Introduction to Embedded Image Processing

EE4065

Homework 2

Mahmut Nedim Göç

150721053

Emre Güner

150722031

Marmara Universty
Faculty of Engineering
Electrical Electronic Engineering

CONTENTS

1. INTRODUCTION

This study describes applying basic image processing techniques on an STM microcontroller during the Embedded Digital Image Processing course. The aim is to explore how core algorithms work within systems that have restricted computing power, limited memory, or strict timing needs - using practical experimentation. Each step was tested under realistic conditions instead of simulations. Focus lies on efficiency rather than complexity, favoring lightweight methods over heavy models. Implementation choices reflect trade-offs between speed and accuracy observed through direct measurement.

The assignment includes four key activities. First, creating histograms a C function that calculates the histogram from a grayscale image kept in the microcontroller memory. Second histogram equalization; explore the theory behind the technique then code it themselves to improve an image's contrast. Each step builds directly on the previous one, guiding users through both understanding and practical use.

The third task explores 2D convolution alongside spatial filtering methods. Instead of standard approaches, low-pass and high-pass filters are tested on a grayscale image through a custom-built convolution process, showing how simple kernels work within limited hardware setups. In contrast, the fourth section looks at median filtering - an effective nonlinear technique for reducing noise - adapted to run smoothly on microcontroller imaging systems.

Overall, this report documents the design, implementation, and experimental results of these operations, showcasing their behavior when executed on an STM32 platform. The presented work aims to strengthen both theoretical understanding and practical embedded-systems programming skills in digital image processing.

2. PROBLEMS

(1) **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.

(2) **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.

(3) **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.

(4) **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.

3. RESULTS

QUESTION 1: HISTOGRAM FORMATION

a) C Function Implementation

In this part of the assignment, a C function was developed to calculate the histogram of a given grayscale image. The histogram represents the distribution of pixel intensities, providing a graphical representation of the tonal distribution in a digital image.

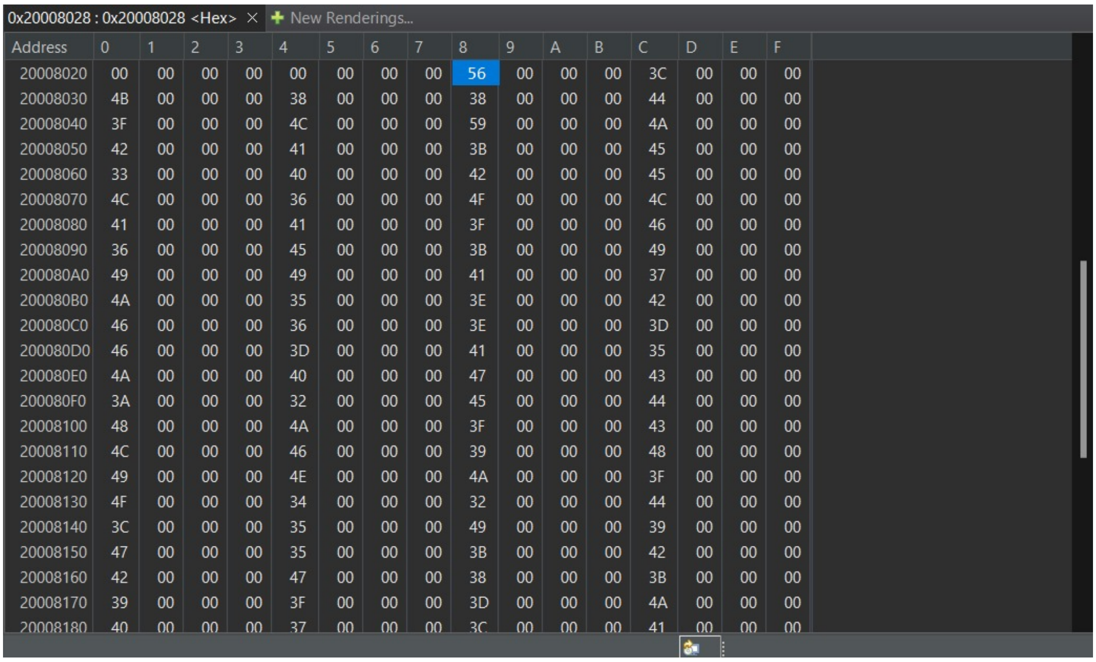
Figure 3.1: Original Image

The algorithm initializes an array of size 256 (representing 8-bit depth) with zeros. It then iterates through the input image array pixel by pixel. For each pixel intensity $I(x, y)$, the corresponding index in the histogram array is incremented. This process counts the frequency of each gray level from 0 to 255.

b) Verification on STM32Cube IDE

To evaluate performance, one unique gray-scale picture got created using a desktop computer while transformed into a C header format; this was added directly to the STM project, holding the picture information stored inside the chip's memory, and the histogram function ran on the microcontroller with results checked afterward using a separate validation step using the debugging features in STMCubeIDE—particularly the Memory view—while an additional method was used to verify what data remained in the histogram's memory after execution.

Figure 3.2 demonstrates the calculated histogram entries as observed during the debug session. The values shown in the memory view confirm that the pixel frequencies were counted correctly by the embedded algorithm.


0x20008028 : 0x20008028 <Hex> New Renderings...

Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
20008020	00	00	00	00	00	00	00	00	56	00	00	00	3C	00	00	00
20008030	4B	00	00	00	38	00	00	00	38	00	00	00	44	00	00	00
20008040	3F	00	00	00	4C	00	00	00	59	00	00	00	4A	00	00	00
20008050	42	00	00	00	41	00	00	00	3B	00	00	00	45	00	00	00
20008060	33	00	00	00	40	00	00	00	42	00	00	00	45	00	00	00
20008070	4C	00	00	00	36	00	00	00	4F	00	00	00	4C	00	00	00
20008080	41	00	00	00	41	00	00	00	3F	00	00	00	46	00	00	00
20008090	36	00	00	00	45	00	00	00	3B	00	00	00	49	00	00	00
200080A0	49	00	00	00	49	00	00	00	41	00	00	00	37	00	00	00
200080B0	4A	00	00	00	35	00	00	00	3E	00	00	00	42	00	00	00
200080C0	46	00	00	00	36	00	00	00	3E	00	00	00	3D	00	00	00
200080D0	46	00	00	00	3D	00	00	00	41	00	00	00	35	00	00	00
200080E0	4A	00	00	00	40	00	00	00	47	00	00	00	43	00	00	00
200080F0	3A	00	00	00	32	00	00	00	45	00	00	00	44	00	00	00
20008100	48	00	00	00	4A	00	00	00	3F	00	00	00	43	00	00	00
20008110	4C	00	00	00	46	00	00	00	39	00	00	00	48	00	00	00
20008120	49	00	00	00	4E	00	00	00	4A	00	00	00	3F	00	00	00
20008130	4F	00	00	00	34	00	00	00	32	00	00	00	44	00	00	00
20008140	3C	00	00	00	35	00	00	00	49	00	00	00	39	00	00	00
20008150	47	00	00	00	35	00	00	00	3B	00	00	00	42	00	00	00
20008160	42	00	00	00	47	00	00	00	38	00	00	00	3B	00	00	00
20008170	39	00	00	00	3F	00	00	00	3D	00	00	00	4A	00	00	00
20008180	40	00	00	00	37	00	00	00	3C	00	00	00	41	00	00	00

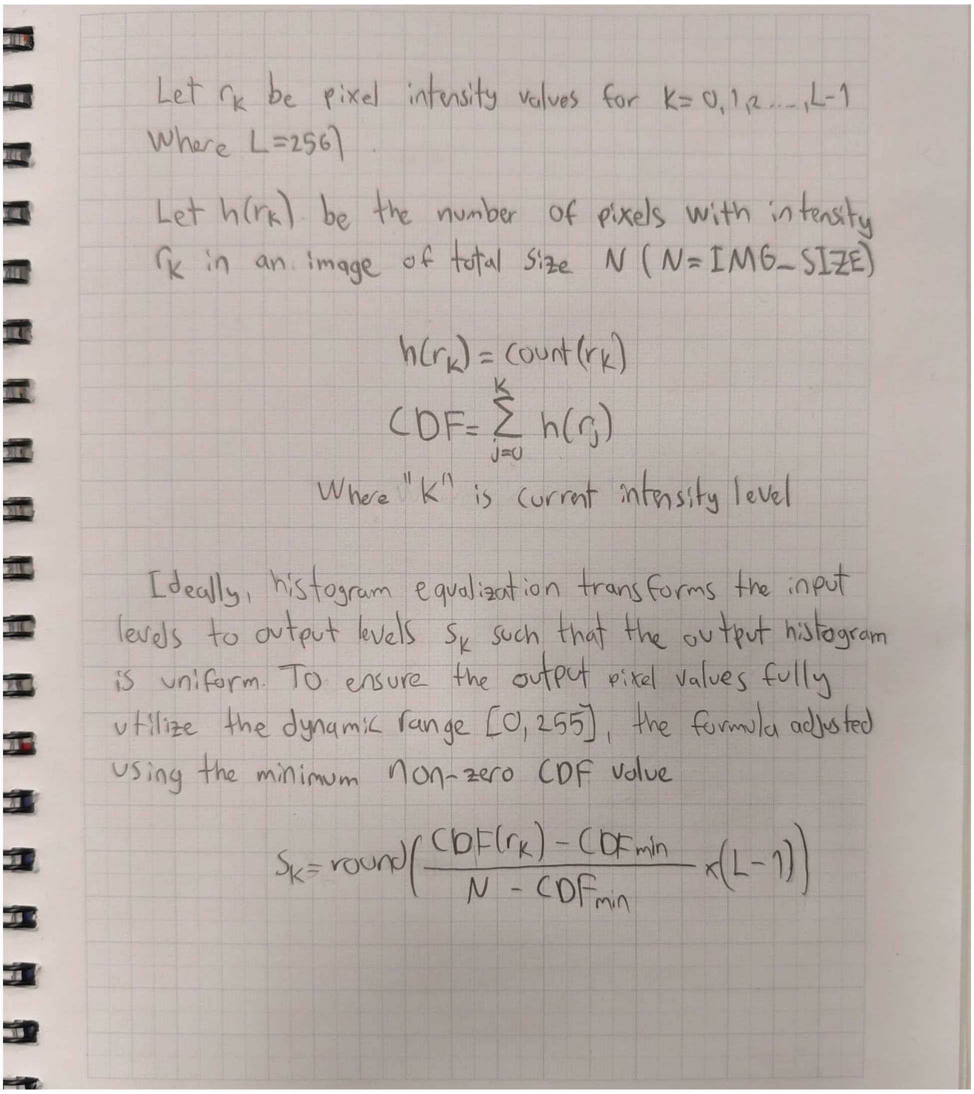
Figure 3.2: STM32Cube IDE Debug Interface showing the calculated histogram entries in memory.

QUESTION 2: HISTOGRAM EQUALIZATION

a) Mathematical Derivation

The histogram equalization method aims to transform an image so that its output histogram is approximately uniform. This is achieved by using the Cumulative Distribution Function (CDF) as the transformation function.

The complete mathematical derivation, showing how input intensity levels r_k are mapped to output levels s_k, was performed by hand. Figure 3.3 presents the photo of this derivation.



Let r_k be pixel intensity values for $k = 0, 1, 2, \ldots, L-1$ where $L = 256$).

Let $h(r_k)$ be the number of pixels with intensity r_k in an image of total size N (N = IMG_SIZE)

$$h(r_k) = \text{count}(r_k)$$

$$CDF = \sum_{j=0}^{k} h(r_j)$$

Where "k" is current intensity level

Ideally, histogram equalization transforms the input levels to output levels S_k such that the output histogram is uniform. To ensure the output pixel values fully utilize the dynamic range $[0, 255]$, the formula adjusted using the minimum non-zero CDF value

$$S_k = \text{round}\left(\frac{CDF(r_k) - CDF_{min}}{N - CDF_{min}} \times (L-1)\right)$$

Figure 3.3: Handwritten derivation of the histogram equalization transformation.

b) C Function Implementation

A C function was developed to apply histogram equalization on the STM32 microcontroller. The implementation involves three main steps:

1. **Histogram Calculation:** The frequency of each pixel intensity in the input image is counted.
2. **CDF Calculation and Mapping:** The cumulative probability is computed for each intensity level.
3. **Image Transformation:** Every pixel in the original image is replaced by its corresponding mapped value from the calculated lookup table.

b) C Function Implementation

A C function was developed to apply histogram equalization on the STM32 microcontroller. The implementation involves three main steps:

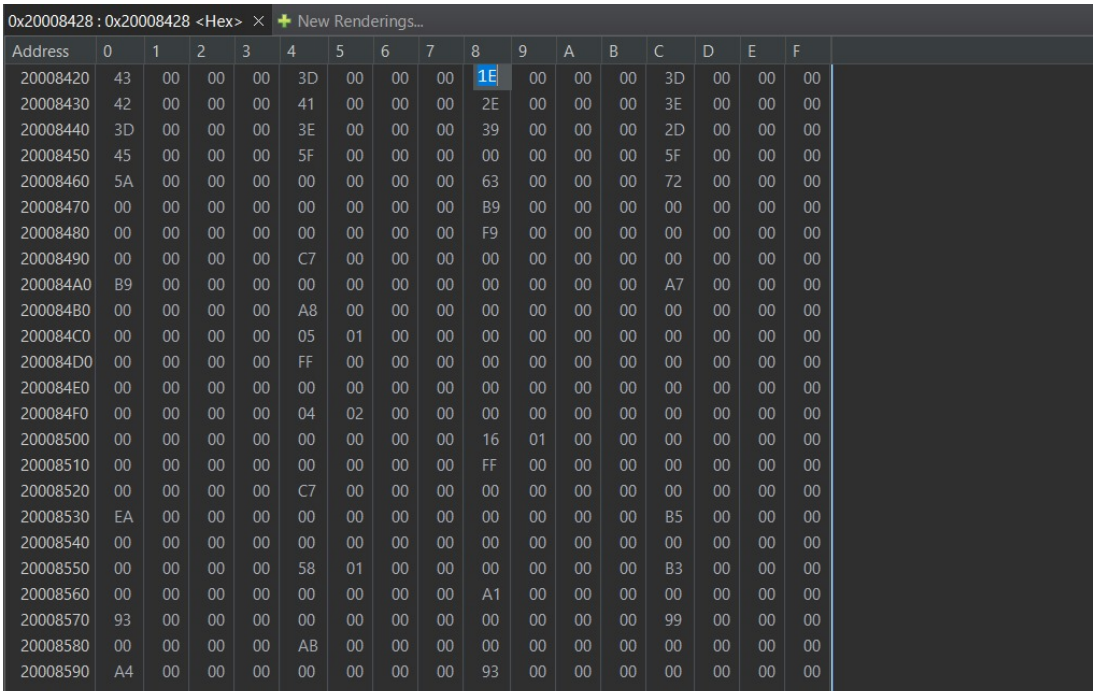
1. **Histogram Calculation:** The frequency of each pixel intensity in the input image is counted.
2. **CDF Calculation and Mapping:** The cumulative probability is computed for each intensity level. A mapping lookup table is generated using the formula derived in part (a).
3. **Image Transformation:** Every pixel in the original image is replaced by its corresponding mapped value from the calculated lookup table.

c) Application and Verification

The implemented function was applied to the grayscale image used in Question 1. Figure 3.4 displays the resulting image after the equalization process, showing the enhanced contrast compared to the original input.

After the equalization process, the histogram of the modified image was recalculated to verify the distribution changes. The resulting histogram entries were examined using the STM32Cube IDE debugger. Figure 3.5 shows the memory content of the new histogram, indicating how the pixel intensities have been redistributed across the dynamic range.

Figure 3.4: The resulting grayscale image after applying histogram equalization.



0x20008428 : 0x20008428 <Hex> New Renderings...

Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
20008420	43	00	00	00	3D	00	00	00	1E	00	00	00	3D	00	00	00
20008430	42	00	00	00	41	00	00	00	2E	00	00	00	3E	00	00	00
20008440	3D	00	00	00	3E	00	00	00	39	00	00	00	2D	00	00	00
20008450	45	00	00	00	5F	00	00	00	00	00	00	00	5F	00	00	00
20008460	5A	00	00	00	00	00	00	00	63	00	00	00	72	00	00	00
20008470	00	00	00	00	00	00	00	00	B9	00	00	00	00	00	00	00
20008480	00	00	00	00	00	00	00	00	F9	00	00	00	00	00	00	00
20008490	00	00	00	00	C7	00	00	00	00	00	00	00	00	00	00	00
200084A0	B9	00	00	00	00	00	00	00	00	00	00	00	A7	00	00	00
200084B0	00	00	00	00	A8	00	00	00	00	00	00	00	00	00	00	00
200084C0	00	00	00	00	05	01	00	00	00	00	00	00	00	00	00	00
200084D0	00	00	00	00	FF	00	00	00	00	00	00	00	00	00	00	00
200084E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
200084F0	00	00	00	00	04	02	00	00	00	00	00	00	00	00	00	00
20008500	00	00	00	00	00	00	00	00	16	01	00	00	00	00	00	00
20008510	00	00	00	00	00	00	00	00	FF	00	00	00	00	00	00	00
20008520	00	00	00	00	C7	00	00	00	00	00	00	00	00	00	00	00
20008530	EA	00	00	00	00	00	00	00	00	00	00	00	B5	00	00	00
20008540	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
20008550	00	00	00	00	58	01	00	00	00	00	00	00	B3	00	00	00
20008560	00	00	00	00	00	00	00	00	A1	00	00	00	00	00	00	00
20008570	93	00	00	00	00	00	00	00	00	00	00	00	99	00	00	00
20008580	00	00	00	00	AB	00	00	00	00	00	00	00	00	00	00	00
20008590	A4	00	00	00	00	00	00	00	93	00	00	00	00	00	00	00

Figure 3.5: STM32Cube IDE Memory View showing the histogram entries after applying histogram equalization.

QUESTION 3: 2D CONVOLUTION AND FILTERING

a) C Function Implementation

A generic C function was developed to perform 2D convolution on the microcontroller. This function iterates through the pixels of the input image (excluding the borders to handle boundary conditions) and applies a 3×3 kernel mask.

For each pixel position (x, y), the new intensity value $g(x, y)$ is calculated using the convolution formula:

$$g(x, y) = \sum_{s=-1}^{1} \sum_{t=-1}^{1} w(s, t) \cdot f(x + s, y + t)$$

where w represents the kernel weights and f is the input image. The function also includes a clamping mechanism to ensure the resulting pixel values remain within the valid 8-bit range $[0, 255]$.

b) Low Pass Filtering

To demonstrate the convolution function, a Low Pass Filter (smoothing filter) was applied to the test image. A standard 3×3 averaging kernel was utilized, where all coefficients are $1/9$. This operation blurs the image and reduces high-frequency noise.

Figure 3.6 shows the visual result of the low pass filtering. Additionally, Figure 3.7 presents the STM32Cube IDE memory view, confirming the filtered pixel values stored in the microcontroller's memory.

c) High Pass Filtering

Subsequently, a High Pass Filter was applied to the original image using the same convolution engine. A kernel designed to highlight edges (e.g., a Laplacian-like mask with a positive center and negative neighbors) was used. This filter accentuates fine details and edges while suppressing constant background intensities.

The resulting edge-detected image is displayed in Figure 3.8. The corresponding memory entries observed via the STM32 debugger are shown in Figure 3.9.

Figure 3.6: Resulting image after applying Low Pass Filtering (Smoothing).

Address	0	2	4	6	8	A	C	E
20000020	0000	0000	0000	0000	0000	0000	0000	0000
20000030	0000	0000	0000	0000	0000	0000	0000	0000
20000040	0000	0000	0000	0000	0000	0000	0000	0000
20000050	0000	0000	0000	0000	0000	0000	0000	0000
20000060	0000	0000	0000	0000	0000	0000	0000	0000
20000070	0000	0000	0000	0000	0000	0000	0000	0000
20000080	0000	0000	0000	0000	0000	0000	0000	0000
20000090	0000	0000	0000	0000	0000	0000	0000	0000
200000A0	0000	0000	0000	0000	005D	554D	4949	4C4E
200000B0	4F4E	4E4E	4E4E	4E50	4C4D	4E4F	4D4D	504E
200000C0	5776	A9D2	E9EE	EEEC	EDEE	EFEF	EFED	ECEA
200000D0	E9E8	E8E9	EAEB	EBEA	E9E8	E8E8	E8E9	EBEB
200000E0	EAE9	E9E8	E8E9	E9E9	E9EA	EBEB	E9E9	E9E9
200000F0	E9E8	E8E8	E8E9	EAEB	EBEA	E9E8	E8E8	E9EA
20000100	EAEB	EBEB	EBEC	ECEA	EAEA	ECEC	EFF1	EECB
20000110	A267	4F43	4A4E	4F52	5252	5252	5252	5250
20000120	4B50	4E52	597C	AE00	0051	4B46	4445	494A
20000130	4A49	4949	4949	4945	4547	4A47	4446	4B50
20000140	71A4	D6EA	EDEB	EAE8	EAEA	EAE9	E9E8	E8E7
20000150	E6E5	E6E6	E7E8	E9E8	E6E5	E5E5	E6E7	E8E9
20000160	E9E8	E8E8	E9EA	ECED	EEEF	EFEE	ECEB	EBEB
20000170	EBEA	E9E9	EAEC	EEF0	F0EF	EEED	ECEC	ECED
20000180	EDEE	EFEF	EFEF	EEED	ECEC	EEF0	F2F2	F3EC
20000190	CC90	5D48	4B4B	494A	4C4B	4B4B	4B4B	4B50
200001A0	484C	494E	4754	8300	004D	4947	4648	4B4C
200001B0	4B4A	4A4A	4A4A	4A40	4242	4845	4342	455E
200001C0	95CC	E8E8	E3E3	E4E5	E7E7	E7E7	E7E6	E6E5
200001D0	E4E4	E4E4	E5E5	E6E5	E3E2	E1E1	E2E2	E3E4
200001E0	E5E5	E4E5	E7E9	ECEE	F0F2	F1F0	EEEE	EEED

Figure 3.7: STM32Cube IDE Memory View showing entries of the Low Pass filtered image.

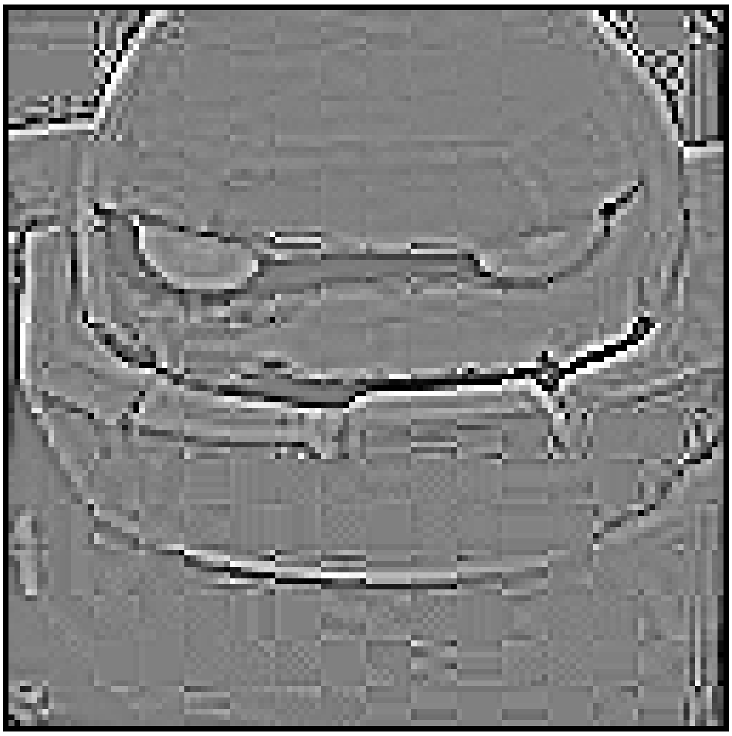
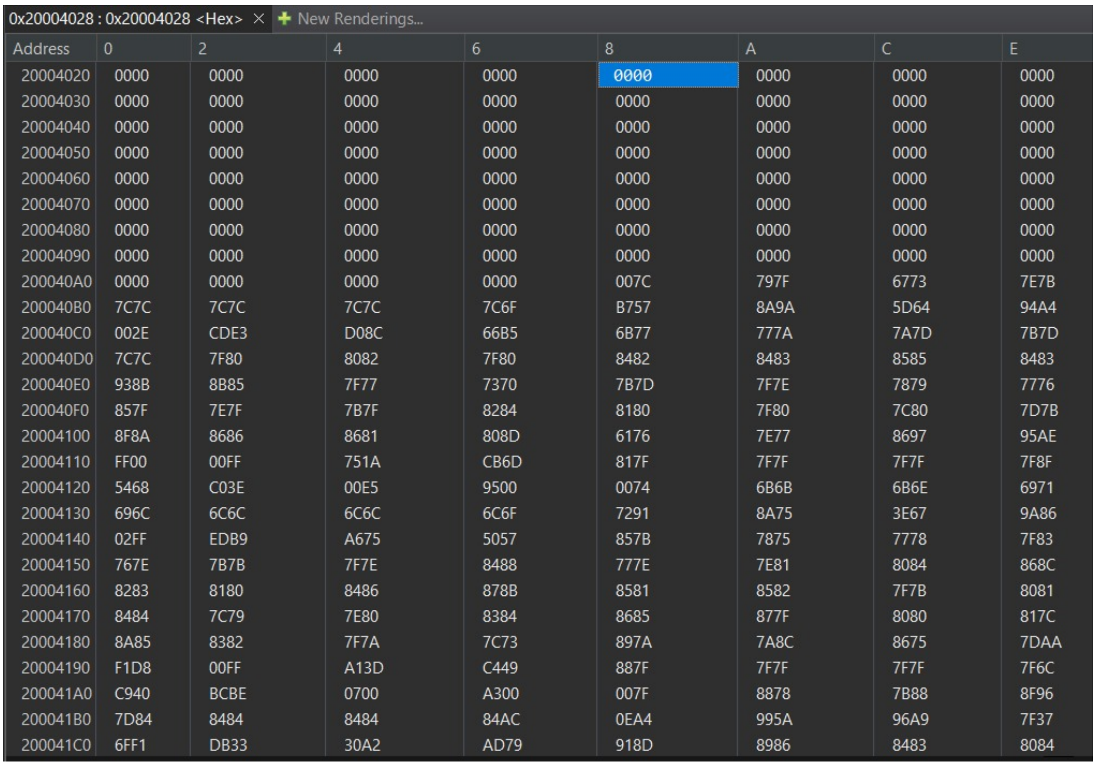


Figure 3.8: Resulting image after applying High Pass Filtering (Edge Detection).



0x20004028 : 0x20004028 <Hex> ✕ + New Renderings...

Address	0	2	4	6	8	A	C	E
20004020	0000	0000	0000	0000	0000	0000	0000	0000
20004030	0000	0000	0000	0000	0000	0000	0000	0000
20004040	0000	0000	0000	0000	0000	0000	0000	0000
20004050	0000	0000	0000	0000	0000	0000	0000	0000
20004060	0000	0000	0000	0000	0000	0000	0000	0000
20004070	0000	0000	0000	0000	0000	0000	0000	0000
20004080	0000	0000	0000	0000	0000	0000	0000	0000
20004090	0000	0000	0000	0000	0000	0000	0000	0000
200040A0	0000	0000	0000	0000	007C	797F	6773	7E7B
200040B0	7C7C	7C7C	7C7C	7C6F	B757	8A9A	5D64	94A4
200040C0	002E	CDE3	D08C	66B5	6B77	777A	7A7D	7B7D
200040D0	7C7C	7F80	8082	7F80	8482	8483	8585	8483
200040E0	938B	8B85	7F77	7370	7B7D	7F7E	7879	7776
200040F0	857F	7E7F	7B7F	8284	8180	7F80	7C80	7D7B
20004100	8F8A	8686	8681	808D	6176	7E77	8697	95AE
20004110	FF00	00FF	751A	CB6D	817F	7F7F	7F7F	7F8F
20004120	5468	C03E	00E5	9500	0074	6B6B	6B6E	6971
20004130	696C	6C6C	6C6C	6C6F	7291	8A75	3E67	9A86
20004140	02FF	EDB9	A675	5057	857B	7875	7778	7F83
20004150	767E	7B7B	7F7E	8488	777E	7E81	8084	868C
20004160	8283	8180	8486	878B	8581	8582	7F7B	8081
20004170	8484	7C79	7E80	8384	8685	877F	8080	817C
20004180	8A85	8382	7F7A	7C73	897A	7A8C	8675	7DAA
20004190	F1D8	00FF	A13D	C449	887F	7F7F	7F7F	7F6C
200041A0	C940	BCBE	0700	A300	007F	8878	7B88	8F96
200041B0	7D84	8484	8484	84AC	0EA4	995A	96A9	7F37
200041C0	6FF1	DB33	30A2	AD79	918D	8986	8483	8084

Figure 3.9: STM32Cube IDE Memory View showing entries of the High Pass filtered image.

QUESTION 4: MEDIAN FILTERING

a) C Function Implementation

A C function was implemented to perform median filtering, which is a non-linear filtering technique often used to remove noise from an image (specifically salt-and-pepper noise) while preserving edges.

The algorithm operates by sliding a 3×3 window over the image. For each pixel location:

1. The intensity values of the 9 pixels within the window are extracted into a temporary array.
2. This array is sorted in ascending order (using a simple sorting algorithm such as bubble sort).
3. The central pixel of the window is replaced by the median value (the middle element of the sorted array).

b) Application and Verification

The median filtering function was applied to the grayscale test image on the STM32 microcontroller. This filter is particularly effective at suppressing impulse noise without blurring sharp edges as much as a mean filter would.

Figure 3.10 illustrates the visual result of the median filter. The corresponding pixel values stored in the microcontroller's memory were inspected using the STM32Cube IDE debugger to validate the implementation. Figure 3.11 presents these memory entries.

Figure 3.10: Resulting image after applying Median Filtering.

0x20008828 : 0x20008828 <Hex> ✕ New Renderings...

Address	0	2	4	6	8	A	C	E
20008820	3D00	0000	2500	0000	0000	0000	0000	0000
20008830	0000	0000	0000	0000	0000	0000	0000	0000
20008840	0000	0000	0000	0000	0000	0000	0000	0000
20008850	0000	0000	0000	0000	0000	0000	0000	0000
20008860	0000	0000	0000	0000	0000	0000	0000	0000
20008870	0000	0000	0000	0000	0000	0000	0000	0000
20008880	0000	0000	0000	0000	0000	0000	0000	0000
20008890	0000	0000	0000	0000	0000	0000	0000	0000
200088A0	0000	0000	0000	0000	005C	524B	4545	4B4D
200088B0	4D4D	4D4D	4D4D	4D4D	4B4D	4D50	4550	5050
200088C0	505E	BAF0	F6F6	F0EB	EBEB	ECED	EDEC	EAE9
200088D0	E7E7	E8E9	EAEB	EBEB	EAE9	E9E9	E9EA	EAEA
200088E0	EAE9	E8E8	E8E8	E8E8	E8E9	EAEA	E9E8	E9E9
200088F0	E9E8	E7E7	E8E9	EBEB	EBEA	E9E8	E8E8	E8E9
20008900	ECEE	EDED	EDEC	ECEC	EBEB	EBED	F0F3	F3E8
20008910	B779	5A4D	4D4D	5052	5252	5252	5252	5252
20008920	4B52	5257	3F8A	C300	0050	4B45	4545	4B4B
20008930	4B4B	4B4B	4B4B	4B4B	4B4D	4D4B	4245	4D50
20008940	52BA	EDF7	F0EB	EAEA	EAEA	EAE9	E8E7	E7E7
20008950	E6E4	E5E6	E7E8	EAEA	E7E4	E4E5	E6E8	EAEA
20008960	EAE9	E8E8	E8EA	ECF0	F0F0	F0EF	ECEB	EBEC
20008970	ECEB	E8E8	E9EC	EFF1	F1F1	F0ED	ECEC	ECEC
20008980	EEEE	F0F0	EFED	ECEC	ECED	EDF1	F3F3	F3F3
20008990	E7A0	5A55	5552	5050	4B4B	4B4B	4B4B	4B4B
200089A0	4545	4557	3C3C	8A00	004D	4D45	454B	5050
200089B0	4B4B	4B4B	4B4B	4B42	4B4B	4B45	3F42	4B50
200089C0	91E4	E9E4	DEDE	E7E7	E8E8	E7E6	E6E6	E6E4
200089D0	E4E4	E4E4	E5E6	E7E4	E1E1	E0E0	E1E1	E2E3
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Figure 3.11: STM32Cube IDE Memory View showing entries of the Median filtered image.

4. CONCLUSION

In this task, image processing methods were built and checked on an STM chip, revealing how standard algorithms perform with limited hardware. Because of histogram creation and balancing, insight into brightness control improved. Although convolution filters were used, they showed blurring and edge-sharpening effects clearly. Despite minimal resources, median filtering proved useful for removing noise through non-linear approaches. As a result, both concept grasp and hands-on coding ability grew, proving basic image techniques can run efficiently in tight, real-time settings.

REFERENCES

(**1**) Embedded Machine Learning with Microcontrollers Applications on STM32 Development Boards, Cem Ünsalan, Berkan Höke and Eren Atmaca

(**2**) Embedded System Design with ARM Cortex-Microcontrollers: Applications with C, C++ and Pyhton, Cem Ünsalan, Hüseyin Deniz Gürhan and Mehmet Erkin