EE326 Lab Report 2: Spatial Transforms and Filtering

11812418 樊青远[1]

[1] *School of Microelectronics, SUSTech Email: fanqy2018@mail.sustech.edu.cn*

Abstract

This report provides the intuitive introduction on different types of spacial transforms and filtering methods implemented in python.

Keywords: spacial transform, Filtering

Contents

EE326 DIP (2021)

DOI: 10.1017/pan.xxxx.xx

Corresponding author
Qingyuan Fan

Advised by
Yajun Yu

1 Objective

The spacial domain refers the normal image space. In this lab, we will attempt to implement the spacial transformation on the input image in two main perspectives: Transform of image intensity and spatial filtering. The transformation of image intensity can be used in the bright, contrast and saturation of image tuning, the spacial filtering can be adapted to noise canceling, and the watermark removing, which are both widely used in professional and amateur fields like medical imaging and general purpose photography.

In the following context, we will try to implement several classic image processing algorithms in the field of spacial transformation and filtering with python , which include histogram equalization, histogram matching, local histogram matching and noise canceling by spacial filtering.

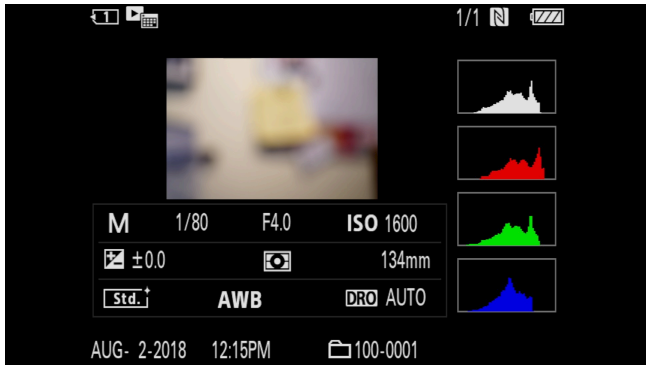


Figure 1. A typical display indication of histogram in SONY camera.

2 Histogram Equalization

2.1 Algorithm

Histogram equalization is a method in image processing of contrast adjustment using the image's histogram.

We start with a discrete grayscale image x and let n_i be the number of occurrences of gray level i. The probability of an occurrence of a pixel of level i in the image is:

$$p_x(i) = p(x = i) = \frac{n_i}{n}, \quad 0 \leq i < L \tag{1}$$

In equation 1, L being the total number of gray levels in the image (in our experiment, we set it to 256), n being the total number of pixels in the image, and $p_x(i)$ is the image's histogram for pixel value i, which has been normalized to $[0, 1]$.

Secondly, we also need to define a cumulative distribution function, which is shown in formula 2:

$$cdf_x(i) = \sum_{j=0}^{i} p_x(x = j) \tag{2}$$

The result of this formula can be also recognized as the image's accumulated normalized histogram.

It's time for us to construct a transformation function $y = T(x)$ to illustrate the flatten projection of the histogram. In formula 3, K is a constant.

$$cdf_y(i) = iK \tag{3}$$

Combing formula 1, 2 and 3, we could get:

$$cdf_y(y') = cdf_y(T(k)) = cdf_x(k) \tag{4}$$

To map the values after transformation back into their original range, the following transformation 5 needs to be applied on the result:

$$y' = y \cdot (\max\{x\} - \min\{x\}) + \min\{x\} \tag{5}$$

2.2 Flow Chart

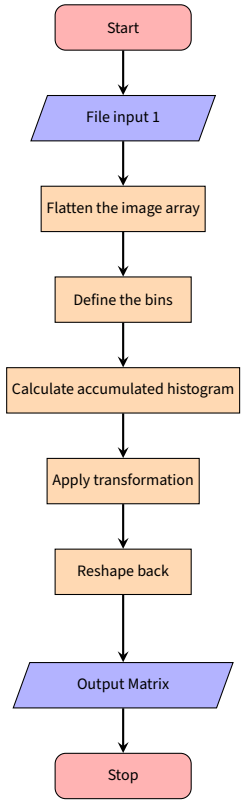


Figure 2. Flow chart of Histogram Equalization

2.3 Python Code

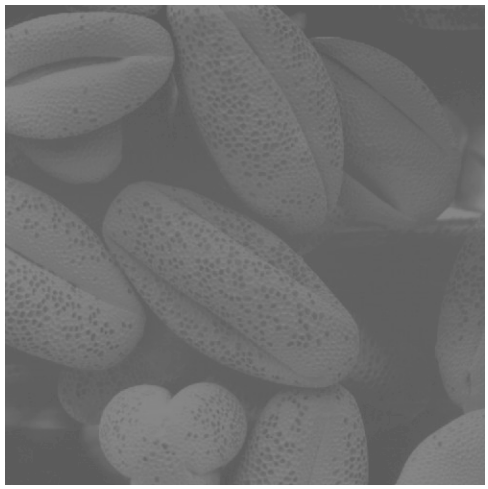
To implement the histogram equalization in python, we divide the function for calculating the accumulation histogram and the transformation into a seperate function called `hist_equ()`. Instantiating this function can also facilitate us to call directly in the subsequent histogram transformation.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
```

```python
from PIL import Image
import time


def hist_equ(img_flat,bins):
    bins_arr = np.arange(bins + 1)
    # generate histogram
    histogram_in_data, histogram_in_index = np.histogram(img_flat,
    ↪  bins=bins_arr)

    # history equ
    # histogram Normalized
    histogram_in_data_normalized = histogram_in_data /
    ↪  np.sum(histogram_in_data)
    # historygram_accumulation
    histogram_in_data_accumulated =
    ↪  np.round(np.add.accumulate(histogram_in_data_normalized) * bins)

    img_after_hist_equ = histogram_in_data_accumulated[img_flat]

    histogram_out_data, histogram_out_index =
    ↪  np.histogram(img_after_hist_equ, bins=bins_arr)
    return histogram_out_data, img_after_hist_equ

def hist_equ_11812418(input_image):
    # Insert code here
    img = Image.open(input_image)
    img_arr = np.asarray(img)
    img_flat = img_arr.flatten() #flatten image to 1D array
    bins = 256
    input_hist, histogram_in_index = np.histogram(img_flat,
    ↪  bins=np.arange(bins + 1))

    output_hist, output_image = hist_equ(img_flat, bins)
    output_image = np.reshape(output_image, img_arr.shape)
    return (output_image, output_hist, input_hist)


def plot_hist(hist,filename,title):
    plt.figure(figsize=(8, 6), dpi=300)
    plt.plot(hist)
    plt.title(title)
    plt.savefig(filename)
    plt.show()

# Q3_1_1
start_time = time.time()
(out_img2,hist_out,hist_in) = hist_equ_11812418("Q3_1_1.tif")

```

```
47  op_image = Image.fromarray(out_img2.astype(np.uint8))
48  print("--- %s seconds ---" % (time.time() - start_time))
49  op_image.save("output/img/hist_equ/Q3_1_1_M.tif")
50
51  plot_hist(hist_in,"output/img/hist_equ/Q3_1_1_hist.png","Histogram before
    ↪  histogram equalization")
52  plot_hist(hist_out,"output/img/hist_equ/Q3_1_1_M_hist.png","Histogram
    ↪  after histogram equalization")
53
54  # Q3_1_2
55  start_time = time.time()
56  (out_img2,hist_out,hist_in) = hist_equ_11812418("Q3_1_2.tif")
57
58  op_image = Image.fromarray(out_img2.astype(np.uint8))
59  print("--- %s seconds ---" % (time.time() - start_time))
60  op_image.save("output/img/hist_equ/Q3_1_2_M.tif")
61
62  plot_hist(hist_in,"output/img/hist_equ/Q3_1_2_hist.png","Histogram before
    ↪  histogram equalization")
63  plot_hist(hist_out,"output/img/hist_equ/Q3_1_2_M_hist.png","Histogram
    ↪  after histogram equalization")
```

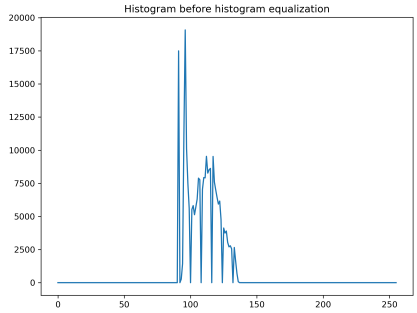
2.4 Result



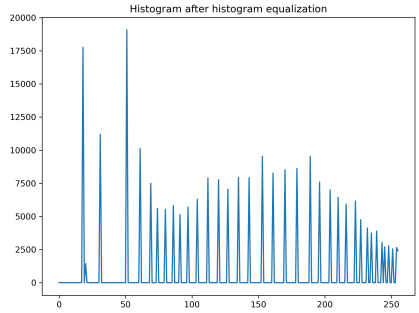
(a) Before histogram equalization.
Q3_1_1.tif

(b) After histogram equalization.
Q3_1_1.tif

Figure 3. Input and output image **(Q3_1_1.tif)**

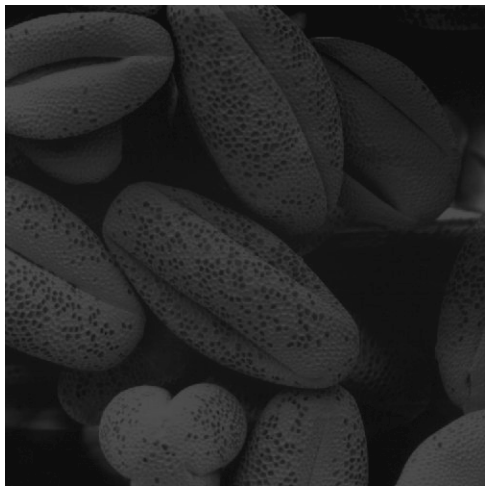


(a) Histogram before histogram equalization. Q3_1_1.tif



(b) Histogram after histogram equalization. Q3_1_1.tif

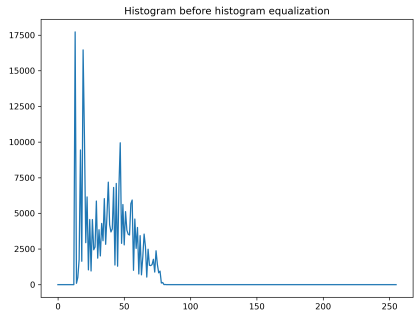
Figure 4. The histograms of input and output image **(Q3_1_1.tif)**



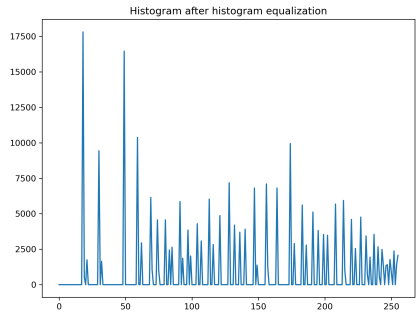
(a) Before histogram equalization. Q3_1_2.tif

(b) After histogram equalization. Q3_1_2.tif

Figure 5. Input and output image **(Q3_1_2.tif)**



(a) Histogram before histogram equalization. Q3_1_2.tif



(b) Histogram after histogram equalization. Q3_1_2.tif

Figure 6. The histograms of input and output image **(Q3_1_2.tif)**

Although the input image (3a, 5a) we use is very different from the histogram, after the histogram equalization, the histogram of the output image (4b, 6b) has been balanced to a greater extent.

Filename	Execution time
Q3_1_2.tif	0.20s
Q3_1_2.tif	0.17s

Table 1. Execution time of histogram equalization method. The test was conducted with a PC equipped with AMD Ryzen5 3600, and the python version is 3.8.

. Nevertheless, since the histogram of the fig 6a is concentrated in the dark part, even though the histogram equalization is carried out, we cannot fully restore the details of the dark part of the image.

As we have make full use of the MKL library provided by numpy, execution time of program shown in table was controlled within 0.2 seconds.

3 Histogram Matching

3.1 Algorithm

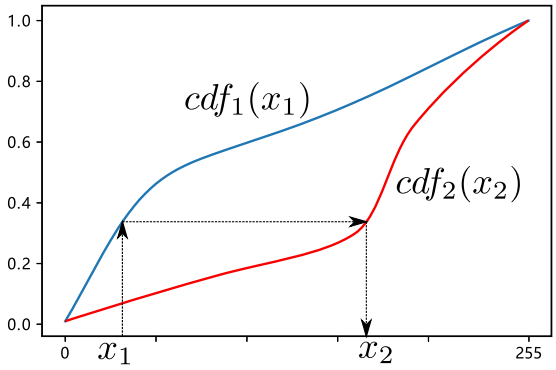


Figure 7. Diagram of histogram matching

Consider a grayscale input image X again. It has a probability density function $p_r(r)$, where r is a grayscale value, and $p_r(r)$ is the probability of that value. This probability can easily be computed from the histogram of the image by the following formula:

$$p_r(r_j) = \frac{n_j}{n} \tag{6}$$

Where n_j is the frequency of the grayscale value r_j, and n is the total number of pixels in the image.

Now a transformation of $p_r(r)$ is needed to convert it to a desired output probability density $p_z(z)$. So each probability density function can easily be mapped to its cumulative distribution function by:

$$S(r_k) = \sum_{j=0}^{k} p_r(r_j), \qquad k = 0, 1, 2, 3, \ldots, L-1 \tag{7}$$

$$G(z_k) = \sum_{j=0}^{k} p_z(z_j), \qquad k = 0, 1, 2, 3, \ldots, L-1 \tag{8}$$

L is the total number of gray level.
The complete formula can be simplified to:

$$S(r_j) = G(z_j) \tag{9}$$

$$z = G^{-1}S(r) \tag{10}$$

3.2 Flow Chart

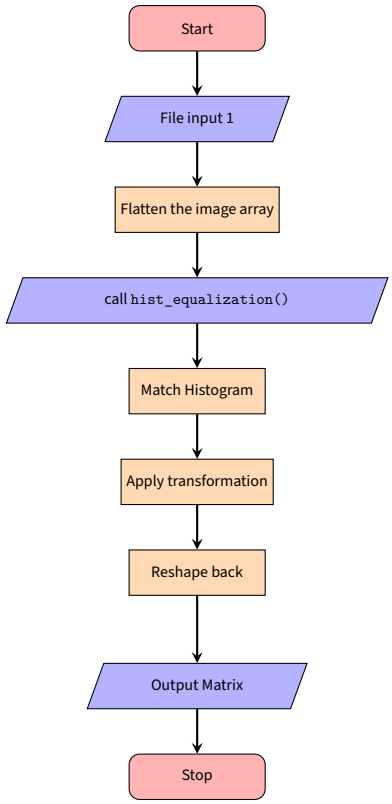


Figure 8. Flow chart of Histogram Matching

3.3 Python Code

In this program, we separate the functions for matching histograms and averaging histograms to achieve modularity of the program.

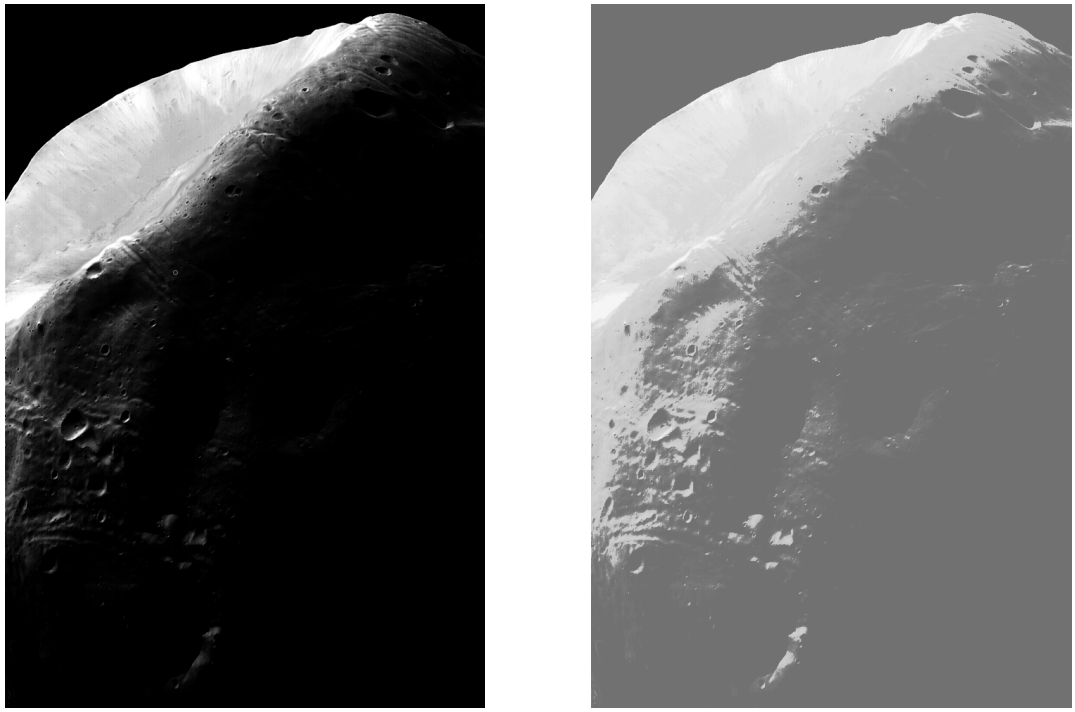
```
1  import time
2  import matplotlib.pyplot as plt
3  import numpy as np
4  from PIL import Image
5
6  # define hist
7  hist = np.concatenate((np.linspace(0, 20, 8 - 0),
```

```
                        np.linspace(20, 0.75, 16 - 8),
                        np.linspace(0.75, 0, 184 - 16),
                        np.linspace(0, 0.5, 200 - 184),
                        np.linspace(0.5, 0, 256 - 200)), axis=0)

def hist_accumulation(hist, bins):
    histogram_in_data_normalized = hist / np.sum(hist)
    # historygram_accumulation
    histogram_in_data_accumulated =
    ↪  np.round(np.add.accumulate(histogram_in_data_normalized) * bins)

    return histogram_in_data_accumulated


def hist_equ(img_flat, bins):
    bins_arr = np.arange(bins + 1)
    # generate histogram
    histogram_in_data, histogram_in_index = np.histogram(img_flat,
    ↪  bins=bins_arr)
    # history equ
    # histogram Normalized
    histogram_in_data_normalized = histogram_in_data /
    ↪  np.sum(histogram_in_data)
    # historygram_accumulation
    histogram_in_data_accumulated =
    ↪  np.round(np.add.accumulate(histogram_in_data_normalized) * bins)
    img_after_hist_equ = histogram_in_data_accumulated[img_flat]
    return histogram_in_data_accumulated, histogram_in_data,
    ↪  img_after_hist_equ


def hist_match(hist_in, hist_desired):
    hist_len = len(hist_in)
    hist_out = np.zeros(hist_len)

    def hist_match_index(i):
        index = min(range(hist_len), key=lambda j: abs(hist_desired[j] -
        ↪  hist_in[i]))
        hist_out[i] = index

    # def hist_match_index_par(i):
    #     return min(range(hist_len), key=lambda j: abs(hist_in[j] -
    ↪  hist_desired[i]))
    #
    # c = Parallel( n_jobs = 4 )( delayed( hist_match_index_par )( item )
    ↪  for item in range(hist_len))
    # return c
    # init pool consume more time

```

```
49      for i in range(hist_len):
50          hist_match_index(i)
51      return hist_out
52
53
54  def hist_match_11812418(input_image, spec_hist):
55      # Insert code here
56      img = Image.open(input_image)
57      img_arr = np.asarray(img)
58      img_flat = img_arr.flatten()  # flatten image to 1D array
59      bins = 256
60
61      # input img hist equ
62      input_img_hist_after_hist_equ, input_hist, input_img_after_hist_equ =
        ↪ hist_equ(img_flat, bins)
63      desired_hist_accumulated = hist_accumulation(spec_hist, bins)
64      # in: input_img_hist_after_hist_equ
65      # desired: desired_hist_accumulated
66
67      hist_lut = np.array(hist_match(input_img_hist_after_hist_equ,
        ↪ desired_hist_accumulated))
68      img_after_hist_match = hist_lut[input_img_after_hist_equ.astype(int) -
        ↪ 1]
69
70      img_new_arr = np.reshape(img_after_hist_match, img_arr.shape)
71      output_hist, histogram_out_index = np.histogram(img_after_hist_match,
        ↪ bins=np.arange(bins + 1))
72
73      return (img_new_arr, output_hist, input_hist)
74
75
76  def plot_hist(hist, filename, title):
77      plt.figure(figsize=(8, 6), dpi=300)
78      plt.plot(hist)
79      plt.title(title)
80      plt.savefig(filename)
81      plt.show()
82
83
84  if __name__ == '__main__':
85      # Q3_2
86      start_time = time.time()
87      (out_img2, hist_out, hist_in) = hist_match_11812418("Q3_2.tif", hist)
88
89      op_image = Image.fromarray(out_img2.astype(np.uint8))
90      print("--- %s seconds ---" % (time.time() - start_time))
91      op_image.save("output/img/hist_matching/Q3_2_M.tif")
92
```

```
93      plot_hist(hist_in, "output/img/hist_matching/Q3_2_hist.png",
        ↪  "Histogram before histogram matching")
94      plot_hist(hist_out, "output/img/hist_matching/Q3_2_M_hist.png",
        ↪  "Histogram after histogram matching")
```

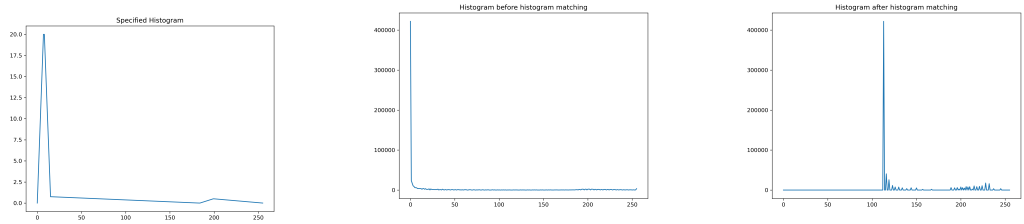
3.4 Result



(a) Before histogram matching. Q3_2.tif

(b) After histogram matching. Q3_2.tif

Figure 9. Input and output image **(Q3_2.tif)**



(a) Specified Histogram

(b) Before histogram matching. Q3_2.tif

(c) After histogram matching. Q3_2.tif

Figure 10. Histogram of input and output image **(Q3_2.tif)**

By using our customized histogram, we recovered some of the dark details of the meteorite in Figure 9a. Although the histogram matching reduces the saturation of the image, because the histogram is more scattered, we can get more details from the processed image.

Methodology	Execution time
Without joblib	0.064s
With joblib (6 pools)	0.078s

Table 2. Execution time of histogram matching method. The test was conducted with a PC equipped with AMD Ryzen5 3600, and the python version is 3.8.

In this question, we realized that finding using for loop to implement the function to find nearest point (formula 10) is asynchronous, which may have the room for us to optimization in this part of the program. We latterly use the native library `joblib` to transform the asynchronous into parallel computation. The rewritten parallel loop is as follows:

```
def hist_match_index_par(i):
    return min(range(hist_len), key=lambda j: abs(hist_in[j] -
    ↪ hist_desired[i]))
    c = Parallel( n_jobs = 6 )( delayed( hist_match_index_par )( item )
    ↪ for item in range(hist_len))
    return c
```

However, due to the initiate process of the parallel pool needs time, so the execution time of the parallel loop is 0.1 seconds slower than the previous for loop. However, when the time saved by parallelism for the program is greater than the time required to open the parallel pool, the parallel loop can greatly improve the speed of the program. We will discuss these optimizations in the next two questions.

4 Local Histogram Equalization

4.1 Algorithm

The core algorithm of Local histogram equalization is basically the same as that of Histogram Equalization. The difference of local histogram equalization is that it divides the input image into several small parts, and implement the histograms equalization algorithm in every slice. This allows us to enhance details over small areas in an image.

4.2 Flow Chart

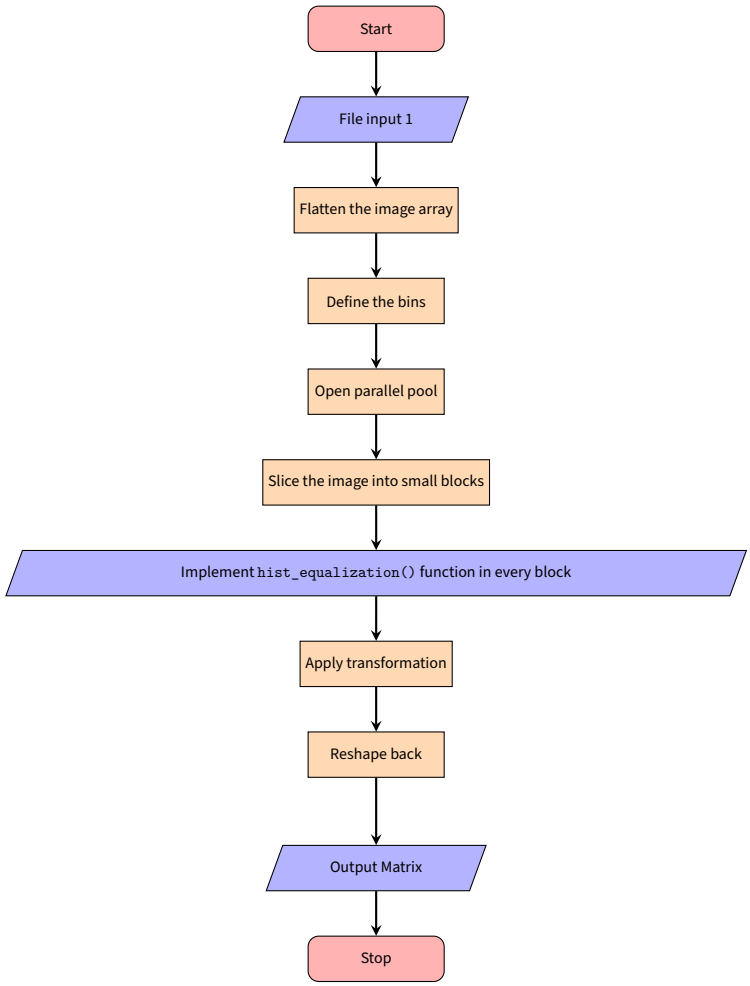


Figure 11. Flow chart of Histogram Equalization

4.3 Python Code

The same as the previous question, in order to maximize the use of hardware, we use the hardware acceleration features of numpy as much as possible, and use joblib package for parallel computing.

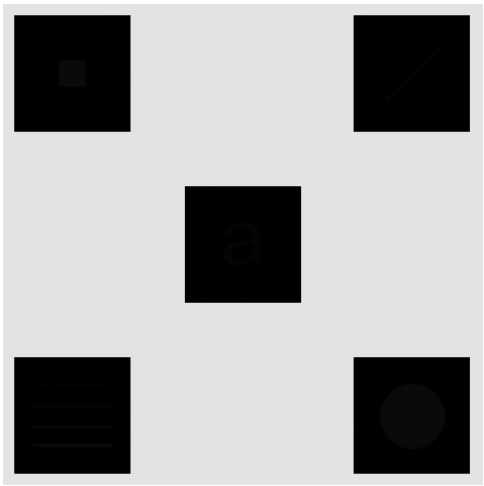
```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from joblib import Parallel, delayed
import time


def hist_equ(img_flat, bins):
    bins_arr = np.arange(bins + 1)
    # generate histogram
    histogram_in_data, histogram_in_index = np.histogram(img_flat,
    ↪  bins=bins_arr)
```

```
12      # history equ
13      # histogram Normalized
14      histogram_in_data_normalized = histogram_in_data /
        ↪  np.sum(histogram_in_data)
15      # historygram_accumulation
16      histogram_in_data_accumulated =
        ↪  np.round(np.add.accumulate(histogram_in_data_normalized) * bins)
17      img_after_hist_equ = histogram_in_data_accumulated[img_flat - 2]
18      return img_after_hist_equ
19
20
21  def local_hist_equ_11812418(in_img, m_size):
22      img = Image.open(in_img)
23      in_img = np.asarray(img)
24      row, col = in_img.shape
25      out_img = np.zeros(in_img.shape, int)
26
27      bin_num = 256
28      bins_arr = range(bin_num + 1)
29
30      histogram_in_data, histogram_in_index = np.histogram(in_img.flatten(),
        ↪  bins=bins_arr)
31
32      # PAR
33      def local_hist_equ_par(i, j):
34          local_img = in_img[i:i + m_size, j:j + m_size]
35
36          return (hist_equ(local_img.flatten(), bin_num)).reshape((m_size,
            ↪  m_size))
37
38      local_hist_equ_par(1, 1)
39
40      par_output =
        ↪  np.array(Parallel(n_jobs=12)(delayed(local_hist_equ_par)(i, j)
41                                              for i in list(range(0, row -
                                                ↪  m_size, m_size)) + [row
                                                ↪  - m_size - 1]
42                                              for j in list(range(0, col -
                                                ↪  m_size, m_size)) + [col
                                                ↪  - m_size - 1]
43                                              ))
44
45      # put array back
46
47      for i in range(int(row / m_size)):
48          for j in range(int(col / m_size)):
49              out_img[i * m_size:(i * m_size + m_size), j * m_size:(j *
                ↪  m_size + m_size)] = par_output[
50                  int(i * row / m_size + j)]
```

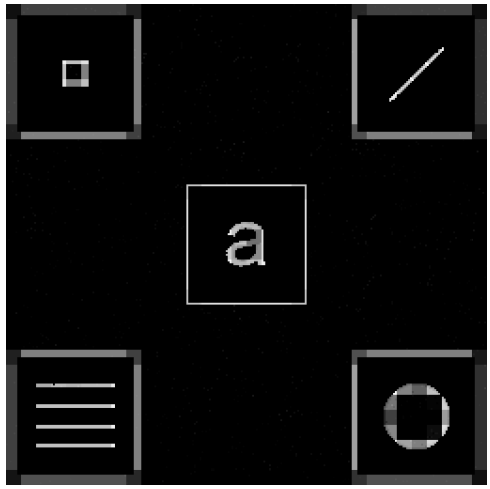
```
51
52      histogram_out_data, histogram_out_index =
        ↪  np.histogram(out_img.flatten(), bins=bins_arr)
53
54      return (out_img, histogram_out_data, histogram_in_data)
55
56
57  def plot_hist(hist, filename, title):
58      plt.figure(figsize=(8, 6), dpi=300)
59      plt.plot(hist)
60      plt.title(title)
61      plt.savefig(filename)
62      plt.show()
63
64
65  start_time = time.time()
66  (out_img2, hist_out, hist_in) = local_hist_equ_11812418("Q3_3.tif", 16)
67
68  op_image = Image.fromarray(out_img2.astype(np.uint8))
69  print("--- %s seconds ---" % (time.time() - start_time))
70  op_image.save("output/img/local_hist_equal/Q3_3_M.tif")
71  op_image.save("output/img/local_hist_equal/Q3_3_M.png")
72
73  plot_hist(hist_in, "output/img/local_hist_equal/Q3_2_hist.png", "Histogram
    ↪  before local histogram equalization")
74  plot_hist(hist_out, "output/img/local_hist_equal/Q3_3_M_hist.png",
    ↪  "Histogram after local histogram equalization")
```

4.4 Result

From the histogram and the image below, we can clearly see that after the local histogram equalization, the readability of the shapes with low contrast in the image has been greatly improved.

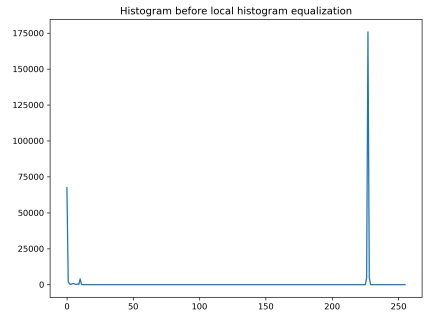


(a) Before Local Histogram Equalization. Q3_3.tif

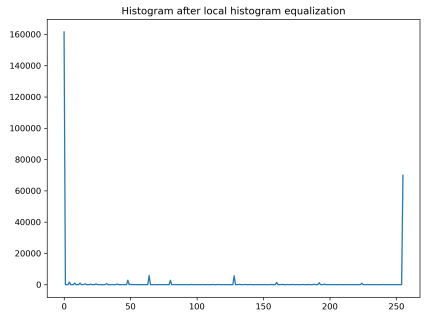


(b) After Local Histogram Equalization. Q3_3.tif m_size=16

Figure 12. Input and output image **(Q3_2.tif)**



(a) Histogram before Local Histogram Equalization. Q3_3.tif



(b) Histogram after Local Histogram Equalization. Q3_3.tif m_size=16

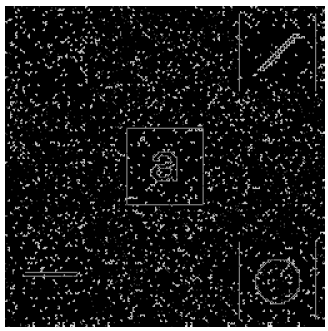
Figure 13. Input and output image **(Q3_3.tif)**

But at the same time we also noticed that after the local histogram equalization, the noise of the image has also increased significantly, which is especially obvious when m_size is small.

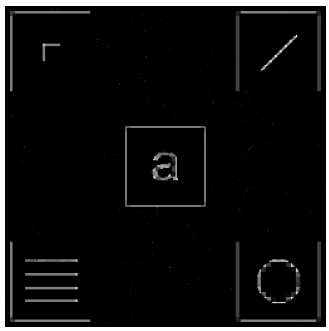
To eliminate this issue, an compromised is reduce the overall brightness of the mapping by 1 or 2, which is implemented by the code below:

```
g_after_hist_equ = histogram_in_data_accumulated[img_flat -2]
```

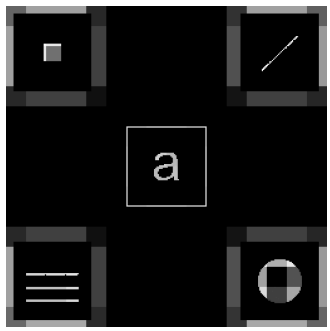
This method can efficiently eliminate the noise that is closed to zero. However, this implementation could also leads to the loss of resolution of image. Therefore, this method cannot be used when m_size is small.



(a) After Local Histogram Equalization. m_size=4



(b) After Local Histogram Equalization. m_size=8



(c) After Local Histogram Equalization. m_size=32

Figure 14. Histogram of input and output image **(Q3_3.tif)**

We also found that the same calculation method can save a lot of time by using highly utilizing hardware-accelerated package (e.g. numpy). According to the table below, using hardware acceleration feature of the matrix operation by numpy and parallel joblib could, the calculation time can be reduced by more than 80 times, which could be time-saving under processing high resolution image.

Methodology	Execution time	CPU Utilization
Without joblib and numpy.sum()	60.2s	12%
Without joblib	5.85s	12%
With joblib (12 pools)	0.71s	97%

Table 3. Execution time of local histogram equalizing method (m_size = 4). The test was conducted with a PC equipped with AMD Ryzen5 3600, and the python version is 3.8.

5 Reduce the Salt-and-pepper Noise

5.1 Algorithm

Salt-and-pepper noise is a form of noise sometimes seen on images. It is also known as impulse noise. This noise can be caused by sharp and sudden disturbances in the image signal. In order to remove these noises, a widely used method is median filtering, which replaces the value of a pixel by the median of the intensity values in the neighborhood of that pixel.

Median filtering is very widely used in digital image processing because, under certain conditions, it preserves edges while removing noise, also having applications in signal processing.

5.2 Flow Chart

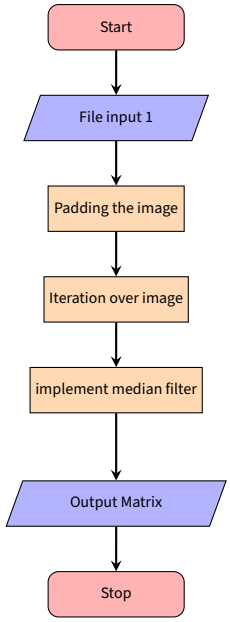


Figure 15. Flow chart of Median Filter

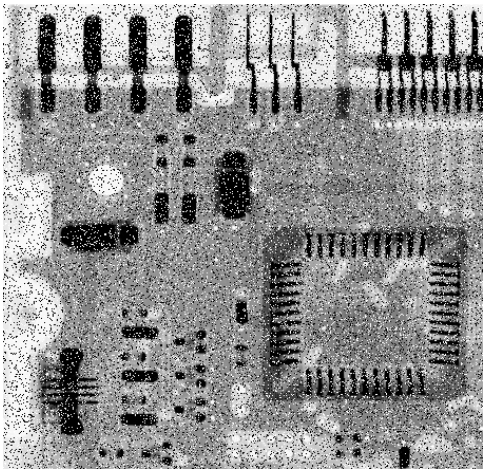
5.3 Python Code

```
import time
import numpy as np
from PIL import Image
from joblib import Parallel, delayed


def reduce_SAP_11812418(input_image, n_size):
```

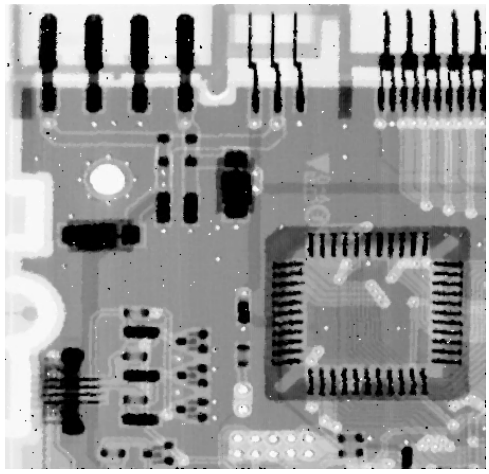
```
    img = Image.open(input_image)
    input_image = np.asarray(img)
    input_image_padding = np.copy(input_image)

    pad_num = int((n_size - 1) / 2)
    input_image_padding = np.pad(input_image_padding, (pad_num, pad_num),
     ↪  mode="constant", constant_values=0)
    m, n = input_image_padding.shape
    output_image = np.copy(input_image_padding)

    # filter_PAR
    def spacial_filter(i, j):
        a = np.median(input_image_padding[i - pad_num:i + pad_num + 1, j -
         ↪  pad_num:j + pad_num + 1])
        return a

    output_img_flat = Parallel(n_jobs=6)(delayed(spacial_filter)(i, j)
                                         for i in range(pad_num, m -
                                          ↪  pad_num)
                                         for j in range(pad_num, n -
                                          ↪  pad_num)
                                         )

    output_image = np.reshape(output_img_flat, input_image.shape)
    output_image = output_image[pad_num:m - pad_num, pad_num:n - pad_num]

    return output_image


start_time = time.time()
img_new_arr = reduce_SAP_11812418("Q3_4.tif", 3)
op_image = Image.fromarray(img_new_arr.astype(np.uint8))
print("--- %s seconds ---" % (time.time() - start_time))
op_image.save("output/img/med_filter/Q3_4_M.tif")
op_image.save("output/img/med_filter/Q3_4_M.png")
```

5.4 Result

Comparing the figure below we could find that the noise has been slightly reduced. However, using too large n_size will blur the image while reducing noise, reducing the resolution of the image. Therefore, in this question, setting n_size to 3 can achieve a better balance between noise reduction and image clarity.



(a) Before median filter. Q3_4.tif



(b) After median filter. Q3_4.tif n_size=3

Figure 16. Input and output image **(Q3_2.tif)**

As shown in the following table, parallel computing can also significantly reduce the running time of the median filter program.

Methodology	Execution time
Without joblib	19.83s
With joblib (6 pools)	1.63s

Table 4. Execution time of median filter. The test was conducted with a PC equipped with AMD Ryzen5 3600, and the python version is 3.8.

6 Conclusion

In this lab different kinds of method for image enhancement had been applied to the given images. Conclusively, we find that different method can enhance different kind of image:

- To enhance the normal image like photograph image, we can apply the histogram equalization, this method generally provides a sensory results.
- To process the image from professional field like astrophotography, the history matching method may be abetter choice when the required histogram is provided.
- For those image care more on detail rather than color like document scanning, we could use local histogram method to enhance it, so we can emphasis its details.
- At any time, using matrix multiplication instead of single data calculation, using parallel calculation instead of serial calculation, using hardware-accelerated functions instead of ordinary functions, can greatly speed up the running time of the program.

7 Supplementary Information

7.1 Code Repository of EE326 Lab

The source code of this lab can be retrieved from https://github.com/sparkcyf/SUSTech_EE326_Digital_image_Processing/.