. It assumes that the image histogram has a bimodal distribution (two peaks). The algorithm exhaustively searches for a threshold that minimizes the weighted within-class variance (σ_w^2), which is mathematically equivalent to maximizing the between-class variance (σ_B^2)[1].

The objective function to maximize is:

$$\sigma_B^2(t) = \omega_{bg}(t)\omega_{fg}(t)[\mu_{bg}(t) - \mu_{fg}(t)]^2$$

where:

- ω_{bg}, ω_{fg}: Probabilities (weights) of the background and foreground classes.

- μ_{bg}, μ_{fg}: Mean intensities of the background and foreground classes.

By maximizing this variance, we ensure that the two classes are as distinct as possible.

Procedure

We implemented a C function named `calcOtsuThreshold` on the STM32 microcontroller. This function first calculates the histogram of the input image using the `calcHistogram` function developed in the previous homework.

The function then iterates through all possible threshold values (t) from 0 to 255. For each t, it updates the weights and means of the background and foreground classes to calculate the between-class variance. The threshold that yields the maximum variance is stored as `t_optimal`.

Listing 2.1: C implementation of Otsu's Method.

```
int calcOtsuThreshold(uint8_t* pIn, uint32_t imgSize) {
    uint32_t histogram[256];
    calcHistogram(pIn, histogram, imgSize);

    float mean = 0;
    for (int i = 0; i < 256; ++i) {
        mean += i * histogram[i];
    }

    float weight_bg = 0;
    float sum_bg = 0;
    float maxVar = 0;
    int t_optimal = 0;

    for (int t = 0; t < 256; ++t) {
        weight_bg += histogram[t];

        if (weight_bg == 0) continue;

        float weight_fg = imgSize - weight_bg;
        if (weight_fg == 0) break;

        sum_bg += (float)(t * histogram[t]);

        float mean_bg = sum_bg / weight_bg;
        float mean_fg = (mean - sum_bg) / weight_fg;

        float variance_between = weight_bg * weight_fg * (mean_bg -
            mean_fg) * (mean_bg - mean_fg);

        if (variance_between > maxVar) {
            maxVar = variance_between;
            t_optimal = t;
        }
    }
    return t_optimal;
}
```

After finding the optimal threshold, we applied it to the image using the `applyThreshold` function. Pixels strictly greater than the threshold were set to 255 (white), and others were set to 0 (black).

Results

To test the implementation, we transferred a grayscale version of the "Monk" image to the STM32. The microcontroller successfully calculated the optimal threshold and returned the binary image shown in Figure 2.1.

Visual inspection confirms that the algorithm successfully separated the monkey (foreground) from the background. The high-contrast areas, such as the face and hands, are clearly defined in white, while the darker background is suppressed to black. The automatic threshold selection proved effective for this image without manual tuning.

(a) Original Grayscale Image **(b)** Otsu's Thresholding Result

Figure 2.1: Visual comparison of the original grayscale image vs. Otsu's binary result.

2.2. Q-2) COLOR IMAGE THRESHOLDING

(20 points) Repeat Question 1 to color images.

Theory

Standard Otsu's thresholding is designed for grayscale (single channel) images based on intensity histograms. To apply this method to a color image (RGB), we must treat each color channel (Red, Green, Blue) as an independent grayscale image.

In this approach, we calculate three separate optimal thresholds (T_R^*, T_G^*, T_B^*). Each pixel component is then threshold individually:

$$R'(x,y) = \begin{cases} 255 & \text{if } R(x,y) > T_R^* \\ 0 & \text{otherwise} \end{cases}$$

This is repeated for Green and Blue channels. Consequently, the resulting image does not contain 256 grayscale levels or the full RGB spectrum. Instead, each pixel can only take one of $2^3 = 8$ primary colors (Black, Red, Green, Blue, Cyan, Magenta, Yellow, White), effectively performing a multi-level segmentation or "posterization" of the image.

Procedure

Handling color images requires significantly more RAM than grayscale images ($128 \times 128 \times 3 \approx 49$ KB). To manage the microcontroller's limited memory, we defined a specific global buffer named `pImage_RGB` to store the incoming color data.

We implemented a channel-wise processing strategy to avoid allocating three separate buffers for R, G, and B channels. We reused the `pImage` buffer (from Q1) as a temporary storage to make manupilations on it. We also formed two helper functions:

- `extractChannel`: Copies a specific channel (R, G, or B) from `pImage_RGB` to `pImage`.

Listing 2.2: Helper C function to extract channels from an RGB image.

```
1 void extractChannel(uint8_t* pRGB, uint8_t* pOut,
2                     int channelOffset, uint32_t numPixels) {
3 for (uint32_t i = 0; i < numPixels; i++) pOut[i] = pRGB[i * 3 +
    channelOffset];
4 }
```

- `insertChannel`: Copies the processed single-channel data from `pImage` back into the correct slots of `pImage_RGB`.

Listing 2.3: Helper C function to insert a channel to an RGB image.

```
void extractChannel(uint8_t* pRGB, uint8_t* pOut,
                    int channelOffset, uint32_t numPixels) {
for (uint32_t i = 0; i < numPixels; i++) pOut[i] = pRGB[i * 3 +
    channelOffset];
}
```

The main processing loop iterates through the three channels. For each channel, it extracts the data, calculates the Otsu threshold, applies it to create a binary mask, and inserts it back into the color image.

Listing 2.4: Main loop for Color Otsu processing.

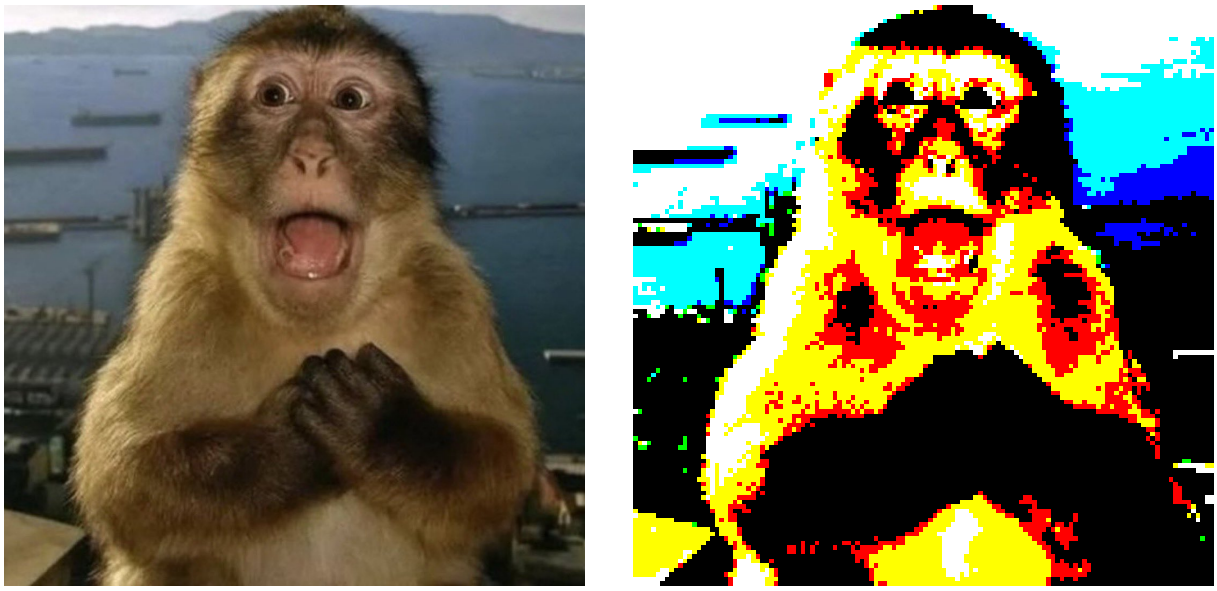
```
for (int ch = 0; ch < 3; ch++) {
    extractChannel((uint8_t*)pImage_RGB, (uint8_t*)pImage, ch, IMG_SIZE);
    int thresh = calculateOtsuThreshold((uint8_t*)pImage, IMG_SIZE);
    applyThreshold((uint8_t*)pImage, (uint8_t*)pImage, thresh, IMG_SIZE);
    insertChannel((uint8_t*)pImage_RGB, (uint8_t*)pImage, ch, IMG_SIZE);
}
```

Results

To verify the multi-channel thresholding approach, we transferred the color version of the image to the STM32. The microcontroller processed the Red, Green, and Blue channels independently using the calculated Otsu thresholds for each channel. The resulting output is displayed in Figure 2.2.

As seen in the result, the image has been reduced to a limited palette of 8 colors. The segmentation effectively highlights the dominant spectral components of the scene:

- **Subject (Monkey):** The fur, originally a mix of brown and gold, has been segmented primarily into Yellow ($R = 255, G = 255, B = 0$) and Red ($R = 255, G = 0, B = 0$) regions. This indicates high intensity in the Red and Green channels for the fur, but low intensity in the Blue channel.
- **Background:** The sea and sky, originally blueish-grey, have been mapped largely to Cyan ($G = 255, B = 255$) and Blue ($B = 255$), correctly reflecting the dominance of the blue channel in these areas.
- **Shadows:** Darker areas such as the deep shadows in the fur or the mouth are mapped to Black ($R = 0, G = 0, B = 0$).



(a) Original Color Image

(b) Color Otsu Result

Figure 2.2: Comparison of the original color image vs. channel-wise Otsu thresholding.

2.3. Q-3) MORPHOLOGICAL OPERATIONS

(60 points) This question is on morphological operations.

(a) Form C functions on the microcontroller to apply dilation, erosion, closing, and opening on a given binary image.

(b) Use the binary image formed in Question 1. Apply morphological operations on it. Return the morphological operation results to PC. Display your results in Python there.

Theory

Morphological operations are non-linear image processing techniques that process images based on their shapes. Unlike linear filters (like the Low-pass filter (LPF) used in Homework 2), morphological operations rely on the relative ordering of pixel values rather than their numerical values. They are typically applied to binary images.

These operations probe an image with a small shape or template called a kernel. In this assignment, we used a 3×3 square kernel. The four fundamental operations are:

- **Dilation ($A \oplus B$):** Adds pixels to the boundaries of objects. If *any* pixel under the 3×3 kernel is white (255), the center pixel becomes white. It fills holes and connects disjoint objects.
- **Erosion ($A \ominus B$):** Removes pixels on object boundaries. If *any* pixel under the kernel is black (0), the center pixel becomes black. It removes small noise and detaches connected objects.
- **Opening ($A \circ B$):** Erosion followed by Dilation. It is useful for removing noise (small white points) from the background while preserving the shape of the main object.
- **Closing ($A \bullet B$):** Dilation followed by Erosion. It is useful for closing small holes inside the foreground objects or connecting components that are close to each other.

Procedure

Implementing four sequential morphological operations requires careful memory management on the STM32. Since the `pImage_RGB` buffer (approx. 49 KB) was not needed for Q3 (as we are processing binary images), we repurposed this large memory block as our output buffer (`pOutputBuffer`) to prevent stack overflow.

We utilized the binary image generated in Q1 (stored in `pImage_Otsu`) as the input for all operations. For composite operations like **Opening** and **Closing**, we used the `pImage` buffer as a temporary intermediate storage, since these operations require two sequential steps. Below is the complete C implementation for all four morphological operations.

Listing 2.5: C implementation of Dilation, Erosion, Opening, and Closing.

```
void dilate(uint8_t* pIn, uint8_t* pOut, int w, int h) {
    for (int y = 1; y < h - 1; y++) {
        for (int x = 1; x < w - 1; x++) {
            uint8_t pixelVal = 0; // Default Black
            // Check 3x3 neighborhood
            for (int ky = -1; ky <= 1; ky++) {
                for (int kx = -1; kx <= 1; kx++) {
                    // If any neighbor is White, result is White
                    if (pIn[(y + ky) * w + (x + kx)] == 255) {
                        pixelVal = 255;
                        break;
                    }
                }
                if (pixelVal == 255) break;
            }
            pOut[y * w + x] = pixelVal;
        }
    }
}

void erode(uint8_t* pIn, uint8_t* pOut, int w, int h) {
    for (int y = 1; y < h - 1; y++) {
        for (int x = 1; x < w - 1; x++) {
            uint8_t pixelVal = 255; // Default White
            // Check 3x3 neighborhood
            for (int ky = -1; ky <= 1; ky++) {
                for (int kx = -1; kx <= 1; kx++) {
                    // If any neighbor is Black, result is Black
                    if (pIn[(y + ky) * w + (x + kx)] == 0) {
                        pixelVal = 0;
                        break;
                    }
                }
                if (pixelVal == 0) break;
            }
            pOut[y * w + x] = pixelVal;
        }
    }
}

void opening(uint8_t* pIn, uint8_t* pOut, int w, int h) {
```

```
    erode(pIn, (uint8_t*)pImage, w, h);
    dilate((uint8_t*)pImage, pOut, w, h);
}

void closing(uint8_t* pIn, uint8_t* pOut, int w, int h) {
    dilate(pIn, (uint8_t*)pImage, w, h);
    erode((uint8_t*)pImage, pOut, w, h);
}
```

Results

We applied the four operations to the binary image obtained from Q1.

1. Dilation: As shown in Figure 2.3a, the white regions (the monkey's body) have expanded. The boundaries appear thicker, and small gaps within the fur texture have been filled in, making the object look more solid.

2. Erosion: Figure 2.3b shows that the white regions have shrunk. Fine details, such as the thinner parts of the fur, have eroded away. The overall object looks thinner compared to the original binary input.

3. Opening: The result in Figure 2.3c demonstrates the noise-removal capability of opening. By eroding first, small isolated pixels are removed, and the subsequent dilation restores the general shape of the larger object. The contours appear smoother.

4. Closing: In Figure 2.3d, small black holes within the white object have been filled. This operation effectively unified the texture of the monkey's fur, creating a more cohesive foreground mask.

(a) Dilation Result

(b) Erosion Result

(c) Opening Result

(d) Closing Result

Figure 2.3: Visual results of morphological operations on the binary image.

3. CONCLUSION

In this homework, we successfully implemented and verified fundamental image segmentation and morphological processing algorithms on the STM32 F446RE microcontroller. Moving beyond the linear filtering techniques of the previous homework, this work focused on non-linear operations essential for object detection and shape analysis. We first established a robust method for binarization using Otsu's Thresholding, which allowed for automatic foreground-background separation without manual parameter tuning. This technique was effectively extended to the color domain, demonstrating how multi-channel processing can be managed on an embedded system by iterating through Red, Green, and Blue channels independently to produce a segmented, 8-color output.

The implementation of Morphological Operations (Dilation, Erosion, Opening, and Closing) provided practical insight into binary image manipulation. By developing these algorithms from scratch in C, we observed how structural elements can be used to filter noise, connect disjoint components, or refine object boundaries. A critical aspect of this assignment was the efficient management of the microcontroller's limited SRAM. We optimized our code by repurposing existing image buffers (specifically the RGB buffer) for output storage, preventing stack overflow errors during memory-intensive operations. Furthermore, the real-time visualization of these results via UART communication confirmed the correctness of our algorithms and highlighted the importance of buffer management between the embedded device and the PC.

BIBLIOGRAPHY

[1] N. Otsu, A threshold selection method from gray-level histograms, IEEE Transactions on Systems, Man, and Cybernetics 9 (1) (1979) 62–66. doi:10.1109/TSMC.1979.4310076.

[2] STMicroelectronics, STM32CubeIDE, https://www.st.com/en/development-tools/stm32cubeide.html, version 1.19.0, Accessed: October 30, 2025 (2025).