MARMARA UNIVERSITY

Embedded Digital Image Processing

EE4065

Homework 1

Mustafa Öztürk

150722016

Emirhan Şener

150721069

Faculty of Engineering
Electrical Electronic Engineering

Fall 2025

ACRONYMS

1D 1-dimensional

2D 2-dimensional

EE4065 Embedded Digital Image Processing

LUT Look-up Table

RAM Random Access Memory

CONTENTS

Acronyms **1**

List of Figures **3**

1 Introduction **4**

2 Problems **5**

2.1 Q-1) 5

2.1.1 Theory 5

2.1.2 Procedure 6

2.2 Q-2) 9

2.2.1 Overall Comparison 16

3 Conclusion **18**

References **19**

LIST OF FIGURES

2.1 Image used for this homework 7

2.2 Side-by-side comparison of the data array in the `monke.h` file (left) and the corresponding data observed in the microcontroller's RAM via the Memory Browser (right). 8

2.3 Comparison of the original image (left) and its negative (right). 10

2.4 Memory browser view of the original `grayscale_img_data` array. 10

2.5 Memory browser view of the `negative_image_data` array. 10

2.6 Result of thresholding with a value of 128. 12

2.7 Memory browser view of the `threshold_image_data` array, showing only `0x00` and `0xFF` values. 12

2.8 Visual results of the Gamma Correction operations. 14

2.9 Memory verification for Gamma Correction operations. 14

2.10 Result of contrast stretching using a piecewise linear function. 16

2.11 Memory browser view of the `piecewise_image_data` array, showing clipped and stretched pixel values. 16

2.12 Overall comparison of all applied transformations. 17

1. INTRODUCTION

This report covers the first homework for Embedded Digital Image Processing (EE4065). Essentially, we explored managing images on a microcontroller, be aware of its limited space. It broke down into stages: getting an image into the microcontroller, then running some simple modifications or transformations or applying some functions on it.

To start, we picked an image using a computer, then transformed it into a C header file full of gray shades. We added this file to our STM32CubeIDE[1] project. After the compilation, the code went onto our NUCLEO-F446RE board. Using the debugger, we verified the complete image data inside the microcontroller's Random Access Memory (RAM) and task one done.

We then built some functions to alter the properties of this image. Specifically, we generated some code for inverting colors, setting thresholds, adjusting overall brightness (gamma correction) using varied gamma values, likewise extending contrast via a custom scale. To confirm everything worked as expected just like the homework asked and we checked the resulting images directly in both computer as images and memory in microcontroller.

2. PROBLEMS

2.1. Q-1)

(40 points) Use the available code in the repository below (with appropriate modifications) for this question. 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 and display some of the image entries in the memory of your microcontroller.

2.1.1. Theory

The fundamental task in embedded image processing is to represent a visual image in a format that a microcontroller can understand and store in its limited memory. A digital image is mathematically represented as a matrix of values, where each value corresponds to a pixel's intensity. For a grayscale image, each pixel holds a single intensity value, typically stored as an 8-bit unsigned number ranging from 0 (black) to 255 (white). [2]

To handle this data on our STM32 microcontroller, the image matrix must be converted into a one-dimensional array. This process involves preparing a grayscale image on a PC and then using a tool to transform its pixel data into a C-language array, which is stored in a **header file (.h)**. By including this header file in our project, the C compiler allocates space for this array in the microcontroller's memory. The microcontroller does not see this data as an image, but simply as a large array of numerical values stored at specific memory addresses.

The final step is to verify this data transfer. Using the debugger in the STM32CubeIDE, we can directly observe the memory locations where this array is stored. This allows us to confirm that the pixel values from our header file have been successfully loaded into the microcontroller's RAM.

2.1.2. Procedure

The procedure for the first question of the assignment was completed in four main steps: preparing the image on a PC, converting it to a C header file, integrating it into an STM32CubeIDE project, and finally verifying its presence in the microcontroller's memory.

Image Preparation and Conversion to Header File

First, a suitable image was selected on the PC. To meet the requirements of the assignment, this image needed to be converted into a grayscale C-style array. The provided Python script, `Image_Header_Library.py`, was initially designed for color formats. Therefore, we added a new function, `grayscale_c_generate`, to handle this specific conversion.

The following Python code snippet was added to the `Image_Header_Library.py` file:

Listing 2.1: Generate grayscale image header function snippet

```python
def grayscale_c_generate(im, outputFileName):
    f = open(outputFileName + ".h", "w+")

    height, width, _ = im.shape

    # Convert image from BGR to Grayscale
    gray_image = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)

    # Flatten the 2D image array to a 1D array
    gray_image_flat = np.reshape(gray_image, (width * height))

    f.write("#include <stdint.h>\n\n")
    f.write("const uint8_t grayscale_img_data[%d] = {\n" % (width *
        height))

    for i in range(width * height):
        f.write("%s, " % hex(gray_image_flat[i]))
        if i != 0 and (i + 1) % 16 == 0:
            f.write("\n")

    f.write("\n};\n\n")

    f.write("/*\n")
    f.write("ImageTypeDef GRAYSCALE_IMG = {\n")
    f.write("    .pData = (uint8_t*)my_image_data,\n")
    f.write("    .width = %d,\n" % (width))
    f.write("    .height = %d,\n" % (height))
    f.write("    .size = %d,\n" % (width*height))
    f.write("    .format = 0 // Assuming 0 is for Grayscale\n")
    f.write("};\n*/\n\n")

    f.close()
    print("Grayscale C header file '%s.h' generated successfully." %
        outputFileName)
```

Using this updated library, we generated a `monke.h` header file containing a 160x120 grayscale image stored in a `const uint8_t` array named `grayscale_img_data`.

Figure 2.1: Image used for this homework

Project Setup in STM32CubeIDE

A new STM32 project was created in STM32CubeIDE. The generated `monke.h` file was then copied into the project's `Core/Inc` directory. To make the image data accessible to our program, we included this header file in `main.c`. A pointer was also created to point to the start of the image array. This was not necessary for the program to function, but it made it easier to locate the data array in the debugger.

The following code was added to the `main.c` file:

Listing 2.2: Code added to main.c for data integration.

```
/* USER CODE BEGIN Includes */
#include "monke.h"
/* USER CODE END Includes */

/* USER CODE BEGIN 2 */
const uint8_t *image_data_ptr = grayscale_img_data;
/* USER CODE END 2 */
```

Results

The primary result for the first question is the successful verification that the image data was correctly loaded into the microcontroller's memory. Following the procedure, the project was compiled and a debugging session was initiated on the NUCLEO-F446RE board, with the program execution paused inside the main loop.

To confirm the data's presence and integrity, we utilized the **Memory Browser** tool within the STM32CubeIDE debugging environment. We configured the browser to monitor the starting address of our image array, `grayscale_img_data`. The tool then displayed the raw hexadecimal content of the RAM at that specific location.

As demonstrated in Figure 2.2, the values observed in the memory window were directly compared against the source values in the `monke.h` file. The comparison showed a perfect match, starting with the initial values of `0xa1, 0xa2, 0xa8, ...`. This successful verification confirms that the image data is correctly stored in the microcontroller's memory and is accessible for the processing tasks required in Question 2.

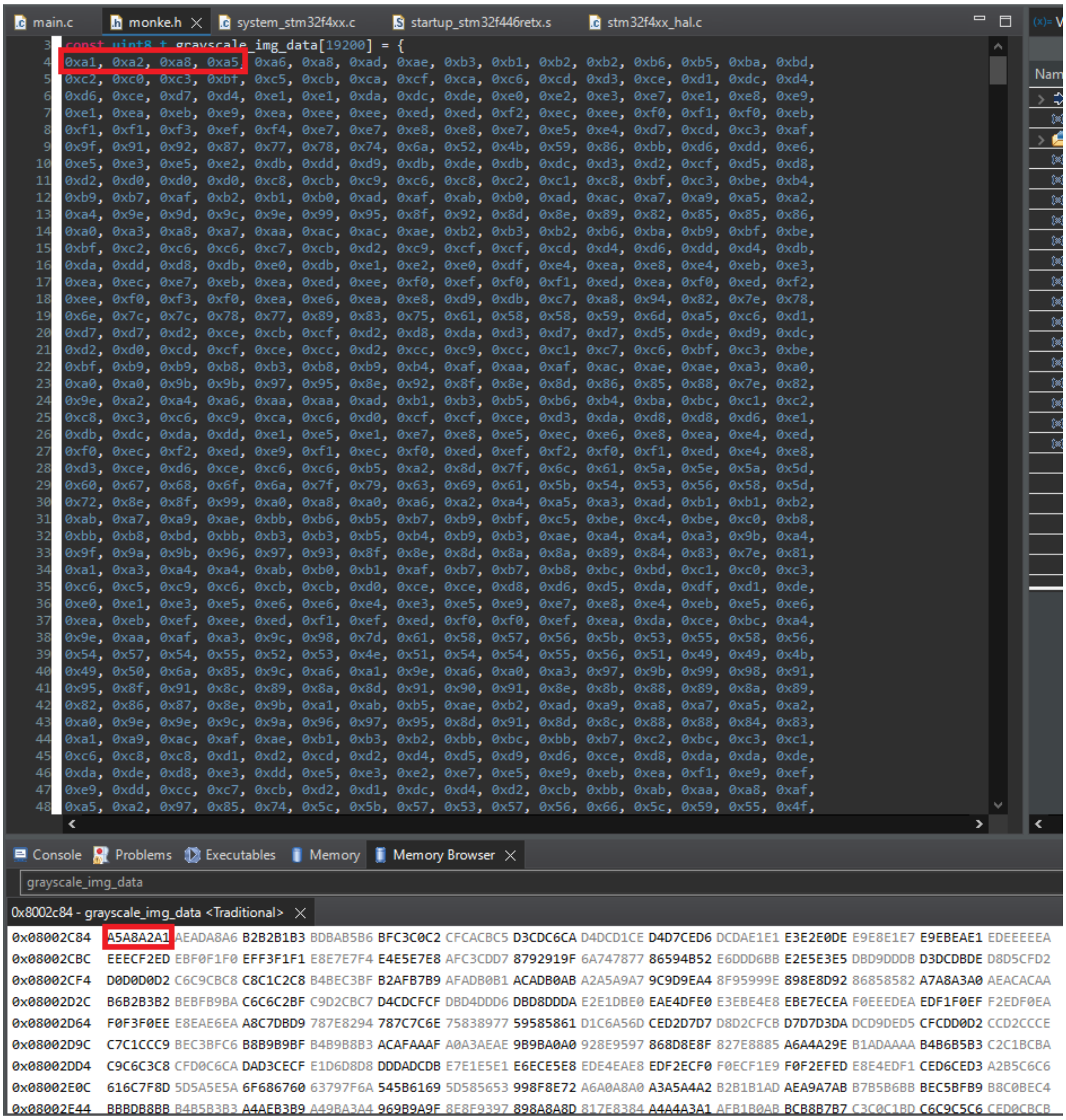


Figure 2.2: Side-by-side comparison of the data array in the `monke.h` file (left) and the corresponding data observed in the microcontroller's RAM via the Memory Browser (right).

2.2. Q-2)

(60 points) Apply the intensity transformations below to your image.

This section details the implementation and results of the intensity transformations applied to the grayscale image data stored in the microcontroller's memory.

Visual Verification Method

To visually analyze the results of our image processing algorithms, we first exported the raw data of each output array (e.g., `negative_image_data`) from the STM32CubeIDE's Memory Browser as a `.bin` file. Then, on the PC, a Python script using the OpenCV[3] and NumPy[4] libraries was used to read this binary file. The script reshapes the 1-dimensional (1D) array of pixel data back into its original 160x120 2-dimensional (2D) image format and saves it as a PNG file. This method allowed us to visually compare the output of each transformation with the original image. You can find the script at `Homework 1/bin/reconstruct_image.py` in this GitHub repository.

(a) **Negative Image**

Theory

The negative transformation inverts the intensity levels of a grayscale image. For an 8-bit image where pixel values range from 0 to 255, the transformation function is given by the equation:

$$f(g) = 255 - g$$

Procedure

We implemented this transformation by iterating through each pixel of the original `grayscale_img_data` array and applying the formula. The result of each operation was stored in a separate array called `negative_image_data`.

Listing 2.3: Code snippet for negative image transformation.

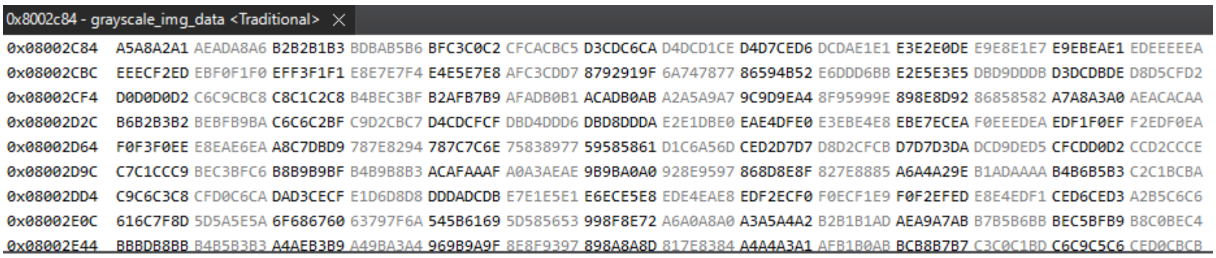
```c
/* Q2.a - Negative Image */
for (int i = 0; i < IMAGE_SIZE; i++){
    negative_image_data[i] = 255 - grayscale_img_data[i];
}
```

Results

The resulting image, shown in Figure 2.3, correctly displays the inverted intensities. The dark areas of the original image appear bright, and the bright areas appear dark, which confirms that the algorithm was implemented successfully.

Figure 2.3: Comparison of the original image (left) and its negative (right).

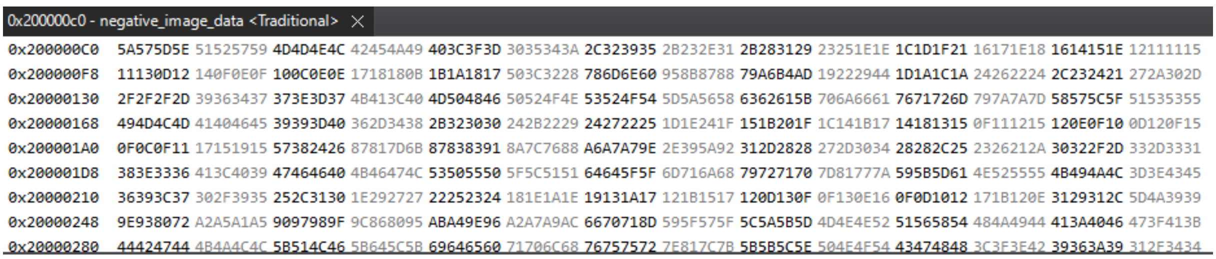
This outcome was also verified numerically by observing the memory locations. As shown in Figure 2.4 and 2.5, the fourth pixel of the original image (at address `0x08002C84`) has a value of `0xA5` (165). The corresponding fourth pixel in the `negative_image_data` array (at address `0x200000C0`) holds the value `0x5A` (90), which correctly matches the expected result of $255 - 165 = 90$. This confirms the accuracy of the implementation.



0x8002c84 - grayscale_img_data <Traditional>

0x08002C84	A5A8A2A1	AEADA8A6	B2B2B1B3	BDBAB5B6	BFC3C0C2	CFCACBC5	D3CDC6CA	D4DCD1CE	D4D7CED6	DCDAE1E1	E3E2E0DE	E9E8E1E7	E9EBEAE1	EDEEEEEA
0x08002CBC	EEECF2ED	EBF0F1F0	EFF3F1F1	E8E7E7F4	E4E5E7E8	AFC3CDD7	8792919F	6A747877	86594B52	E6DDD6BB	E2E5E3E5	DBD9DDDB	D3DCDBDE	D8D5CFD2
0x08002CF4	D0D0D0D2	C6C9CBC8	C8C1C2C8	B4BEC3BF	B2AFB7B9	AFADB0B1	ACADB0AB	A2A5A9A7	9C9D9EA4	8F95999E	898E8D92	86858582	A7A8A3A0	AEACACAA
0x08002D2C	B6B2B3B2	BEBFB9BA	C6C6C2BF	C9D2CBC7	D4CDCFCF	DBD4DDD6	DBD8DDDA	E2E1DBE0	EAE4DFE0	E3EBE4E8	EBE7ECEA	F0EEEDEA	EDF1F0EF	F2EDF0EA
0x08002D64	F0F3F0EE	E8EAE6EA	A8C7DBD9	787E8294	787C7C6E	75838977	59585861	D1C6A56D	CED2D7D7	D8D2CFCB	D7D7D3DA	DCD9DED5	CFCDD0D2	CCD2CCCE
0x08002D9C	C7C1CCC9	BEC3BFC6	B8B9B9BF	B4B9B8B3	ACAFAAAF	A0A3AEAE	9B9BA0A0	928E9597	868D8E8F	827E8885	A6A4A29E	B1ADAAAA	B4B6B5B3	C2C1BCBA
0x08002DD4	C9C6C3C8	CFD0C6CA	DAD3CECF	E1D6D8D8	DDDADCDB	E7E1E5E1	E6ECE5E8	EDE4EAE8	EDF2ECF0	F0ECF1E9	F0F2EFED	E8E4EDF1	CED6CED3	A2B5C6C6
0x08002E0C	616C7F8D	5D5A5E5A	6F686760	63797F6A	545B6169	5D585653	998F8E72	A6A0A8A0	A3A5A4A2	B2B1B1AD	AEA9A7AB	B7B5B6BB	BEC5BFB9	B8C0BEC4
0x08002E44	BBBDB8BB	B4B5B3B3	A4AEB3B9	A49BA3A4	969B9A9F	8E8F9397	898A8A8D	817E8384	A4A4A3A1	AFB1B0AB	BCB8B7B7	C3C0C1BD	C6C9C5C6	CED0CBCB

Figure 2.4: Memory browser view of the original `grayscale_img_data` array.



0x200000c0 - negative_image_data <Traditional>

0x200000C0	5A575D5E	51525759	4D4D4E4C	42454A49	403C3F3D	3035343A	2C323935	2B232E31	2B283129	23251E1E	1C1D1F21	16171E18	1614151E	12111115
0x200000F8	11130D12	140F0E0F	100C0E0E	1718180B	1B1A1817	503C3228	786D6E60	958B8788	79A6B4AD	19222944	1D1A1C1A	24262224	2C232421	272A302D
0x20000130	2F2F2F2D	39363437	373E3D37	4B413C40	4D504846	50524F4E	53524F54	5D5A5658	6362615B	706A6661	7671726D	797A7A7D	58575C5F	51535355
0x20000168	494D4C4D	41404645	39393D40	362D3438	2B323030	242B2229	24272225	1D1E241F	151B201F	1C141B17	14181315	0F111215	120E0F10	0D120F15
0x200001A0	0F0C0F11	17151915	57382426	87817D6B	87838391	8A7C7688	A6A7A79E	2E395A92	312D2828	272D3034	28282C25	2326212A	30322F2D	332D3331
0x200001D8	383E3336	413C4039	47464640	4B46474C	53505550	5F5C5151	64645F5F	6D716A68	79727170	7D81777A	595B5D61	4E525555	4B494A4C	3D3E4345
0x20000210	36393C37	302F3935	252C3130	1E292727	22252324	181E1A1E	19131A17	121B1517	120D130F	0F130E16	0F0D1012	171B120E	3129312C	5D4A3939
0x20000248	9E938072	A2A5A1A5	9097989F	9C868095	ABA49E96	A2A7A9AC	6670718D	595F575F	5C5A5B5D	4D4E4E52	51565854	484A4944	413A4046	473F413B
0x20000280	44424744	4B4A4C4C	5B514C46	5B645C5B	69646560	71706C68	76757572	7E817C7B	5B5B5C5E	504E4F54	43474848	3C3F3E42	39363A39	312F3434

Figure 2.5: Memory browser view of the `negative_image_data` array.

(b) **Thresholding the Image**

Theory

Thresholding is a simple segmentation method used to create an image. A threshold value, T, is chosen. Any input pixel g with an intensity greater than T is set to the maximum value (255), and any pixel with an intensity less than or equal to T is set to the minimum value (0). The function is defined as:

$$f(g) = \begin{cases} 255 & \text{if } g > T \\ 0 & \text{if } g \leq T \end{cases}$$

Procedure

We chose a threshold value of $T = 128$. The code iterates through the original image and applies this condition to each pixel and stores the result in the `threshold_image_data` array.

Listing 2.4: Code snippet for thresholding.

```
    /* Q2.b - Thresholding */
    uint8_t threshold_value = 128;
    for (int i = 0; i < IMAGE_SIZE; i++){
        if (grayscale_img_data[i] > threshold_value){
            threshold_image_data[i] = 255;
        } else {
            threshold_image_data[i] = 0;
        }
    }
```

Results

The output, shown in Figure 2.6, is a high-contrast, binary image composed only of black and white pixels. This demonstrates the correct application of the thresholding algorithm.

Figure 2.6: Result of thresholding with a value of 128.

The numerical verification in the memory browser confirms this result. The memory dump in Figure 2.7 shows that the entire `threshold_image_data` array consists exclusively of `0x00` and `0xFF` values. For instance, original pixels with values greater than 128 (e.g., `0xA5`) were correctly mapped to `0xFF`, while pixels with values less than or equal to 128 were mapped to `0x00`.

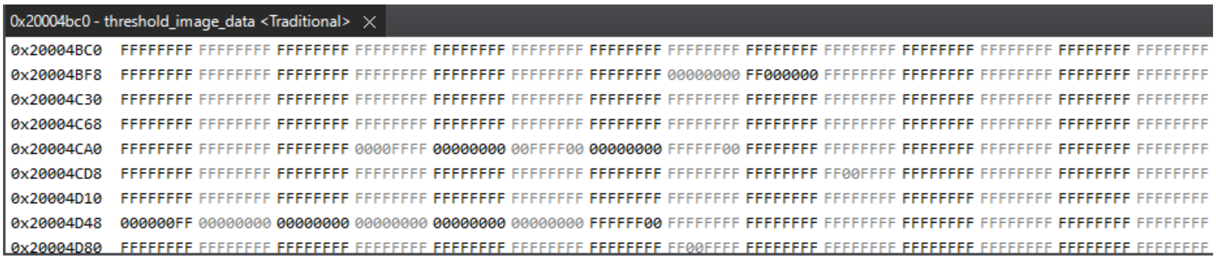


Figure 2.7: Memory browser view of the `threshold_image_data` array, showing only `0x00` and `0xFF` values.

(c) **Gamma Correction with Gamma Being 3 and 1/3**

Theory

Gamma correction is a non-linear operation used to adjust image brightness and contrast. The transformation is defined by $f(g) = c \cdot g^{\gamma}$, where c is a constant and γ is the gamma value. A $\gamma > 1$ darkens the image, while a $\gamma < 1$ brightens it.

Procedure

For $\gamma = 3.0$, we just used `powf()` (which is a function of `Math.h`) function which creates a Look-up Table (LUT) to calculate it easier. For $\gamma = 1/3$, we implemented a more efficient piecewise linear approximation to avoid the computationally expensive root function as suggested in our class. This method uses simple linear equations between key points to approximate the gamma curve.

Listing 2.5: Code snippet for Gamma Correction

```
/* Q2.c: Gamma Correction */

	// Gamma = 3
	uint8_t gamma_lut_dark[256];
	for (int i = 0; i < 256; i++) {
		gamma_lut_dark[i] = (uint8_t)(powf(i / 255.0f, 3.0f
		    ) * 255.0f);
	}
	for (int i = 0; i < IMAGE_SIZE; i++) {
		gamma_dark_image_data[i] = gamma_lut_dark[
		    grayscale_img_data[i]];
	}

	// Gamma = 1/3 - piecewise linear approach (since taking
	    root is slower process as we've talked during the class)
	// Points chosen: (x1,y1)=(0,0), (x2,y2)=(64,161), (x3,y3)
	    =(192,230), (x4,y4)=(255,255)
	const int x1=0,   y1=0;
	const int x2=64,  y2=161;
	const int x3=192, y3=230;
	const int x4=255, y4=255;

	// slopes
	const float m1 = (float)(y2 - y1) / (x2 - x1);
	const float m2 = (float)(y3 - y2) / (x3 - x2);
	const float m3 = (float)(y4 - y3) / (x4 - x3);

	for (int i = 0; i < IMAGE_SIZE; i++){
		uint8_t pixel_value = grayscale_img_data[i];

		if (pixel_value <= x2){ // 0-64
			// y = m*(x-x1) + y1
			gamma_bright_image_data[i] = (uint8_t)(m1 *
			    (pixel_value - x1) + y1);
		}
		else if (pixel_value <= x3){ // 64-192
			// y = m*(x-x2) + y2
			gamma_bright_image_data[i] = (uint8_t)(m2 *
			    (pixel_value - x2) + y2);
		}
		else{ // 192-255
			// y = m*(x-x3) + y3
			gamma_bright_image_data[i] = (uint8_t)(m3 *
			    (pixel_value - x3) + y3);
		}
	}
```

Results

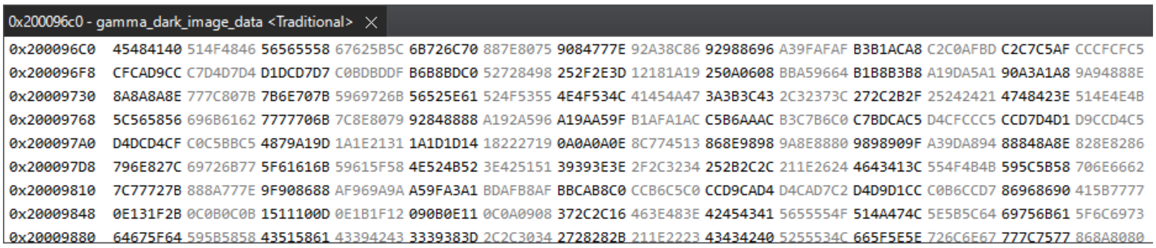
The results in Figure 2.8 show that the transformations worked as expected. Applying a gamma of 3.0 resulted in a darker image with increased contrast in the brighter regions. Applying a gamma of 1/3 brightened the image, making details in the darker regions more visible.

(a) Result for γ = 3.0 (darker).

(b) Result for γ = 1/3 (brighter).

Figure 2.8: Visual results of the Gamma Correction operations.

The memory dump for the dark image (Figure 2.9a) shows that pixel values are generally lower than their original ones so confirming the darkening effect. At the other side, the memory dump for the brightened image (Figure 2.9b) contains higher pixel values which matches the expected brightening effect of our piecewise linear approximation. For example an original mid-gray pixel value of `0x8B` (139) was transformed to approximately `0xC9` (201) demonstrating the non-linear increase in brightness.



(a) Memory view of the darker image (γ = 3.0).

(b) Memory view of the brighter image (γ = 1/3).

Figure 2.9: Memory verification for Gamma Correction operations.

(d) **Piecewise Linear Transformation for Part in (b)**

Theory

This transformation, also known as contrast stretching, enhances the contrast of an image by expanding a specific range of intensity levels to fill the entire dynamic range. We chose to stretch the mid-tones, defined between points (x_1, y_1) and (x_2, y_2), to the full 0-255 range.

Procedure

We selected an input range of $[50, 150]$ to be stretched to the output range of $[0, 255]$. The C code implements this by mapping pixels within this range linearly, while clipping values outside of it.

Listing 2.6: Code snippet for piecewise linear transformation.

```c
/* Q2.d: Piecewise Linear Transformation */
int r1 = 50,  s1 = 0;
int r2 = 150, s2 = 255;

for (int i = 0; i < IMAGE_SIZE; i++){
    uint8_t pixel_value = grayscale_img_data[i];
    if (pixel_value < r1){
        piecewise_image_data[i] = s1;
    } else if (pixel_value > r2){
        piecewise_image_data[i] = s2;
    } else {
        piecewise_image_data[i] = (uint8_t)(((float)(
            pixel_value - r1) / (r2 - r1)) * (s2 - s1) + s1);
    }
}
```

Results

The resulting image in Figure 2.10 has noticeably higher contrast. The mid-gray tones from the original image are now spread across the full black-to-white spectrum, making the details in those areas much clearer.

Figure 2.10: Result of contrast stretching using a piecewise linear function.

This was also validated by inspecting the memory content. As seen in Figure 2.11, original pixels with intensities above 150 (e.g., the bright sky area) were correctly clipped to `0xFF`, and those below 50 were clipped to `0x00`. Pixels within the [50, 150] range were re-mapped to values spanning the full [0, 255] range.

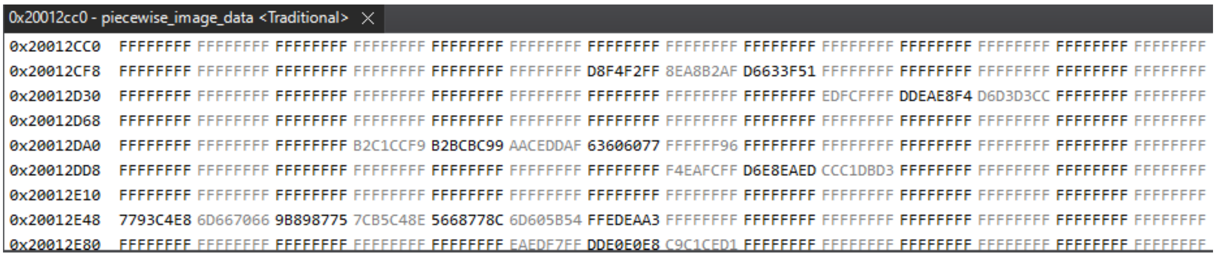


Figure 2.11: Memory browser view of the `piecewise_image_data` array, showing clipped and stretched pixel values.

2.2.1. Overall Comparison

Figure 2.12 provides a consolidated view of the original image and all the transformations applied. It is clear how each algorithm alters the pixel intensities to achieve a different visual effect.

(a) Original **(b)** Negative **(c)** Thresholding

(d) Gamma (Darker) **(e)** Gamma (Brighter) **(f)** Piecewise Linear Transformation

Figure 2.12: Overall comparison of all applied transformations.

3. CONCLUSION

This homework assignment was a very practical introduction to embedded digital image processing. In the first part, we successfully managed to take an image, convert it into a C header file using a Python script and load it into the memory of our NUCLEO-F446RE microcontroller. The most important part of this step was using the STM32CubeIDE's memory browser to verify that the raw pixel data was correctly placed in the RAM.

In the second part, we applied several common intensity transformations directly on the image data in the microcontroller's memory. We implemented algorithms for negative, thresholding, gamma correction, and piecewise linear contrast stretching. We were able to confirm that our C code worked correctly by exporting the resulting data arrays back to the PC and reconstructing them into images. This allowed us to visually compare the "before" and "after" images as shown in our results.

Overall, this assignment helped us to understand the complete workflow from preparing data on a PC to processing it on a microcontroller. We also learned how to verify our results numerically by observing memory locations. The challenge of implementing gamma correction using different methods showed us that we must think about performance and memory efficiency and not just getting the correct result directly. We feel we now have a solid foundation for the more complex topics in this course.

BIBLIOGRAPHY

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

[2] C. Ünsalan, H. D. Gürhan, M. E. Yücel, Embedded System Design with Arm Cortex-M Microcontrollers: Applications with C, C++ and MicroPython, Springer Nature Switzerland AG, 2022. doi:10.1007/978-3-030-88439-0.

[3] G. Bradski, The OpenCV Library, http://opencv.org, accessed: October 30, 2025 (2000).

[4] The NumPy Developers, NumPy - The fundamental package for scientific computing with Python, https://numpy.org/, accessed: October 30, 2025 (2025).