A Brief Introduction to Histogram Transforming and Median Filtering

Wang Zhuoyang[1]

[1] *12112907, Department of Electrical and Electronic Engineering, SUSTech. Email: glverfer@outlook.com*

Abstract

This is my third lab report for the Digital Image Processing course. The report covers an overview of several commonly used image processing methods related to histograms and one filtering method, specifically Median Filtering. The report includes an explanation of the algorithm principles, along with the pseudo-code and Python code for each. Additionally, there will be an analysis of the results with accompanying remarks and comparisons, as needed.

Keywords: DIP; Histogram Processing; Filtering

Contents

Digital Image Processing Laboratory Reports (2023)

Corresponding author
Wang Zhuoyang

1 Introduction

Background

This experiment involves the analysis and verification of four algorithms. The first three manipulate the histogram of an image to adjust its global or local dynamic range, in order to make the information in the image more clearly visible. The fourth algorithm is an image filtering technique, aimed at removing salt-and-pepper noise from the image.

Methodology and Result

We will sequentially analyze the principles and algorithm processes of histogram equalization, histogram matching, local histogram equalization, and median filtering. We will implement these algorithms using Python and verify their correctness, demonstrate the necessity of using different algorithms, and compare the efficiency differences between different implementations of the same algorithm.

The experimental results are divided into several aspects. Firstly, we analyze the influence of input parameters (kernel size) on the results for the last two algorithms. Secondly, we compare the efficiency of different implementation methods. Additionally, we compare the applicability of various algorithms for special cases.

Disclaimer

Though part of this article has been edited for language by ChatGPT, **these are in no way related to** the specific content of the text, such as the algorithm analysis, implementation, and optimization discussed in the article. This statement is hereby declared.

2 Histogram Equalization

2.1 Algorithm Principle and Pseudo-code

Analyze the 2-D Grayscale Image from a Probabilistic Perspective

A two-dimensional grayscale image can be viewed as a matrix of uint8 values, where each element represents a pixel. We can analyze the image from a probabilistic and statistical perspective by counting the number of pixels with each of the 256 gray values and plotting the histogram to clearly represent the grayscale distribution of the image. Below, we provide an example image in Figure 1.

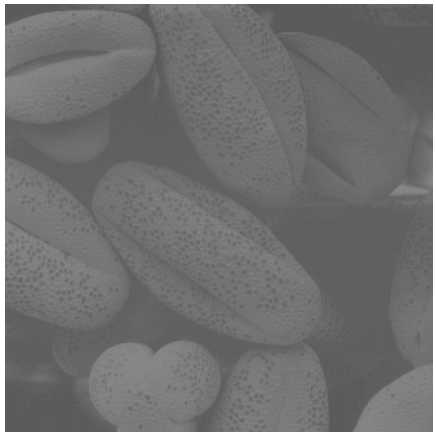


Figure 1. Image 1-1, 500x500, grayscale.

This image has an obvious grayish color (limited dynamic range), which is due to the majority of its pixels being in the middle gray range, with few high or low grayscale pixels. Plotting the histogram of its grayscale probability density clearly shows this point (Figure 2).

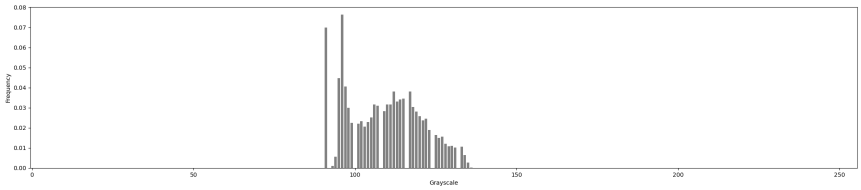


Figure 2. The Histogram of frequency of the pixel gray values in Image 1-1.

The above histogram can be roughly viewed as the probability density function (PDF) of the pixel grayscale distribution. However, strictly speaking, since gray values are discrete, the histogram obtained from the specific image is actually a result of sampling the PDF. Nevertheless, for the generality of algorithms and to account for the fact that grayscale transformations cannot distinguish pixels with the same original color, we simply treat it as the PDF.

Transformations on the Grayscale Domain

For a grayscale image, we can define a transformation T that maps the original grayscale range to a new grayscale range:

$$s = T(r) \tag{1}$$

By applying function T to each grayscale value in the image, we can obtain a new image. For the original image with a too compact grayscale distribution as shown in the previous example, we hope to make its grayscale distribution more uniform to achieve the goal of stretching the

dynamic range and enhancing the image. Histogram equalization algorithm aims to find such a transformation for the input image that makes the histogram distribution more uniform.

Obtain a Uniform Distribution from a Given Distribution

We define the gray values of the original image as a random variable R, with a cumulative distribution function (CDF) of $F_R(r)$, which can be obtained by statistical analysis of the original image.

We define S as a random variable that describes the gray values of the new image:

$$S = F_R(R) \tag{2}$$

and compute its cumulative distribution function:

$$F_S(s) = P\{S \leq s\} = P\{F_R(R) \leq s\} = P\{R \leq F_R^{-1}(s)\} = F_R(F_R^{-1}(s)) = s \tag{3}$$

from which we can find out that S satisfied a uniform distribution:

$$S \sim U(0, 1) \tag{4}$$

which is what we want eventually. So if we use the known $F_R(r)$ as the transform function, we can obtain a zero-to-one uniform distribution from any other given distributions. Finally, we can obtain the required transformation function for the histogram equalization algorithm by multiplying it by 255 and then rounding it to an integer, where $F_R(r)$ is in fact defined as following:

$$F_R(r) = \sum_{k=0}^{r} f_R(k) \tag{5}$$

where $f_R(k)$ is the frequency of the grayscale k, $0 \leq k \leq 255$, $k \in \mathbb{Z}$.

Pseudo-code

Below is the pseudo code that describes the original algorithm (but does not necessarily represent the final implementation).

Algorithm 1: Histogram Equalization

Input: img
Output: img_new

$img\_pdf \leftarrow$ Calculate the PDF of img
$img\_cdf \leftarrow$ Calculate the cumulative summation of img_pdf
foreach $(x, y) \in img$ **do**
    $img\_new[x, y] \leftarrow 255 \times img\_cdf[img[x, y]]$
end

2.2 Python Implementation

Brute-force Implementation

Firstly, we can follow the algorithmic process completely.

In the first step of calculating the PDF, we need to iterate through each pixel and count the grayscale distribution. In the second step of calculating the CDF, we need to loop 256 times to compute the prefix sum. In the third step of applying the transformation, we iterate through each pixel and apply the aforementioned transformation function to it. For example, when we calculate the PDF (Code 1):

```python
def getPDF(img):
    res = np.zeros(256)
    for i in range(256):
        res[i] = np.sum(img == i)
    return res / (img.shape[0] * img.shape[1])
```

Listing 1. PDF Calculation (Brute-force)

Other codes is similar and of no use, there is no need to provide them.

Using Characteristics of NumPy

In fact, we can use NumPy to speed up the calculation and reduce the amount of code needed (Code 2).

```python
def getPDF(img):
    res, bins = np.histogram(img, bins=256, range=(-0.5, 255.5))
    return res / (img.shape[0] * img.shape[1])

def hist_equ(img):
    img = img.astype(np.int32)

    img_pdf = getPDF(img)  # PDF (regardless of size)
    img_cdf = np.cumsum(img_pdf)  # Cumulative Sum

    res = 255 * img_cdf[img]

    res = res.astype(np.uint8)
    return (res, getPDF(res), img_pdf)
```

Listing 2. Histogram Equalization

We can use $np.histogram$ to calculate the PDF, and use $np.cumsum$ to calculate the cummlative summation.

Complexity Analysis

The time complexity of the algorithm is always O(nm)O(nm), and the space complexity is also O(nm)O(nm). However, the use of NumPy results in a significantly smaller constant factor when compared to running the algorithm in native Python. However, in all honesty, we cannot provide meaningful results as we cannot accurately estimate the average complexity of the matrix operations that occur behind NumPy.

As images must be stored, which is the costliest factor, the space complexity is always $O(nm)$. However, it appears that this does not hold any meaningful or essential analytical significance. Of course, if we exclude the necessity of input and output, the algorithm's space complexity is $O(k)$, where k represents the number of discrete gray values.

However, given that we are employing Python, the time complexity is still open for discussion, making the significance of discussing the space complexity of such a straightforward problem relatively negligible.

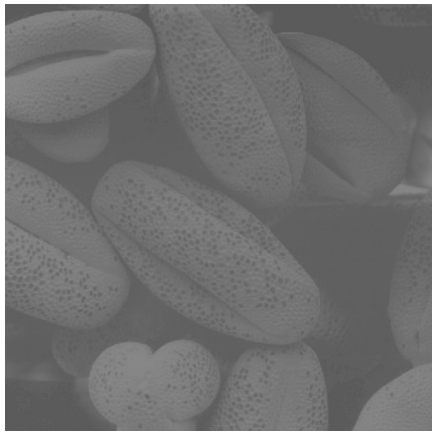
2.3 Results

Performance

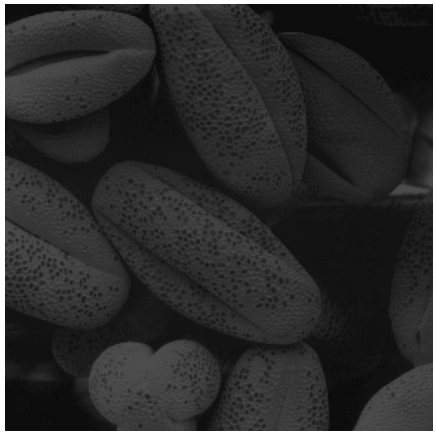
We apply histogram equalization on the two example images (Figure 3).

The result of the two images are as following (Figure 4).

To better observe the performance of the program, we plotted a histogram comparison between the original image and the resulting image (Figure 5, 6). The gray histogram represents the original image, while the orange histogram represents the equalized image.

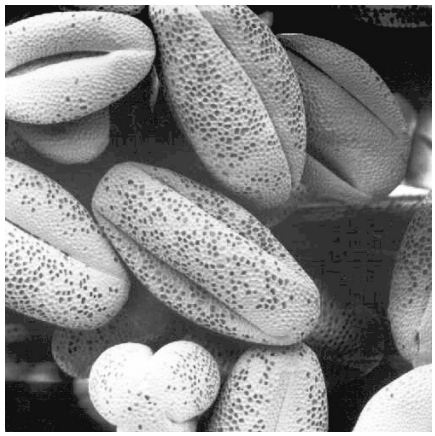


(a) Image 1-1.

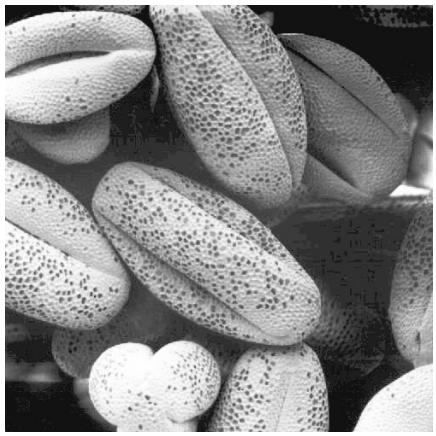


(b) Image 1-2.

Figure 3. The raw images.



(a) The equalized image from Image 1-1.



(b) The equalized image from Image 1-2.

Figure 4. The output images of histogram equalization.

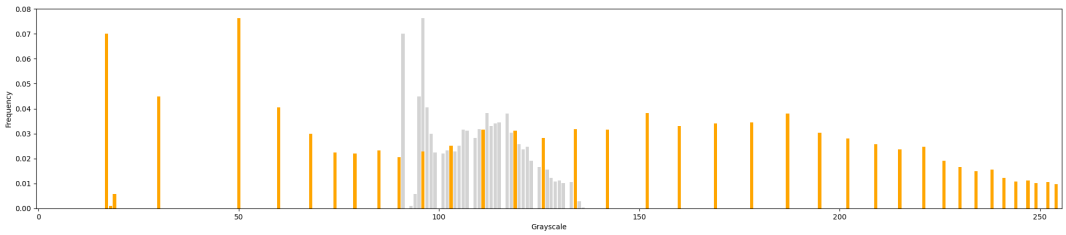


Figure 5. The histogram comparison between the raw image 1-1 and the output image.

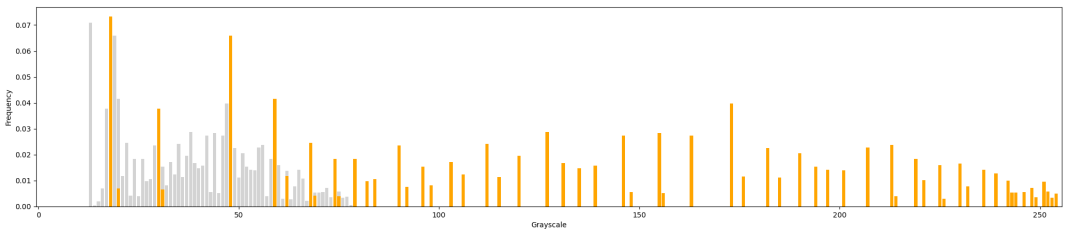


Figure 6. The histogram comparison between the raw image 1-2 and the output image.

The histogram reveals that the original image has a limited grayscale distribution, causing it to appear either too gray or too dark. However, after equalization, the histogram is more broadly distributed over the entire value range, resulting in increased contrast and overall improvement in image quality.

Time Cost

We run the code for 100100 times and calculate its average time cost. The result is about 0.00980.0098 seconds for the first image and about 0.01020.0102 seconds for the second image (both 500x500 px).

Meanwhile, we attempted to solve the PDF and prefix sum using a loop. However, the average runtime clocked in at approximately 0.2534 seconds, revealing the sluggish performance of the glue language.

3 Histogram Matching

3.1 Algorithm Principle and Pseudo-code

Disadvantages of Histogram Equalization

The histogram equalization algorithm is not universally applicable because it cannot achieve true balance in the strictest sense.

There are two perspectives to explain the reasons for this:

Firstly, from a probability statistics perspective, as the gray values are discrete, the specific gray values of an image are essentially samples obtained by sampling the probability distribution. Therefore, the obtained results are transformations of the samples and cannot obtain the ideal completely linear gray distribution. From this perspective, if the gray values were continuous or tend to be continuous, we would obtain more ideal results.

Secondly, we can understand this from the perspective of transformations. As the gray transformation operates on pixels with the same gray value together, there will inevitably be fluctuations in gray levels, making it impossible to achieve perfect balance. Based on this, if the gray values were continuous, this problem could be avoided, as the range of each gray value would change from a discrete 1 to an infinitesimal value.

Due to the existence of the above problems, uniform equalization alone may not yield the desired results and may even lead to more extreme cases. For example, if pure black pixels occupy most of the image, the resulting transformation would attempt to shift them towards higher gray values. However, as the black region does not originally contain any useful information, this would be meaningless and could lead to an overly bright image.

To solve this problem, we should use the histogram matching algorithm. Instead of blindly equalizing the histogram, we set a target histogram based on our desired result. In the aforementioned example, we can set a higher peak in the black region, allowing the majority of black pixels to remain in their original position while stretching the brighter gray level where the useful information lies.

Obtain an Specified Distribution from a Uniform Distribution

We can first perform histogram equalization on the original image, and then obtain the desired distribution from a uniform distribution. The process of obtaining the target distribution from a uniform distribution is similar to the one mentioned earlier. We denote:

$$S \sim U(0,1) \Rightarrow F_S(s) = s \tag{6}$$

The target r.v and its distribution:

$$Z = F_Z(z) \tag{7}$$

Then we apply the inverse function of $F_Z(z)$ on S and see what will happen:

$$P\{Z \leq z\} = P\{F_z^{-1}(S) \leq z\} = P\{S \leq F_z(z)\} = F_S(F_Z(z)) = F_Z(z) \tag{8}$$

which is what we want.

So if we have the raw image with a grayscale distribution of $F_R(r)$, we can obtain $Z = T(r) = F_Z^{-1}(F_R(r))$ which satisfied the distribution $F_Z(z)$ set by us.

Pseudo-code

Below is the pseudo code that describes the original algorithm (but does not necessarily represent the final implementation).

Algorithm 2: Histogram Matching

Input: img, pdf_target
Output: img_new

$img\_pdf \leftarrow$ Calculate the PDF of img
$img\_cdf \leftarrow$ Calculate the cumulative summation of img_pdf
$cdf\_target \leftarrow$ Calculate the cumulative summation of pdf_target
$cdf\_target\_inv \leftarrow$ Calculate the inverse of cdf_target
$trans\_func \leftarrow cdf\_target\_inv(img\_cdf)$
foreach $(x, y) \in img$ **do**
  $img\_new[x, y] \leftarrow trans\_func[img[x, y]]$
end

3.2 Python Implementation

Calculate PDF and CDF

This part is the same as the histogram equalization algorithm. The code is given below (Code 3):

```
def getPDF(img):
    res, bins = np.histogram(img, bins=256, range=(-0.5, 255.5))
    return res / (img.shape[0] * img.shape[1])

def hist_match(img):
    img = img.astype(np.int32)

    img_pdf = getPDF(img)
    img_cdf = np.cumsum(img_pdf)
    spec_cdf = np.cumsum(spec_hist)
    ...
```

Listing 3. Histogram Matching (Part I)

Apply the Inverse Function

First of all, we know from the basic properties of the CDF that it is monotonically increasing, and its inverse function can be mathematically derived. However, the grayscale values we use are discretely stored in an array, so a special method is needed when taking the inverse function.

In fact, for a given, discrete grayscale value y, finding $x = f^(-1)(y)$ can be seen as finding the smallest index value x where $f(x)$ is not less than y. We can use $np.searchsorted$ to get the smallest index value idx where $f[idx]$ is larger than the argument. To use it, we also need to consider the

case when the values are equal. In this case, we can add a very small value to the original function (less than $\frac{1}{255}$). Of course, we can also multiply them all by 255 and add a value less than 1. This might look better (Code 4).

```
    ...
    tran = np.searchsorted(255 * spec_cdf, 255 * img_cdf + 0.5, side='left')
     - 1
    res = tran[img]

    res = res.astype(np.uint8)
    return (res, getPDF(res), img_pdf)
```

Listing 4. Histogram Matching (Part II)

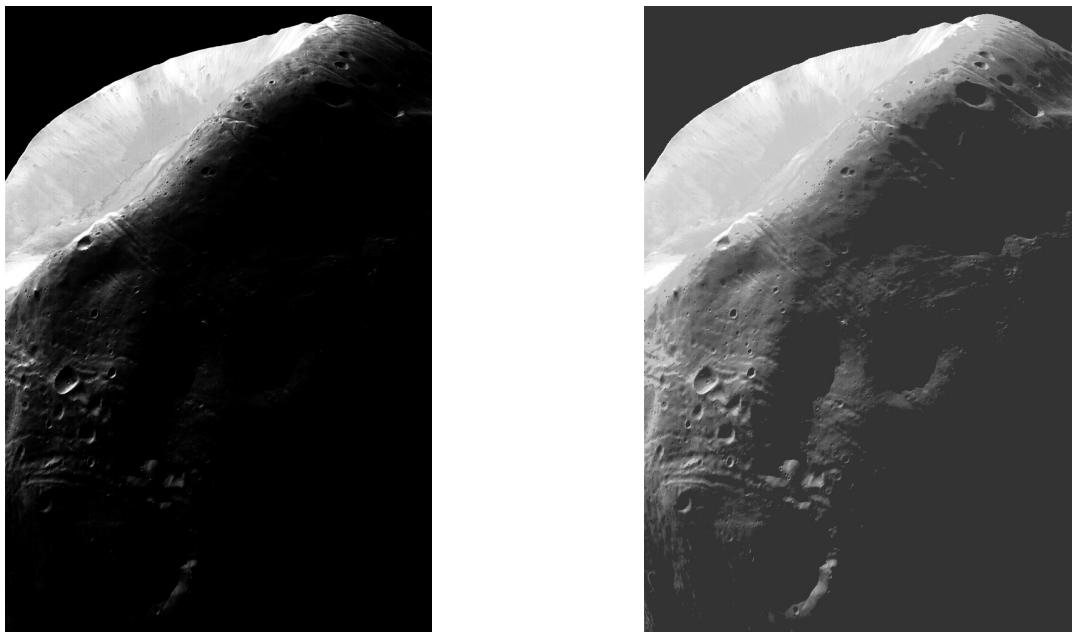
Complexity Analysis

Obviously, the time complexity is still $O(nm)$ as before, and without considering image transformation, it is $O(k)$; the space complexity is also the same, and there is no need for further analysis.

As there is an extra transformation compared to histogram equalization, both algorithms have a slightly larger constant than histogram equalization.

3.3 Results

The raw image and the result are shown in Figure 7.



(a) The raw image, Image 2.

(b) The matched image from Image 2.

Figure 7. The output images of histogram matching.

If we use the histogram equalization algorithm to process this image, we will get a different result, which is shown in Figure 8. The result will be balanced by brightening up a large number of dark pixels, leading to an overall too bright image.

The solution is to use histogram matching, which specifies a target histogram that we want the result to resemble as closely as possible. In this example, the histogram function used is shown in Equation 9, Figure 9 and Code 5.

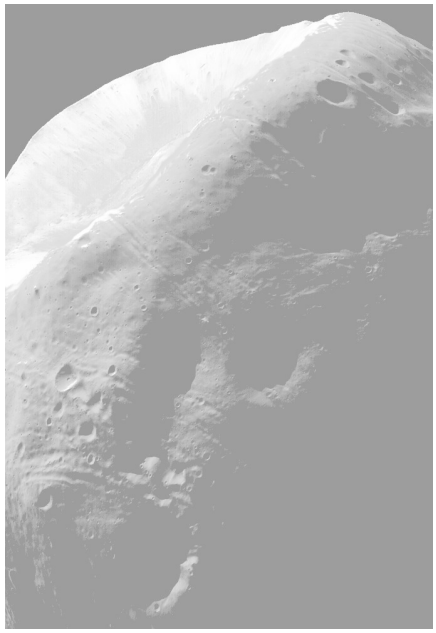


Figure 8. The output image when applying histogram equalization on the Image 2.

$$15.38 \times spec\_hist[k] = \begin{cases} \frac{7}{5}k, & 0 \leq k < 5 \\ -\frac{21}{50}(k-5)+7, & 5 \leq k < 20 \\ -\frac{1}{230}(k-20)+0.7, & 20 \leq k < 181 \\ \frac{3}{110}(k-181), & 181 \leq k < 203 \\ -\frac{3}{260}(k-203)+0.6, & 203 \leq k \leq 255 \end{cases} \tag{9}$$

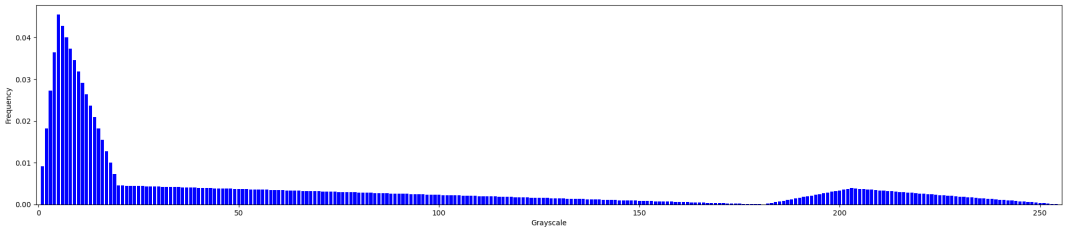


Figure 9. The specified histogram.

```
spec_hist = np.concatenate((
    np.linspace(0, 7, 5 - 0 + 1),
    np.linspace(7, 0.7, 20 - 5 + 1)[1:],
    np.linspace(0.7, 0, 181 - 20 + 1)[1:],
    np.linspace(0, 0.6, 203 - 181 + 1)[1:],
    np.linspace(0.6, 0, 255 - 203 + 1)[1:]
), axis=0)
spec_hist /= np.sum(spec_hist)
```

Listing 5. Histogram Matching (Specified Histogram)

And finally here is the comparison between the histograms of the raw image and the output image (Figure 10).

The average time cost of one-hundred-time execution is about 0.0240 seconds.

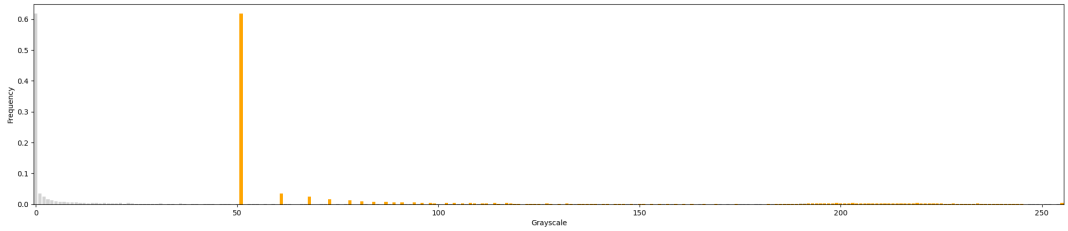


Figure 10. The histogram comparison between the raw image and the output image.

4 Local Histogram Equalization

4.1 Algorithm Principle and Pseudo-code

Globally or Locally

Local histogram equalization, strictly speaking, has a different range of applications from the previous algorithm. This algorithm is mainly used to find some details of the image locally. When the overall contrast of the image is high, but the contrast near the effective information is insufficient, the overall histogram equalization algorithm is obviously not very effective. At this time, it is necessary to use local histogram equalization, which only performs histogram equalization on a small area around each pixel, without being affected by the high overall contrast of the image.

The idea of this algorithm is very simple. First, we define a kernel radius r, which represents the size of the area around each pixel that we want to perform histogram equalization on. We know that the center of this kernel should be the pixel we want to assign a value to, so the size of the kernel should be an odd number. For convenience, we define the radius r, which represents the side length of the entire area as $2 \times r - 1$.

And then we can apply histogram equalization on the kernel as before, and only update the value of the center pixel, which is important.

Pseudo-code

Here is the pseudo-code (Algorithm ?????????????3).

Algorithm 3: Local Histogram Equalization

Input: $img, radius$
Output: img_new

foreach $(x, y) \in img$ **do**
  foreach $(i, j) \in the\ kernel\ centered\ at\ (x, y)$ **do**
    Do the histogram equalization on the kernel area
    $img\_new[x, y] \leftarrow kernel\_new[x, y]$
  end
end

Of course, the pseudocode is just for illustrating the algorithm process, in which we perform many meaningless calculations. For example, when we perform histogram equalization on the kernel, we don't actually need to consider the equalization results of other pixels except for the center pixel, because we only need to update the value of the center pixel, and this result does not depend on the equalization results of other pixels.

In addition, we can also use some NumPy tricks to eliminate loops in the program, greatly speeding up the computation.

4.2 Python Implementation

Optimized Approach

I don't want to implement the algorithm using loops, so here is the optimized approach (Code ???6).

```
def local_hist_equal(img, radius):  # radius should be an integer and larger than 1
    size = radius * 2 - 1
    img = np.pad(img.astype(np.int32), ((radius - 1, radius - 1), (radius - 1, radius - 1)), 'reflect')

    pts = [(idx // size - radius + 1, idx % size - radius + 1) for idx in range(size * size)]
    mvd = np.array([np.roll(np.roll(img, x, axis=0), y, axis=1) for (x, y) in pts])
    res = np.sum(mvd <= img, axis=0) / (size * size) * 255

    res = res[(radius - 1):(1 - radius), (radius - 1):(1 - radius)].astype(np.uint8)
    return (res, getPDF(res), getPDF(img))
```

Listing 6. Local Histogram Equalization

Explanation will be given line by line below.

The third line expands the image boundary by $r-1$ pixels outward because the algorithm uses the pixel neighborhood.

The fifth line generates all relative coordinates of pixels within the kernel using a generator. For example, with a radius of 2, the generator will produce:

$$[(-1,-1),(-1,0),(-1,1),(0,-1),(0,0),(0,1),(1,-1),(1,0),(1,1)]$$

The method is to iterate from 0 to $(radius \times 2-1)^2$, then perform integer division and modulo operations on the kernel size, and finally add the offset.

In the sixth line, taking advantage of the convenience of matrix operations, each offset coordinate in pts is applied to the original image, translating it according to the offset. This means that all the pixels in the kernel are moved to the position of each pixel in the image.

Finally, we can compare the mvd directly with the img to obtain an array of Boolean values representing whether the grayscale of the pixels inside the kernel exceeds that of the center pixel.

By summing this array, we can obtain the number of pixels surrounding each pixel whose grayscale does not exceed its own grayscale within the kernel.

Dividing this by the size of the kernel yields the frequency of that grayscale. Multiplying this frequency by 255 yields the grayscale after the center pixel has been equalized.

Complexity Analysis

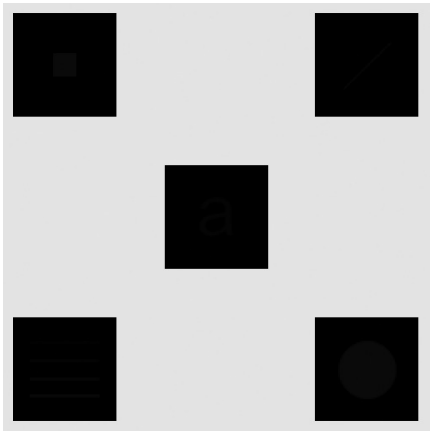
The time complexity of the naive algorithm is $O(nm \times (2r-1)^2)$, which cannot be changed. However, matrix acceleration greatly improves the speed. If matrix calculations are regarded as ordinary operations with a time complexity of 1, the time complexity of this method is "$O(1)$".

There is no need to analyze the spatial complexity, as its cost will not exceed the storage cost of the image itself.

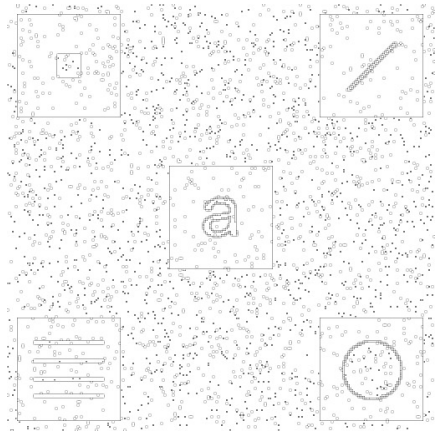
4.3 Results

Performance

The raw image and the result are shown in Figure 11 and the histogram is shown in 12. The kernel radius we pick is 2, which means the kernel is of a size of 3x3.



(a) The raw image, Image 3.



(b) The locally equalized image from Image 3 (radius = 2).

Figure 11. The output images of local histogram equalization.

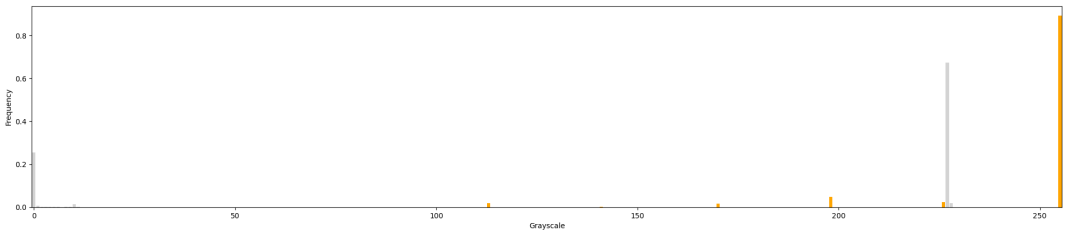


Figure 12. The histogram of the raw image and the output image of local histogram equalization.

As we can see, the information in the result image is not visible in the original image because the gray values of the pixels inside the black blocks are too close to each other. Moreover, the overall contrast of the image is high due to the presence of the background, making it unsuitable for the global processing method used previously.

However, after applying local histogram equalization, the algorithm ignores the global contrast relationship and increases the local contrast, thereby revealing the hidden and valuable information.

From the comparison of histograms, we can also see that the higher weighted gray values in local histogram equalization do not have a significant impact on other parts, and the local histogram equalization algorithm is different from the two histogram processing algorithms discussed earlier. It may cause pixels that were originally the same color to become inconsistent, resulting in a "non-conservation" histogram.

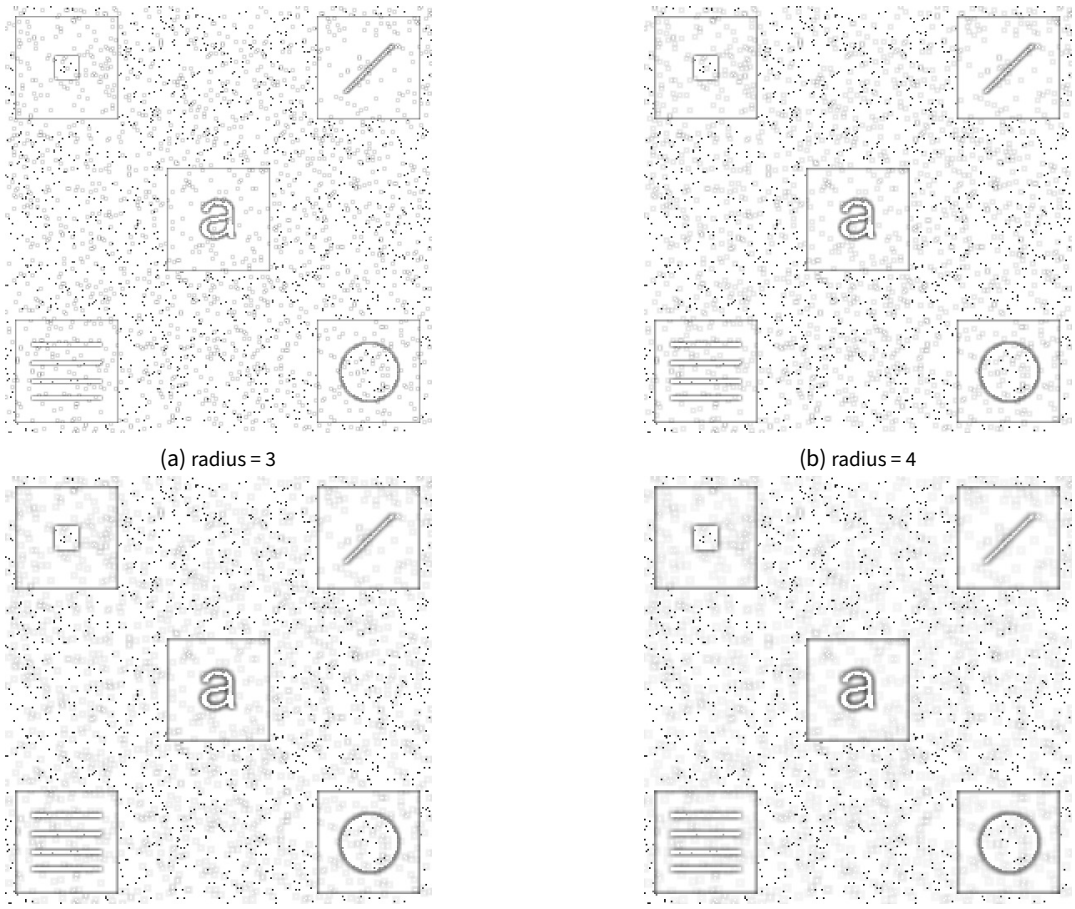
Time Cost

The average time cost is about 0.0254 seconds. The speed improvement mainly comes from the use of NumPy. If loops are used for implementation, there will be a significant time cost, and processing slightly larger images may result in delays of several seconds.

Effect of Changes in Kernel Radius

Local histogram equalization allows us to specify the size of the kernel. Here, we will set the kernel radius to 3, 4, 5, and 6 and simply compare the results 13.

By adjusting the size of the kernel radius, we can observe that some edges in the image become more pronounced. This is because as the kernel size increases, the algorithm becomes more global in its processing, and the gradually deepening “shadows” highlight areas of high contrast in the



(a) radius = 3 (b) radius = 4

(c) radius = 5 (d) radius = 6

Figure 13. Changing the kernel radius in local histogram equalization.

image.

5 Median Filtering

5.1 Algorithm Principle and Pseudo-code

Salt-and-pepper Noise

Salt-and-pepper noise is a type of image noise that appears as random white and black pixels scattered throughout an image, similar to grains of salt and pepper. This type of noise is commonly caused by errors in the image acquisition process, such as corrupted pixels in a digital camera sensor or errors in the transmission of image data.

To eliminate salt-and-pepper noise, we can take the neighborhood of each pixel and then sort all the pixels in the neighborhood by grayscale. The median value is then taken as the new value for that pixel. This is because salt-and-pepper noise mainly consists of a few black and white pixels, and during the sorting process, these two grayscale values will be ranked at the front or the end. Taking the median value can effectively remove the salt-and-pepper noise.

Pseudo-code

Here is the pseudo-code (Algorithm 4).

Algorithm 4: Median Filtering

Input: $img, radius$
Output: img_new

foreach $(x, y) \in img$ **do**
  foreach $(i, j) \in the\ kernel\ centered\ at\ (x, y)$ **do**
    $img\_new[x, y] \leftarrow GetTheMedian(kernel)$
  end
end

5.2 Python Implementation

A Few Modifications on Local Histogram Equalization

In fact, the implementation of this algorithm is very similar to that of local histogram equalization, because both actually involve calculating the surrounding kernel for each pixel (Code 7). We only need to replace the comparison and summation operations in local histogram equalization with $np.median$ to calculate the median, which is very simple.

```
def reduce_SAP(img, radius):  # radius should be an integer and larger than 1
    size = radius * 2 - 1
    img = np.pad(img.astype(np.int32), ((radius - 1, radius - 1), (radius - 1, radius - 1)), 'reflect')

    pts = [(idx // size - radius + 1, idx % size - radius + 1) for idx in range(size * size)]
    mvd = np.array([np.roll(np.roll(img, x, axis=0), y, axis=1) for (x, y) in pts])
    res = np.median(mvd, axis=0)  # [!] The only difference

    res = res[(radius - 1):(1 - radius), (radius - 1):(1 - radius)].astype(np.uint8)
    return res
```

Listing 7. Median Filtering

The seventh line is the only difference between this code and the previous one.

Complexity Analysis

The time and space complexities are the same as local histogram equalization, and will not be discussed further here.

5.3 Results

Performance

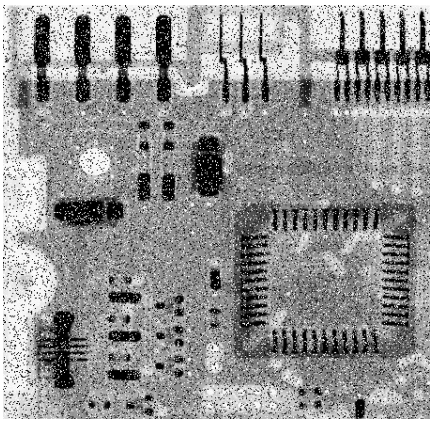
The raw image and the result are shown in Figure 14. The kernel radius we pick is 2, which means the kernel is of a size of 3x3.

As you can see, the obvious spots that were present before have been removed.

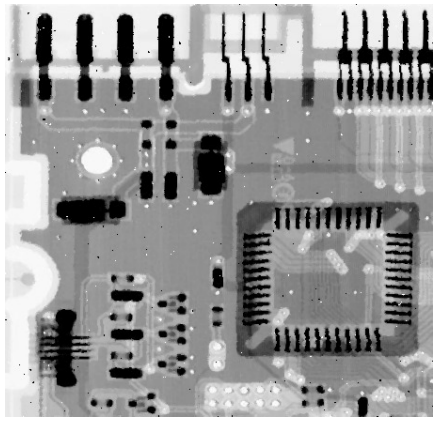
Time Cost

The average time cost is about 0.0281 seconds. The speed improvement mainly comes from the use of NumPy. If loops are used for implementation, there will be a significant time cost, and processing slightly larger images may result in delays of several seconds.

The code structure is similar to local histogram equalization, but median filtering is slightly slower because the process of taking the median is slower than the comparison and addition process.



(a) The raw image, Image 4.

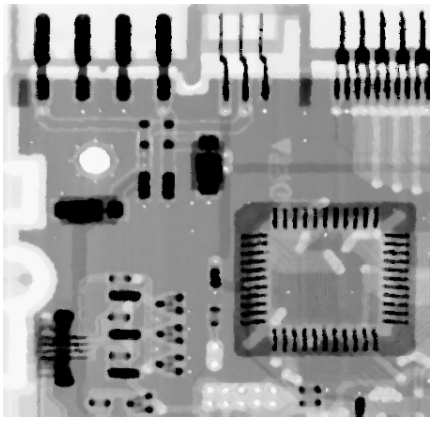


(b) The locally equalized image from Image 4 (radius = 2).

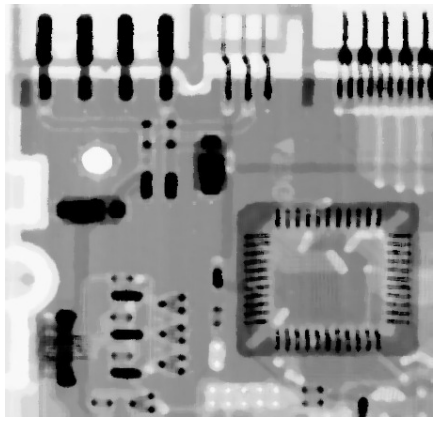
Figure 14. The output images of median filtering.

Effect of Changes in Kernel Radius

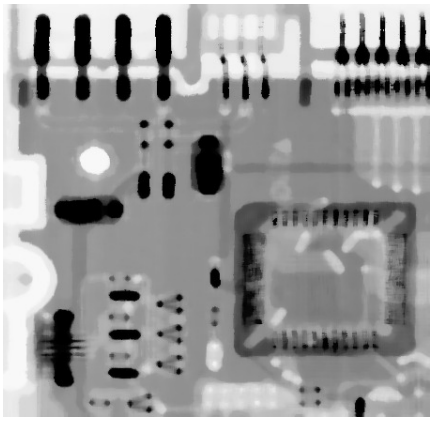
Median filtering also allows us to specify the size of the kernel. Here, we will set the kernel radius to 3, 4, 5, and 6 and simply compare the results 15.



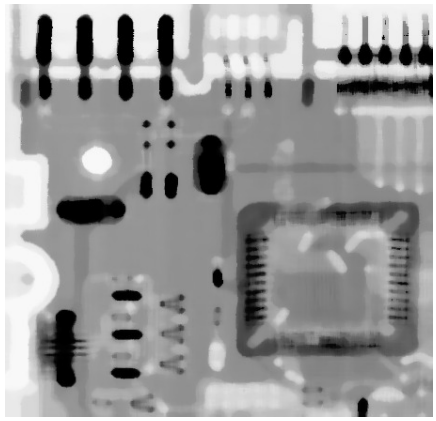
(a) radius = 3



(b) radius = 4



(c) radius = 5



(d) radius = 6

Figure 15. Changing the kernel radius in median filtering.

By changing the size of the kernel, we can observe that the result becomes smoother and the speckles eventually disappear completely. However, at the same time, details are gradually blurred

and lost. This is because median filtering is equivalent to applying a low-pass filter to the image.

6 Conclusion

In this article, we have implemented four image processing algorithms: histogram equalization, histogram matching, local histogram equalization, and median filtering.

Histogram equalization is used to adjust the dynamic range of low-contrast images to enhance them. The principle is to treat the gray level of the image as a random variable and adjust its probability density function through transformation to make it uniform, which is very inspiring.

Histogram matching can make up for some extreme cases where histogram equalization is not applicable. Histogram equalization treats input images equally, while histogram matching allows for manually fixing target gray level distributions for different images to achieve correction and image enhancement.

Local histogram equalization can ignore global contrast and enhance the local contrast of the image, avoiding the influence of other high-contrast parts of the image. The implementation takes advantage of the characteristics of matrix operation libraries to greatly accelerate calculation without using loop structures. The size of the kernel determines the degree of importance of the algorithm for global situations.

Median filtering is used to remove salt-and-pepper noise by filtering out extreme values through taking the median. The implementation method is similar to local histogram equalization, and adjusting the size of the kernel will determine the degree of detail preservation.

The above methods mainly embody the ideas of transformation, gray-level domain, convolution kernel, and filtering.