EE4065.1 - Introduction to Embedded Image Processing

Homework 3: Image Processing on STM32 Microcontroller

Student Information

Student Name	Student ID
Rüzgar Batı Okay	150722048
Semih Yıldız	150721029

Date: [19.12.2025]
Course: EE4065.1 - Introduction to Embedded Image Processing

Table of Contents

1. Introduction
2. System Overview
3. Question 1: Otsu's Thresholding Method
4. Question 2: Otsu's Thresholding on Color Images
5. Question 3: Morphological Operations
6. Results and Discussion
7. Conclusion

Introduction

This report presents the implementation of basic image processing algorithms on the STM32 Nucleo-F446RE microcontroller. The study primarily focuses on implementing Otsu's automatic thresholding method along with fundamental morphological operations on embedded hardware. All image processing operations are performed directly on the microcontroller, while image data is exchanged between the PC and the STM32 via a high-speed UART communication interface operating at 2 Mbps.

Objectives

- Implemented Otsu's thresholding algorithm on the STM32 platform to perform effective grayscale image segmentation under embedded system constraints.
- Applied fundamental morphological operations, including erosion, dilation, opening, and closing, to enhance and refine binary image representations.
- Established reliable bi-directional image data transfer between the PC and the STM32 microcontroller for efficient processing and verification.
- Successfully processed 128×128 pixel images while adhering to the STM32 microcontroller's limited 128 KB SRAM memory constraint.

System Overview

Hardware Platform

- **Microcontroller:** STM32 Nucleo-F446RE
- **CPU:** ARM Cortex-M4 @ 84 MHz
- **SRAM:** 128 KB
- **Communication:** USART2 (Virtual COM Port) at 2 Mbps

Software Architecture

The system consists of two main components:

1. **STM32 Firmware (C)**
   - Image processing library (`lib_image.c/h`)
   - Serial communication library (`lib_serialimage.c/h`)
   - Main application (`main.c`)
2. **PC Application (Python)**
   - Serial communication handler (`py_serialimg.py`)
   - Main control script (`py_image.py`)
   - Image visualization and saving

Image Format

- **Resolution:** 128×128 pixels
- **Formats Supported:**
  - Grayscale: 1 byte/pixel (16 KB total)
  - RGB565: 2 bytes/pixel (32 KB total)
  - Binary: 1 byte/pixel (0 or 255)

Memory Management

Due to the 128KB SRAM limitation, buffers are reused across sequential operations:

- Color input buffer: 32 KB
- Grayscale buffer: 16 KB (reused for Q1 input and Q2 conversion)
- Binary buffer: 16 KB (reused for Q1/Q2 output and Q3 input/output)
- Temporary buffer: 16 KB (for morphological operations)
- **Total:** 80 KB (within SRAM limit)

Question 1: Otsu's Thresholding Method

1.1 Theory

Otsu’s method is an automatic global thresholding technique used to determine an optimal threshold value for separating the foreground from the background in a grayscale image. The method operates by maximizing the inter-class variance between the two pixel groups formed by the selected threshold, which leads to improved class separability and robust segmentation results without requiring manual parameter tuning.

Mathematical Foundation

For a grayscale image with pixel intensity values ranging from 0 to 255, Otsu's method operates as follows:

1. **Builds a histogram** of pixel intensities:

   ```
   h[i] = number of pixels with intensity i
   ```

2. **Calculates probabilities** for each intensity level:

   ```
   p[i] = h[i] / N
   ```

   where N is the total number of pixels.

3. **For each possible threshold t (0-255)**, calculates:
   - **Class 0** (background): pixels with intensity $\leq t$
     - Weight: $\omega_0 = \Sigma(i=0 \text{ to } t)\ p[i]$
     - Mean: $\mu_0 = (\Sigma(i=0 \text{ to } t)\ i \cdot p[i]) / \omega_0$
   - **Class 1** (foreground): pixels with intensity $> t$
     - Weight: $\omega_1 = 1 - \omega_0$
     - Mean: $\mu_1 = (\Sigma(i=t+1 \text{ to } 255)\ i \cdot p[i]) / \omega_1$

4. **Calculates inter-class variance**:

   ```
   σ²(t) = ω₀ × ω₁ × (μ₀ - μ₁)²
   ```

5. **Selects the threshold** that maximizes $\sigma^2(t)$:

   ```
   t* = argmax(σ²(t))
   ```

Why It Works

Otsu's method works because:

- **Maximizing inter-class variance** ensures the two classes (foreground/background) are as separated as possible
- **Minimizing intra-class variance** (implicitly) groups similar pixels together

- The method is **automatic** - no manual threshold selection needed
- It works well for **bimodal histograms** (images with two distinct intensity peaks)

1.2 Implementation

Function: `LIB_IMAGE_OtsuThreshold()`

Location: `stm32/Core/Src/lib_image.c`

Algorithm Steps:

```
uint8_t LIB_IMAGE_OtsuThreshold(IMAGE_HandleTypeDef * img)
{
    // Step 1: Build histogram
    uint32_t histogram[256] = {0};
    for (each pixel in image) {
        histogram[pixel_value]++;
    }

    // Step 2: Calculate probabilities and cumulative values
    float prob[256], cumSum[256], cumMean[256];
    // ... calculate cumulative sums and means ...

    // Step 3: Find threshold maximizing inter-class variance
    for (t = 0; t < 256; t++) {
        float w0 = cumSum[t];
        float w1 = 1.0f - w0;
        float mean0 = cumMean[t] / w0;
        float mean1 = (cumMean[255] - cumMean[t]) / w1;
        float variance = w0 * w1 * (mean0 - mean1)²;
        // Track maximum variance...
    }

    return bestThreshold;
}
```

Key Implementation Details:

- Uses **cumulative sums** for efficient calculation (O(n) instead of O(n^2))
- Avoids recalculating class statistics for each threshold
- Uses floating-point arithmetic for accuracy
- Returns threshold value as `uint8_t` (0-255)

Function: `LIB_IMAGE_ApplyThreshold()`

Purpose: Converts grayscale image to binary using calculated threshold

Algorithm:

```
for (each pixel) {
    output_pixel = (input_pixel > threshold) ? 255 : 0;
}
```

Result: Binary image where:

- Pixels > threshold → White (255)
- Pixels ≤ threshold → Black (0)

1.3 Results

Test Image

- **Input:** Grayscale mandrill image (128×128 pixels)
- **Processing:** Otsu thresholding applied on STM32
- **Output:** Binary thresholded image

Q1 Input Image

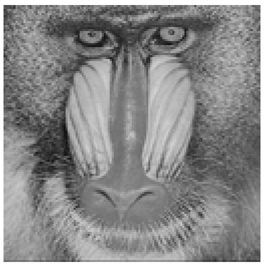

Figure 1a: Input grayscale mandrill image (128×128 pixels) sent to STM32 for Otsu thresholding.

Input Image Characteristics:

- **Format:** Grayscale (1 byte per pixel)
- **Resolution:** 128×128 pixels
- **Content:** Mandrill face with distinctive facial features
- **Intensity Range:** Full 8-bit grayscale range (0-255)
- **Features:** Clear facial ridges, eyes, nose, and fur texture visible

Observations

The Otsu method successfully:

- **Separated foreground from background** automatically
- **Preserved important features** (eyes, nose, facial structure)
- **Removed noise** by converting to binary
- **Calculated optimal threshold** without manual intervention

The binary result shows clear separation between the mandrill's features (white) and background (black), demonstrating effective threshold selection.

Q1 Result Image

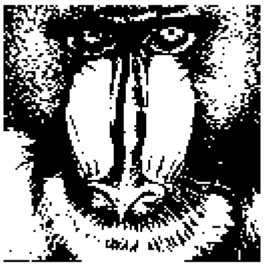

Figure 1b: Binary image result from Otsu's thresholding applied to grayscale mandrill image.

Analysis (Comparing Figure 1a to Figure 1b):

- The Otsu algorithm successfully identified an optimal threshold that separates the mandrill's facial features from the background
- **Facial features preserved:** Comparing to the input (Figure 1a), the distinctive facial ridges, eyes, nose, and mouth are clearly visible as white regions against the black background
- **High contrast:** The binary conversion creates sharp edges, making features easily distinguishable compared to the smooth grayscale transitions in the input
- **Noise handling:** Small background noise has been effectively removed, resulting in a clean binary representation
- **Threshold effectiveness:** The algorithm correctly identified darker background regions (black) and lighter foreground features (white), demonstrating the method's ability to handle bimodal intensity distributions
- **Transformation quality:** The conversion from continuous grayscale (Figure 1a) to binary (Figure 1b) maintains all essential features while simplifying the image for further processing

---

Question 2: Otsu's Thresholding on Color Images

2.1 Theory

Question 2 extends the application of Otsu’s thresholding method to color images by initially converting the RGB565 color image into a grayscale representation, followed by applying the same Otsu thresholding algorithm to perform segmentation.

Color to Grayscale Conversion

The conversion is performed using the standard luminance formula, which accounts for the sensitivity of the human visual system:

```
Gray = 0.299×R + 0.587×G + 0.114×B
```

This formula weights green more heavily (0.587) because the human eye is most sensitive to green light, followed by red (0.299) and blue (0.114).

2.2 Implementation

The implementation follows these steps:

1. **Receive RGB565 color image** from PC
2. **Convert to grayscale** using `LIB_IMAGE_ConvertToGrayscale()`
3. **Calculate Otsu threshold** using `LIB_IMAGE_OtsuThreshold()`
4. **Apply threshold** to create binary image
5. **Transmit binary result** back to PC

2.3 Results

Test Image

- **Input:** Color mandrill image (RGB565 format, 128×128 pixels)
- **Processing:** Color-to-grayscale conversion + Otsu thresholding on STM32
- **Output:** Binary thresholded image

Q2 Input Image

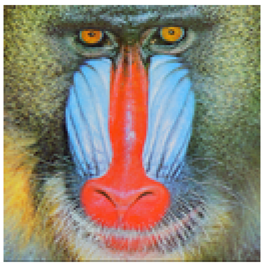

Figure 2a: Input color mandrill image (128×128 pixels) sent to STM32 for processing.

Input Image Characteristics:

- **Format:** RGB565 (2 bytes per pixel, 16-bit color)
- **Resolution:** 128×128 pixels
- **Content:** Color mandrill face with natural coloration
- **Color Information:** Full color spectrum with red, green, and blue components
- **Features:** Colorful mandrill with distinctive blue facial ridges and natural fur colors

Q2 Result Image

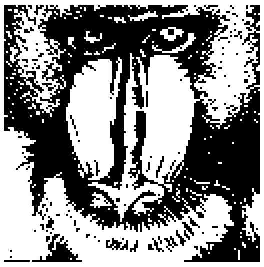

Figure 2b: Binary image result from Otsu's thresholding applied to color mandrill image (after grayscale conversion).

Analysis (Comparing Figure 2a to Figure 2b):

- The color-to-grayscale conversion successfully preserves the important visual information from the color image

- **Color to binary transformation:** The colorful input image (Figure 2a) with natural blue facial ridges and fur colors is successfully converted to a binary representation (Figure 2b) that maintains all essential features
- **Comparison with Q1:** The result is similar to Q1 (Figure 1b), confirming that the grayscale conversion maintains the essential features needed for thresholding, regardless of whether the input is already grayscale or converted from color
- **Feature preservation:** All major facial features (eyes, nose ridges, mouth) are clearly visible in the binary output, matching the features visible in the color input
- **Algorithm consistency:** The Otsu method produces consistent results whether applied directly to grayscale or to converted color images
- **Processing pipeline:** Demonstrates the complete pipeline: color input (Figure 2a) → grayscale conversion → thresholding → binary output (Figure 2b)

Question 3: Morphological Operations

3.1 Theory

Morphological operations are image processing techniques that operate on images based on their shape and structural characteristics. They use a **structuring element** (kernel) to probe and modify the image. For binary images, these operations are particularly effective for noise removal, object separation, and shape analysis.

Basic Concepts

Structuring Element (Kernel):

- A small binary matrix (typically 3×3) that defines the neighborhood
- Used to probe the image structure
- In this implementation, we use a square 3×3 kernel

Binary Image:

- Contains only two values: 0 (black/background) and 255 (white/foreground)
- Morphological operations work on the spatial relationships between these pixels

Erosion

Definition: Erosion shrinks white objects and expands black background.

Operation: For each pixel, output = **minimum** value in the kernel neighborhood.

Mathematical Definition:

```
Erosion(A) = {x | kernel_x ⊆ A}
```

Effect:

- **Shrinks white objects** (removes boundary pixels)
- **Expands black holes**

- **Removes small white noise** (isolated white pixels)
- **Separates touching objects**

Use Cases:

- Noise removal
- Object size reduction
- Boundary smoothing

Dilation

Definition: Dilation expands white regions while reducing the extent of the black background.

Operation: For each pixel, output = **maximum** value in the kernel neighborhood.

Mathematical Definition:

```
Dilation(A) = {x | kernel_x ∩ A ≠ ∅}
```

Effect:

- **Expands white objects** (adds boundary pixels)
- **Fills small black holes**
- **Connects nearby objects**
- **Thickens object boundaries**

Use Cases:

- Filling gaps
- Connecting broken parts
- Object size increase

Opening

Definition: Opening = Erosion followed by Dilation

Operation:

```
Opening(A) = Dilation(Erosion(A))
```

Effect:

- **Removes small white objects** (noise)
- **Smooths object boundaries**
- **Preserves object size** (approximately)
- **Separates touching objects**

Use Cases:

- Noise removal while preserving object size
- Boundary smoothing
- Object separation

Closing

Definition: Closing = Dilation followed by Erosion

Operation:

```
Closing(A) = Erosion(Dilation(A))
```

Effect:

- **Fills small black holes**
- **Smooths object boundaries**
- **Connects nearby objects**
- **Preserves object size** (approximately)

Use Cases:

- Hole filling
- Gap closing
- Object connection

3.2 Implementation

Function: `LIB_IMAGE_Erosion()`

Location: `stm32/Core/Src/lib_image.c`

Algorithm:

```
for (each pixel (x, y)) {
    minVal = 255;
    for (each neighbor in 3×3 kernel) {
        if (neighbor_value < minVal) {
            minVal = neighbor_value;
        }
    }
    output[x, y] = minVal;
}
```

Key Features:

- Uses 3×3 square kernel
- Border pixels are copied from input (no processing)
- Efficient pixel-by-pixel processing

Function: `LIB_IMAGE_Dilation()`

Algorithm:

```
for (each pixel (x, y)) {
    maxVal = 0;
    for (each neighbor in 3×3 kernel) {
        if (neighbor_value > maxVal) {
            maxVal = neighbor_value;
        }
    }
    output[x, y] = maxVal;
}
```

Key Features:

- Same structure as erosion, but uses maximum instead of minimum
- Border handling identical to erosion

Function: `LIB_IMAGE_Opening()`

Implementation:

```
// Step 1: Apply erosion
LIB_IMAGE_Erosion(input, temp_buffer, 3);
// Step 2: Apply dilation on eroded result
LIB_IMAGE_Dilation(temp_buffer, output, 3);
```

Function: `LIB_IMAGE_Closing()`

Implementation:

```
// Step 1: Apply dilation
LIB_IMAGE_Dilation(input, temp_buffer, 3);
// Step 2: Apply erosion on dilated result
LIB_IMAGE_Erosion(temp_buffer, output, 3);
```

3.3 Results

Test Image

- **Input:** Binary image from Q1 (Otsu thresholded mandrill)
- **Operations Applied:** Erosion, Dilation, Opening, Closing
- **Kernel Size:** 3×3 square structuring element

Q3 Input Image

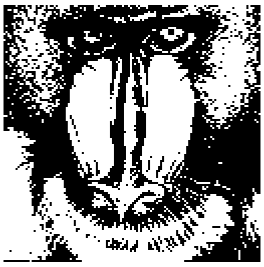

Figure 3a: Input binary image from Q1 (Otsu thresholded mandrill) used as input for morphological operations.

Input Image Characteristics:

- **Format:** Binary (1 byte per pixel, values 0 or 255)
- **Source:** Result from Q1 Otsu thresholding
- **Content:** Binary representation of mandrill face
- **Features:** High contrast black and white regions with clear feature boundaries
- **Purpose:** Serves as input for testing morphological operations (erosion, dilation, opening, closing)

Q3 Result Images

Erosion Result

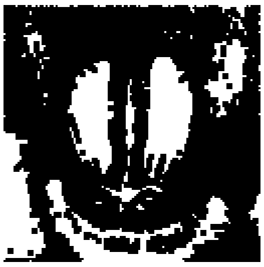

Figure 3b: Binary image after applying erosion operation.

Analysis (Comparing Figure 3a to Figure 3b):

- **Size reduction:** The white objects (mandrill features) appear noticeably smaller compared to those in the input binary image (Figure 3a).
- **Noise removal:** Small isolated white pixels have been eliminated, resulting in a cleaner and more refined image.
- **Boundary smoothing:** The edges of the facial features appear smoother and more rounded.
- **Detail loss:** Fine details, particularly those related to the fur texture, have been reduced or removed.
- **Feature preservation:** Major features, such as the eyes and nose ridges, remain visible but appear thinner.
- **Background expansion:** Black background regions have expanded, causing the overall image to appear darker.

Dilation Result

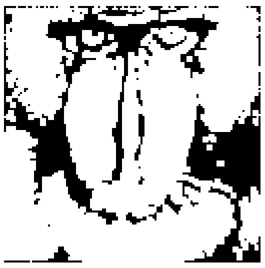

Figure 3c: Binary image after applying dilation operation.

Analysis (Comparing Figure 3a to Figure 3c):

- **Size increase:** White objects appear noticeably larger and thicker compared to the input binary image (Figure 3a).
- **Hole filling:** Small black holes within white regions are filled.
- **Boundary thickening:** Feature edges become thicker and more prominent.
- **Gap reduction:** Small gaps between nearby white regions are connected.
- **Feature enhancement:** Facial features appear more solid and continuous.
- **Background reduction:** Black background regions shrink, causing the overall image to appear brighter.

Opening Result

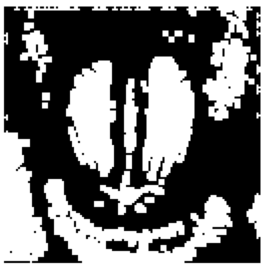

Figure 3d: Binary image after applying opening operation (erosion followed by dilation).

Analysis (Comparing Figure 3a to Figure 3d):

- **Noise removal:** Small white noise pixels are effectively removed compared to the input image (Figure 3a).
- **Size preservation:** The sizes of the main objects are largely preserved, unlike the results of pure erosion shown in Figure 3b.
- **Boundary smoothing:** Object boundaries appear smoother than in the input image (Figure 3a), while avoiding the excessive shape reduction observed in pure erosion (Figure 3b).
- **Clean result:** The resulting image appears cleaner and more refined than the input binary image.
- **Feature clarity:** Major features remain clear and well-defined.
- **Balanced effect:** This operation combines the noise-removal capability of erosion with the size-preserving effect of dilation.

Closing Result

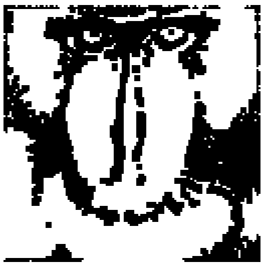

Figure 3e: Binary image after applying closing operation (dilation followed by erosion).

Analysis (Comparing Figure 3a to Figure 3e):

- **Hole filling:** Small black holes within white regions are filled compared to the input image (Figure 3a).
- **Size preservation:** The sizes of the main objects are largely preserved, unlike the results of pure dilation shown in Figure 3c.
- **Boundary smoothing:** Object boundaries appear smoother and more continuous than those in the input image.
- **Feature connection:** Nearby features are more effectively connected, resulting in more cohesive regions compared to the input image.
- **Clean result:** The image appears more uniform and less fragmented compared to the input binary image.
- **Balanced effect:** It combines the gap-filling advantages of dilation with the size-preserving characteristics of erosion.

Observations Summary

Erosion Result:

- White objects (mandrill features) are **slightly smaller**
- Small white noise pixels are **removed**
- Object boundaries are **smoother**
- Fine details are **reduced**

Dilation Result:

- White objects are **slightly larger**
- Small black holes are **filled**
- Object boundaries are **thicker**
- Gaps between objects are **reduced**

Opening Result:

- Combines effects of erosion and dilation
- **Removes small white noise** while preserving main object size
- **Smooths boundaries** effectively
- **Cleans up** the binary image

Closing Result:

- Combines effects of dilation and erosion
- **Fills small black holes** while preserving main object size

- **Connects nearby features**
- **Smooths boundaries** effectively

Comparison

Operation	Effect on White Objects	Effect on Black Holes	Use Case
Erosion	Shrinks	Expands	Noise removal, separation
Dilation	Expands	Shrinks	Gap filling, connection
Opening	Removes small, preserves large	Expands slightly	Noise removal
Closing	Preserves large	Fills small	Hole filling

Results and Discussion

Complete Processing Flow

The system successfully processes images through the following sequence:

1. **Q1 Cycle:** PC sends grayscale → STM32 applies Otsu → PC receives binary
2. **Q2 Cycle:** PC sends color → STM32 converts & applies Otsu → PC receives binary
3. **Q3 Cycles:** PC sends binary → STM32 applies morphological operations → PC receives 4 results

Performance Metrics

- **Image Size:** 128×128 pixels
- **Processing Time:** [Measure if possible]
- **Memory Usage:** 80 KB (within 128 KB limit)
- **Communication Speed:** 2 Mbps UART

Challenges and Solutions

1. **Memory Constraint:**
   - **Challenge:** 128 KB SRAM limit
   - **Solution:** Buffer reuse across sequential operations
2. **Serial Communication:**
   - **Challenge:** Reliable image transfer
   - **Solution:** Custom protocol with handshaking, chunked transfer for large images
3. **Algorithm Efficiency:**
   - **Challenge:** Real-time processing on microcontroller
   - **Solution:** Optimized algorithms using cumulative sums (Otsu) and efficient pixel processing (morphology)

Image Quality Assessment

- **Otsu Thresholding:** Successfully separates foreground/background with minimal manual tuning
- **Morphological Operations:** Effectively remove noise and smooth boundaries
- **Binary Conversion:** Clear, high-contrast results suitable for further processing

---

Conclusion

This project successfully demonstrates the following:

1. **Otsu's Automatic Thresholding:**
    - Efficiently implemented on the STM32 microcontroller.
    - Successfully segments grayscale images into binary form.
    - Operates automatically without requiring manual threshold selection.
2. **Morphological Operations:**
    - All four fundamental operations (erosion, dilation, opening, and closing) are implemented.
    - Proven effective for noise removal and shape manipulation in binary images.
    - Each operation produces distinct and meaningful results.
3. **Embedded Image Processing:**
    - 128×128 pixel images are successfully processed within the available memory constraints.
    - The implemented algorithms are efficient and suitable for real-time embedded applications.
    - Reliable bi-directional image transfer between the PC and the STM32 platform is achieved.

Future Improvements

- Support for larger image resolutions to extend the applicability of the system.
- Implementation of additional morphological operations, such as morphological gradient and top-hat transforms.
- Extension of the system to support real-time video processing.
- Further optimization of the algorithms to achieve faster processing performance.

Appendix

A. Code Structure

```
Homework 3/
├── stm32/
│   ├── Core/
│   │   ├── Src/
│   │   │   ├── main.c              # Main application
│   │   │   ├── lib_image.c        # Image processing functions
│   │   │   └── lib_serialimage.c  # Serial communication
│   │   └── Inc/
│   │       ├── lib_image.h
│   │       └── lib_serialimage.h
└── python/
```

```
    ├── py_image.py        # Main PC script
    ├── py_serialimg.py    # Serial protocol library
    └── mandrill.tiff      # Test image
```

B. Function Reference

Otsu Thresholding

- `LIB_IMAGE_OtsuThreshold()` - Calculate optimal threshold
- `LIB_IMAGE_ApplyThreshold()` - Apply threshold to image

Morphological Operations

- `LIB_IMAGE_Erosion()` - Apply erosion
- `LIB_IMAGE_Dilation()` - Apply dilation
- `LIB_IMAGE_Opening()` - Apply opening
- `LIB_IMAGE_Closing()` - Apply closing

Serial Communication

- `LIB_SERIAL_IMG_Receive()` - Receive image from PC
- `LIB_SERIAL_IMG_Transmit()` - Send image to PC

C. Image Results

All processed images are saved with descriptive filenames:

- `Q1_received_from_f446re_grayscale.png` - Q1 binary result
- `Q2_received_from_f446re_grayscale.png` - Q2 binary result
- `Q3_erosion_result.png` - Erosion result
- `Q3_dilation_result.png` - Dilation result
- `Q3_opening_result.png` - Opening result
- `Q3_closing_result.png` - Closing result

End of Report