MARMARA UNIVERSITY
ENGINEERING FACULTY

EE 4065

Introduction to Embedded Digital Image Processing

Homework 3 Report

NAME:	*Baran*	*Muhammet Yücel*
SURNAME:	*ORMAN*	*ÇELİK*
NUMBER:	*150721063*	*150721024*

1. QUESTION

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

In this section, we implemented Otsu's method to automatically determine the optimal threshold value for separating the foreground from the background in a grayscale image. The algorithm aims to maximize the **Between-Class Variance** (σ_B^2)

The core logic is implemented in the compute_otsu_threshold function in image_processing.c.

1. **Histogram Calculation:** First, we iterate through the image pixels to calculate the histogram (probability distribution of intensity levels).
2. **Variance Maximization:** We iterate through all possible threshold values (t from 0 to 255). For each t, we calculate:
    - Weight of background (ω_B) and foreground (ω_F).
    - Mean of background (μ_B) and foreground (μ_F).
    - Between-class variance: $\sigma_B^2 = \omega_B \times \omega_F(\mu_B - \mu_F)$.
3. **Optimal Threshold:** The t value that yields the maximum variance is selected as the optimal threshold.

Key Code Segment:

The function uses floating-point arithmetic to ensure precision during the variance calculation but operates on uint8_t pixel data for efficiency.

```
uint8_t compute_otsu_threshold(const uint8_t* image, uint32_t num_pixels)
{
    uint32_t hist[256] = { 0 };
    for (uint32_t i = 0; i < num_pixels; i++)
        hist[image[i]]++;

    uint32_t total = num_pixels;

    uint32_t sum_total = 0;
    for (uint32_t i = 0; i < 256; i++)
        sum_total += i * hist[i];

    uint32_t wB = 0;
    uint32_t sumB = 0;

    double max_between = 0.0;
    uint8_t threshold = 0;

    for (uint32_t t = 0; t < 256; t++)
    {
        wB += hist[t];
        if (wB == 0) continue;

        uint32_t wF = total - wB;
        if (wF == 0) break;

        sumB += t * hist[t];

        double mB = (double)sumB / wB;
        double mF = (double)(sum_total - sumB) / wF;

        double diff = mB - mF;
        double between = (double)wB * (double)wF * diff * diff;

        if (between > max_between)
        {
            max_between = between;
            threshold = (uint8_t)t;
        }
    }

    return threshold;
}
```

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.

The grayscale image is transferred from the PC to the STM32 via UART using a custom Python script. The MCU processes the image and sends back the binary result.

- **Input:** A grayscale image of size 128×128.
- **Process:** Pixels with intensity $> T$ are set to 255 (White), others to 0 (Black).
- **Result:** A binary image where the object is clearly separated from the background.

Figure 1 Original Grayscale Image vs. Otsu Thresholded Binary Image

2. QUESTION

a) Repeat Question 1 to color images.

For color images (RGB565 format), we applied the **Independent Channel Thresholding** approach. Since a 3D color histogram is computationally expensive for an MCU, we treated each color channel (Red, Green, Blue) as an independent grayscale image.

Implementation Strategy (Apply_Otsu_On_ColorImage_RGB565):

1. **Extraction:** Since Otsu's method works on intensity histograms, we first need to isolate a single color channel. The Extract_Channel_From_RGB565 function iterates through the 16-bit pixels, applies bit-masking and shifting to separate the 5-bit Red, 6-bit Green, or 5-bit Blue components, and stores them in a temporary 8-bit buffer.

```
void Extract_Channel_From_RGB565(const uint16_t* src,
    uint8_t* dst,
    uint32_t num_pixels,
    char channel)
{
    for (uint32_t i = 0; i < num_pixels; i++)
    {
        uint16_t pixel = src[i];

        // Bit-masking and Shifting to isolate components
        uint8_t R8 = ((pixel >> 11) & 0x1F) << 3; // R5 -> R8
        uint8_t G8 = ((pixel >> 5) & 0x3F) << 2; // G6 -> G8
        uint8_t B8 = (pixel & 0x1F) << 3; // B5 -> B8

        switch (channel)
        {
        case 'R': dst[i] = R8; break;
        case 'G': dst[i] = G8; break;
        case 'B': dst[i] = B8; break;
        default:  dst[i] = 0;  break;
        }
    }
}
```

2. **Processing:** Once the channel data is in the 8-bit buffer (tempImg1), we apply the standard apply_otsu_threshold function developed in Question 1. This function calculates the histogram, finds

the optimal threshold that maximizes between-class variance, and binarizes the buffer (pixels become 0 or 255).

```
void apply_otsu_threshold_gray(const uint8_t* in,
    uint8_t* out,
    uint32_t num_pixels)
{
    uint8_t T = compute_otsu_threshold_gray(in, num_pixels);

    for (uint32_t i = 0; i < num_pixels; i++)
        out[i] = (in[i] > T) ? 255 : 0;
}
```

3. **Reconstruction:** The final step is to write the processed binary data back into the original image structure. The Write_Channel_To_RGB565 function takes the binarized 8-bit values and inserts them back into their respective bit positions within the 16-bit RGB565 pixel, preserving the other two channels.

```
void Write_Channel_To_RGB565(uint16_t* dst,
    const uint8_t* src,
    uint32_t num_pixels,
    char channel)
{
    for (uint32_t i = 0; i < num_pixels; i++)
    {
        uint16_t pixel = dst[i];

        // Extract current components
        uint8_t R8 = ((pixel >> 11) & 0x1F) << 3;
        uint8_t G8 = ((pixel >> 5) & 0x3F) << 2;
        uint8_t B8 = (pixel & 0x1F) << 3;

        // Update only the target channel
        switch (channel)
        {
        case 'R': R8 = src[i]; break;
        case 'G': G8 = src[i]; break;
        case 'B': B8 = src[i]; break;
        }

        // Pack back to RGB565 (5-6-5 format)
        uint16_t R5 = R8 >> 3;
        uint16_t G6 = G8 >> 2;
        uint16_t B5 = B8 >> 3;

        dst[i] = (R5 << 11) | (G6 << 5) | B5;
    }
}
```

4. This process is repeated for the Green and Blue channels sequentially.

This modular approach allows us to reuse the robust Otsu logic developed in Q1 without rewriting the core algorithm.

```
void Apply_Otsu_On_ColorImage_RGB565(uint16_t* pRGB565,
    uint32_t num_pixels,
    uint8_t* tempImg1,
    uint8_t* tempImg2)
{
    const char channels[3] = { 'R', 'G', 'B' };

    for (int c = 0; c < 3; c++)
    {
        // 1) İlgili kanalı tempImg1 buffer'ına çıkar
        Extract_Channel_From_RGB565(pRGB565,
            tempImg1,
            num_pixels,
            channels[c]);

        // 2) Otsu ile binary yap (Sonuç tempImg2'ye)
        apply_otsu_threshold(tempImg1,
            tempImg2,
            num_pixels);
```

```
        // 3) Binary kanalı RGB565 görüntüye geri yaz
        Write_Channel_To_RGB565(pRGB565,
            tempImg2,
            num_pixels,
            channels[c]);
    }
}
```

Figure 2 Original Color Image vs. Color Otsu Result

3. QUESTION

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

We implemented four fundamental morphological operations. To ensure data integrity, "Opening" and "Closing" operations utilize a temporary buffer to store intermediate results, preventing the overwriting of source data before the second stage is complete.

1. **Dilation (Max Filter):** The dilation function iterates through the image with a 3×3 kernel. It searches for the maximum pixel value within the neighborhood. For binary images, if any neighbor is White (255), the center pixel becomes White.

```
void apply_dilation_gray_7x7(const uint8_t* in,
    uint8_t* out,
    uint32_t width,
    uint32_t height)
{
    for (uint32_t y = 0; y < height; y++)
    {
        for (uint32_t x = 0; x < width; x++)
        {
            uint8_t maxVal = 0;

            // 3x3 çekirdek taraması
            for (int32_t ky = -3; ky <= 3; ky++)
            {
                int32_t yy = y + ky;
                if (yy < 0 || yy >= (int32_t)height) continue;

                for (int32_t kx = -3; kx <= 3; kx++)
                {
                    int32_t xx = x + kx;
                    if (xx < 0 || xx >= (int32_t)width) continue;

                    uint8_t val = in[yy * width + xx];
                    if (val > maxVal)
                        maxVal = val;
                }
            }

            out[y * width + x] = maxVal;
        }
    }
}
```

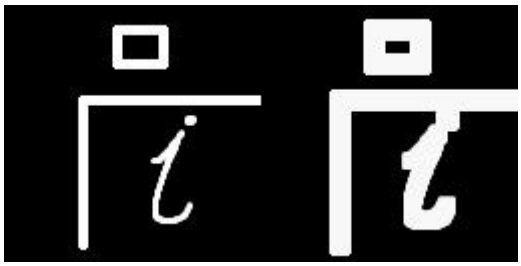

Figure 3 Dilation Implementation

2. **Erosion (Min Filter):** The erosion function works similarly but searches for the minimum pixel value. If any neighbor is Black (0), the center pixel becomes Black. This effectively shrinks bright regions.

```
apply_erosion_gray_7x7(const uint8_t* in,
    uint8_t* out,
    uint32_t width,
    uint32_t height)
{
    for (uint32_t y = 0; y < height; y++)
    {
        for (uint32_t x = 0; x < width; x++)
        {
            uint8_t minVal = 255;

            // 3x3 komşuluk taraması
            for (int32_t ky = -2; ky <= 2; ky++)
            {
                int32_t yy = y + ky;
                if (yy < 0 || yy >= (int32_t)height) continue;

                for (int32_t kx = -2; kx <= 2; kx++)
                {
                    int32_t xx = x + kx;
                    if (xx < 0 || xx >= (int32_t)width) continue;

                    uint8_t val = in[yy * width + xx];
                    if (val < minVal)
                        minVal = val;
                }
            }

            out[y * width + x] = minVal;
        }
    }
}
```

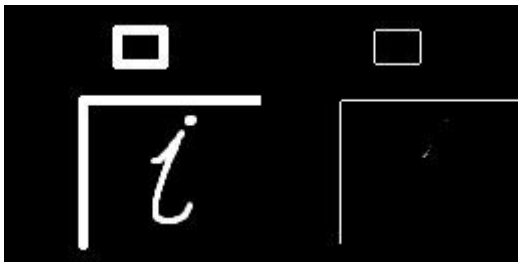

Figure 4 Erosion Implementation

3. **Opening and Closing (Composite Operations):** These functions chain erosion and dilation. A static temporary buffer (temp) is used to hold the result of the first operation.

```
void apply_opening_gray_7x7(const uint8_t* in, uint8_t* out, uint32_t width, uint32_t height)
{
    static uint8_t temp[128 * 128]; // Intermediate buffer
    apply_erosion_gray_7x7(in, temp, width, height); // Step 1: Erosion
```

```
    apply_dilation_gray_7x7(temp, out, width, height); // Step 2: Dilation
}

void apply_closing_gray_7x7(const uint8_t* in, uint8_t* out, uint32_t width, uint32_t height)
{
    static uint8_t temp[128 * 128]; // Intermediate buffer
    apply_dilation_gray_7x7(in, temp, width, height); // Step 1: Dilation
    apply_erosion_gray_7x7(temp, out, width, height); // Step 2: Erosion
}
```

Figure 5 Opening and Closing Implementation

4. SYSTEM ARCHITECTURE AND CODE ORGANIZATION

To manage the increasing complexity of the project (multiple homework questions, image processing algorithms, and communication protocols), we adopted a **Modular Architecture** that prioritizes code reusability and stability.

a) Modular Design

Instead of writing all logic inside the main loop, we created a dedicated application layer (app_homeworks.c and .h).

1. **Separation of Concerns:** The core image processing algorithms (e.g., Otsu, Morphological Filters) are encapsulated in image_processing.c, while the homework execution logic (receiving data, calling the algorithm, sending data back) is managed in app_homeworks.c.
2. **Buffer Management:** Large image buffers are passed as pointers to these functions. This approach avoids excessive use of global variables and prevents stack overflow issues by reusing the same memory areas for different tasks.

b) Task Selection Mechanism

To switch between different homework questions (Q1, Q2, Q3), we utilized a flexible **Configuration-Based Approach** in main.c.

1. **Static Configuration:** A global variable currentMode or direct function calls within the main loop determine the active task.
2. **Focused Debugging:** By setting the active mode in the code before compilation, we can isolate and test specific algorithms (e.g., running only "Otsu Color" or "Morphological Dilation"). This ensures that the MCU resources are fully dedicated to the target operation without the overhead of complex runtime switching logic.

Example Execution Loop

```
while (1)
{
    // The active task is selected via the currentMode variable
    switch (currentMode)
    {
    case MODE_Q1_OTSU_GRAY:
        App_Run_Q1_OtsuGray(&img, original_gray_image, processed_image, NUM_PIXELS);
        break;
    case MODE_Q2_OTSU_COLOR:
        App_Run_Q2_OtsuColor(&img, (uint16_t*)pImage_RGB565, original_gray_image,
processed_image, NUM_PIXELS);
        break;
        // Other cases...
    }
}
```