MARMARA UNIVERSITY
ENGINEERING FACULTY

EE 4065

Introduction to Embedded Digital Image Processing

Homework 2 Report

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

1. Question

a) Histogram Calculation Function

We implemented the calculate_histogram function with a focus on efficient memory use, ensuring our large image array stays in **Flash Memory** while only the essential counters use **RAM**.

We used four main variables:

image_data: This is a **const uint8_t*** (pointer to an 8-bit unsigned integer). We used **const** to store the massive image array in **Read-Only Flash Memory**, avoiding costly RAM usage. It holds the 0-255 pixel values.

num_pixels: A **uint32_t** variable that gives the total count of pixels in the image. It controls the main loop, ensuring we iterate over every single pixel.

histogram_output: This is a **uint32_t*** pointer. It points to the 256-element array that holds the final counts. The use of **uint32_t** guarantees the counter can handle well over the 76,800 pixels we have. Critically, this array must be in **RAM** because its values are constantly incremented.

pixel_value: A temporary **uint8_t** variable used inside the loop. It captures the current pixel's brightness (0-255).

The function's speed comes from its **Direct Addressing** technique: it uses the pixel_value directly as the index to increment the corresponding counter in the output array (histogram_output[pixel_value]++;), making the process a single, fast operation per pixel.

```
void calculate_histogram(const uint8_t* image_data, uint32_t num_pixels, uint32_t* histogram_output)
{
    // 1. Adım: Histogram dizisini sıfırlama
    // 0-255 arası her piksel değeri için bir sayaç
    for (int i = 0; i < 256; i++)
    {
        histogram_output[i] = 0;
    }

    // 2. Adım: Görüntüyü tara ve sayaçları artır
    for (uint32_t i = 0; i < num_pixels; i++)
    {
        // Mevcut pikselin parlaklık değerini al (0-255)
        uint8_t pixel_value = image_data[i];

        // Bu parlaklık değerine karşılık gelen sayacı artır
        histogram_output[pixel_value]++;
    }
}
```

Figure 1- Histogram Calculation Function

b) Histogram Calculation for Grayscale Image

Our approach shifts the burden of large image storage from the resource-limited MCU RAM to a **dynamic, continuous data transfer** system between the **PC (running Python)** and the **Microcontroller (MCU)**.

We moved away from using const and extern for static Flash storage. Instead, the process relies on a robust communication loop:

Fixed Data Size: To ensure predictable RAM usage and constant processing time on the MCU, any image taken by the PC is first **resized to a fixed 128 X 128 pixels** in grayscale format. This standardizes the data payload to exactly **16,384 bytes**.

Continuous Loop: The program runs in an infinite loop, managing the data flow:

- The **PC sends** the 16,384 byte image to the MCU's input buffer.
- The **MCU processes** the image data using the applied filters or algorithms.
- The **MCU sends** the 16,384 byte processed image back to the PC's output buffer.

PC Visualization: Upon receiving the processed data, the PC **saves the raw bytes as a .png file** with a special name. This allows us to observe the results of the MCU's work in real-time.

This dynamic system minimizes the MCU's RAM commitment (32 KB for two image buffers), preventing the risk of memory overflow while enabling continuous, real-time image processing.

&*img_hedef.pData <Unsigned Integer>

Address	0
20000238	00
20000239	00
2000023A	00
2000023B	00
2000023C	00
2000023D	00
2000023E	00
2000023F	00
20000240	00
20000241	00
20000242	FF

Figure 2 - Pixel Values of Histogram Calculated Image in Memory as Unsigned Integer

2. Question

a) Histogram Equalization Method

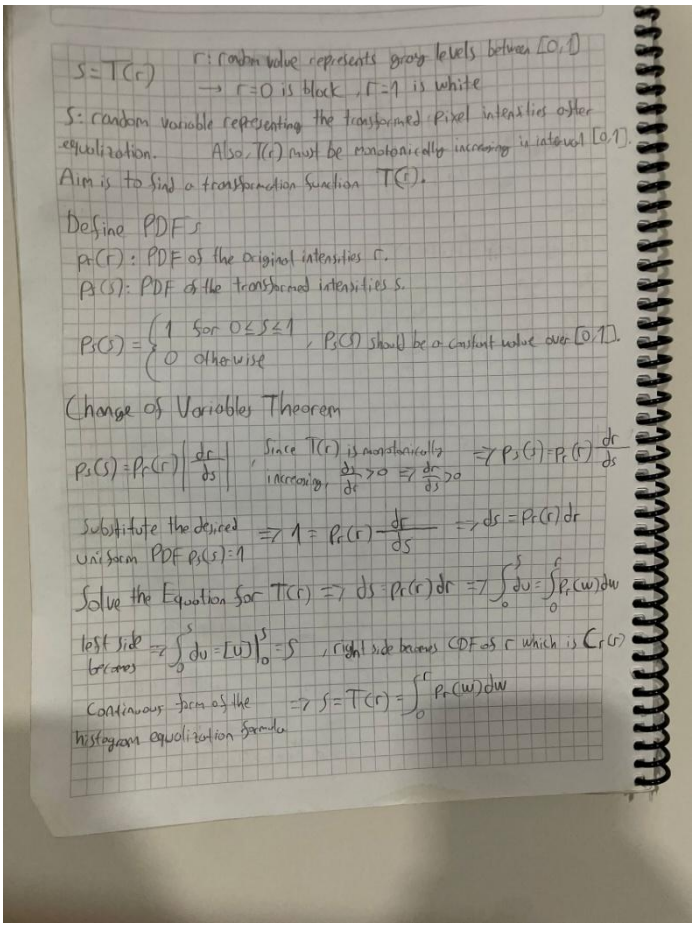

Figure 3 - Deriving the Histogram Equalization Method

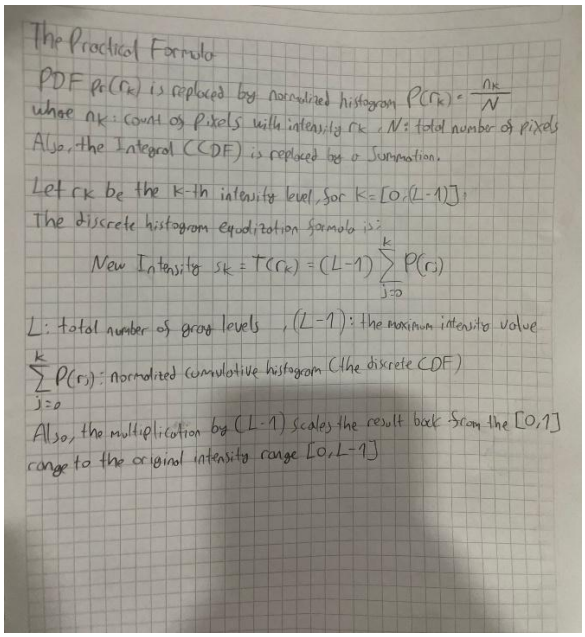

Figure 4 - Practical Formula of Histogram Equalization

Before going further, the next steps include memory snapshots of the grayscale image before and after processing. To show the difference between these values, we will add some pixels' grayscale values beforehand for comparison purposes.

original_gray_image : 0x2000C070 <Unsigned Integer>

Address	0
2000C070	84
2000C071	62
2000C072	73
2000C073	67
2000C074	92
2000C075	54
2000C076	128
2000C077	105
2000C078	126
2000C079	96
2000C07A	109
2000C07B	146

Figure 5 Grayscale Values before processings

b) Histogram Equalization Function

The apply_histogram_equalization function enhances image contrast by redistributing the pixel intensities, using the calculated histogram (histogram_input) and the image arrays (original_image, equalized_image).

The process relies on calculating the **Cumulative Distribution Function (CDF)** and building a **Lookup Table (LUT)** to map old pixel values to new ones.

First, we calculate the **CDF** iteratively, summing the counts from the histogram_input array into the temporary cdf[256] array.

Next, we create the **256-element LUT** (lookup_table[256]) by applying the core equalization formula to the CDF values. For optimization, the **scale_factor** (255.0f / num_pixels) is pre-calculated. We use a **+ 0.5f** offset before casting to **uint8_t** to ensure correct mathematical rounding during the mapping.

Finally, we iterate through every pixel of the original_image. We use the original pixel's value as a direct index to fetch its new, contrast-enhanced value from the lookup_table, writing the result into the **equalized_image** array. This single-pass lookup ensures rapid image transformation.

```
void apply_histogram_equalization(const uint8_t* original_image,
                                  uint8_t* equalized_image,
                                  uint32_t* histogram_input,
                                  uint32_t num_pixels)
{
    // Adım 1: Histogramdan CDF (Kümülatif Dağılım Fonksiyonu) hesapla
    // (Teorideki (1/N) * Toplam(n_j) kısmı)
    uint32_t cdf[256];
    cdf[0] = histogram_input[0];
    for (int i = 1; i < 256; i++)
    {
        cdf[i] = cdf[i - 1] + histogram_input[i];
    }

    // Adım 2: Dönüsüm için Arama Tablosu (Lookup Table - LUT) olustur
    // Bu, Q2a'daki s_k = 255 * (1/N) * CDF(k) formülünü uygular.
    // (Optimizasyon: Ölçekleme faktörünü (255.0f / num_pixels) önceden hesapla)
    uint8_t lookup_table[256];
    float scale_factor = 255.0f / (float)num_pixels;

    for (int i = 0; i < 256; i++)
    {
        // (float)cdf[i] * scale_factor -> Bu, (L-1) * CDF(k) işlemini yapar
        // + 0.5f eklemek, en yakın tam sayıya doğru yuvarlama (rounding) sağlar.
        lookup_table[i] = (uint8_t)( (float)cdf[i] * scale_factor + 0.5f );
    }

    // Adım 3: LUT kullanarak yeni görüntüyü olustur
    // Orijinal görüntüdeki her pikseli (değer: v) al
    // ve yeni görüntüye lookup_table[v] değerini yaz.
    for (uint32_t i = 0; i < num_pixels; i++)
    {
        equalized_image[i] = lookup_table[ original_image[i] ];
    }
}
```

Figure 6 - Histogram Equalization Function

c) Histogram Equalization Applied Grayscale Image

The **histogram equalization applied image** aims to enhance the overall **contrast** by remapping the original image's pixel values (0-255) to a more **uniform** distribution.

This transformation results in two primary differences from the original grayscale image:

> **Increased Contrast and Detail:** The image will appear **sharper** because the process spreads out the gray tones. In the original image, if many pixels were clustered in a narrow range (low contrast), the equalized image will assign these clustered values to a wider range of 0-255 tones. This makes previously **hidden details** in dark or bright areas more visible and distinct.
>
> **Flattened Histogram:** The histogram of the new image will be **flatter and wider**. While the original histogram likely had tall, narrow peaks, the new distribution, ideally, uses the full range of 0-255 more evenly. This efficient use of the entire **dynamic range** is what ultimately boosts the visual contrast.

In essence, the equalized image looks **more dynamic** because the function forces the pixels to occupy more of the available brightness space.

Figure 7 - Histogram Equalization Applied Grayscale Image

processed_image : 0x20008470 <Unsigned Integer>

Address	0
20008470	32
20008471	12
20008472	21
20008473	16
20008474	44
20008475	7
20008476	132
20008477	67
20008478	126
20008479	50
2000847A	75
2000847B	177

Figure 8 - Pixel Values of Histogram Equalized Image in Memory as Unsigned Integer

3. Question

a) 2D Convolution Function

The **apply_2d_convolution** function processes the **original_image** to create a **filtered_image** using a **3x3 kernel**. The implementation focuses on three key processes: iteration, boundary control, and the convolution mechanism itself.

The function iterates over every pixel in the image using nested

loops (**for y**, **for x**). For each image pixel, we use a second set of nested loops (**for ky**, **for kx**) to traverse the neighboring pixels covered by the 3x3 kernel.

A crucial part is **Boundary Handling**: we implement **Zero Padding** by checking if the neighbor coordinates fall outside the image bounds. If they do, we simply skip them (**continue**), which mathematically treats them as zero and ensures the edge pixels are correctly processed.

The core operation is **Convolution**. To correctly perform convolution (as opposed to correlation), we explicitly **flip the kernel** by calculating mirrored coordinates (**kx_flipped = -kx, ky_flipped = -ky**). We then multiply the value of the original pixel by the corresponding **flipped** kernel element and accumulate the result into a **sum**.

Finally, after the weighted sum is complete, the float result is **clamped** to the valid 0 to 255 range. The final step writes the result to the **filtered_image** array after adding **+ 0.5f** to ensure proper rounding when casting the float sum back to an **uint8_t pixel** value.

```c
void apply_2d_convolution(const uint8_t* original_image, uint8_t* filtered_image,int width,int height, const float* kernel)
{
    // Görüntünün her pikseli (satır satır) üzerinde gezin
    for (int y = 0; y < height; y++)
    {
        for (int x = 0; x < width; x++)
        {
            // Bu (x, y) pikseli için konvolüsyon toplamını hesapla
            float sum = 0.0f;
            // 3x3'lük kernel üzerinde gezin (-1'den +1'e)
            for (int ky = -1; ky <= 1; ky++)
            {
                for (int kx = -1; kx <= 1; kx++)
                {
                    // Komşu pikselin koordinatını bul
                    int pixel_x = x + kx;
                    int pixel_y = y + ky;

                    // KENAR KONTROLÜ (Zero Padding)
                    if (pixel_x < 0 || pixel_x >= width ||
                        pixel_y < 0 || pixel_y >= height)
                    {
                        continue; // Sınır dışı ise 0 say, toplama katkı yok
                    }
                    // Komşu pikselin 1D dizideki indisini bul
                    int image_index = pixel_y * width + pixel_x;
                    // * ÖNEMLİ FARK: GERÇEK KONVOLÜSYON İÇİN KERNEL'İ FLIP ET *
                    // Korelasyon: kernel_index = (ky + 1) * 3 + (kx + 1);
                    // Konvolüsyon: kernel_index = (-ky + 1) * 3 + (-kx + 1);
                    int kx_flipped = -kx;
                    int ky_flipped = -ky;
                    int kernel_index = (ky_flipped + 1) * 3 + (kx_flipped + 1);
                    // Toplama ekle
                    sum += (float)original_image[image_index] * kernel[kernel_index];
                }
            }
            // Sonucu [0, 255] aralığına sıkıştır (Clamp)
            if (sum < 0.0f)   sum = 0.0f;
            if (sum > 255.0f) sum = 255.0f;
            // Sonucu çıktı görüntüsüne yaz (yuvarlama için +0.5f)
            filtered_image[y * width + x] = (uint8_t)(sum + 0.5f);
```

Figure 9 - 2D Convolution Function

```c
// Q3 Filtre Kernelleri (Bunlar Flash'ta saklanır, RAM kullanmaz)
// Q3b: Low-Pass Filtre (Averaging / Box Blur)
const float kernel_low_pass[9] = {
    1.0f/9.0f, 1.0f/9.0f, 1.0f/9.0f,
    1.0f/9.0f, 1.0f/9.0f, 1.0f/9.0f,
    1.0f/9.0f, 1.0f/9.0f, 1.0f/9.0f
};

// Q3c: High-Pass Filtre (Laplacian - Kenar Tespiti)
const float kernel_high_pass[9] = {
     0.0f, -1.0f,  0.0f,
    -1.0f,  4.0f, -1.0f,
     0.0f, -1.0f,  0.0f
};
```

Figure 10 - Low-Pass and High-Pass Filters

b) Low-Pass Filtering Application

This filter, defined as kernel_low_pass, contains all elements set to **1/9** (whose sum is 1.0). Its purpose is **smoothing** or **blurring**. It works by calculating the **average** of the 9 pixels in the neighborhood, which effectively reduces high-frequency content like **sharp details** and **noise** in the image.

Figure 11 - Low-Pass Filter Applied Grayscale Image

processed_image : 0x20008070 <Unsigned Integer>

Address	0
20008070	35
20008071	52
20008072	48
20008073	50
20008074	45
20008075	58
20008076	62
20008077	76
20008078	74
20008079	78
2000807A	86
2000807B	85

Figure 12 - Pixel Values of Low-Pass Filter Applied Image in Memory as Unsigned Integer

c) High-Pass Filtering Application

This kernel, kernel_high_pass, is a **Laplacian** operator with a central value of **4.0** and surrounding elements of **-1.0** (the sum is 0.0). Its purpose is **edge detection**. It responds strongly to rapid changes in pixel intensity, highlighting the boundaries (edges) in the image while suppressing low-frequency (smooth) areas.

Figure 13 - High-Pass Filter Applied Grayscale Image

processed_image : 0x20008070 <Unsigned Integer>

Address	0
20008070	182
20008071	14
20008072	80
20008073	35
20008074	177
20008075	0
20008076	230
20008077	73
20008078	196
20008079	14
2000807A	69
2000807B	215

Figure 14 - Pixel Values of High-Pass Filter Applied Image in Memory as Unsigned Integer

4. Question

a) Median Filtering Function

The apply_median_filtering function reduces noise in the original_image by replacing each pixel with the **median** value of its 3x3 neighborhood, storing the result in the filtered_image.

The operation is performed using three main steps:

Window Population and Boundary Handling: We iterate over every pixel in the image. For each pixel, we fill the temporary 9-element array, **window[9]**, with the values of the 3×3 neighborhood pixels. For pixels outside the image boundaries, we implement **Zero Padding**, treating them as a black pixel value of **0**. This ensures the edges of the image are processed consistently.

Sorting and Median Calculation: Once the window is populated, we call a separate function (sort_9_elements(window)) to sort the 9 pixel values in ascending order. The **median value** is then simply the middle element of the sorted array, which corresponds to the **5th element** at index **window[4]** (since we are zero-indexed).

```
static void sort_9_elements(uint8_t* arr)
{
    int i, j;
    uint8_t key;
    for (i = 1; i < 9; i++) {
        key = arr[i];
        j = i - 1;

        // key'den büyük olan elemanları bir pozisyon ileri kaydır
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Figure 15 - Sorting Function for Median Filtering Algorithm

Output: Finally, this calculated **median_value** replaces the original pixel's value and is written to the corresponding location in the filtered_image array. Median filtering is highly effective at removing **salt-and-pepper noise** while preserving edges better than standard averaging filters.

```
void apply_median_filtering(const uint8_t* original_image, uint8_t* filtered_image,int width,int height)
{
    // 3x3'lük penceredeki pikselleri geçici olarak tutacak dizi
    uint8_t window[9];

    // Görüntünün her pikseli (satır satır) üzerinde gezin
    for (int y = 0; y < height; y++)
    {
        for (int x = 0; x < width; x++)
        {
            // 1. Adım: 3x3'lük pencereyi doldur
            int i = 0; // window dizisinin indeksi
            for (int ky = -1; ky <= 1; ky++)
            {
                for (int kx = -1; kx <= 1; kx++)
                {
                    int pixel_x = x + kx;
                    int pixel_y = y + ky;

                    // KENAR KONTROLÜ (Zero Padding)
                    // Görüntü dışındaysak 0 (siyah) olarak kabul et
                    if (pixel_x >= 0 && pixel_x < width && pixel_y >= 0 && pixel_y < height)
                    {
                        window[i] = original_image[pixel_y * width + pixel_x];
                    }
                    else
                    {
                        window[i] = 0;
                    }
                    i++;
                }
            }

            // 2. Adım: Penceredeki 9 pikseli sırala
            sort_9_elements(window);

            // 3. Adım: Ortanca değeri (median) bul (9 elemanlı dizide 5. eleman)
            // İndeks 0'dan başladığı için [4] olur.
            uint8_t median_value = window[4];

            // 4. Adım: Sonucu çıktı görüntüsüne yaz
            filtered_image[y * width + x] = median_value;
        }
```

Figure 16 - Median Filtering Function

b) Median Filtering Applied Grayscale Image

Median filtering is a **non-linear** process used to reduce **noise** while attempting to preserve image edges.

The **"median filtering applied image"** will primarily show two key characteristics compared to the original:

Noise Reduction: The image will appear **smoother** and cleaner. Isolated, extreme pixel values, typical of **salt-and-pepper noise**, will be largely eliminated. This occurs because the median operation forces outlier values to the beginning or end of the sorted window, making them irrelevant to the median (middle) value chosen for replacement.

Edge Preservation: Unlike blurring filters, median filtering effectively **preserves sharp edges**. The filter is known for its ability to smooth out noise without significantly blurring boundaries between objects, resulting in an image that is both clean and maintains good structural integrity.

In essence, the filtered image will look **cleaner and less granular** than the original, having successfully suppressed random noise without the typical side effect of significant edge blurring.

Figure 17 - Median Filter Applied Grayscale Image

processed_image : 0x20008070 <Hex>

Address	0
20008070	00
20008071	49
20008072	43
20008073	44
20008074	36
20008075	36
20008076	36
20008077	69
20008078	60
20008079	6B
2000807A	6D
2000807B	6D

Figure 18 - Pixel Values of Median Filter Applied Image in Memory as Unsigned Integer