MARMARA
UNIVERSITY

Embedded Digital Image Processing

EE4065

Homework 2

Mustafa Öztürk

150722016

Emirhan Şener

150721069

Faculty of Engineering
Electrical Electronic Engineering

Fall 2025

ACRONYMS

2D 2-dimensional

EE4065 Embedded Digital Image Processing

HPF High-pass filter

LPF Low-pass filter

LUT Look-up Table

RAM Random Access Memory

UART Universal Asynchronous Receiver Transmitter

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

This report is for the second assignment in the Embedded Digital Image Processing (EE4065) course. The main objective of this homework was to implement several fundamental image processing algorithms in C, running them directly on our NUCLEO-F446RE microcontroller. This assignment builds on our previous work by moving from simply loading data to actively transforming it.

For this homework, we used a dynamic data transfer method, thanks to one of the groups taking this class [1]. Instead of compiling a static header file into the program, our STM32 application receives a grayscale image from a PC Python script via Universal Asynchronous Receiver Transmitter (UART) at runtime. Once the image is loaded into the microcontroller's Random Access Memory (RAM), the processing functions are applied. This system allows us to test our algorithms much more quickly and flexibly.

This report is structured around the four main tasks assigned in the homework document. We will first present our C function for calculating an image's histogram (Q1). Next, we will cover the theory and implementation of histogram equalization (Q2). Following that, we demonstrate 2-dimensional (2D) convolution by applying both low-pass and high-pass filters (Q3). Finally, we implement a median filter (Q4). For each part, we provide our C code and the results, which we verified both visually (by sending the processed image back to the PC) and numerically (by observing the memory entries in the STM32CubeIDE[2] debugger).

2. PROBLEMS

2.1. Q-1) HISTOGRAM FORMATION

(20 points) This question is on histogram formation.

(a) Form a C function on the microcontroller to calculate histogram of a given grayscale image.

(b) Form a grayscale image of your choice with appropriate size on PC. Store it as a header file. Then, add this header file to your new project. Calculate its histogram. Show histogram entries (at least some of them) on STM32Cube IDE.

Theory

The histogram of an image is a fundamental tool in digital image processing. It represents the distribution of pixel intensities (brightness levels) within the image. For an 8-bit grayscale image, the histogram is an array of 256 integers, where each entry $h(k)$ (for k from 0 to 255) stores the total count of pixels in the image that have the intensity value k.

Mathematically, this is represented by:

$$h(g_k) = n_k$$

where g_k is the k-th intensity level and n_k is the number of pixels with that intensity. Analyzing this histogram allows us to understand the image's contrast and brightness. An image with a histogram bunched up in one area typically has low contrast.

Procedure

First, the STM32CubeIDE project was configured to receive data via UART (at 2,000,000 baud) using the NUCLEO-F446RE's COM port. A Python script sends the 128x128 grayscale image, which is received by the microcontroller and stored in a `volatile uint8_t` buffer named `pImage`.

To fulfill part (a), the following C function, `calcHistogram`, was created. This function iterates through the entire image array one time. For each pixel it reads, it increments the corresponding counter in the `histogram_data` array.

Listing 2.1: C function to calculate the image histogram.

```
void calcHistogram(uint8_t* pIn, uint32_t* pHist, uint32_t imgSize){
    for (int i = 0; i < 256; i++) {
        pHist[i] = 0;
    }

    for (int i = 0; i < imgSize; i++) {
        pHist[pIn[i]]++;
    }
}
```

This function was called in `main.c` immediately after the image was successfully received from the PC, populating the 256-element `histogram_data` array.

Results

To satisfy the requirement of part (b), we verified the result numerically. After running the program and pausing the debugger, we inspected the `histogram_data` array directly in the STM32CubeIDE's "Expressions" window. As shown in Figure 2.1, the array was successfully populated with the pixel counts for each intensity level.

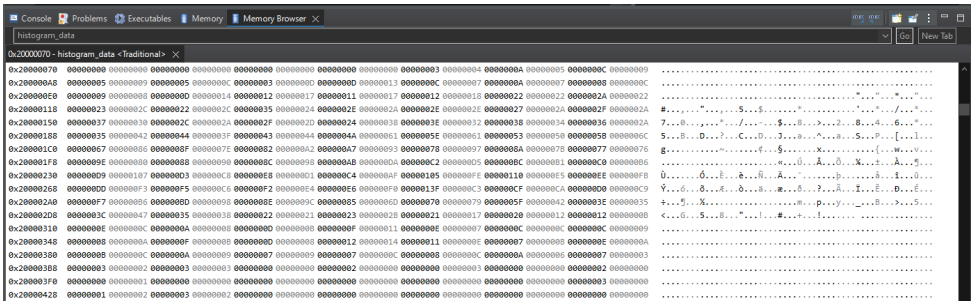


Figure 2.1: Histogram entries for the original image

To better visualize the result for this report, we also exported the 1024 bytes of the `histogram_data` array as a `.bin` file from the Memory Browser. This binary file was then

read by a Python script, which used Matplotlib[3] to generate the complete histogram plot shown in Figure 2.2. The graph clearly shows the pixel distribution of the original image with most pixels concentrated in the mid-to-high gray values indicating low contrast.

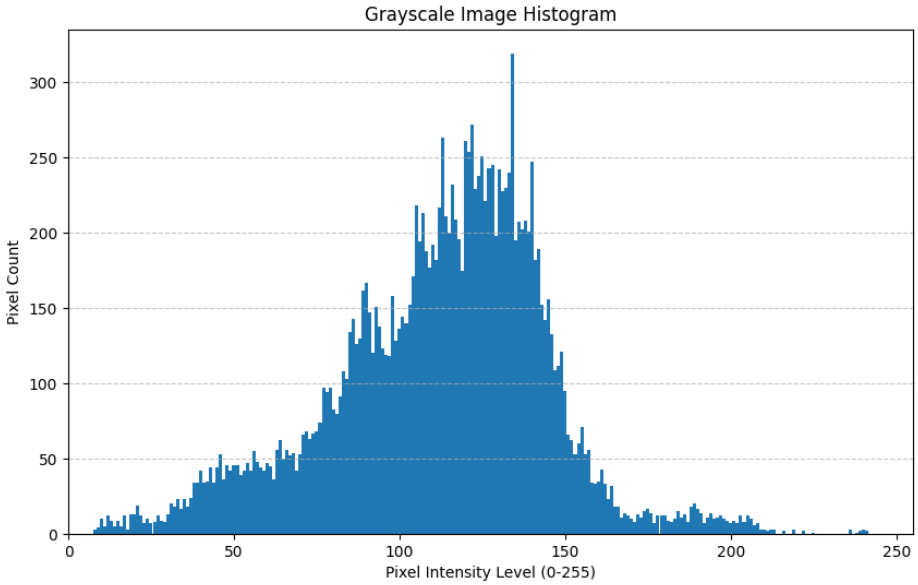


Figure 2.2: Full histogram of the original image, plotted using data exported from the microcontroller.

Also below is the python script that's used to convert a .bin file into a histogram plot:

Listing 2.2: Python script for plotting a binary file into a histogram

```
# plot_histogram.py
import numpy as np
import matplotlib.pyplot as plt
import os

BIN_FILE_NAME = "histogram.bin"
PLOT_FILE_NAME = "histogram_plot.png"
EXPECTED_ELEMENTS = 256

def plot_histogram_from_bin(bin_file):
    if not os.path.exists(bin_file):
        print(f"Error: '{bin_file}' not found.")
        return

    hist_data = np.fromfile(bin_file, dtype=np.uint32)

    if hist_data.size != EXPECTED_ELEMENTS:
        print(f"Error: Read {hist_data.size} elements, but expected {
            EXPECTED_ELEMENTS}.")
        return

    print(f"Successfully read {hist_data.size} histogram bins.")

    # Create an array for the x-axis (0 to 255)
    pixel_levels = np.arange(EXPECTED_ELEMENTS)

    # Plot the histogram
```

```
    plt.figure(figsize=(10, 6))
    plt.bar(pixel_levels, hist_data, width=1.0)
    plt.title("Grayscale Image Equalized Histogram")
    plt.xlabel("Pixel Intensity Level (0-255)")
    plt.ylabel("Pixel Count")
    plt.grid(axis='y', linestyle='--', alpha=0.7)
    plt.xlim([0, 255])


    plt.savefig(PLOT_FILE_NAME)
    print(f"Plot saved as '{PLOT_FILE_NAME}'")

    plt.show()

if __name__ == "__main__":
    plot_histogram_from_bin(BIN_FILE_NAME)
```

2.2. Q-2) HISTOGRAM EQUALIZATION

(30 points) This question is on histogram equalization.

(a) Derive the histogram equalization method by pencil and paper. Post your result here by taking the photo of your derivation on the paper.

(b) Form a C function on the microcontroller to apply histogram equalization on a given grayscale image.

(c) Use the grayscale image formed in the previous question. Apply histogram equalization to it. Calculate its histogram. Show histogram entries (at least some of them) on STM32Cube IDE.

a) Theory and Derivation

Histogram equalization is a method used to improve image contrast by "spreading out" the most frequent intensity values. The goal is to obtain a histogram that is more uniform, or "flatter." This is achieved by mapping the input pixel intensities using a transformation function based on the image's Cumulative Distribution Function (CDF).

As requested by the assignment derivation of this transformation method is shown in Figure 2.3. This derivation starts from the probability density function (PDF) of the input pixels, $p_r(g_k)$, and derives the final transformation function $s_k = T(g_k)$ that produces a uniform output distribution.

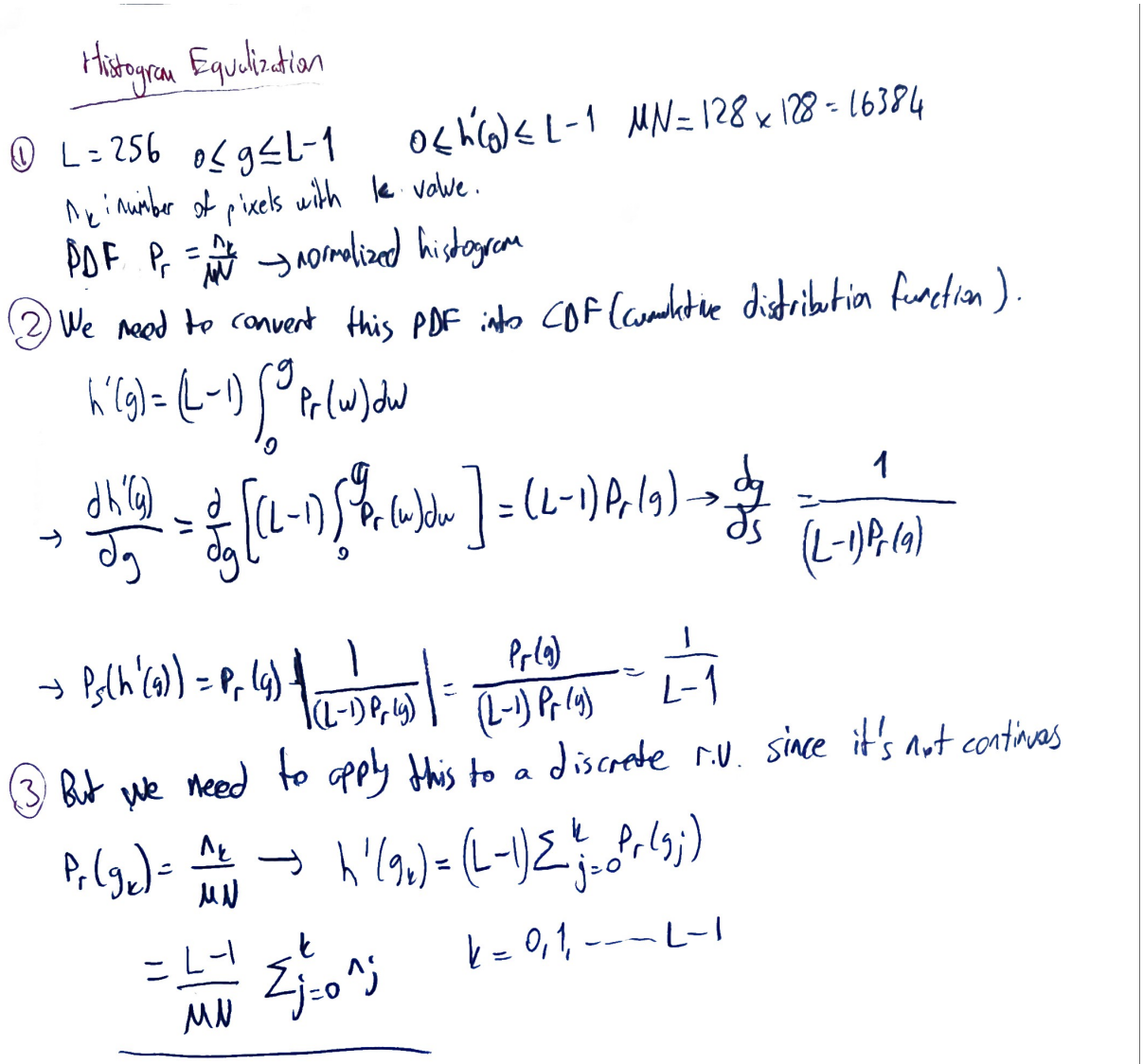


Figure 2.3: Mathematical derivation of the histogram equalization method.

b) C Function for Histogram Equalization

Based on the mathematical derivation, we formed the C function `equalHistogram`. This function implements the discrete formula $s_k = \frac{L-1}{MN} \sum_{j=0}^{k} n_j$.

First, it calls the `calcHistogram` function (from Q1) to get the n_k values. Second, it calculates the CDF ($\sum n_j$) into a temporary static array. Third, it calculates the `scale_factor` ($\frac{L-1}{MN}$). Finally, it creates a 256-element Look-up Table (LUT) where each new pixel value is pre-calculated. The main image processing loop then becomes very fast, simply mapping each input pixel to its new value via the LUT.

Listing 2.3: C function for histogram equalization.

```
void equalHistogram(uint8_t* pIn, uint8_t* pOut, uint32_t* pHist_in,
    uint32_t imgSize){
    calcHistogram(pIn, pHist_in, imgSize);

    static uint32_t cdf[256]; // keep cdf away from stack
    cdf[0] = pHist_in[0];
    for (int i = 1; i < 256; i++) {
        cdf[i] = cdf[i - 1] + pHist_in[i];
    }

    // Formula: s_k = ((L-1) / MN) * CDF(k)
    // L-1 = 255
    // MN = imgSize
    float scale_factor = 255.0f / (float)imgSize;

    uint8_t lut[256];
    for (int k = 0; k < 256; k++) {
        lut[k] = (uint8_t)(scale_factor * (float)cdf[k]);
    }

    for (int i = 0; i < imgSize; i++) {
        pOut[i] = lut[pIn[i]];
    }
}
```

c) Application and Results

This function was applied to the original image (`pImage`) and the result was saved in a new buffer, `pImage_Equalized`. This new, processed image was then transmitted back to the PC via UART for visual inspection. As shown in Figure 2.4, the resulting image has significantly higher contrast than the original, and the details in the darker areas are much more visible.

(a) Original Image **(b)** Equalized Image

Figure 2.4: Visual comparison of the original vs. histogram equalized image.

Finally, to fulfill the last part of the question, we calculated the histogram of this equalized image by calling `calcHistogram` again, this time on the `pImage_Equalized` buffer. The result was stored in `histogram_data_equalized`.

Figure 2.5 shows the entries of this new histogram. To better visualize this new distribution, we also exported the `histogram_data_equalized` array as a `.bin` file and plotted it. The resulting graph (Figure 2.6) clearly shows that the pixel intensities are now much more widely and uniformly distributed across the entire 0-255 range, confirming the success of the equalization algorithm.

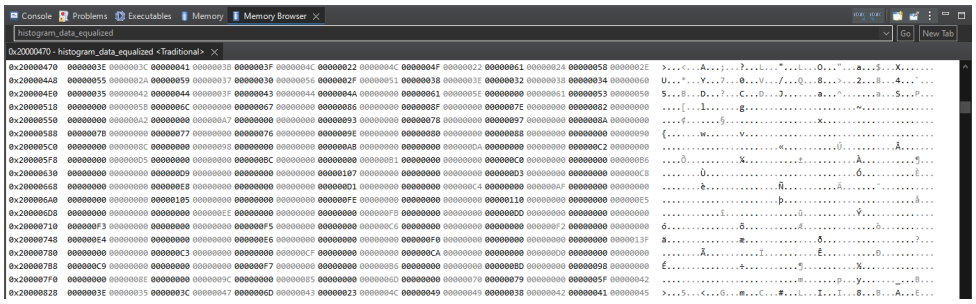


Figure 2.5: Histogram entries of the equalized image

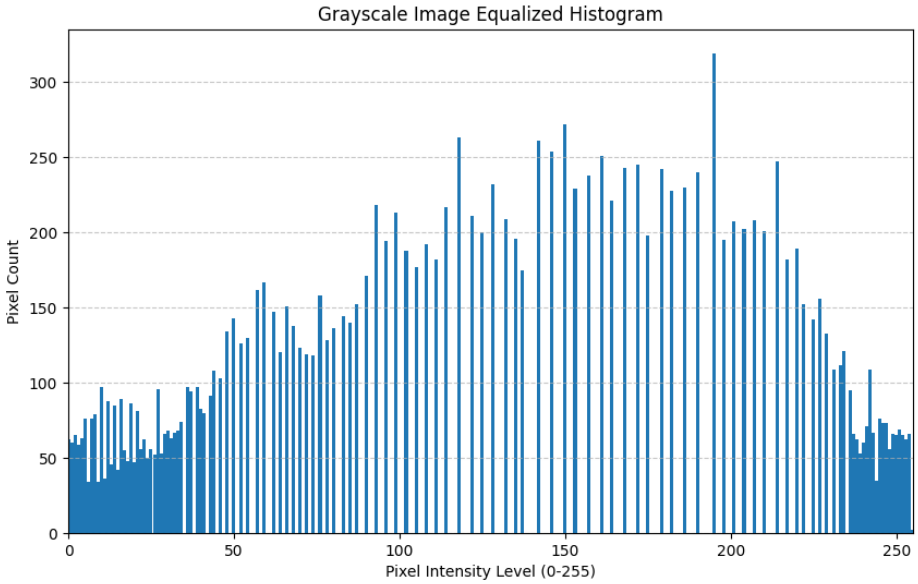


Figure 2.6: Full histogram plot of the equalized image.

2.3. Q-3) 2D CONVOLUTION AND FILTERING

(30 points) This question is on 2D convolution and filtering.

(a) Form a C function on the microcontroller to apply 2D convolution on a given grayscale image.

(b) Use the grayscale image formed in the previous question. Apply low pass filtering to it. Show filtered image entries (at least some of them) on STM32Cube IDE.

(c) Use the grayscale image formed in the previous question. Apply high pass filtering to it. Show filtered image entries (at least some of them) on STM32Cube IDE.

a) Theory and C Function

2D convolution is a fundamental operation in image processing. It works by sliding a small matrix called a "kernel" or "filter" over every pixel of the input image. The new value for the center pixel is calculated by taking a weighted sum of its neighbors where the weights are defined by the kernel. This function allows us to perform many different operations such as blurring, sharpening, and edge detection, simply by changing the kernel matrix.

As required by part (a), we created a C function named `convolve`. This function takes the input image (`pIn`), an output buffer (`pOut`), the image dimensions, a 9 element kernel array, and a divisor. It iterates through every pixel and applies the 3x3 kernel to calculate the new pixel value. The result is divided by the divisor and then clamped between 0 and 255 to ensure it remains a valid 8 bit pixel value.

Listing 2.4: C function for 2D convolution.

```
void convolve(uint8_t* pIn, uint8_t* pOut, int w, int h, const float*
    kernel, float divisor){
    float sum;
    int pixel_pos;

    for (int y = 1; y < h-1; y++) {
        for (int x = 1; x < w-1; x++) {

            sum = 0.0f;

            sum += (float)pIn[(y-1)*w + (x-1)] * kernel[0];
            sum += (float)pIn[(y-1)*w + (x)  ] * kernel[1];
            sum += (float)pIn[(y-1)*w + (x+1)] * kernel[2];

            sum += (float)pIn[(y)*w   + (x-1)] * kernel[3];
```

```
            sum += (float)pIn[(y)*w   + (x)  ] * kernel[4];
            sum += (float)pIn[(y)*w   + (x+1)] * kernel[5];

            sum += (float)pIn[(y+1)*w + (x-1)] * kernel[6];
            sum += (float)pIn[(y+1)*w + (x)  ] * kernel[7];
            sum += (float)pIn[(y+1)*w + (x+1)] * kernel[8];

            sum = sum / divisor;

            if (sum < 0)   sum = 0;
            if (sum > 255) sum = 255;

            pixel_pos = y*w + x;
            pOut[pixel_pos] = (uint8_t)sum;
        }
    }
}
```

b) Low Pass Filtering

A Low-pass filter (LPF), or mean filter, is used to blur an image and reduce high-frequency noise. It works by replacing each pixel's value with the average value of its 3x3 neighborhood.

Theory: The kernel used for a standard 3x3 mean filter is:

$$K_{LPF} = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Procedure: We defined this kernel as `KERNEL_LPF` in our code. We then called our generic `convolve` function, passing the original image as input, `pImage_LowPass` as the output buffer, and the LPF kernel.

Listing 2.5: Calling convolve for Low-Pass Filter.

```
// Defined globally:
// const float KERNEL_LPF[9] = { 1, 1, 1, 1, 1, 1, 1, 1, 1 };
// const float DIVISOR_LPF = 9.0f;

// Called in main():
convolve((uint8_t*)pImage, pImage_LowPass, IMG_WIDTH, IMG_HEIGHT,
    KERNEL_LPF, DIVISOR_LPF);
```

Results: The processed image was sent back to the PC via UART. The visual result in Figure 2.7 clearly shows that the original image has been blurred, and sharp details have been smoothed out.

(a) Original Image **(b)** Low-pass filtered Image

Figure 2.7: Visual comparison of the original vs. low pass filtered image.

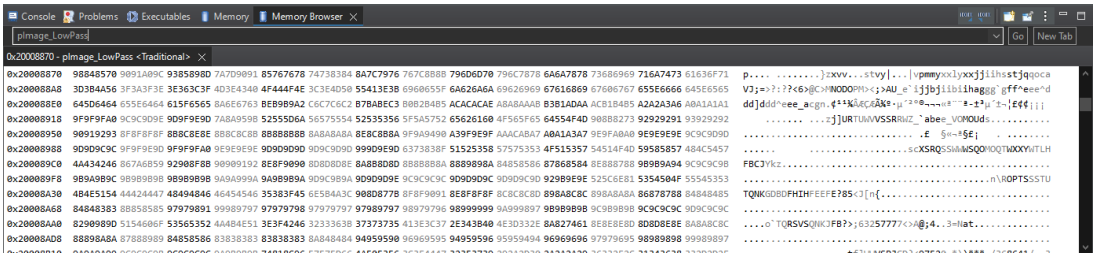


Figure 2.8: Memory browser view of the `pImage_LowPass` array.

c) High Pass Filtering

A High-pass filter (HPF) does the opposite of an LPF. It is used to sharpen an image or detect edges. It works by highlighting areas where pixel intensity changes sharply and suppressing areas that are uniform.

Theory: We used a Laplacian kernel, which is a common HPF for edge detection.

$$K_{HPF} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

This kernel calculates the difference between the center pixel and its surrounding neighbors. In a flat (non-edge) area, all pixels are similar, so the result is close to 0 (black). On an edge, the center pixel is very different, resulting in a high value (white).

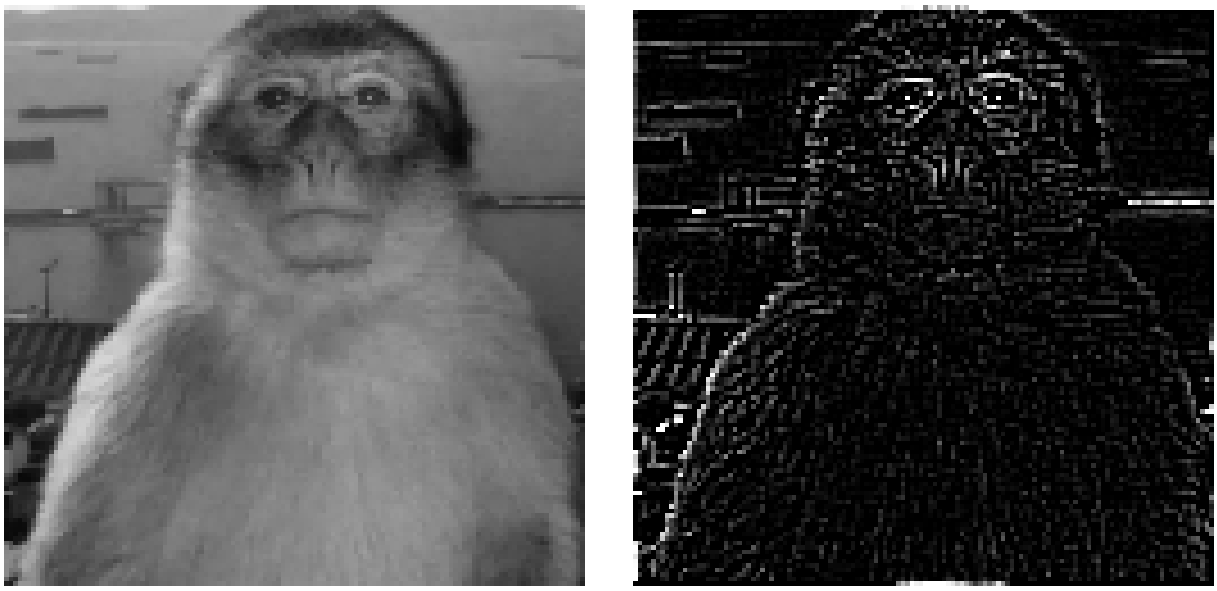
Procedure: We defined this kernel as `KERNEL_HPF` and called the `convolve` function, saving the result to the `pImage_HighPass` buffer.

Listing 2.6: Calling convolve for High-Pass Filter.

```
// Defined globally:
// const float KERNEL_HPF[9] = { -1, -1, -1, -1, 8, -1, -1, -1, -1 };
// const float DIVISOR_HPF = 1.0f;

// Called in main():
convolve((uint8_t*)pImage, pImage_HighPass, IMG_WIDTH, IMG_HEIGHT,
    KERNEL_HPF, DIVISOR_HPF);
```

Results: The resulting image sent back to the PC is shown in Figure 2.9. As expected the filter has removed all the flat surfaces and only left the bright lines that correspond to the edges in the original image.



(a) Original Image **(b)** High-pass filtered Image

Figure 2.9: Visual comparison of the original vs. high pass filtered image.

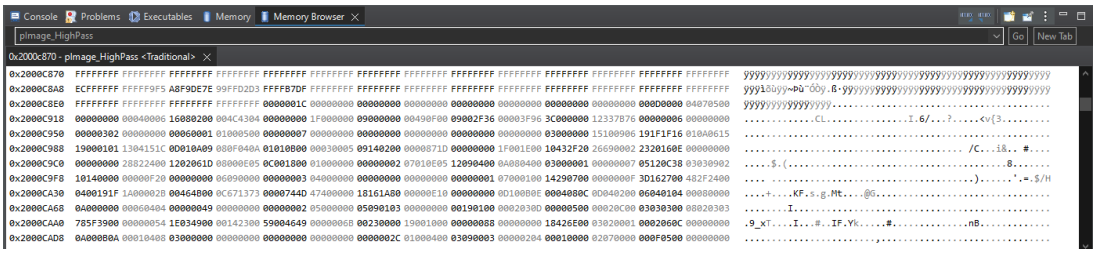


Figure 2.10: Memory browser view of the `pImage_HighPass` array

2.4. Q-4) MEDIAN FILTERING

(20 points) This question is on median filtering.

(a) Form a C function on the microcontroller to apply median filtering on a given grayscale image.

(b) Use the grayscale image formed in the previous question. Apply median filtering to it. Show filtered image entries (at least some of them) on STM32Cube IDE.

a) Theory and C Function

The median filter is a non-linear filtering technique and often used to remove noise from an image or signal. It preserves edges while removing noise, unlike linear smoothing filters (like the LPF in Q3) which tend to blur edges.

The main idea is to run through the signal entry by entry, replacing each entry with the median of neighboring entries. For 2D images, we use a window (kernel), typically 3x3. The 9 pixel values within the window are sorted numerically, and the value in the center of the sorted list (the 5th element) becomes the new value for the center pixel.

To implement this on the microcontroller, we first created a helper function, `sort`, which implements a simple bubble sort algorithm to sort an array of 9 elements.

Listing 2.7: Helper function to sort the 3x3 window array.

```
void sort(uint8_t arr[9]) {
    for (int i = 0; i < 9 - 1; i++) {
        for (int j = 0; j < 9 - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                uint8_t temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

Then, we implemented the main `medianFilter` function. This function iterates through the image pixels, collects the 3x3 neighborhood into a temporary `window` array, sorts it using our helper function and assigns the median value to the output image.

Listing 2.8: C function for Median Filtering.

```
void medianFilter(uint8_t* pIn, uint8_t* pOut, int w, int h)
{
    uint8_t window[9];
```

```
4
5       for (int y = 1; y < h - 1; y++) {
6           for (int x = 1; x < w - 1; x++) {
7
8               window[0] = pIn[(y-1)*w + (x-1)];
9               window[1] = pIn[(y-1)*w + (x)  ];
10              window[2] = pIn[(y-1)*w + (x+1)];
11              window[3] = pIn[(y)*w   + (x-1)];
12              window[4] = pIn[(y)*w   + (x)  ];
13              window[5] = pIn[(y)*w   + (x+1)];
14              window[6] = pIn[(y+1)*w + (x-1)];
15              window[7] = pIn[(y+1)*w + (x)  ];
16              window[8] = pIn[(y+1)*w + (x+1)];
17
18              sort(window);
19              pOut[y*w + x] = window[4];
20          }
21      }
22  }
```

b) Application and Results

We applied the `medianFilter` function to the original image and stored the result in the `pImage_Median` buffer. The processed image was transmitted back to the PC. The visual result is shown in Figure 2.11. Although the input image did not have significant noise the filter's smoothing effect is visible.

(a) Original Image **(b)** Median filtered Image

Figure 2.11: Visual comparison of the original vs. median filtered image.

The operation was also verified numerically. Figure 2.12 shows the memory contents of the `pImage_Median` array. By comparing this with the original data, we can observe that pixel values have been replaced by the median of their neighbors.

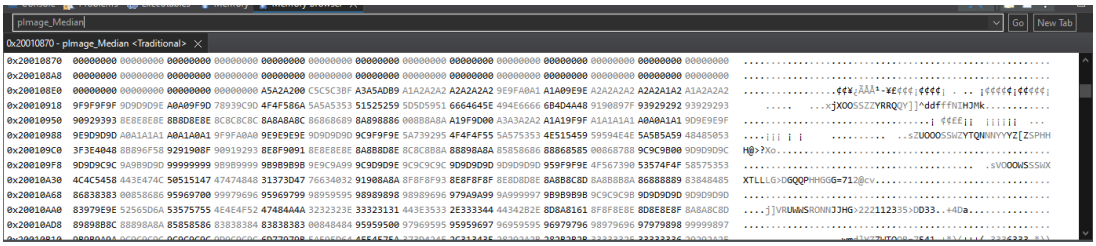


Figure 2.12: Memory browser view of the `pImage_Median` array.

3. CONCLUSION

In this assignment we implemented various image processing algorithms directly on the STM32 microcontroller. We switched from using static header files to a UART communication system. This allowed us to transfer images between the PC and the board in real-time. It made the process of testing our code much faster and more flexible.

We successfully completed four main processing tasks. First, we calculated the histogram to analyze the pixel distribution of an image. Second, we applied histogram equalization to improve the contrast of the image. Third, we implemented 2D convolution for spatial filtering. We used a LPF to blur the image and a HPF to detect edges. Finally, we wrote a Median Filter function to remove noise while preserving the edges.

We verified every step of our work carefully. We compared the processed images visually on the PC. We also checked the raw data in the microcontroller's memory using the STM32CubeIDE debugger. This homework taught us the practical side of embedded image processing.

BIBLIOGRAPHY

[1] Yusuf Zivaroğlu and Taner Kahyaoğlu, EE4065 STM32 Nucleo F446RE Image Transfer (STM-to-PC and PC-to-STM), https://github.com/EmbeddedIntelligenceTeam/EE4065_STM32_nucleo_F446RE-Image-Transfer-STM-to-PC-and-PC-to-STM, accessed: November 16, 2025 (2025).

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

[3] J. D. Hunter, Matplotlib: A 2d graphics environment, Computing in Science & Engineering 9 (3) (2007) 90–95. doi:10.1109/MCSE.2007.55.