



Marmara University
Faculty of Engineering
Electrical and Electronics Engineering

EE4065.1: INTRODUCTION TO EMBEDDED IMAGE PROCESSING#

Homework #2 Report & Results

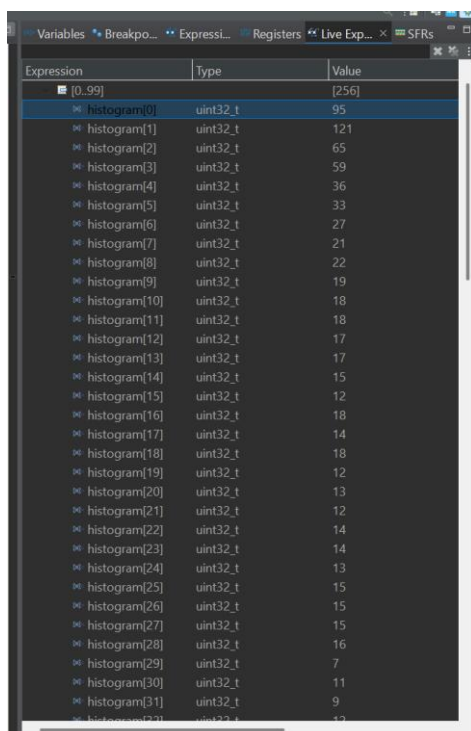
Name Surname	Student ID
Yusuf Yiğit Söyleyici	150723524
Mehmet Açar	150719020

Q1) Histogram Formation

a) C Function for Histogram Calculation The following function was implemented to calculate the histogram of a grayscale image. It initializes a 256-element array to zero and iterates through every pixel in the 128x96 image buffer, incrementing the count for the corresponding intensity value.

```
65 void CalculateHistogram(IMAGE_HandleTypeDef *pImg, uint32_t *pHist)
66 {
67     for(int i = 0; i < 256; i++)
68     {
69         pHist[i] = 0;
70     }
71
72     uint32_t totalPixels = pImg->width * pImg->height;
73
74     for(uint32_t i = 0; i < totalPixels; i++)
75     {
76         uint8_t intensity = pImg->pData[i];
77
78         pHist[intensity]++;
79     }
80 }
81 }
```

b) Histogram Results (STM32CubeIDE) The image was transferred from the PC to the STM32 via UART. A breakpoint was placed after the CalculateHistogram function execution. The histogram array was inspected in the "Live Expressions" tab.



The screenshot shows the 'Live Expressions' window in STM32CubeIDE. The window has tabs for 'Variables', 'Breakpoints', 'Expressions', 'Registers', 'Live Expressions', and 'SFRs'. The 'Live Expressions' tab is active, displaying a table of histogram data. The table has three columns: 'Expression', 'Type', and 'Value'. The data is organized into a tree structure under the expression '[0..99]'. The first row is expanded, showing the 'histogram' array. The array contains 256 elements, each of type 'uint32_t'. The values for the first 32 elements are listed in the table.

Expression	Type	Value
[0..99]		[256]
histogram[0]	uint32_t	95
histogram[1]	uint32_t	121
histogram[2]	uint32_t	65
histogram[3]	uint32_t	59
histogram[4]	uint32_t	36
histogram[5]	uint32_t	33
histogram[6]	uint32_t	27
histogram[7]	uint32_t	21
histogram[8]	uint32_t	22
histogram[9]	uint32_t	19
histogram[10]	uint32_t	18
histogram[11]	uint32_t	18
histogram[12]	uint32_t	17
histogram[13]	uint32_t	17
histogram[14]	uint32_t	15
histogram[15]	uint32_t	12
histogram[16]	uint32_t	18
histogram[17]	uint32_t	14
histogram[18]	uint32_t	18
histogram[19]	uint32_t	12
histogram[20]	uint32_t	13
histogram[21]	uint32_t	12
histogram[22]	uint32_t	14
histogram[23]	uint32_t	14
histogram[24]	uint32_t	13
histogram[25]	uint32_t	15
histogram[26]	uint32_t	15
histogram[27]	uint32_t	15
histogram[28]	uint32_t	16
histogram[29]	uint32_t	7
histogram[30]	uint32_t	11
histogram[31]	uint32_t	9
histogram[32]	uint32_t	12

Q2) Histogram Equalization

a) Derivation of Method

pdf of histogram (k) = $\frac{\# \text{ of } k \text{ valued pixels} \rightarrow n}{\# \text{ of total pixels} \rightarrow M}$

$CDF = 255 \sum_{j=0}^k \frac{n}{M}$ (new value of k valued pixels after histogram equalization applied)

→ ~~let's say~~ from prob theory, if we have transformation $s = T(r)$, the probability density of the output variable s , denoted as $p_s(s)$ is related to input by

$$p_s(s) = p_r(r) \left| \frac{dr}{ds} \right|$$
$$\frac{ds}{dr} = \frac{d}{dr} \left[255 \int_0^r p_r(u) du \right] = 255 p_r(r)$$
$$\Rightarrow p_s(s) = p_r(r) \left| \frac{1}{255 p_r(r)} \right| = \frac{1}{255}$$

→ since $p_s(s)$ is a constant, the probability of every value in the output is equal. This means the output histogram is perfectly flat (uniform).

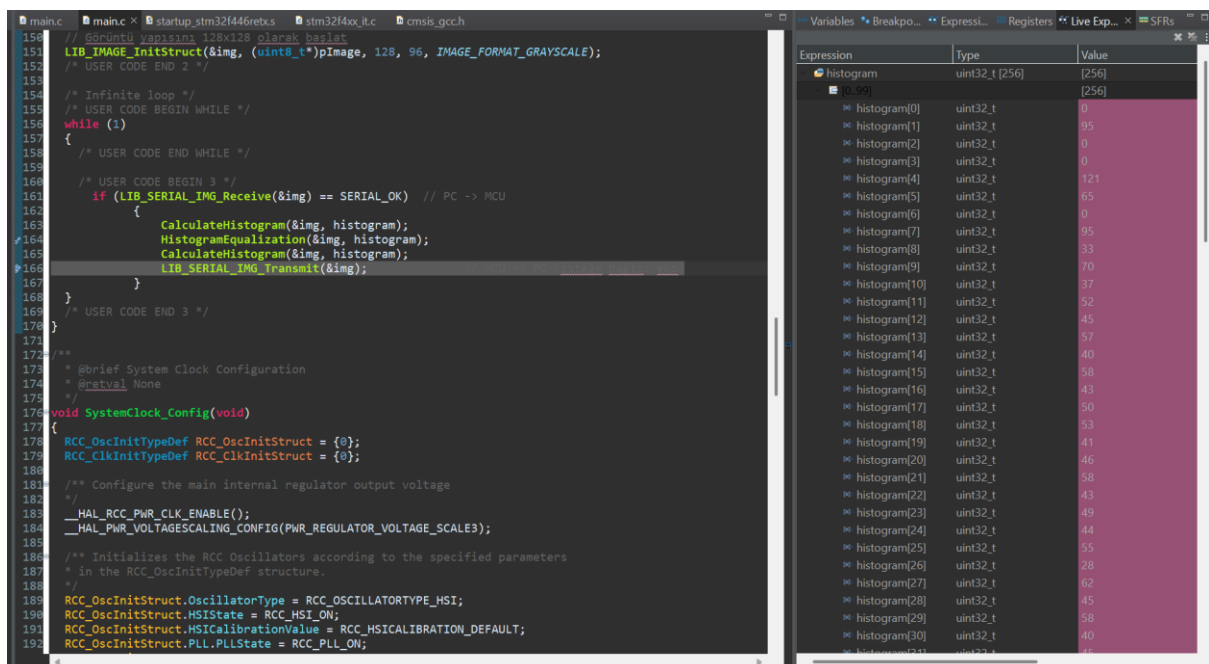
b) C Function for Histogram Equalization The equalization function first calculates the Cumulative Distribution Function (CDF) of the histogram. It then creates a lookup table (map) to normalize the CDF values to the range $[0, 255]$. Finally, it replaces every pixel in the original image with its mapped value.

```

83 void HistogramEqualization(IMAGE_HandleTypeDef *pImg, uint32_t *pHist)
84 {
85     uint32_t totalPixels = pImg->width * pImg->height;
86
87     uint32_t cdf[256];
88
89     cdf[0] = pHist[0];
90     for(int i = 1; i < 256; i++)
91     {
92         cdf[i] = cdf[i-1] + pHist[i];
93     }
94
95     uint8_t map[256];
96     for(int i = 0; i < 256; i++)
97     {
98
99         map[i] = (uint8_t)( (cdf[i] * 255) / totalPixels );
100     }
101
102     for(uint32_t i = 0; i < totalPixels; i++)
103     {
104         uint8_t oldPixel = pImg->pData[i];
105         pImg->pData[i] = map[oldPixel];
106     }
107 }

```

c) Equalized Histogram Results After applying the equalization function, the histogram was recalculated. As seen in the screenshot below, the pixel counts have shifted. Note that some indices now have a count of 0 (gaps), while others have increased counts. This confirms that the contrast stretching algorithm successfully redistributed the intensity values.



Q3) 2D Convolution and Filtering

a) C Function for 2D Convolution A generic convolution function was created that accepts a kernel, a scaling factor, and an offset. The function uses a secondary buffer (pImageOutput) to store results to avoid modifying the source data during calculation. The borders (Row 0/127 and Col 0/95) are skipped.

```
109 void ApplyConvolution(uint8_t* pIn, uint8_t* pOut, int width, int height, const int8_t* kernel, int scale, int offset)
110 {
111     {
112         for (int x = 1; x < width - 1; x++)
113         {
114             int sum = 0;
115             // Kernel Index: 0 1 2
116             //                3 4 5
117             //                6 7 8
118
119             int k = 0;
120             for (int ky = -1; ky <= 1; ky++)
121             {
122                 for (int kx = -1; kx <= 1; kx++)
123                 {
124                     int pIdx = (y + ky) * width + (x + kx);
125                     sum += pIn[pIdx] * kernel[k];
126                     k++;
127                 }
128             }
129
130             if (scale != 0) sum /= scale;
131
132             sum += offset;
133
134             if (sum < 0) sum = 0;
135             if (sum > 255) sum = 255;
136
137             pOut[y * width + x] = (uint8_t)sum;
138         }
139     }
140 }
141
142 }
```

b) Low Pass Filtering Results A standard Box Blur (Average) kernel was applied.

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

The screenshot shows an IDE with a C program for STM32F446RE. The code includes a while loop for user code and a function for system clock configuration. The live expression table on the right shows the values of pImageOutput[i] for i from 111 to 143.

Expression	Type	Value
pImageOutput[111] uint8_t	uint8_t	0 '\000'
pImageOutput[112] uint8_t	uint8_t	0 '\000'
pImageOutput[113] uint8_t	uint8_t	0 '\000'
pImageOutput[114] uint8_t	uint8_t	0 '\000'
pImageOutput[115] uint8_t	uint8_t	0 '\000'
pImageOutput[116] uint8_t	uint8_t	0 '\000'
pImageOutput[117] uint8_t	uint8_t	0 '\000'
pImageOutput[118] uint8_t	uint8_t	0 '\000'
pImageOutput[119] uint8_t	uint8_t	0 '\000'
pImageOutput[120] uint8_t	uint8_t	0 '\000'
pImageOutput[121] uint8_t	uint8_t	0 '\000'
pImageOutput[122] uint8_t	uint8_t	0 '\000'
pImageOutput[123] uint8_t	uint8_t	0 '\000'
pImageOutput[124] uint8_t	uint8_t	0 '\000'
pImageOutput[125] uint8_t	uint8_t	0 '\000'
pImageOutput[126] uint8_t	uint8_t	0 '\000'
pImageOutput[127] uint8_t	uint8_t	0 '\000'
pImageOutput[128] uint8_t	uint8_t	0 '\000'
pImageOutput[129] uint8_t	uint8_t	235 'e'
pImageOutput[130] uint8_t	uint8_t	236 'f'
pImageOutput[131] uint8_t	uint8_t	236 'f'
pImageOutput[132] uint8_t	uint8_t	236 'f'
pImageOutput[133] uint8_t	uint8_t	236 'f'
pImageOutput[134] uint8_t	uint8_t	236 'f'
pImageOutput[135] uint8_t	uint8_t	235 'e'
pImageOutput[136] uint8_t	uint8_t	235 'e'
pImageOutput[137] uint8_t	uint8_t	234 'd'
pImageOutput[138] uint8_t	uint8_t	233 'c'
pImageOutput[139] uint8_t	uint8_t	233 'c'
pImageOutput[140] uint8_t	uint8_t	233 'c'
pImageOutput[141] uint8_t	uint8_t	233 'c'
pImageOutput[142] uint8_t	uint8_t	234 'd'
pImageOutput[143] uint8_t	uint8_t	235 'e'

Value of 0 until 129th index shows that we applied convolution correctly since the first row doesn't included in convolution.

c) High Pass Filtering Results A Laplacian Edge Detection kernel was applied.

$$Kernel = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

The screenshot shows an IDE with a C program for STM32F4xx. The code includes a while loop for image processing, applying a Laplacian edge detection kernel. The variable watch window on the right shows the values of pImageOutput array elements. The first 129 elements are 0, and the remaining elements show non-zero values, indicating successful edge detection.

Expression	Type	Value
pImageOutput[111]	uint8_t	0 '\000'
pImageOutput[112]	uint8_t	0 '\000'
pImageOutput[113]	uint8_t	0 '\000'
pImageOutput[114]	uint8_t	0 '\000'
pImageOutput[115]	uint8_t	0 '\000'
pImageOutput[116]	uint8_t	0 '\000'
pImageOutput[117]	uint8_t	0 '\000'
pImageOutput[118]	uint8_t	0 '\000'
pImageOutput[119]	uint8_t	0 '\000'
pImageOutput[120]	uint8_t	0 '\000'
pImageOutput[121]	uint8_t	0 '\000'
pImageOutput[122]	uint8_t	0 '\000'
pImageOutput[123]	uint8_t	0 '\000'
pImageOutput[124]	uint8_t	0 '\000'
pImageOutput[125]	uint8_t	0 '\000'
pImageOutput[126]	uint8_t	0 '\000'
pImageOutput[127]	uint8_t	0 '\000'
pImageOutput[128]	uint8_t	0 '\000'
pImageOutput[129]	uint8_t	3 '\003'
pImageOutput[130]	uint8_t	2 '\002'
pImageOutput[131]	uint8_t	0 '\0'
pImageOutput[132]	uint8_t	3 '\003'
pImageOutput[133]	uint8_t	0 '\0'
pImageOutput[134]	uint8_t	0 '\0'
pImageOutput[135]	uint8_t	0 '\0'
pImageOutput[136]	uint8_t	0 '\0'
pImageOutput[137]	uint8_t	6 '\006'
pImageOutput[138]	uint8_t	0 '\0'
pImageOutput[139]	uint8_t	4 '\004'
pImageOutput[140]	uint8_t	0 '\0'
pImageOutput[141]	uint8_t	1 '\001'
pImageOutput[142]	uint8_t	0 '\0'
pImageOutput[143]	uint8_t	1 '\001'

Value of 0 until 129th index shows that we applied convolution correctly since the first row doesn't included in convolution.

Q4) Median Filtering

a) **C Function for Median Filtering** The Median Filter collects the 9 neighbors of a pixel into a temporary array, sorts them using a Bubble Sort algorithm, and selects the middle value (index 4). This method is non-linear and is effective for removing salt-and-pepper noise.

```
144 void ApplyMedianFilter(uint8_t* pIn, uint8_t* pOut, int width, int height)
145 {
146     uint8_t window[9];
147
148     for(int y = 1; y < height - 1; y++)
149     {
150         for(int x = 1; x < width - 1; x++)
151         {
152             int k = 0;
153             for(int ky = -1; ky <= 1; ky++)
154             {
155                 for(int kx = -1; kx <= 1; kx++)
156                 {
157                     window[k] = pIn[ (y + ky) * width + (x + kx) ];
158                     k++;
159                 }
160             }
161
162             for(int i = 0; i < 9; i++)
163             {
164                 for(int j = i + 1; j < 9; j++)
165                 {
166                     if(window[i] > window[j])
167                     {
168                         uint8_t temp = window[i];
169                         window[i] = window[j];
170                         window[j] = temp;
171                     }
172                 }
173             }
174
175             pOut[y * width + x] = window[4];
176         }
177     }
178 }
179
180 /* USER CODE END Q4 */
```


b) Median Filter Results The filter was applied to the image. The output buffer was inspected.

The screenshot shows an IDE with a C program for STM32. The code includes functions for histogram calculation, convolution, and median filtering. A loop is currently executing, copying data from the output buffer to the input buffer. The debugger window on the right shows the values of the `pImageOutput` array elements, which are mostly 0, with some non-zero values starting from index 129.

```

261 /* USER CODE BEGIN 3 */
262 if (LIB_SERIAL_IMG_Receive(&img) == SERIAL_OK) // PC -> MCU
263 {
264     /*
265     CalculateHistogram(&img, histogram);
266     HistogramEqualization(&img, histogram);
267     CalculateHistogram(&img, histogram);
268     */
269     ApplyConvolution(img.pData, pImageOutput, 128, 96, Kernel_LowPass, 9, 0);
270     ApplyConvolution(img.pData, pImageOutput, 128, 96, Kernel_HighPass, 1, 0);
271     /*
272     ApplyMedianFilter(img.pData, pImageOutput, 128, 96);
273     */
274     for(int i = 0; i < 128 * 96; i++) {
275         img.pData[i] = pImageOutput[i];
276     }
277     LIB_SERIAL_IMG_Transmit(&img); // MCU -> PC (instead of &img)
278 }
279 /* USER CODE END 3 */
280
281 /**
282 * @brief System Clock Configuration
283 * @retval None
284 */
285 void SystemClock_Config(void)
286 {
287     RCC_OscInitTypeDef RCC_OscInitStruct = {0};
288     RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
289
290     /** Configure the main internal regulator output voltage
291     */
292     __HAL_RCC_PWR_CLK_ENABLE();
293     __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE3);
294
295     /** Initializes the RCC Oscillators according to the specified parameters
296     * in the RCC_OscInitTypeDef structure.
297     */
298     RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
299     RCC_OscInitStruct.HSIState = RCC_HSI_ON;

```

Expression	Type	Value
pImageOutput[115]	uint8_t	0 '\000'
pImageOutput[116]	uint8_t	0 '\000'
pImageOutput[117]	uint8_t	0 '\000'
pImageOutput[118]	uint8_t	0 '\000'
pImageOutput[119]	uint8_t	0 '\000'
pImageOutput[120]	uint8_t	0 '\000'
pImageOutput[121]	uint8_t	0 '\000'
pImageOutput[122]	uint8_t	0 '\000'
pImageOutput[123]	uint8_t	0 '\000'
pImageOutput[124]	uint8_t	0 '\000'
pImageOutput[125]	uint8_t	0 '\000'
pImageOutput[126]	uint8_t	0 '\000'
pImageOutput[127]	uint8_t	0 '\000'
pImageOutput[128]	uint8_t	0 '\000'
pImageOutput[129]	uint8_t	235 'e'
pImageOutput[130]	uint8_t	237 'Y'
pImageOutput[131]	uint8_t	237 'Y'
pImageOutput[132]	uint8_t	237 'Y'
pImageOutput[133]	uint8_t	236 'Y'
pImageOutput[134]	uint8_t	236 'Y'
pImageOutput[135]	uint8_t	235 'e'
pImageOutput[136]	uint8_t	235 'e'
pImageOutput[137]	uint8_t	235 'e'
pImageOutput[138]	uint8_t	234 'd'
pImageOutput[139]	uint8_t	233 'c'
pImageOutput[140]	uint8_t	233 'c'
pImageOutput[141]	uint8_t	234 'd'
pImageOutput[142]	uint8_t	234 'd'
pImageOutput[143]	uint8_t	236 'Y'
pImageOutput[144]	uint8_t	237 'Y'
pImageOutput[145]	uint8_t	238 'Y'
pImageOutput[146]	uint8_t	238 'Y'
pImageOutput[147]	uint8_t	238 'Y'
pImageOutput[148]	uint8_t	237 'Y'

Conclusion

This project successfully established a bidirectional image processing pipeline between a PC and the STM32 Nucleo-F446RE, validating the microcontroller's capability to perform both statistical and spatial image manipulations. Through the implementation of histogram equalization, the system demonstrated effective contrast enhancement, which was mathematically verified by the redistribution of intensity values observed in the debugger. Furthermore, the successful execution of 2D convolution (Low/High Pass) and Median filtering using a secondary output buffer highlighted the importance of correct memory management in embedded C programming, ultimately confirming the system's reliability in handling fundamental digital image processing algorithms in a resource-constrained environment.