A Brief Introduction to Common Digital Image Restoration Algorithms

Wang Zhuoyang[1]

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

Abstract

No abstract and see *Introduction*. And it is true that there should not be such an unreasonable thing as a lab report with a bunch of requirements attached to it in this world.

Keywords: Image Restoration; Digital Image Processing.

Contents

Digital Image Processing Laboratory Reports (2023)

Corresponding author
Wang Zhuoyang

1 Introduction

Application Scenarios

Image restoration techniques are used in a variety of applications where it is necessary to improve the quality of digital images that have been degraded or corrupted. In fields such as medical imaging, forensics, and astronomy, high-quality images are essential for accurate analysis and interpretation.

In the medical field, image restoration techniques can be used to enhance medical images such as CT scans or MRI scans. These techniques can help to improve the clarity of the images, making it easier for doctors to diagnose and treat medical conditions.

In forensic investigations, image restoration techniques can be used to enhance low-quality images, such as those captured by surveillance cameras. These techniques can help investigators to identify suspects or uncover important details that may have been otherwise impossible to see.

In astronomy, image restoration techniques are used to improve the quality of images captured by telescopes. By removing distortions caused by atmospheric turbulence or other factors, astronomers can obtain clearer images of celestial objects, allowing for more accurate measurements and analysis.

Overall, image restoration techniques have a wide range of applications in fields where high-quality images are essential. From medical imaging to forensics and astronomy, these techniques are helping to improve our understanding of the world around us.

Methodology

In this report, we will introduce common image restoration algorithms, including basic spatial filtering, adaptive spatial filters, mathematical model-based filtering in the frequency domain, and filters derived from optimized statistical measures.

Additionally, we will apply these methods to solve three practical problems: removing mixed noise, eliminating atmospheric interference, and rectifying linear motion blur.

Results

We observe and analyze each problem, and apply appropriate algorithms for image restoration, obtaining filtered results under various filter parameters (if any). We also calculate statistical measures such as SNR and MSE and compare them to conduct a comparative analysis. The corresponding results and analysis are shown in the following figures.

2 Denoising

2.1 Common Spatial Filtering Approaches

Pseudo-code and Complexity

Considering that the overall framework of spatial filtering algorithms is consistent, we will only provide a pseudo-code paradigm (Algorithm 1) once. The specific algorithms in the following sections can be incorporated accordingly.

Algorithm 1: Spatial filtering with operations in the kernel S_{xy}.

Input: img
Output: res

foreach $(x, y) \in img$ **do**
  foreach $(i, j) \in the\ kernel\ S_{xy}\ centered\ at\ (x, y)$ **do**
    $res[x, y] \leftarrow$ SomeFunction(S_{xy})
  end
end
return res

Similarly, we will perform complexity analysis only once. We assume that the time complexity of the current spatial filtering algorithm within the kernel S_{xy} is O_S.

For each point in the image, we need to perform a calculation. Let the width and height of the image be N and M, respectively. The overall time complexity of the algorithm can be expressed as

$$O(N \times M \times O_S) \tag{1}$$

The calculation of space complexity is not significant and depends on the implementation of the algorithm. In most cases, the space resources occupied by the calculation of different pixel values can be reused. In such cases, the space complexity is roughly $O(N \times M)$ multiplied by a constant.

Arithmetic Mean Filter

The arithmetic mean filter is the most basic spatial filtering method, which computes the arithmetic mean of the elements within the kernel and assigns it as the gray value of the corresponding pixel.

The algorithm is widely used and commonly applied in image blurring scenarios that do not require high image quality. The biggest issue with the arithmetic mean filter is that the shape of the kernel is square rather than circular, resulting in anisotropic filtering results.

Its behavior can be described by the following formula:

$$\hat{f}(x, y) = \frac{1}{mn} \sum_{(s,t) \in S_{xy}} g(s, t) \tag{2}$$

Pseudo-code has been provided at the beginning of this section. And here is the Python code for the algorithm (Code 1).

```python
def arithmetic_mean_filter(img, kernel_dim, pad_mode='constant', regulator=
    NonRegulator):
    return window_process(img, np.ones(kernel_dim) / kernel_dim[0] /
    kernel_dim[1], pad_mode=pad_mode, regulator=regulator)
```

Listing 1. Arithmetic Mean Filter.

The *window_process* is a generic function for performing kernel operations, which is implemented as follows (Code 2). It can handle different linear operations on kernels with different

centers and sizes. It will not be provided subsequently.

```
def window_process(img, kernel, center=None, process_func=None, pad_mode='
    constant', inter_type=np.int32, regulator=NonRegulator):
    def pf_sum(x): return np.sum(x, axis=0)
    size = kernel.shape
    rad = np.array(size) // 2
    if center is None: center = rad
    if process_func is None: process_func = pf_sum
    img_pad = np.pad(img.astype(inter_type), ((rad[0], rad[0]), (rad[1], rad
    [1])), mode=pad_mode)
    kernel_spanned = np.expand_dims(np.reshape(kernel, -1), axis=(1, 2))
    pts = [(center[0] - idx // size[1], center[1] - idx % size[1]) for idx
    in range(size[0] * size[1])]
    moved = np.array([np.roll(np.roll(img_pad, x, axis=0), y, axis=1) for (x
    , y) in pts])
    res = process_func(moved * kernel_spanned)
    return regulator(res[rad[0]:-rad[0], rad[1]:-rad[1]])
```

Listing 2. Kernel operating.

Geometric Mean Filter

The geometric mean filter is a type of spatial filter that calculates the geometric average of pixel values within a kernel. It is effective in reducing noise while preserving edges and textures in the image.

It is commonly used for images with additive Gaussian noise.

Its behavior can be described by the following formula:

$$\hat{f}(x,y) = \left[\prod_{(s,t)\in S_{xy}} g(s,t) \right]^{\frac{1}{mn}} \tag{3}$$

Pseudo-code has been provided at the beginning of this section. And here is the Python code for the algorithm (Code 3).

```
def geometric_mean_filter(img, kernel_dim, pad_mode='constant', regulator=
    NonRegulator):
    def pf_prod(x): return np.prod(x, axis=0)
    return regulator(window_process(img ** (1.0 / kernel_dim[0] / kernel_dim
    [1]), np.ones(kernel_dim), process_func=pf_prod, pad_mode=pad_mode,
    inter_type=np.float32))
```

Listing 3. Geometric Mean Filter.

Harmonic Mean Filter

The harmonic mean filter is a type of spatial filter that calculates the harmonic mean of pixel values within a kernel. It is effective in removing the effects of high-frequency noise, but may over-smooth the image.

It is commonly used for images with impulse noise.

Its behavior can be described by the following formula:

$$\hat{f}(x,y) = \frac{mn}{\sum_{(s,t)\in S_{xy}} \frac{1}{g(s,t)}} \tag{4}$$

Pseudo-code has been provided at the beginning of this section. And here is the Python code for the algorithm (Code 4).

```python
def harmonic_mean_filter(img, kernel_dim, pad_mode='constant', regulator=
    NonRegulator):
    return regulator(kernel_dim[0] * kernel_dim[1] / window_process(1.0 /
    img, np.ones(kernel_dim), pad_mode=pad_mode, inter_type=np.float32))
```

Listing 4. Harmonic Mean Filter.

Contra-harmonic Mean Filter

The contra-harmonic mean filter is a type of spatial filter that calculates the weighted average of pixel values within a kernel. It is effective in removing the effects of both high- and low-frequency noise, but may cause over-smoothing or ringing artifacts.

It is commonly used for images with both impulse and Gaussian noise.

Its behavior can be described by the following formula:

$$\hat{f}(x,y) = \frac{\sum_{(s,t)\in S_{xy}} g(s,t)^{Q+1}}{\sum_{(s,t)\in S_{xy}} g(s,t)^{Q}} \tag{5}$$

Pseudo-code has been provided at the beginning of this section. And here is the Python code for the algorithm (Code 5).

```python
def contraharmonic_mean_filter(img, kernel_dim, q, pad_mode='constant',
    regulator=NonRegulator):
    u = window_process(np.float_power(img, q + 1), np.ones(kernel_dim),
    pad_mode=pad_mode, inter_type=np.float32)
    d = window_process(np.float_power(img, q), np.ones(kernel_dim), pad_mode
    =pad_mode, inter_type=np.float32)
    return regulator(u / d)
```

Listing 5. Contra-harmonic Mean Filter.

Max / Min / Midpoint Filter

These filters are effective in removing small details and enhancing edges in the image. Max filter is effective in the presence of pepper noise and Min filter is effective in the presence of salt noise.

These three filters share the same basic structure, and can be implemented by replacing the processing function. Here we take the max filter as an example. Its behavior can be described by the following formula:

$$\hat{f}(x,y) = \max_{(s,t)\in S_{xy}} \{g(s,t)\} \tag{6}$$

Pseudo-code has been provided at the beginning of this section. And here is the Python code for the algorithm (Code 6).

```python
def max_filter(img, kernel_dim, pad_mode='constant', regulator=NonRegulator)
    :
    def pf_max(x): return np.max(x, axis=0)
    return window_process(img, np.ones(kernel_dim), process_func=pf_max,
    pad_mode=pad_mode, regulator=regulator)
```

Listing 6. Max Filter.

Median Filter

The median filter is a type of spatial filter that replaces each pixel value within a kernel with the median pixel value within that kernel. It is effective in removing impulse noise while preserving image details and textures.

It is commonly used for images with impulse noise like pepper-and-salt noise.

Its behavior can be described by the following formula:

$$\hat{f}(x,y) = \text{median}_{(s,t)\in S_{xy}}\{g(s,t)\} \tag{7}$$

Pseudo-code has been provided at the beginning of this section. And here is the Python code for the algorithm (Code 7).

```
def median_filter(img, kernel_dim, pad_mode='constant', regulator=
    NonRegulator):
    def pf_median(x): return np.median(x, axis=0)
    return window_process(img, np.ones(kernel_dim), process_func=pf_median,
    pad_mode=pad_mode, regulator=regulator)
```

Listing 7. Median Filter.

Alpha-trimmed Mean Filter

The alpha-trimmed mean filter is a type of spatial filter that calculates the trimmed mean of pixel values within a kernel. It is effective in removing both Gaussian and impulse noise, but may cause over-smoothing or loss of image details if the trimming parameter is set too high.

This method is useful in situations involving multiple types of noise, such as a combination of salt-and-pepper and Gaussian noise.

Its behavior can be described by the following formula:

$$\hat{f}(x,y) = \frac{1}{mn-d}\sum_{(s,t)\in S_r g(s,t)} \tag{8}$$

where S_r represents the pixel values after removing the top $d/2$ largest and bottom $d/2$ smallest values within the kernel S_{xy}.

Pseudo-code has been provided at the beginning of this section. The Python code is emitted.

2.2 Adaptive Filters

Adaptive Local Noise Reduction Filter

This filter is a type of spatial domain filter that adapts its filtering behavior based on the local image characteristics. It is particularly effective in reducing noise in regions with varying levels of noise and detail. The filter works by applying a weighted average to the pixels in a local neighborhood around each pixel in the image. The weights are determined by estimating the noise variance in the local neighborhood, and the filter adaptively adjusts these weights based on the local image characteristics.

The basic idea of this filter is to determine the gray value of a pixel based on statistical measures such as the mean and variance of the original image and the window it belongs to. Therefore, the following quantities are defined:

1. $g(x,y)$, the value of the noisy image at (x,y);
2. σ_η^2, the variance of the noise corrupting $f(x,y)$ to form $g(x,y)$;
3. m_L, the local mean of the pixels in S_{xy};
4. σ_L^2, the local variance of the pixels in S_{xy}.

During the filtering process, the following rules will be followed:

1. If σ_η^2 is zero, the filter should return simply the value of $g(x,y)$;
2. If the local variance is high relative to σ_η^2, the filter should return a value close to $g(x,y)$;
3. If the two variances are equal, the filter returns the arithmetic mean value of the pixels in S_{xy}.

According to the above rules, the following transformation formula that satisfies the requirements can be derived:

$$\hat{f}(x,y) = g(x,y) - \frac{\sigma_\eta^2}{\sigma_L^2}\left[g(x,y) - m_L\right] \tag{9}$$

And the algorithm workflow is as the following (Algorithm 2).

Algorithm 2: Adaptive Local Noise Reduction Filter.

Input: g, σ_η
Output: res

$m_L \leftarrow \frac{1}{mn}\sum_{(x,y)\in G_{xy}} g(x,y)$
foreach $(x,y) \in g$ **do**
  foreach $(i,j) \in the\ kernel\ S_{xy}\ centered\ at\ (x,y)$ **do**
    $\sigma_L^2 \leftarrow$ the variance of the pixels in S_{xy}
    $res[x,y] \leftarrow g(x,y) - \frac{\sigma_\eta^2}{\sigma_L^2}\left[g(x,y) - m_L\right]$
  end
end
return res

The actual implementation is similar to the previous simple spatial filtering. The time complexity is $O(M \times N \times K \times L)$, where K and L are the length and width of the kernel, and the space complexity is the same as the amount of space occupied by the image. The code is shown below (Code 8).

```
def adaptive_local_noise_reduction_filter(img, kernel_dim, V_noise, pad_mode
    ='constant', regulator=NonRegulator):
    def pf_var(x): return np.var(x, axis=0)
    m_L = window_process(img, np.ones(kernel_dim) / kernel_dim[0] /
    kernel_dim[1], pad_mode=pad_mode, inter_type=np.float32)
    V_L = window_process(img, np.ones(kernel_dim), process_func=pf_var,
    pad_mode=pad_mode, inter_type=np.float32)
    return regulator(img - (img - m_L) * V_noise / V_L)
```

Listing 8. Adaptive Local Noise Reduction Filter.

Adaptive Median Filter

This filter is also a type of spatial domain filter that adapts its filtering behavior based on the local image characteristics. It is particularly effective in removing impulse noise, which is a type of noise that causes sudden spikes or drops in the pixel values. The filter works by first comparing the pixel value at the center of a local neighborhood with the values of the surrounding pixels. If the center pixel is found to be an outlier, the filter applies a median filter operation to the local neighborhood, otherwise it leaves the center pixel unchanged. The size of the local neighborhood is increased iteratively until an acceptable output is obtained.

The algorithm is like the following (Algorithm 3).

The algorithm runs slowly and the worst-case time complexity is approximately $O(M \times N \times (3^2 + 5^2 + \cdots + S_{max}^2))$, similar to the previous method. The space complexity is also similar. In practice, parallelization can be considered to speed up the calculation.

The code is provided in Code 9.

```
def adaptive_median_filter(img, s_max, pad_mode='constant', regulator=
    NonRegulator):
```

Algorithm 3: Adaptive Median Filter.

Input: z, S_{max}
Output: res

foreach $(x, y) \in z$ **do**
  foreach $(i, j) \in$ *the kernel* S_{xy} *centered at* (x, y) **do**
    Stage A:
    $z_{med} \leftarrow$ the median value in S_{xy}
    $z_{min} \leftarrow$ the min value in S_{xy}
    $z_{max} \leftarrow$ the max value in S_{xy}
    if $z_{med} \neq z_{min}$ *and* $z_{med} \neq z_{max}$ **then**
      Goto Stage B
    end
    else
      Increase the size of window
      if *window size* $\leq S_{max}$ **then**
        Goto Stage A
      end
      else
        $res[x, y] \leftarrow z_{med}$
        Continue
      end
    end
    Stage B:
    if $z[x, y] \neq z_{min}$ *and* $z[x, y] \neq z_{max}$ **then**
      $res[x, y] \leftarrow z[x, y]$
    end
    else
      $res[x, y] \leftarrow z_{med}$
    end
  end
end
return res

```
    r_max = s_max // 2
    img_pad = np.pad(img.astype(np.int32), ((r_max, r_max), (r_max, r_max)),
     mode=pad_mode)
    res = np.zeros(img.shape)
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            r, c = i + r_max, j + r_max
            r_now = 1
            while True:
                z_xy = img_pad[r, c]
                z_win = img_pad[r - r_now : r + r_now + 1, c - r_now : c +
r_now + 1]
                z_min = np.min(z_win)
                z_max = np.max(z_win)
                z_med = np.median(z_win).astype(np.int32)
                if z_med == z_min or z_med == z_max:
                    r_now += 1
                    if r_now > r_max:
                        res[i, j] = z_med
                        break
                else:
                    if z_xy == z_min or z_xy == z_max:
```

```
                        res[i, j] = z_med
                    else:
                        res[i, j] = z_xy
                    break
    return regulator(res)
```

Listing 9. Adaptive Median Filter.

2.3 Results

Image with Pepper Noise

The raw image is shown in Figure 1.

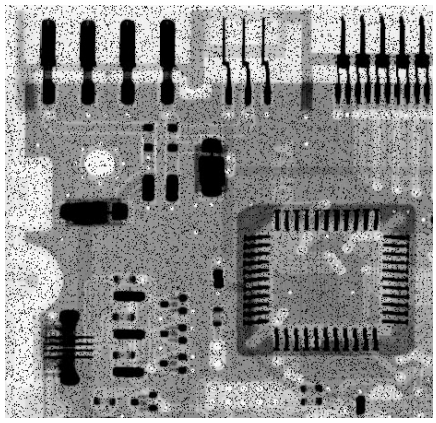
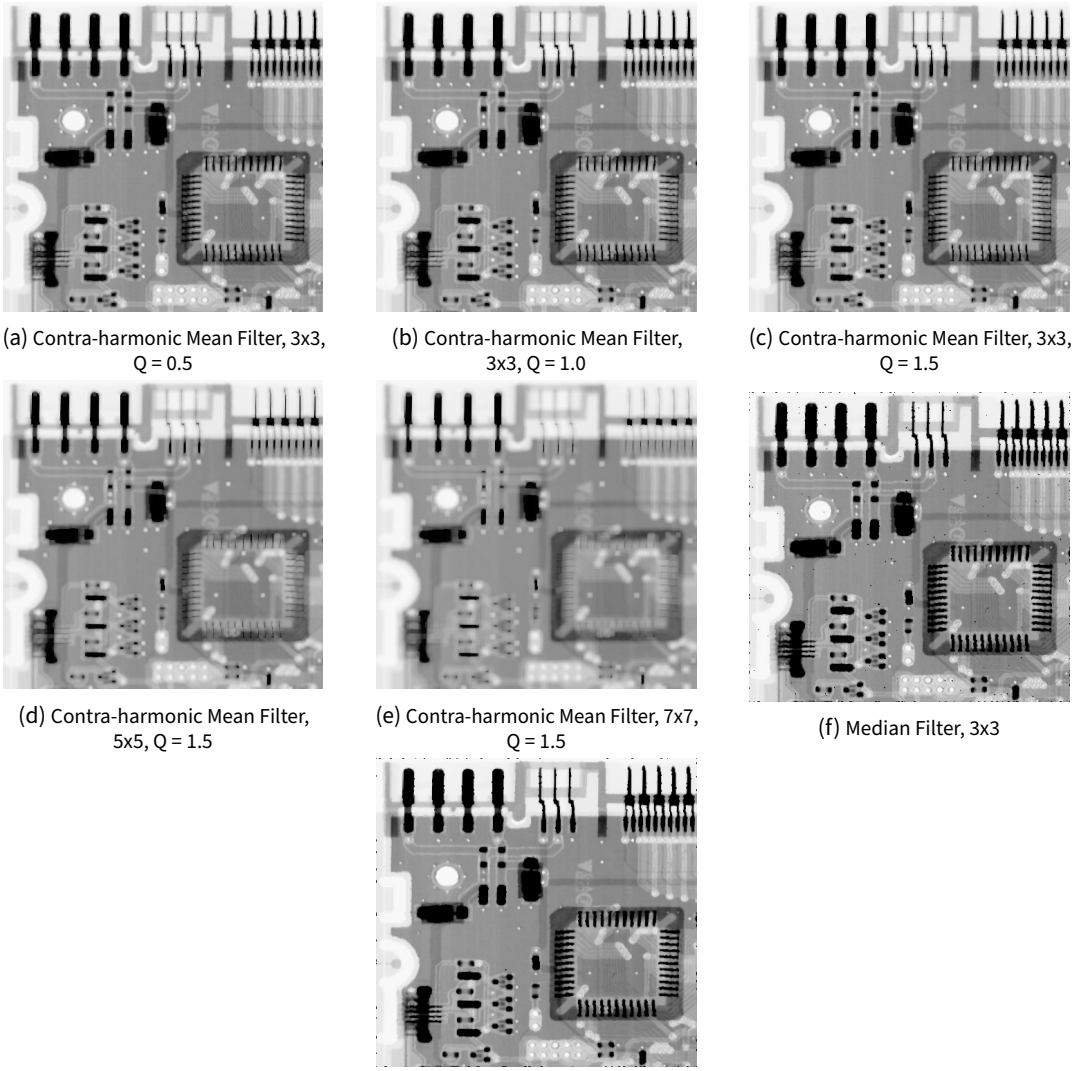


Figure 1. Raw Image Q6_1_1.

By observing the sample image in Figure 1, it is apparent that the image is mainly affected by pepper noise. We can use commonly used spatial filters for denoising, and the results are shown in Figure 2. Considering the nature of pepper noise, we mainly utilized contra-harmonic filters with different sizes and positive coefficient Q values, as well as single and double median filtering methods.

```
img = persistence.load_gray('res/Q6_1_1.tiff')
persistence.save_gray('output/ans/Q6_1_1_contraharmonic_mean_3x3_1.5.jpg',
    spatial.contraharmonic_mean_filter(img, (3, 3), 1.5, regulator=regulator
    .GrayCuttingRegulator))
persistence.save_gray('output/ans/Q6_1_1_contraharmonic_mean_5x5_1.5.jpg',
    spatial.contraharmonic_mean_filter(img, (5, 5), 1.5, regulator=regulator
    .GrayCuttingRegulator))
persistence.save_gray('output/ans/Q6_1_1_contraharmonic_mean_7x7_1.5.jpg',
    spatial.contraharmonic_mean_filter(img, (7, 7), 1.5, regulator=regulator
    .GrayCuttingRegulator))
persistence.save_gray('output/ans/Q6_1_1_contraharmonic_mean_3x3_0.5.jpg',
    spatial.contraharmonic_mean_filter(img, (3, 3), 0.5, regulator=regulator
    .GrayCuttingRegulator))
persistence.save_gray('output/ans/Q6_1_1_contraharmonic_mean_3x3_1.jpg',
    spatial.contraharmonic_mean_filter(img, (3, 3), 1, regulator=regulator.
    GrayCuttingRegulator))
persistence.save_gray('output/ans/Q6_1_1_median_3x3.jpg', spatial.
    median_filter(img, (3, 3), regulator=regulator.GrayCuttingRegulator))
persistence.save_gray('output/ans/Q6_1_1_median_3x3_twice.jpg', spatial.
    median_filter(spatial.median_filter(img, (3, 3), regulator=regulator.
    GrayCuttingRegulator), (3, 3), regulator=regulator.GrayCuttingRegulator)
    )
```

Listing 10. Spatial filtering for the raw image with pepper noise.



(a) Contra-harmonic Mean Filter, 3x3, Q = 0.5

(b) Contra-harmonic Mean Filter, 3x3, Q = 1.0

(c) Contra-harmonic Mean Filter, 3x3, Q = 1.5

(d) Contra-harmonic Mean Filter, 5x5, Q = 1.5

(e) Contra-harmonic Mean Filter, 7x7, Q = 1.5

(f) Median Filter, 3x3

(g) Median Filter, 3x3, Twice

Figure 2. The results for the degraded image with pepper noise.

Comparing the results, it can be observed that the contra-harmonic mean filter produces smoother results, with darker regions tending to recess inward. The filtering effect increases with increasing Q values, but if Q is too large, black regions may become incomplete. Using a larger kernel has a similar effect. The results of median filtering are clearer and more definitive. Even after a single pass, there are still some bad points, which are basically eliminated after multiple passes.

Image with Salt Noise

The raw image is shown in Figure 3.

By observing the sample image in Figure 3, it is apparent that the image is mainly affected by salt noise. We can use commonly used spatial filters for denoising, and the results are shown in Figure 4. Considering the nature of salt noise, we mainly utilized contra-harmonic filters with different sizes and negative coefficient Q values, as well as single and double median filtering methods.

```
img = persistence.load_gray('res/Q6_1_2.tiff')
persistence.save_gray('output/ans/Q6_1_2_contraharmonic_mean_3x3_-1.5.jpg',
    spatial.contraharmonic_mean_filter(img, (3, 3), -1.5, regulator=
    regulator.GrayCuttingRegulator))
```

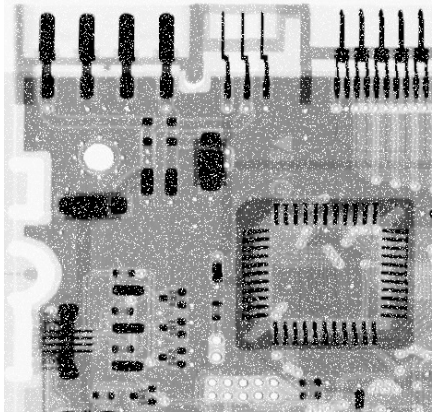


Figure 3. Raw Image Q6_1_2.

```
persistence.save_gray('output/ans/Q6_1_2_contraharmonic_mean_5x5_-1.5.jpg',
    spatial.contraharmonic_mean_filter(img, (5, 5), -1.5, regulator=
    regulator.GrayCuttingRegulator))
persistence.save_gray('output/ans/Q6_1_2_contraharmonic_mean_7x7_-1.5.jpg',
    spatial.contraharmonic_mean_filter(img, (7, 7), -1.5, regulator=
    regulator.GrayCuttingRegulator))
persistence.save_gray('output/ans/Q6_1_2_contraharmonic_mean_3x3_-1.jpg',
    spatial.contraharmonic_mean_filter(img, (3, 3), -1, regulator=regulator.
    GrayCuttingRegulator))
persistence.save_gray('output/ans/Q6_1_2_contraharmonic_mean_3x3_-2.jpg',
    spatial.contraharmonic_mean_filter(img, (3, 3), -2, regulator=regulator.
    GrayCuttingRegulator))
persistence.save_gray('output/ans/Q6_1_2_median_3x3.jpg', spatial.
    median_filter(img, (3, 3), regulator=regulator.GrayCuttingRegulator))
persistence.save_gray('output/ans/Q6_1_2_median_3x3_twice.jpg', spatial.
    median_filter(spatial.median_filter(img, (3, 3), regulator=regulator.
    GrayCuttingRegulator), (3, 3), regulator=regulator.GrayCuttingRegulator)
    )
```

Listing 11. Spatial filtering for the raw image with salt noise.

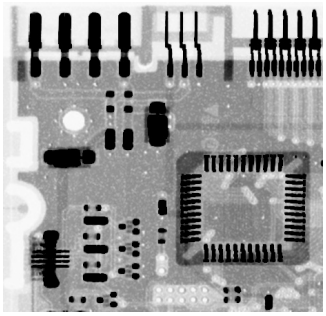
After comparing the results, we can observe that the filtering effect follows a pattern similar to the previous image when the parameter is changed. The difference is that using negative Q values (i.e., closer to the harmonic mean filter) tends to stretch the darker regions, which can cause certain dark regions to merge into a single area when the kernel is large.

Image with Pepper-and-salt Noise

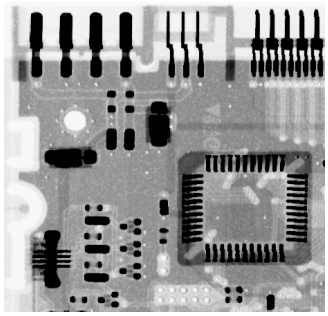
The raw image is shown in Figure 5.

By observing the sample image in Figure 5, it is apparent that the image is mainly affected by pepper-and-salt noise. We can use commonly used spatial filters for denoising, and the results are shown in Figure 8. Considering the nature of pepper-and-salt noise, we mainly utilized median filter and adaptive median filter to restore the image.

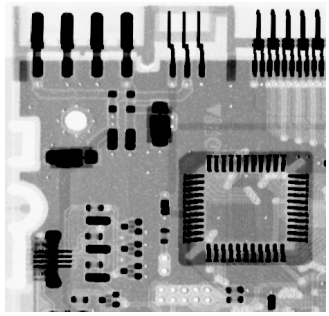
```
img = persistence.load_gray('res/Q6_1_3.tiff')
persistence.save_gray('output/ans/Q6_1_3_median_3x3.jpg', spatial.
    median_filter(img, (3, 3), regulator=regulator.GrayCuttingRegulator))
persistence.save_gray('output/ans/Q6_1_3_median_3x3_twice.jpg', spatial.
    median_filter(spatial.median_filter(img, (3, 3), regulator=regulator.
    GrayCuttingRegulator), (3, 3), regulator=regulator.GrayCuttingRegulator)
    )
persistence.save_gray('output/ans/Q6_1_3_median_3x3_thrice.jpg', spatial.
    median_filter(spatial.median_filter(spatial.median_filter(img, (3, 3),
```



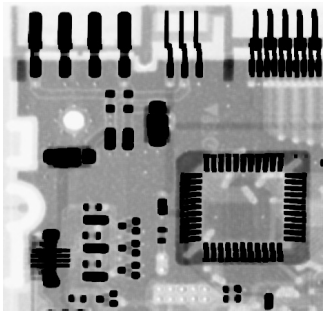
(a) Contra-harmonic Mean Filter, 3x3, Q = -1.0



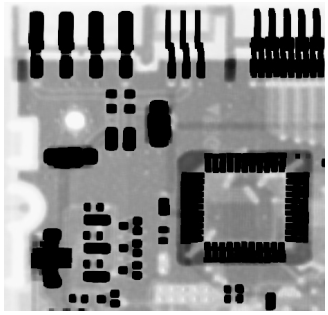
(b) Contra-harmonic Mean Filter, 3x3, Q = -1.5



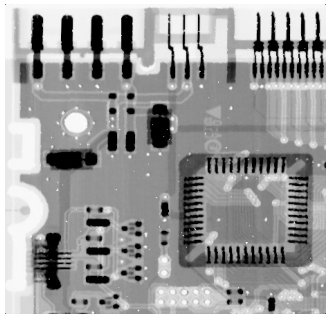
(c) Contra-harmonic Mean Filter, 3x3, Q = -2.0



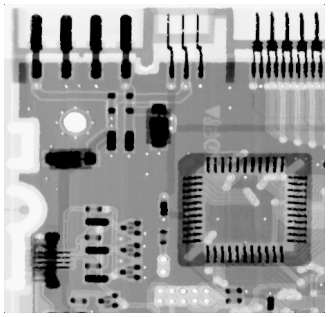
(d) Contra-harmonic Mean Filter, 5x5, Q = -1.5



(e) Contra-harmonic Mean Filter, 7x7, Q = -1.5



(f) Median Filter, 3x3



(g) Median Filter, 3x3, Twice

Figure 4. The results for the degraded image with pepper noise.

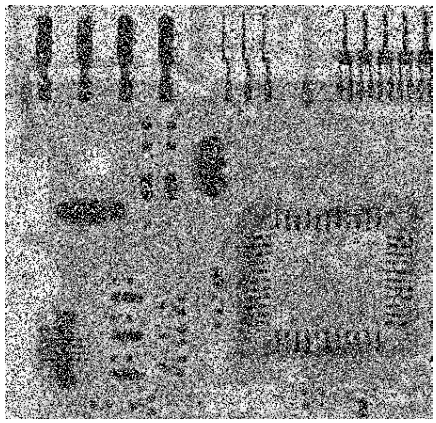
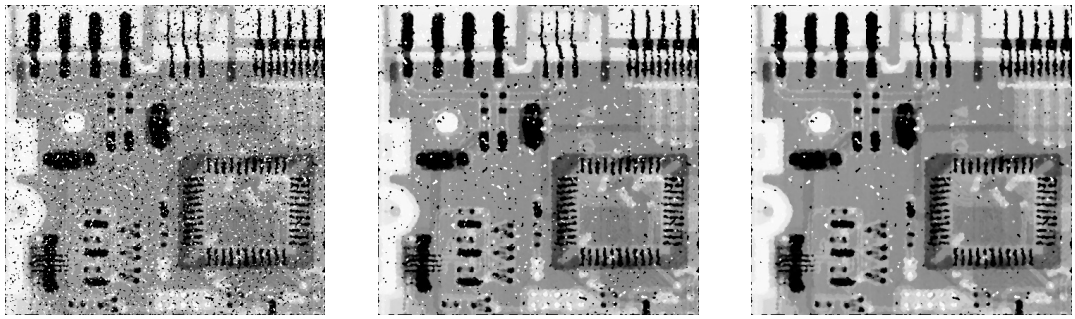


Figure 5. Raw Image Q6_1_3.

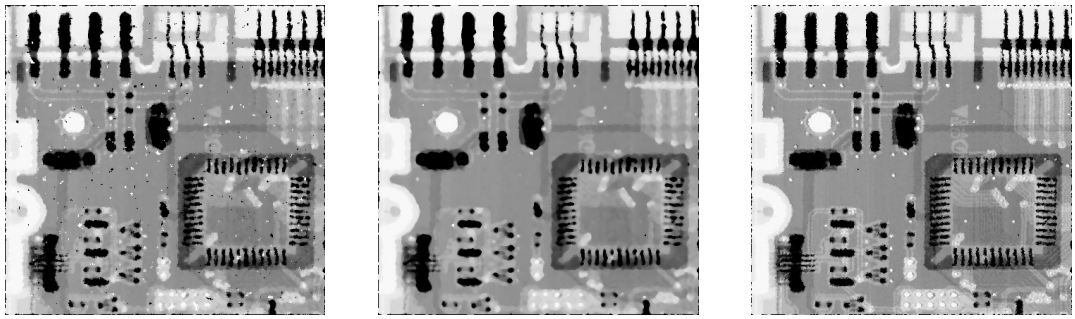
```
    regulator=regulator.GrayCuttingRegulator), (3, 3), regulator=regulator.
    GrayCuttingRegulator), (3, 3), regulator=regulator.GrayCuttingRegulator)
```

```
    )
persistence.save_gray('output/ans/Q6_1_3_median_5x5.jpg', spatial.
    median_filter(img, (5, 5), regulator=regulator.GrayCuttingRegulator))
persistence.save_gray('output/ans/Q6_1_3_median_5x5_twice.jpg', spatial.
    median_filter(spatial.median_filter(img, (5, 5), regulator=regulator.
    GrayCuttingRegulator), (5, 5), regulator=regulator.GrayCuttingRegulator)
    )
persistence.save_gray('output/ans/Q6_1_3_adaptive_median_7.jpg', spatial.
    adaptive_median_filter(img, 7, regulator=regulator.GrayCuttingRegulator)
    )
```

Listing 12. Spatial filtering for the raw image with mixed impulse noise.



(a) Median Filter, 3x3 (b) Median Filter, 3x3, Twice (c) Median Filter, 3x3, Thrice



(d) Median Filter, 5x5 (e) Median Filter, 5x5, Twice (f) Adaptive Median Filter, $S_{max} = 7$

Figure 6. The results for the degraded image with mixed impulse noise.

After observing the results, we found that the pure median filter seems to be inadequate. One solution is to apply the median filter multiple times to obtain a relatively acceptable result. Another solution is to use the adaptive median filter, which can achieve the effect that requires multiple applications of the median filter with a larger window size. Furthermore, the edges in multiple areas are more orderly, without the occurrence of jagged edges and deformations caused by the previous method.

Image with Pepper-and-salt Noise and Gaussian Noise

The raw image is shown in Figure 7.

By observing the sample image in Figure 7, it is apparent that the image is mainly affected by pepper-and-salt noise and some other types of noise. We can use commonly used spatial filters for denoising, and the results are shown in Figure **??**. Considering the nature of the raw image, we mainly utilized adaptive median filter and adaptive local noise reduction filter to restore the image.

```
img = persistence.load_gray('res/Q6_1_4.tiff')
res_adaptive_median_7 = spatial.adaptive_median_filter(img, 7, regulator=
    regulator.GrayCuttingRegulator)
```

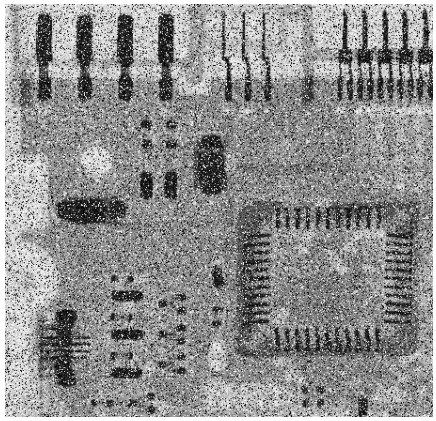
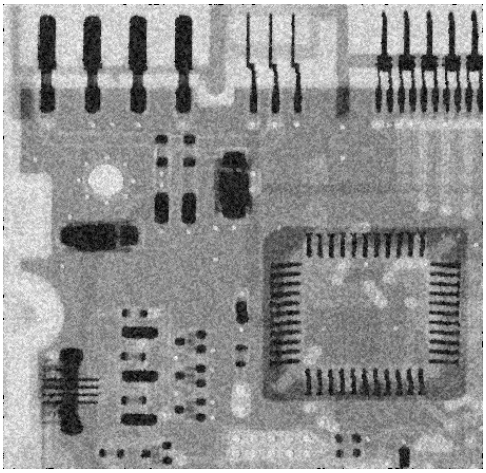


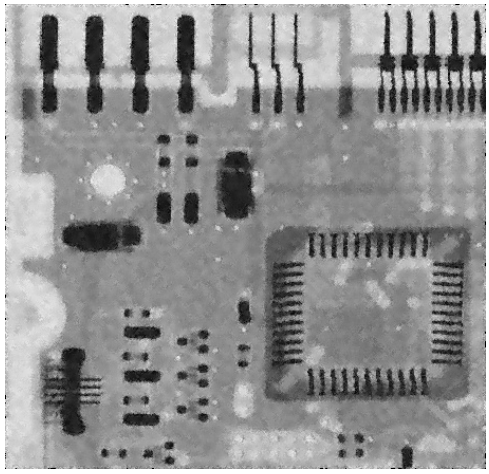
Figure 7. Raw Image Q6_1_4.

```
persistence.save_gray('output/ans/
    Q6_1_4_adaptive_local_noise_reduction_7x7_after_adaptive_median_7.jpg',
    spatial.adaptive_local_noise_reduction_filter(res_adaptive_median_7, (7,
     7), 250, regulator=regulator.GrayCuttingRegulator))
```

Listing 13. Spatial filtering for the raw image with multiple noise.



(a) Adaptive Median Filter, S_{max} = 7



(b) Adaptive Local Noise Reduction (7x7) after the Adaptive Median Filter

Figure 8. The results for the degraded image with multiple noise.

After applying the adaptive median filter to the original image, the result is still very good, but with noticeable noise. Therefore, we applied the adaptive local noise reduction filter on top of it, adaptively filtering out the noise and achieving better results.

In fact, we can also use the alpha-trimmed mean filter mentioned earlier in combination, which has good results for mixed noise containing impulse noise. This will not be discussed here.

3 Atmosphere Turbulence Removing

3.1 Principle

For Linear Position-Invariant Degradations, we can describe the degraded model through convolution and obtain the original image by inverse transformation in the frequency domain.

Atmospheric turbulence is a common image disturbance that causes the image to be blurred in

some way. Some measurement methods have been used to model atmospheric turbulence, and the Fourier transform of its unit impulse response is as follows:

$$H(\mu,\nu) = e^{-k(\mu^2+\nu^2)^{5/6}} \tag{10}$$

where k is a constant that depends on the nature of the turbulence.

Figure 9. Raw Image Q6_2.

Now we have a raw image with an atmosphere turbulence whose k is given as 0.0025 (Figure 9). What we need to do is to remove the turbulence. Given the H, we have many approaches to achieve the target, which would be talked about in the following sections.

3.2 Transfer-function-based Filtering Approaches

Full Inverse Filter

Inverse Filtering is to directly divide the fourier transform of the raw image $G(\mu,\nu)$ with $H(\mu,\nu)$ to obtain the fourier transform of the degraded image $\hat{F}(\mu,\nu)$.

The downside of this approach is that the original image may contain noise, so the result is in fact like this:

$$\hat{F}(\mu,\nu) = F(\mu,\nu) + \frac{N(\mu,\nu)}{H(\mu,\nu)} \tag{11}$$

Meanwhile, due to the direct division, the result will show great uncertainty in the parts where the values of H are close to zero, which can lead to complete errors in the entire image in severe cases - and most of the cases tend to be like this.

We roughly describe the algorithm process as follows (Algorithm 4). Other algorithm frameworks in the following text are similar, with only changes in mathematical forms, which will not be repeated here.

Algorithm 4: Frequency domain processing.

Input: img, H
Output: res

1 $F \leftarrow$ fftshift(fft2(img))
2 (Pad if needed.)
3 Do something with F and H. For example in full inverse filtering, $res \leftarrow$ ifft2(fftshift(F/H))
4 **return** res

Based on this, we provide the Python code for the inverse filter (Code 14).

```
img = persistence.load_gray('res/Q6_2.tiff')
r, c = img.shape
F = frequency.fft2d(img)
k = 0.0025
H = np.exp(-k * np.float_power(np.sum((np.array(np.meshgrid(range(c), range(
    r))) - np.array([np.ones((r, c)) * c // 2, np.ones((r, c)) * r // 2]))
    ** 2, axis=0), 5 / 6))
```

Listing 14. Generating the F and H for the atmosphere turbulence.

In addition, to address the issue of atmosphere turbulence mentioned earlier, we provide the Python code for calculating the H of atmosphere turbulence (Code 15).

```
def full_inverse_filter(F, H, regulator=GrayScalingRegulator):
    return regulator(ifft2d(F / H))
```

Listing 15. Full inverse filter.

So the solution code is as below (Code **??**).

```
persistence.save_gray('output/ans/Q6_2_inverse.jpg', frequency.
    full_inverse_filter(F, H))
```

Listing 16. The solution for the atmosphere turbulence using full inverse filter.

The result of the algorithm is shown in Figure 10.

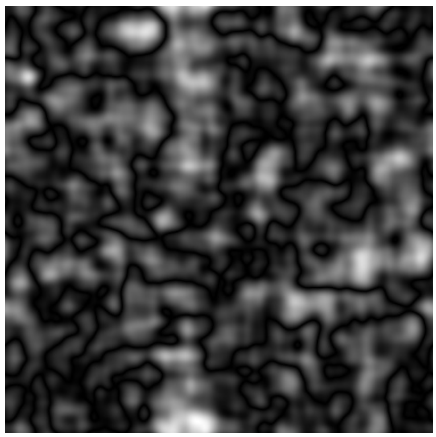


Figure 10. The result for the full inverse filter.

As analyzed earlier, we obtain meaningless results. The main reason for this is that the parts of the H function far from the center are all very close to zero, making it unsuitable as a denominator.

And the complexity analysis is insignificant here, which will be omitted in the following texts.

Radially-limited Inverse Filter

A very straightforward approach to solve the issues of the full inverse filter is to remove the values of H that are far from the center, for instance, by setting them to 1, and then apply the inverse filter to ensure that this part of the values will not cause such significant interference in the results.

The code is shown as below (Code).

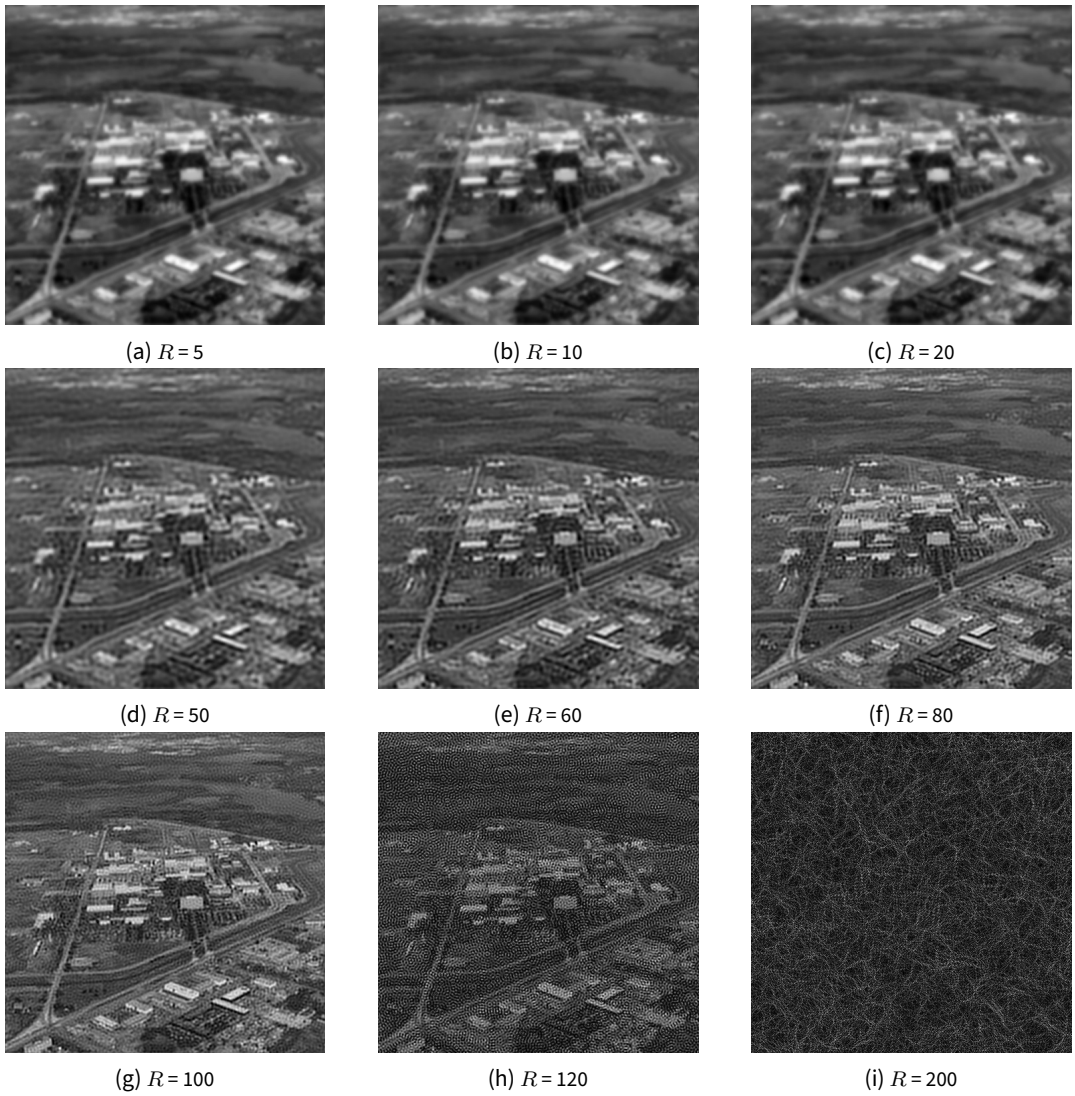
```
# In frequency.py
def radially_limited_inverse_filter(F, H, radius, regulator=
    GrayScalingRegulator):
    H_tmp = np.copy(H)
    r, c = F.shape
```

```
    H_tmp[np.sum((np.array(np.meshgrid(range(c), range(r))) - np.array([np.
    ones((r, c)) * c // 2, np.ones((r, c)) * r // 2])) ** 2, axis=0) >
    radius * radius] = 1
    return full_inverse_filter(F, H_tmp, regulator=regulator)

# In main.py
for R in [5, 10, 20, 50, 60, 80, 100, 120, 200]:
    persistence.save_gray('output/ans/Q6_2_radially_limited_inverse_{}.jpg'.
    format(R), frequency.radially_limited_inverse_filter(F, H, R))
```

Listing 17. Radially-limited inverse filter.

And here is the result by setting different radius values (Figure 11).



(a) $R = 5$ (b) $R = 10$ (c) $R = 20$

(d) $R = 50$ (e) $R = 60$ (f) $R = 80$

(g) $R = 100$ (h) $R = 120$ (i) $R = 200$

Figure 11. The results for the radially-limited inverse filter.

We tried several radii, ignoring all parts of H outside the radius, and the results were consistent with expectations. The image became increasingly clear but was gradually mixed with interference, eventually becoming completely unviewable.

Overall, the best results were obtained with a radius of around 80 to 100. Beyond that, regular patterns resembling convolutions with a $sinc$ function were observed, followed by irregular patterns that completely obscured the useful information.

Minimum Mean Square Error (Wiener) Filter

Furthermore, we can take a target-oriented approach. For example, if we set the goal of minimizing the mean squared error after filtering:

$$e^2 = E\{(f - \hat{f})^2\} \tag{12}$$

From which we can derive a filter called the Wiener filter, with the following mathematical expression:

$$\hat{F}(\mu,\nu) = \left[\frac{1}{H(\mu,\nu)}\frac{|H(\mu,\nu)|^2}{|H(\mu,\nu)|^2 + S_\eta(\mu,\nu)/S_f(\mu,\nu)}\right] G(\mu,\nu) \tag{13}$$

In fact, the term $S_\eta(\mu,\nu)/S_f(\mu,\nu)$ is an constant related to the nature of the noise and the raw image, so we can substitute it by a K:

$$\hat{F}(\mu,\nu) = \left[\frac{1}{H(\mu,\nu)}\frac{|H(\mu,\nu)|^2}{|H(\mu,\nu)|^2 + K}\right] G(\mu,\nu) \tag{14}$$

The Python code for the Wiener Filter is as follows (Code **??**).

```
# In frequency.py
def wiener_filter(F, H, K, regulator=GrayScalingRegulator):
    H2 = H * np.conj(H)
    return regulator(ifft2d(F * H2 / (H * (H2 + K))))

# In main.y
for K in [1e-1, 1e-3, 1e-4, 5e-5, 1e-5, 1e-6]:
    persistence.save_gray('output/ans/Q6_2_wiener_{}.jpg'.format(K),
    frequency.wiener_filter(F, H, K))
```

Listing 18. Wiener filter.

We tried various K and the results are in Figure 12.

Observing the results, we find that the Wiener filter produces visually better results. For the choice of K, a value between 1e-5 and 5e-5 is suitable in this example (although this depends on the specific situation). In fact, the evaluation of results in image restoration can be carried out using corresponding standards, which will be discussed later.

Constrained Least Squares Filter

In Wiener filter, the power spectra of the undegraded image and noise must be known. Although a constant estimate is sometimes useful, it is not always suitable.

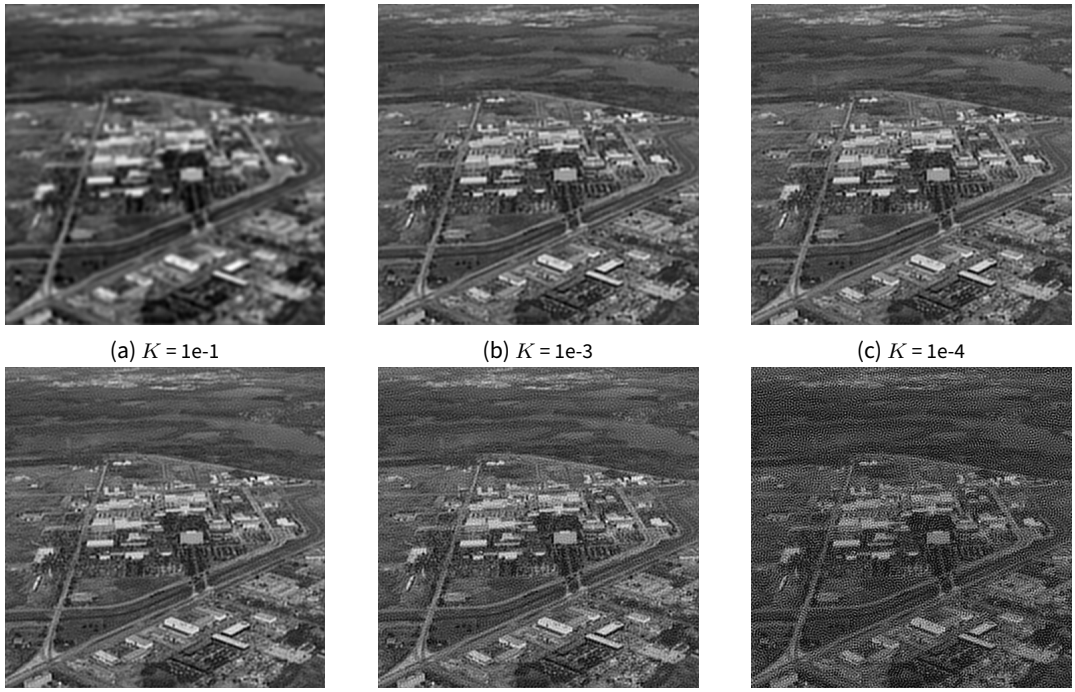
Constrained least squares filtering just requires the mean and variance of the noise. And it is optimum for the particular image processed.

The mathematical description of the constraint obtained by solving the constraints listed by the definition is:

$$\hat{F}(\mu,\nu) = \left[\frac{H^*(\mu,\nu)}{|H(\mu,\nu)|^2 + \gamma|P(\mu,\nu)|^2}\right] G(\mu,\nu) \tag{15}$$

where γ is an adjustable parameter, and $P(\mu,\nu)$ is the fourier transform of the Laplacian operator

$$p(x,y) = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix} \tag{16}$$



(a) K = 1e-1 (b) K = 1e-3 (c) K = 1e-4

(d) K = 5e-5 (e) K = 1e-5 (f) K = 1e-6

Figure 12. The results for the Wiener filter.

Geometric Mean Filter

The Geometric Mean Filter is a more general form, and its mathematical description is as follows.

$$\hat{F}(\mu,\nu) = \left[\frac{H^*(\mu,\nu)}{|H(\mu,\nu)|^2}\right]^{\alpha} \left[\frac{H^*(\mu,\nu)}{|H(\mu,\nu)|^2 + \beta[S_\eta(\mu,\nu)/S_f(\mu,\nu)]}\right]^{1-\alpha} G(\mu,\nu) \tag{17}$$

When α is 0, it reduces to the Wiener filter. When α is 1, it reduces to the inverse filter. When α is $1/2$, it behaves as the geometric mean filter.

It will not be discussed further here.

3.3 Remarks and Measures

Signal-noise Ratio (SNR)

$$SNR = \frac{\sum_{\mu=0}^{M-1}\sum_{\nu=0}^{N-1}|F(\mu,\nu)|^2}{\sum_{\mu=0}^{M-1}\sum_{\nu=0}^{N-1}|N(\mu,\nu)|^2} = \frac{\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}\hat{f}(x,y)^2}{\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}[f(x,y)-\hat{f}(x,y)]^2} = \tag{18}$$

This ratio gives a measure of the level of information bearing signal power to the level of noise power.

Due to time constraints, we will not analyze specific cases or the above results in detail, but only record them.

Mean Square Error (MSE)

The Mean Square Error is defined like

$$MSE = \frac{1}{MN}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}[f(x,y)-\hat{f}(x,y)]^2 \tag{19}$$

Root Mean Square Error (RMSE) and Root Mean Square Signal-to-noise Ratio (RMS-SNR) may be

defined accordingly.

4 Linear Motion Deblurring

4.1 Principle

Modeling

Suppose that an image $f(x, y)$ undergoes planar motion $x_0(t)$ and $y_0(t)$ are the time-varying components of motion in the x- and y-directions, respectively.

The optical imaging process is perfect. T is the duration of the exposure. The blurred image $g(x, y)$ is

$$g(x,y) = \int_0^T f[x - x_0(t), y - y_0(t)]\mathrm{d}t \tag{20}$$

Suppose the motion is at a rate given by $x_0(t) = at/T$ and $y_0(t) = bt/T$, we can derive the H for the degradation as below:

$$H(\mu,\nu) = \frac{T}{\pi(\mu a + \nu b)} \sin[\pi(\mu a + \nu b)] e^{-j\pi(\mu a + \nu b)} \tag{21}$$

Now we can restore the image by employing the approaches we talked about in previous tasks. This time we need to process the raw image in Figure 13.

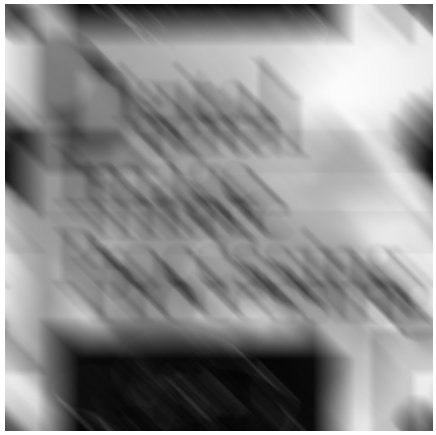


Figure 13. Raw Image Q6_3_1.

The parameters for the motion is given as $a = b = 0.1$ and $T = 1$.

Noise Estimation

In addition, we also need to process two other images that contain significant noise (Figure 14).

In this case, the impact of noise will be amplified. If we process it as the previous image, the result will have no useful information.

Therefore, we need to perform denoising first. We can take a small area (Figure 15) and estimate the noise (Figure 16) before further processing.

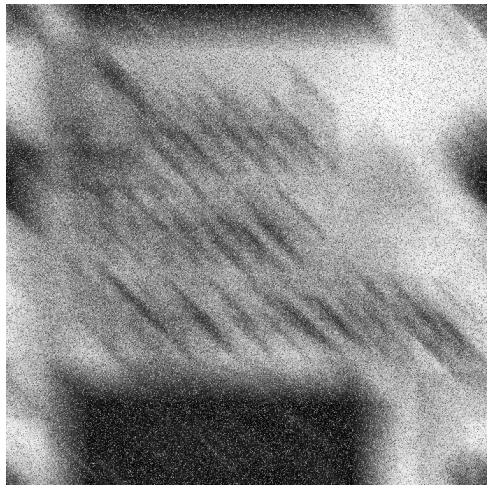
Through observation, we can see that the noise distribution is similar to a normal distribution. We calculate its variance as a reference for filtering: the variance is approximately 347.

4.2 Results

Blurred Image

The algorithm used here is the three frequency domain filtering methods mentioned earlier, so I will not go into detail. Pseudo-code and complexity analysis can be found in the previous text.

(a) Raw Image Q6_3_2.



(b) Raw Image Q6_3_3.

Figure 14. Raw images with noise.

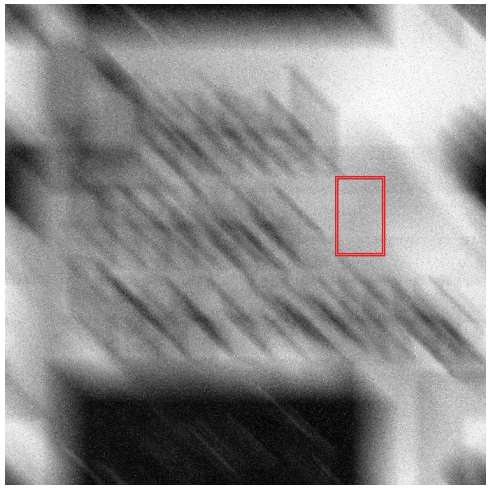


Figure 15. The sample area.

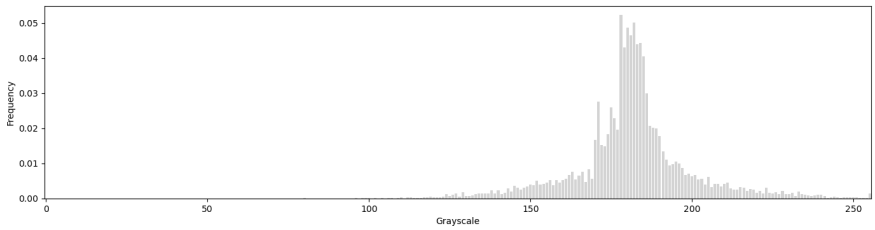


Figure 16. The histogram for the selected area.

For the first image, we applied both inverse filter and Wiener filter with different parameters. The code is shown in Code 19 and the results are shown in Figure 17.

```
def linear_motion_deblurring_FH(img, a, b, T):
    r, c = img.shape
    F = frequency.fft2d(img)
    u, v = np.array(np.meshgrid(np.linspace(1, c, c), np.linspace(1, r, r)))
    A = u * a + v * b
    H = T / (np.pi * A) * np.sin(np.pi * A) * np.exp(-1j * np.pi * A)
```

```
    return F, H

img = persistence.load_gray('res/Q6_3_1.tiff')
F, H = linear_motion_deblurring_FH(img, 0.1, 0.1, 1.0)

persistence.save_gray('output/ans/Q6_3_1_inverse.jpg', frequency.
    full_inverse_filter(F, H))

for K in [1e-3, 1e-5, 1e-9, 1e-12, 1e-15, 1e-16, 1e-17, 1e-19]:
    persistence.save_gray('output/ans/Q6_3_1_wiener_{}.jpg'.format(K),
    frequency.wiener_filter(F, H, K))
```

Listing 19. Deblurring the first image.

(a) Inverse Filter (b) Wiener Filter, K = 1e-3 (c) Wiener Filter, K = 1e-5

(d) Wiener Filter, K = 1e-9 (e) Wiener Filter, K = 1e-12 (f) Wiener Filter, K = 1e-15

(g) Wiener Filter, K = 1e-16 (h) Wiener Filter, K = 1e-17 (i) Wiener Filter, K = 1e-19

Figure 17. The results for the first blurred image.

From the results, we can see that the inverse filter is completely ineffective in this case, while the Wiener filter performs well. The optimal value of K is around $1e-16$, and setting it too small will result in loss of useful information.

Blurred Image Accompanied With Noise

As mentioned before, this image contains noise with a certain variance. If we apply frequency domain filtering directly, the result will be a gray and unclear image. Therefore, we need to denoise the image first. After repeated experiments, the final method flow is described as follows (Algorithm 5).

Algorithm 5: The workflow to process the blurred image with noise.

1. Do Adaptive Median Filtering for 5 times.
2. Do Wiener Filtering.
3. Do Histogram Equalization.
4. Do Geometric Mean Filtering, multiply the result with a constant k and add it to the equalized image.
5. (Do image scaling and cut the too high or too low gray values.)
6. Do Adaptive Local Noise Reduction Filtering.

The code is shown in Code 20.

```python
# In histogram.py
def compute_pdf(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, regulator=GrayCuttingRegulator):
    img = img.astype(np.int32)

    img_pdf = compute_pdf(img)  # PDF (regardless of size)
    img_cdf = np.cumsum(img_pdf)  # Cumulative Sum
    res = 255 * img_cdf[img]

    return regulator(res)

# In main.py
img = persistence.load_gray('res/Q6_3_2.tiff')

for i in range(5):
    print(i)
    img = spatial.adaptive_median_filter(img, 5, regulator=regulator.
    GrayScalingRegulator)

F, H = linear_motion_deblurring_FH(img, 0.1, 0.1, 1.0)
img = frequency.wiener_filter(F, H, 1e-16, regulator=regulator.
    GrayScalingRegulator)

img = histogram.hist_equ(img)

img = regulator.GrayScalingToRegulator(-100, 300)(img + 0.5 * spatial.
    geometric_mean_filter(img, (3, 3), regulator=regulator.
    GrayCuttingRegulator))
img = regulator.GrayCuttingRegulator(img)

img = spatial.adaptive_local_noise_reduction_filter(img, (5, 5), 1500,
    regulator=regulator.GrayScalingToRegulator(-100, 300))
img = regulator.GrayCuttingRegulator(img)

persistence.save_gray('output/ans/Q6_3_2_output.jpg', img)
```

Listing 20. Deblurring the image with noise.

And the result is shown in Figure 18.

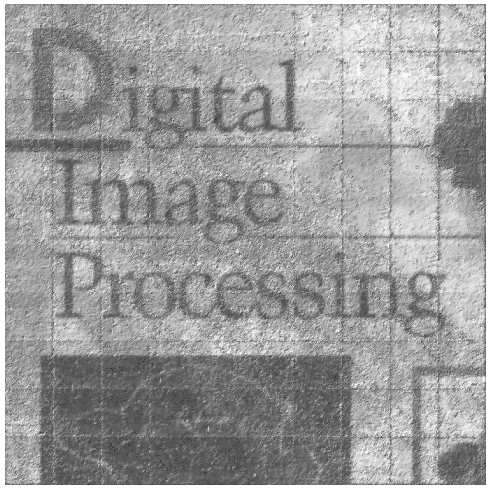


Figure 18. The result for the deblurring process with the second image.

Blurred Image Accompanied With Heavy Noise

The processing method for the third image is basically the same as for the second image, except that the number of iterations for the adaptive median filter is adjusted to 10. Other than that, there are only small adjustments to the parameters. Here is the code (Code 21).

```
img = persistence.load_gray('res/Q6_3_3.tiff')

for i in range(10):
    print(i)
    img = spatial.adaptive_median_filter(img, 5, regulator=regulator.
    GrayScalingRegulator)

F, H = linear_motion_deblurring_FH(img, 0.1, 0.1, 1.0)
img = frequency.wiener_filter(F, H, 1e-16, regulator=regulator.
    GrayScalingRegulator)

img = histogram.hist_equ(img)

img = regulator.GrayScalingToRegulator(-100, 300)(img + 0.7 * spatial.
    geometric_mean_filter(img, (3, 3), regulator=regulator.
    GrayCuttingRegulator))
img = regulator.GrayCuttingRegulator(img)

img = spatial.adaptive_local_noise_reduction_filter(img, (5, 5), 2500,
    regulator=regulator.GrayScalingToRegulator(-200, 350))
img = regulator.GrayCuttingRegulator(img)

persistence.save_gray('output/ans/Q6_3_3_output.jpg', img)
```

Listing 21. Deblurring the image with heavy noise.

The result is given below (Figure 19).

5 Conclusion

In this experiment, we investigated various image restoration techniques, including spatial and frequency domain filters. Our results demonstrate that different types of filters can have a significant impact on the quality of restored images. We found that spatial domain filters such as the median filter and the adaptive median filter were effective in removing salt and pepper noise while preserving image details. On the other hand, frequency domain filters, such as the Wiener filter,

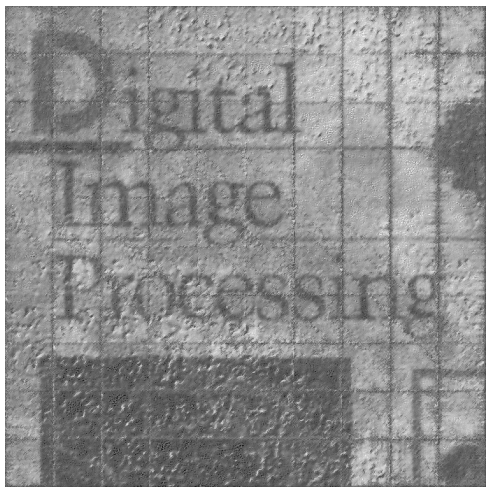


Figure 19. The result for the deblurring process with the third image.

were effective in removing Gaussian noise, but required careful selection of parameters for optimal results.

We also observed that some filters, such as the inverse filter, were not suitable for image restoration in our experiment. In addition, we found that image denoising is an essential preprocessing step for effective image restoration, particularly for images with high levels of noise.

Overall, our findings suggest that a combination of spatial and frequency domain filters, along with appropriate preprocessing steps, can significantly improve the quality of restored images. However, the selection of the most suitable filter depends on the type and level of noise in the image, as well as the specific restoration task at hand.