



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 Söleyici	150723524
Mehmet Açıar	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
68     for(int i = 0; i < 256; i++)
69     {
70         pHist[i] = 0;
71     }
72
73     uint32_t totalPixels = pImg->width * pImg->height;
74
75     for(uint32_t i = 0; i < totalPixels; i++)
76     {
77         uint8_t intensity = pImg->pData[i];
78
79         pHist[intensity]++;
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 STM32CubeIDE interface with the 'Live Expressions' tab selected. A table displays the histogram array 'histogram' with 256 entries. The first entry is highlighted, showing its type as 'uint32_t' and value as 95. The table has three columns: Expression, Type, and Value. The 'Expression' column lists indices from 0 to 31, and the 'Value' column lists the corresponding counts for each intensity level.

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

Q2) Histogram Equalization

a) Derivation of Method

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

$$CDF = 255 \sum_{j=0}^k \frac{n_j}{M} \left(\begin{array}{l} \text{new value of } k \text{ valued pixels} \\ \text{after histogram equalization applied} \end{array} \right)$$

\rightarrow ~~law~~ 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

$$p_s(s) = p_r(r) \left| \frac{ds}{dr} \right|$$

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

$$\Rightarrow p_s(s) = p_r(r) \left| \frac{1}{255 p_r(r)} \right| = \frac{1}{255}$$

\rightarrow 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).

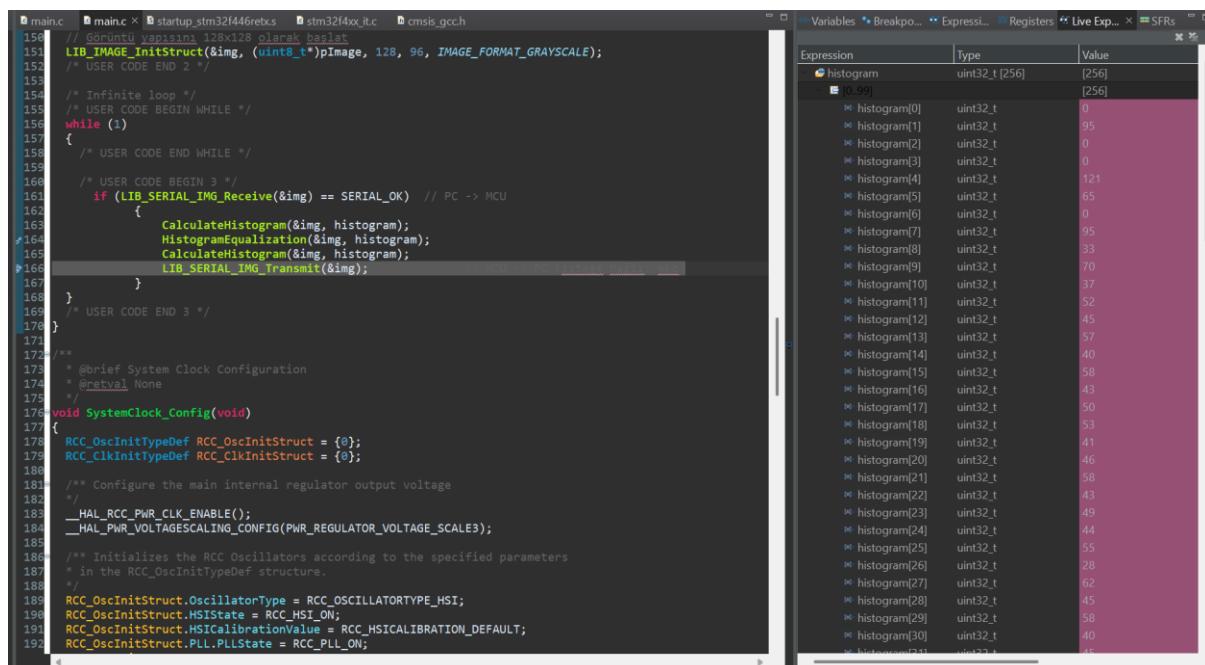
b) C Function for Histogram Equalization The equalization function first calculates the Cumulative Distribution Function (CDF) of the histogram. It then creates a lookup table (map) to normalize the CDF values to the range [0, 255]. Finally, it replaces every pixel in the original image with its mapped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



The screenshot shows a debugger interface with two panes. The left pane displays the C code for the `main.c` file, specifically the `HistogramEqualization` function. The right pane is a variable viewer showing the `histogram` variable. The `histogram` is defined as a `uint32_t [256]` array. The table shows the following data:

Index	Value
0	0
1	95
2	0
3	0
4	121
5	65
6	0
7	95
8	33
9	70
10	37
11	52
12	45
13	57
14	40
15	58
16	43
17	50
18	53
19	41
20	46
21	58
22	43
23	49
24	44
25	55
26	28
27	62
28	45
29	58
30	40

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

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

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

Expression	Type	Value
pImageOutput[111 uint8_t]		0 '\x00'
pImageOutput[112 uint8_t]		0 '\x00'
pImageOutput[113 uint8_t]		0 '\x00'
pImageOutput[114 uint8_t]		0 '\x00'
pImageOutput[115 uint8_t]		0 '\x00'
pImageOutput[116 uint8_t]		0 '\x00'
pImageOutput[117 uint8_t]		0 '\x00'
pImageOutput[118 uint8_t]		0 '\x00'
pImageOutput[119 uint8_t]		0 '\x00'
pImageOutput[120 uint8_t]		0 '\x00'
pImageOutput[121 uint8_t]		0 '\x00'
pImageOutput[122 uint8_t]		0 '\x00'
pImageOutput[123 uint8_t]		0 '\x00'
pImageOutput[124 uint8_t]		0 '\x00'
pImageOutput[125 uint8_t]		0 '\x00'
pImageOutput[126 uint8_t]		0 '\x00'
pImageOutput[127 uint8_t]		0 '\x00'
pImageOutput[128 uint8_t]		0 '\x00'
pImageOutput[129 uint8_t]		0 '\x00'
pImageOutput[130 uint8_t]		235 '\xe'
pImageOutput[131 uint8_t]		236 '\f'
pImageOutput[132 uint8_t]		236 '\f'
pImageOutput[133 uint8_t]		236 '\f'
pImageOutput[134 uint8_t]		236 '\f'
pImageOutput[135 uint8_t]		235 '\e'
pImageOutput[136 uint8_t]		235 '\e'
pImageOutput[137 uint8_t]		234 '\d'
pImageOutput[138 uint8_t]		233 '\c'
pImageOutput[139 uint8_t]		233 '\c'
pImageOutput[140 uint8_t]		233 '\c'
pImageOutput[141 uint8_t]		233 '\c'
pImageOutput[142 uint8_t]		234 '\d'
pImageOutput[143 uint8_t]		235 '\e'
plmImageOutput[444 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 a debugger interface with two main panes. The left pane displays assembly code for the main.c file, specifically the main loop and serial communication logic. The right pane shows a memory dump titled 'Live Exp...' with columns for Expression, type, and Value. The memory dump shows the first 144 bytes of memory, where the first 129 bytes are all zero, followed by a series of values starting at index 129.

```

main.c
213 /* Infinite loop */
214 /* USER CODE BEGIN WHILE */
215 while (1)
216 {
217     /* USER CODE END WHILE */
218
219     /* USER CODE BEGIN 3 */
220     if (LIB_SERIAL_IMG_Receive(&img) == SERIAL_OK) // PC -> MCU
221     {
222         //CalculateHistogram(&img, histogram);
223         //HistogramEqualization(&img, histogram);
224         //CalculateHistogram(&img, histogram);
225         ApplyConvolution(img.pData, pImageOutput, 128, 96, Kernel_LowPass, 0, 0);
226         ApplyConvolution(img.pData, pImageOutput, 128, 96, Kernel_HighPass, 1, 0);
227
228         for(int i = 0; i < 128 * 96; i++) {
229             img.pData[i] = pImageOutput[i];
230         }
231         LIB_SERIAL_IMG_Transmit(&img);           // MCU -> PC (data[0..128*96], end)
232     }
233     /* USER CODE END 3 */
234 }
235
236 /**
237 * @brief System Clock Configuration
238 * @retval None
239 */
240 void SystemClock_Config(void)
241 {
242     RCC_OscInitTypeDef RCC_OscInitStruct = {0};
243     RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
244
245     /** Configure the main internal regulator output voltage
246     */
247     __HAL_RCC_PWR_CLK_ENABLE();
248     __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE3);
249
250     /** Initializes the RCC Oscillators according to the specified parameters
251     * in the RCC_OscInitTypeDef structure.
252     */
253     RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
254 }
```

Expression	Type	Value
pImageOutput[111]	uint8_t	0 \x00'
pImageOutput[112]	uint8_t	0 \x00'
pImageOutput[113]	uint8_t	0 \x00'
pImageOutput[114]	uint8_t	0 \x00'
pImageOutput[115]	uint8_t	0 \x00'
pImageOutput[116]	uint8_t	0 \x00'
pImageOutput[117]	uint8_t	0 \x00'
pImageOutput[118]	uint8_t	0 \x00'
pImageOutput[119]	uint8_t	0 \x00'
pImageOutput[120]	uint8_t	0 \x00'
pImageOutput[121]	uint8_t	0 \x00'
pImageOutput[122]	uint8_t	0 \x00'
pImageOutput[123]	uint8_t	0 \x00'
pImageOutput[124]	uint8_t	0 \x00'
pImageOutput[125]	uint8_t	0 \x00'
pImageOutput[126]	uint8_t	0 \x00'
pImageOutput[127]	uint8_t	0 \x00'
pImageOutput[128]	uint8_t	0 \x00'
pImageOutput[129]	uint8_t	3 \x003'
pImageOutput[130]	uint8_t	2 \x002'
pImageOutput[131]	uint8_t	0 \x00'
pImageOutput[132]	uint8_t	3 \x003'
pImageOutput[133]	uint8_t	0 \x00'
pImageOutput[134]	uint8_t	0 \x00'
pImageOutput[135]	uint8_t	0 \x00'
pImageOutput[136]	uint8_t	0 \x00'
pImageOutput[137]	uint8_t	6 \x006'
pImageOutput[138]	uint8_t	0 \x00'
pImageOutput[139]	uint8_t	4 \x004'
pImageOutput[140]	uint8_t	0 \x00'
pImageOutput[141]	uint8_t	1 \x001'
pImageOutput[142]	uint8_t	0 \x00'
pImageOutput[143]	uint8_t	1 \x001'

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
153             int k = 0;
154             for(int ky = -1; ky <= 1; ky++)
155             {
156                 for(int kx = -1; kx <= 1; kx++)
157                 {
158                     window[k] = pIn[(y + ky) * width + (x + kx)];
159                     k++;
160                 }
161             }
162
163             for(int i = 0; i < 9; i++)
164             {
165                 for(int j = i + 1; j < 9; j++)
166                 {
167                     if(window[i] > window[j])
168                     {
169                         uint8_t temp = window[i];
170                         window[i] = window[j];
171                         window[j] = temp;
172                     }
173                 }
174             }
175
176             pOut[y * width + x] = window[4];
177         }
178     }
179 }
```

/* USER CODE END Q */

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

The screenshot shows a debugger interface with two panes. The left pane displays the C code for the `main.c` file, specifically the section for handling serial communication between the PC and the STM32 MCU. The right pane shows a memory dump of the `pImageOutput` buffer, which contains 128x96 uint8_t values. The values are mostly zero, with some non-zero values appearing after the loop where the median filter is applied.

```

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, 0, 0);
270
271      ApplyConvolution(img.pData, pImageOutput, 128, 96, Kernel_HighPass, 1, 0);
272
273      /*
274      ApplyMedianFilter(img.pData, pImageOutput, 128, 96);
275
276      for(int i = 0; i < 128 * 96; i++) {
277          img.pData[i] = pImageOutput[i];
278      }
279
280      LIB_SERIAL_IMG_Transmit(&img); // MCU -> PC (isteğe bağlı: eko)
281  }
282  /* USER CODE END 3 */
283
284 /**
285 * @brief System Clock Configuration
286 */
287 void SystemClock_Config(void)
288 {
289     RCC_OscInitTypeDef RCC_OscInitStruct = {0};
290     RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
291
292     /** Configure the main internal regulator output voltage
293     */
294     HAL_RCC_PWR_CLK_ENABLE();
295     HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE3);
296
297     /** Initializes the RCC Oscillators according to the specified parameters
298     * in the RCC_OscInitTypeDef structure.
299     */
300
301     RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
302     RCC_OscInitStruct.HSIStrict = RCC_HSI_ON;
303
304 }

```

Expression	Type	Value
pImageOutput[115]	uint8_t	0 \x000'
pImageOutput[116]	uint8_t	0 \x000'
pImageOutput[117]	uint8_t	0 \x000'
pImageOutput[118]	uint8_t	0 \x000'
pImageOutput[119]	uint8_t	0 \x000'
pImageOutput[120]	uint8_t	0 \x000'
pImageOutput[121]	uint8_t	0 \x000'
pImageOutput[122]	uint8_t	0 \x000'
pImageOutput[123]	uint8_t	0 \x000'
pImageOutput[124]	uint8_t	0 \x000'
pImageOutput[125]	uint8_t	0 \x000'
pImageOutput[126]	uint8_t	0 \x000'
pImageOutput[127]	uint8_t	0 \x000'
pImageOutput[128]	uint8_t	0 \x000'
pImageOutput[129]	uint8_t	235 \x0e'
pImageOutput[130]	uint8_t	237 \x0f'
pImageOutput[131]	uint8_t	237 \x0f'
pImageOutput[132]	uint8_t	237 \x0f'
pImageOutput[133]	uint8_t	236 \x0f'
pImageOutput[134]	uint8_t	236 \x0f'
pImageOutput[135]	uint8_t	235 \x0e'
pImageOutput[136]	uint8_t	235 \x0e'
pImageOutput[137]	uint8_t	235 \x0e'
pImageOutput[138]	uint8_t	234 \x0d'
pImageOutput[139]	uint8_t	233 \x0c'
pImageOutput[140]	uint8_t	233 \x0c'
pImageOutput[141]	uint8_t	234 \x0d'
pImageOutput[142]	uint8_t	234 \x0d'
pImageOutput[143]	uint8_t	236 \x0f'
pImageOutput[144]	uint8_t	237 \x0f'
pImageOutput[145]	uint8_t	238 \x0f'
pImageOutput[146]	uint8_t	238 \x0f'
pImageOutput[147]	uint8_t	238 \x0f'

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.