Lab3 Spatial Transforms and Filtering

Hong-Jia Yang
Southern University of Science and Technology

Abstract—Digital images spatial transforms and filtering is of great significance in digital images analysis. This report summaries main methods to achieve images spatial transofrms and filtering with Python: In historgram processing part, histogram equalization, histogram matching and local histogram processing will be discussed. In spatial filtering part, different kinds of filters will be used to reduce noise. Detailed principle and derivation will be shown in this report to explain these methods. The block diagrams of the algorithm, difference between the results from these methods, the efficiency of the methods and some code details will be discussed together in this report. For each method, considering the efficiency of the algorithm, different improved methods are proposed.

Index Terms—Spatial transforms, image filtering, Python.

1 Introduction

In many cases, people need to further process the digital images. For example, we can see satellite images taken by space telescope more clearly and see more details through the spatial transforms and filtering. By using these technologies to process digital images, we can get ideal picture effects and gain more information.

In spatial processing, we mainly discuss histogram processing and filtering. Histogram contains histogram equalization, histogram matching and local histogram processing. It is based on the statistical significance of histogram and relevant mathematical formula derivation. We change the distribution of the intensity value to get better results. Histogram equalization is a method to adjust contrast with the intensity value better distributed on the histogram; Histogram matching converts the histogram of an image into a specified histogram, which allows the image to have the histogram we want. Local histogram equalization refers to the enhancement of details in a small part of the image.

Spatial filtering use a mask to slide on the image and process the image. It mainly changes the relationship between adjacent pixels to achieve functions such as denoising. Different kinds of filter will produce different results.

For the above methods, through principle derivation, results analysis and improvement, we summarize the advantages and disadvantages of each method, discuss the efficiency of the algorithm, and do some improvements. From the experimental results, the performance of the three histogram methods is consistent with the effect discussed above. Histogram equalization can automatically process images, improve contrast with high efficiency, but it has great limitation in special image. Histogram matching can make the image have the required histogram, but need to create specified histograms for different images. Local histogram equalization enhances the details of small areas in the image with relatively low efficient. Spatial filtering can filter out noise, and median filter is especially effective for salt and pepper noise.

Through this experiment, we know that we need to adopt appropriate method to deal with images having different characteristics in order to get ideal results.

Fig. 1. Surface of Mars photographed by NASA

2 Histogram Processing

2.1 Histogram Equalization

The histogram of a digital image is with intensity levels in the range [0,L-1], which is discrete function $h(r_k) = n_k$, where r_k is the kth intensity value and n_k is the number of pixels in the image with intensity r_K. M and N are the row and column dimensions of the image. Define a normalized histogram:

$$p(r_k) = n_k/MN \quad k = 0, 1, \cdots L-1$$

Histogram is simple to calculate both in software and economic hardware implementations.

My python function to calculate an image's histogram list is as follow:

```
def get_hist(input_image):
 M,N=input_image.shape
 histogram=np.zeros([256], np.float32)
 for i in range(M):
  for j in range(N):
   histogram[int(input_image[i][j])] += 1
 return histogram
```

*Thanks

- *This work was supported in Department of Electrical and Electronic Engineering in Southern University of Science and Technology with Professor Yajun Yu, and the tutorial assistant and all classmates.*

It create a numpy array in one dimension of length 256, we traverse the image for each pixel and add the value by 1 in the array's location correspond to that pixel's gray value.

This function only give a list recording the histogram message of the image, for intuitive viewing, I design a function to plot the histogram:

```
def show_hist(histogram_data,path):
 plt.bar(np.arange(256), ...
     histogram_data,color='dodgerblue',width=0.7)

 plt.grid(axis="y",color='black', ...
     linestyle='-.', linewidth=1)
 plt.xlim((0, 256))
 bwith=2
 ax=plt.gca()
 ax.spines['bottom'].set_linewidth(bwith)
 ax.spines['left'].set_linewidth(bwith)
 ax.spines['top'].set_linewidth(bwith)
 ax.spines['right'].set_linewidth(bwith)

 plt.xlabel('Intensity')
 plt.ylabel('Number of pixels')
 plt.title('Histogram of the Image')
 plt.savefig(path)
 plt.show()
```

From the definition, an image with low contrast has a narrow histogram and histogram in the high-contrast image cover a wide range of the intensity scale with the distribution of pixels is not too far from uniform. We can develop a transformation function to improve the contrast

Firstly, we introduce the histogram equalization, which is the basic for the histogram operation. Note that the histogram matching and local histogram equalization are all based on this.

The design idea of the algorithm is that we firstly discuss the continuous case, and then discretize and approximate it.

The transformations(intensity mappings) is:

$s = T(r), 0 \leq r \leq L-1$

produces an output intensity level s for every pixel in the input image having intensity r.

One issue that needs attention is there are limitations for this transformations to insure the output intensity values will never be less than corresponding input values and the range of output intensities is the same as the input. When we deal with integer intensity values, we should round all results to their nearest integer values and look for the closest integer matches.

Let $p_r(r)$ and $p_s(s)$ denote the PDFs of r and s, the relation between them is:

$$p_s(s) = p_r(r)\left|\frac{\mathrm{d}r}{\mathrm{d}s}\right|$$

An important transformation function of histogram equalization is:

$$s = T(r) = (L-1)\int_0^r p_r(w)\mathrm{d}w$$

w is a dummy variable of integration and the right side of the equation is the CDF of random variable r. From Leibniz's rule:

$$\frac{\mathrm{d}s}{\mathrm{d}r} = \frac{\mathrm{d}T(r)}{\mathrm{d}r} = (L-1)\frac{\mathrm{d}}{\mathrm{d}r}\left[\int_0^r p_r(w)\mathrm{d}w\right] = (L-1)p_r(r)$$

Thus,

$$p_s(s) = p_r(r)\left|\frac{\mathrm{d}r}{\mathrm{d}s}\right| = p_r(r)\left|\frac{1}{(L-1)p_r(r)}\right| = \frac{1}{L-1}$$

This is a uniform probability density function. This means that performing the important intensity transformation function yields a random variable s characterized by a uniform PDF.

The discrete form of the transformation is:

$$s_k = T(r_k) = (L-1)\sum_{j=0}^{k} p_r(r_j) = \frac{(L-1)}{MN}\sum_{j=0}^{k} n_j$$

mapping each pixel in the input image with intensity r_k into a corresponding pixel with level s_k in the output image.

The python code for the histogram equalization is:

```
def hist_equ_11911214(input_image):
 M,N=input_image.shape
 input_histogram=get_hist(input_image)
 input_normalized_histogram=input_histogram/(M*N)
 s_k=np.int32(np.round(255*
          np.cumsum(input_normalized_histogram)))
 output_image=input_image.copy()
 for i in range(M):
  for j in range(N):
   output_image[i][j]=s_k[input_image[i][j]]
 output_histogram=get_hist(output_image)
 return output_image, output_histogram, ...
     input_histogram
```

My flow-chart for this algorithm is:

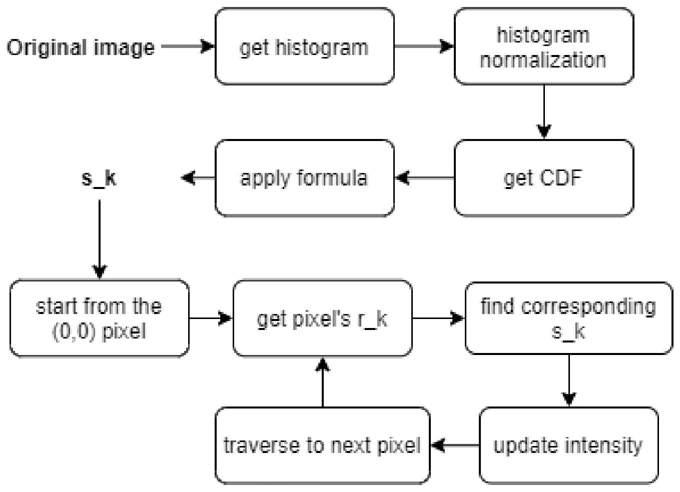


Fig. 2. Flow-chart for histogram equalization

The main process for this algorithm is that we firstly using function 'get_hist' above to generate the image's histogram list. Then we do the normalization and use python function 'cumsum' to directly get CDF. According to the formula, we get s_k and traverse all pixels and update their intensity value.

Obviously, the histogram equalization is fully "automatic." It only bases on the formula above. It is relatively simple, we will discuss this later.

Also, we use the opencv function to achieve the histogram equalization and compare our results together:

```
dst = cv2.equalizeHist(img)
cv2.imwrite('./Q3_1_1_11911214_cv.tif', dst)
```

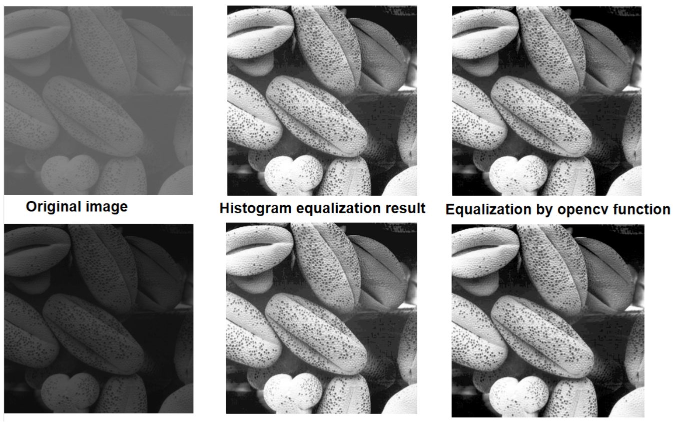

Original image
Histogram equalization result
Equalization by opencv function

Fig. 3. Histogram equalization results comparison

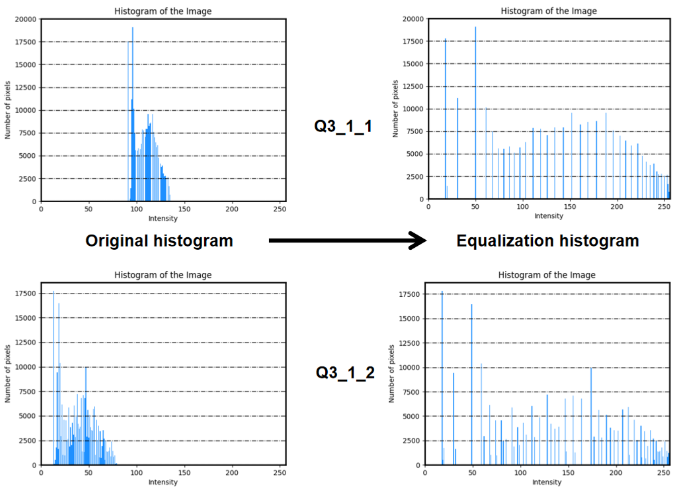


Fig. 4. Histogram equalization effect in histogram

From the result, after the histogram equalization process, the contrast of the output images increases significantly, and we can see more details. From the histograms, we can see that the gray value range is expanded, covering almost the whole gray value range. We also find that the two output images are very similar. This is because after equalization, they have an approximate wide range of gray values.

Also, our results is similar to the opencv function's result, which prove our algorithm is correct.

2.2 Histogram Matching

Sometimes applying the base histogram equalization is not the best approach. In particular, sometimes we want to make the shape of the histogram we wish the processed image to have. Thus, histogram matching is introduced.

Suppose the specified histogram is $p_z(z)$. we should let:

$$s = T(r) = (L-1)\int_0^r p_r(w)\mathrm{d}w$$

$$G(z) = (L-1)\int_0^z p_z(t)\mathrm{d}t = s$$

so

$$z = G^{-1}[T(r)] = G^{-1}(s)$$

In discrete condition,

$$s_k = T(r_k) = (L-1)\sum_{j=0}^{k} p_r(r_j) = \frac{(L-1)}{MN}\sum_{j=0}^{k} n_j$$

$$G(z_q) = (L-1)\sum_{i=0}^{q} p_z(z_i)$$

For a value of q,

$$G(z_q) = s_k$$
$$z_q = G^{-1}(s_k)$$

In practical, due to the gray level is defined as an integer, we do not need to calculate $G^{-1}(s_k)$. We can simply use two data tables to storage T(r) and G(z) separately. For each s_k, we find the most matching value in z_q. Note that in discrete condition, we get the useful results at the cost of some crude approximations.

For the question requirement, we are asked to enhance an image of planetary surface. **We design the specified function firstly according to the textbook's function's shape**.

```python
def create_spec_hist():
 #worse specified function
 spec_hist = np.zeros(256);
 for i in range(0, 5):
  spec_hist[i]=70000*i/5;
 for i in range(5, 20):
  spec_hist[i]=-(13000/3)*i+(275000/3);
 for i in range(20, 180):
  spec_hist[i]=-(125/4)*i+5625
 for i in range(180, 200):
  spec_hist[i]=200*i-36000
 for i in range(200, 256):
  spec_hist[i]=-(500/7)*i+(128000/7)
 spec_hist=spec_hist / np.sum(spec_hist)
 return spec_hist  # get probability density ...
     function
```

All the data for this specified function is estimated according to the shape. This function return the normalization histogram list(for we do not know the specified pixels for the image, we use normalization histogram) for the specified histogram.

The python code for the histogram matching is shown as follow:

```python
def hist_match_11911214(input_image,spec_hist):
 M,N=input_image.shape
 input_histogram=get_hist(input_image)
 input_normalized_histogram=input_histogram/(M*N)
 s_k=np.int32(255*np.cumsum(input_normalized_histogram
 #spec_hist has been normalized
 G_z=np.int32(255*np.cumsum(spec_hist))
 table=np.zeros(256)

 #slow version
 for i in range(256):
  for j in range(256):
   if s_k[i] > G_z[j]:
    table[i] = j
   if s_k[i] == G_z[j]:
    table[i] = j
    break

```

```
output_image=input_image.copy()
for i in range(M):
 for j in range(N):
  output_image[i][j]=table[input_image[i][j]]
output_histogram=get_hist(output_image)
return ...
    output_image,output_histogram,input_histogram
```

The flow-chart for histogram matching is:

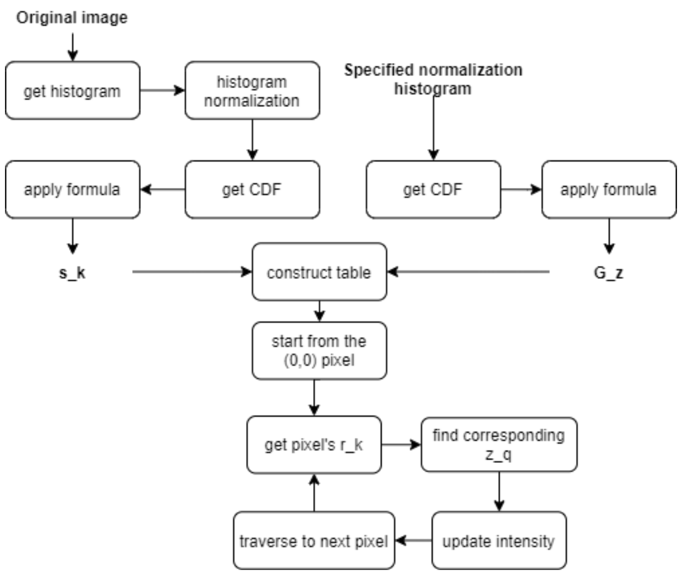


Fig. 5. Flow-chart for histogram matching

The difference from histogram matching with equalization in code is the process of constructing mapping table. All other operations are similar.

In the mapping process(as shown in the code under 'slow version'), for each intensity value, we do the following operation: if in G_z no value equal to $s_k[i]$,take the largest value in G_z that smaller than $s_k[i]$, record in the table j(largest value in G_z corresponding j) to the i(current intensity), else if s_k and G_z correspond exactly, get the corresponding relationship.

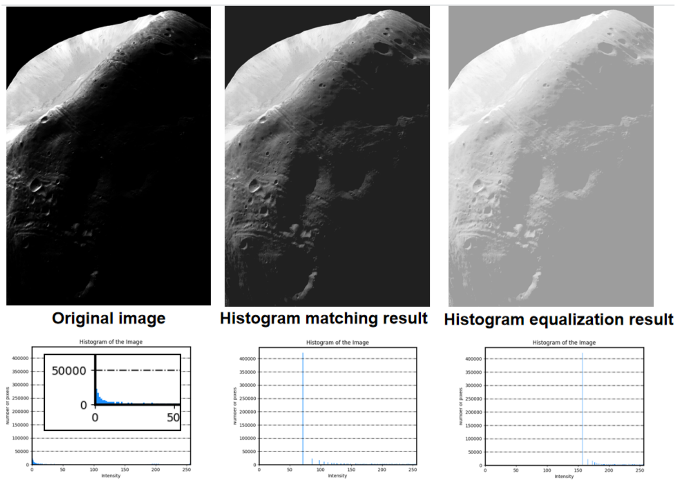


Fig. 6. Histogram matching result

We can see that the contrast of the input image is low. In the histogram, the intensity values are distributed on the left side of the histogram and concentrated. The use of histogram equalization can not well expand its contrast.

After histogram matching according to the specified histogram, the contrast of the output image is greater, and we can see more details. We can also see from the histogram that the intensity distribution has been greatly improved after the algorithm.

Alternative plan: Here we writing another specified histogram to do the histogram matching. For the histogram in textbook may be not the best function. The code for this histogram is shown as follow:

```
    def create_spec_hist():
     #better specified function
     spec_hist = np.zeros(256)
     for i in range(256):
      if i < 40:
       spec_hist[i] =i*0.0005+0.003
      elif i < 75:
       spec_hist[i] =0.041-0.0005*i
      elif i < 170:
       spec_hist[i] =0.003
      elif i < 190:
       spec_hist[i] =0.003+(i-170)*0.00005
      elif i < 210:
       spec_hist[i] =0.004+(i-210)*0.00005
      else:
       spec_hist[i] = 0.003
     spec_hist=spec_hist / np.sum(spec_hist)
     return spec_hist
```

The result of histogram matching for the two specified histograms is shown as follow:

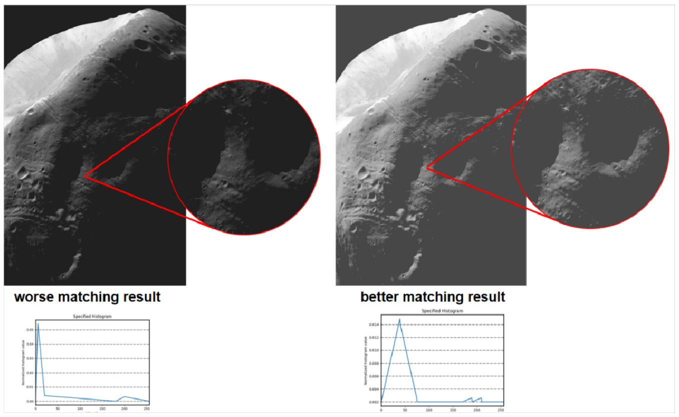


Fig. 7. Histogram matching result with different specified histogram

For the specified histogram designing, we enlarge the range for the relatively dark intensity. **According to our result, the new specified histogram's result shows more details in the image. We can see more potholes in the image.**

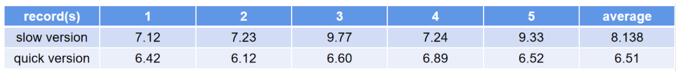
Improvement in algorithm:

To improve the efficiency of the algorithm, we modify the table establish part, which is shown in the code below. We replace original slow version with this quicker one.

```
#quicker vision
k=0
for i in range(256):
 for j in range(k,256):
  if s_k[i] > G_z[j]:
   table[i] = j
  if s_k[i] == G_z[j]:
   table[i] = j
```

```
     k=j
     break
```

The main difference is that we introduce another variable k, to record current j value. So in the first for loop for the i, each time we do not need to access all 256 j value, instead with access the j value from k to the end, which can save time. The performance for our improvement is shown as follow:



record(s)	1	2	3	4	5	average
slow version	7.12	7.23	9.77	7.24	9.33	8.138
quick version	6.42	6.12	6.60	6.89	6.52	6.51

Fig. 8. Comparison of time spent before and after optimization for histogram matching

2.3 Local Histogram Processing

In the above two methods, pixels are modified by a transformation function based on the intensity distribution of an entire image. Sometimes, these global method may enhance the noise and cause bad results, the global transformation does not necessarily guarantee the desired local enhancement and we should design transformation function based on the neighborhood of every pixel.

We define a neighborhood and move its center from pixel to pixel. For each neighbor, we calculate the histogram only in neighborhood and map the intensity of the pixel centered in the neighborhood.

To achieve the local histogram equalization, we should firstly pad the original image. This is because when we use neighbor for boundary pixel, it may not have enough neighborhood pixels. Figure below explains the process for padding:

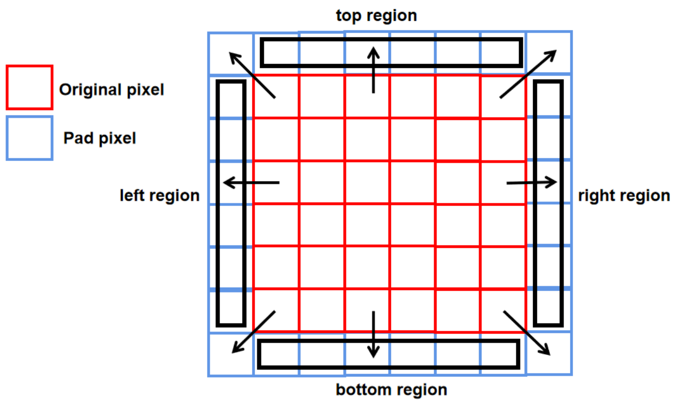


Fig. 9. Pad the original image

For different neighborhood size, we use a formula:

$$padnumber = int(size/2)$$

to determine how many rows and columns should be padded. This formula can ensure the local neighborhood process for odd and even number.

The python code for padding and the local histogram equalization is shown as follow:

```
def local_hist_equ_11911214(input_image,m_size):
 M,N=input_image.shape
 pad=int(m_size/2)
 pad_img=np.zeros((M+2*pad,N+2*pad),np.uint8)
 for i in range(pad):
  #top region
  pad_img[i][pad:(N+pad)]=input_image[0]
  #left region
  pad_img[pad:(M+pad),i]=input_image[:,0]
 for i in range(pad):
  #bottom region
  pad_img[i+pad+M][pad:(N+pad)]=input_image[M-1]
  #right region
  pad_img[pad:(M+pad),i+pad+N]=input_image[:,N-1]
 pad_img[0:pad,0:pad]=input_image[0][0]
 pad_img[0:pad,N:N+pad]=input_image[0][N-1]
 pad_img[M:M+pad,0:pad]=input_image[M-1][0]
 pad_img[M:M+pad,N:N+pad]=input_image[M-1][N-1]
 #center region
 for i in range(M):
  for j in range(N):
   pad_img[i+pad][j+pad]=input_image[i][j]
 output_image=np.zeros((M,N),np.uint8)

 #slow vision
 for i in range(pad,pad+M):
  for j in range(pad,pad+N):
   local_hist=np.zeros(256)
   for x in range(m_size):
    for y in range(m_size):
     local_hist[pad_img[i+x-pad][j+y-pad]]+=1
     tr=np.cumsum(local_hist)
     pix_value=int(round(255*tr[pad_img[i][j]]
                                  /(m_size**2)))

 input_histogram=get_hist(input_image)
 output_histogram=get_hist(output_image)
 return ...
      output_image,output_histogram,input_histogram
```

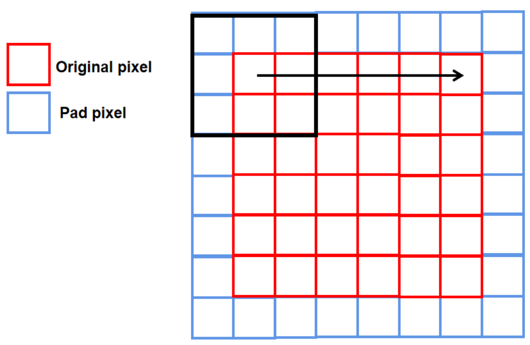


Fig. 10. local histogram equalization

The flow-chart for the algorithm is:

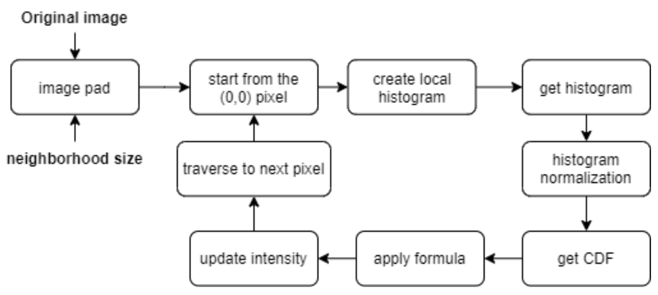


Fig. 11. Flow-chart for local histogram equalization

For each neighbor, we create a local histogram according to the given size. We do the histogram equalization in each neighborhood and update the intensity for this neighbor's 'center' pixel. Note that for even neighborhood size, 'center' pixel in not in the center.

The result for the local histogram equalization is:

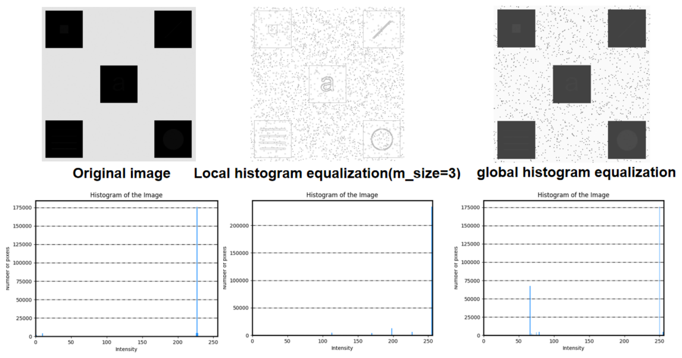


Fig. 12. Result for local histogram equalization

The original image has noise. When global histogram equalization is used, the noise is significantly increased, but it does not show new important details. And the histogram for the global equalization shown no improvement, the intensity distribution is still not wide. However, when we use the local histogram equalization, we can see the objects contained in the black box, and we get the desired information through processing. And the intensity in histogram is improved.

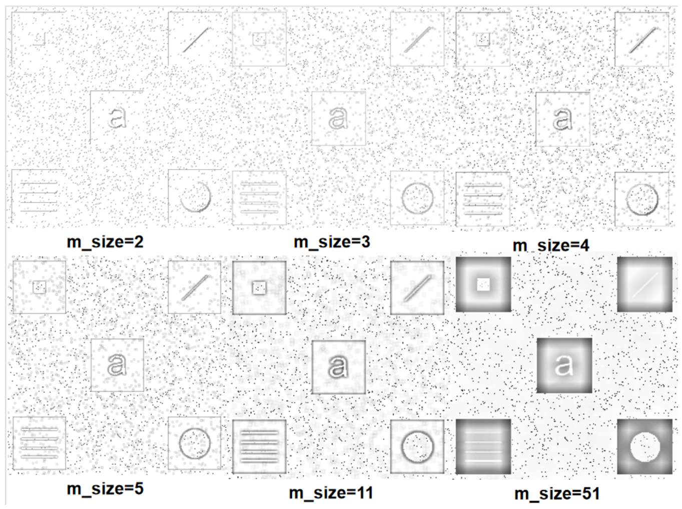


Fig. 13. Results for local histogram equalization with different sizes

Also, we compare local histogram equalization results with different neighborhood size. From the result, we can find that with the size of neighborhood increasing, the details in the five boxes become blurred, which verify our purpose for the local histogram equalization, that is, large neighborhood will cause detail defect.

Improvement in algorithm: One improvement for the local histogram equalization is in the process of moving neighborhood horizontally, as shown in figure below:

When we traverse into next neighborhood, we only need to delete the three left pixels in the last neighborhood and add three new right pixels in current neighborhood, which can reduce the time for construct neighborhood local histogram, especially for relatively large size neighborhood.

The code for this quicker way is shown as follow:

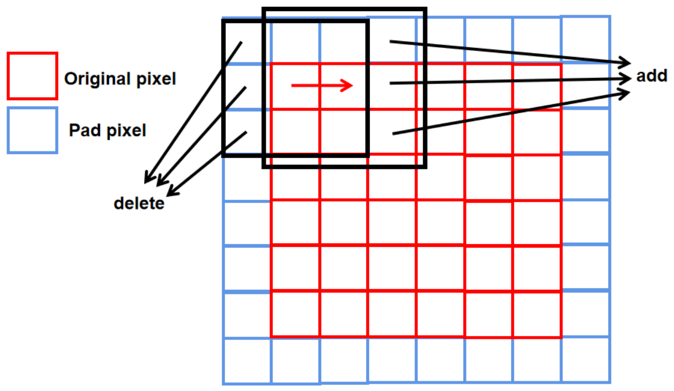
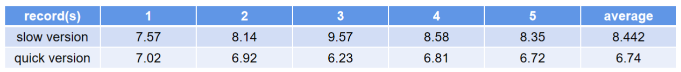


Fig. 14. Improvement in local histogram equalization

```
#quicker vision
local_hist = np.zeros(256)
for i in range(pad,pad+M):
 for j in range(pad,pad+N):
  if j<(pad+N-1)andj>pad:
   for x in range(m_size):
    local_hist[pad_img[i+x-pad][j+(0)-pad-1]]-=1
    local_hist[pad_img[i+x-pad][j+(m_size-1)-pad]]+=1
  else:
   local_hist = np.zeros(256)
   for x in range(m_size):
    for y in range(m_size):
     local_hist[pad_img[i+x-pad][j+y-pad]]+=1
  tr=np.cumsum(local_hist)
  pix_value=int(round(255*tr[pad_img[i][j]]
                                   /(m_size**2)))
  output_image[i-pad,j-pad]=pix_value
```

It should be noted that this method can only be used for horizontal pixel blocks that are not boundaries. When writing code, need to pay special attention to whether the coordinates are correct.

And the testing results shows that our quicker way shorten the time for local histogram equalization.



record(s)	1	2	3	4	5	average
slow version	7.57	8.14	9.57	8.58	8.35	8.442
quick version	7.02	6.92	6.23	6.81	6.72	6.74

Fig. 15. Comparison of time spent before and after optimization for local histogram equalization

We will discuss the change for complexity of our improvement. In the original slow version, we should do 9 operations for the neighborhood size of 3, while in this quicker vision, we only need to do 6 operations, 3 delete and 3 adding. **When the neighborhood size increases, our algorithm's advantages will be more obvious.** For neighborhood size of n, we will save the operations from n^2 to $2*n$, that is, almost takes $2/n$ of the original operations. This is also reflected in time.

3 Spatial Filtering

Spatial filtering is for image enhancement. A spatial filter consists of a neighborhood and a predefined operation that is performed on the image pixels encompassed by the neighborhood. The result of filtering operation is given to the center of the neighborhood.

In general, linear spatial filtering of an image of size $M * N$ with a filter of size $m * n(m = 2a + 1, n = 2b + 1)$ is given by the expression:

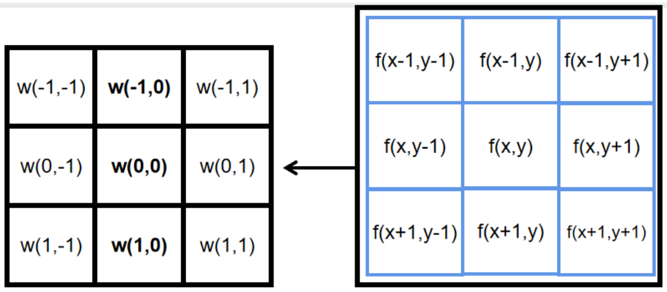


Fig. 16. Spatial filtering

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

Smoothing linear filters(aberage filters) outputs the average of the pixels contained in the neighborhood of the filter mask. This process results in an image with reduced "sharp" transitions in intensities.

Order-statistic(nonlinear) filters bases on the ordering of the pixels' intensity in the neighborhood. They provide excellent noise-reduction capabilities. The averaging filter may blurry the image and its noise reduction performance is poor.

Firstly, we write python function for finding neighborhood average and median value:

```
def mean(mask):
  M,N=mask.shape;
  sum=0
  for i in range(M):
    for j in range(N):
      sum+=mask[i][j]
      sum=sum/(M*N)
  return sum
```

For median method, we use two sorting method: bubble sorting and selected sorting. This will be discussed later.

```
def median(mask):
  list=mask.flatten()
  #Selected Sorting
  l = len(list)
  for i in range(l):
    min = i
    for j in range(i, l):
      if list[j] < list[min]:
        min = j
      list[i], list[min] = list[min], list[i]
  return list[int(l/2)]
```

```
def median(mask):
  list=mask.flatten()
  #Bubble Sorting
  l=len(list)
  for i in range(l-1,0,-1):
      for j in range(i):
          if list[j]>list[j+1]:
              list[j],list[j+1]=list[j+1],list[j]
  return list[int(l/2)]
```

The code for the spatial filtering of median is shown as follow:

```
def reduce_SAP_11911214(input_image, n_size):
 #pad
 M,N=input_image.shape
 pad=int(n_size/2)
 pad_img=np.zeros((M+2*pad,N+2*pad),np.uint8)
 for i in range(pad):
  #top region
  pad_img[i][pad:(N+pad)]=input_image[0]
  #left region
  pad_img[pad:(M+pad),i]=input_image[:,0]
 for i in range(pad):
  #bottom region
  pad_img[i+pad+M][pad:(N+pad)]=input_image[M-1]
  #right region
  pad_img[pad:(M+pad),i+pad+N]=input_image[:,N-1]
 pad_img[0:pad,0:pad]=input_image[0][0]
 pad_img[0:pad,N:N+pad]=input_image[0][N-1]
 pad_img[M:M+pad,0:pad]=input_image[M-1][0]
 pad_img[M:M+pad,N:N+pad]=input_image[M-1][N-1]
 #center region
 for i in range(M):
  for j in range(N):
   pad_img[i+pad][j+pad]=input_image[i][j]
 output_image=np.zeros((M,N),np.uint8)
 #filtering
 for i in range(M):
  for j in range(N):
   output_image[i][j]=median(pad_img[i:i+pad*2+1,
                j:j+pad*2+1])#simplified ...
                    formula
 return output_image
```

The flow-chart for median filtering is:

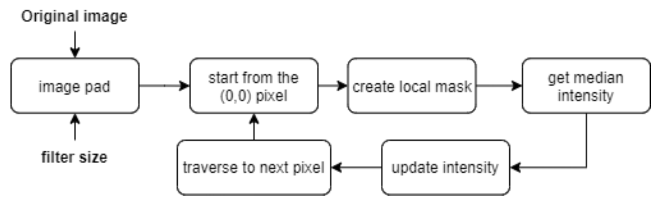


Fig. 17. Flow chart for median filtering

Also, similar to the local histogram equalization, we firstly pad the image according to the given size. Then we traverse each pixel and construct the neighborhood, finding median intensity and replace the intensity of the center pixel's intensity with this median. The result is shown in Fig 18 and 19.

The average filter blurs the image, although the intensity of the noise is reduced. Applying the median filtering, we can see that the noise of the output image is greatly eliminated and the original details of the image are well preserved. With the increase of the filter size, the noise intensity of the output image decreases gradually, but the image becomes blurred, and some details are removed as noise. Also, with the filter size increases, more time should be used for the median filter.

Now we will discuss the sorting method in the median filter. **According to the data structure and algorithm analysis, for a small amount of data, selected sorting and bubble sorting are relatively effective algorithm.** So we test the performance of these two sorting algorithm for median filter and the results is shown in FIg 20.

The average algorithm complexity for these two sorting are all $O(n^2)$. We find that the performance of the two sorting method is relatively similar.

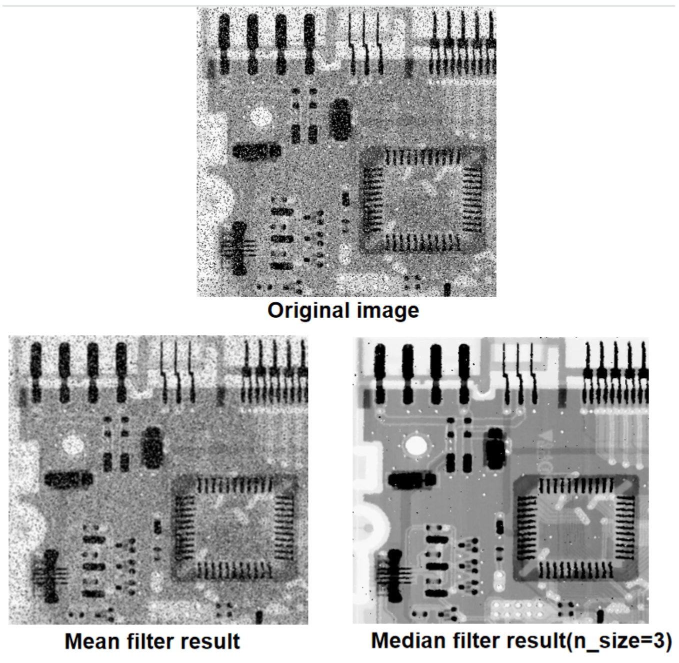


Fig. 18. Result from average filter and median filter

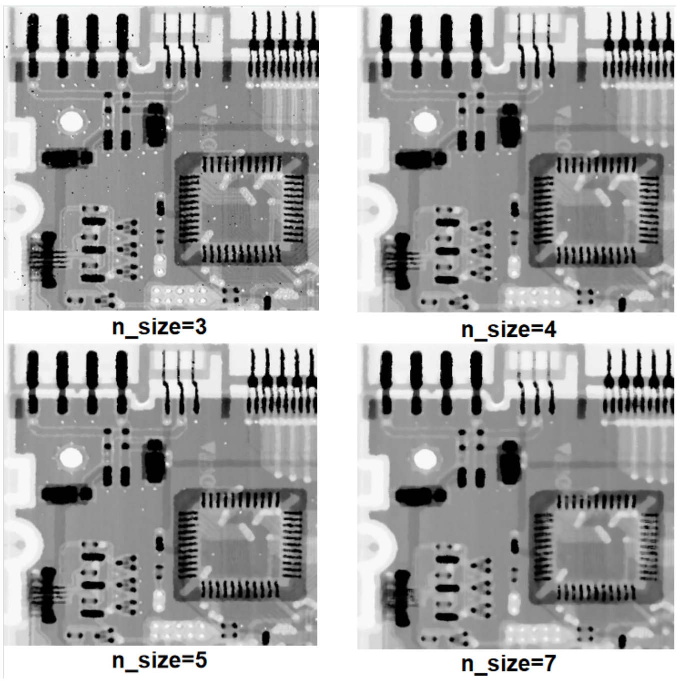


Fig. 19. Result of median filter with different filter size

4 Conclusion

In this lab, we learned how to use python achieve the image histogram equalization, histogram matching, local histogram equalization and spatial filter for median filter. We analyze the properties of each method and have a deeper understanding based on their principle, derivation process, flow-charts, codes and result.

For histogram equalization, we mainly discuss the principle of the algorithm according to the statistical formula. For histogram matching, we design different specified function to test the performance. For local histogram equalization, we introduce the process

record(s)	1	2	3	4	5	average
Bubble Sort	6.06	7.44	6.64	6.99	6.69	6.76
Selected Sort	5.49	6.05	6.72	5.29	6.17	5.944

Fig. 20. Comparison of time spent for different sorting algorithm

for image pad and improve the traverse algorithm.

Finally, we compare the effect of different methods from their results and algorithm. We do a lot improvement in algorithm and use detailed images and time spend data to explain our conclusion.

Histogram equalization is very useful and effective for most images. However, for some special images, such as those with concentrated intensity , the result is not good. For histogram matching, the output results are generally satisfactory according to our specified histogram. However, this method is inconvenient to use. For local histogram equalization, it can show more local details than global histogram equalization. But its efficiency is relatively low. Median filter has good adaptability to eliminate salt noise and pepper noise.

For different images, we need to adopt different strategies to continuously optimize the image processing effect, and comprehensively consider the cost of time and computing ability brought by image processing.

When writing the code, I encountered many problems, such as the details of the algorithm, the boundary of pixel coordinates and so on. Through continuous attempts and verification, I finally solved these problems and improved my programming ability. Now I have a deeper understanding towards image histogram operation and spatial filter. This is a course that requires careful thinking.