EE326 Lab Report 5: Image Restoration

11812418 樊青远[1]

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

Abstract

This report provides the introduction, theory and the Python code implementation of several image restoration methods that is widely used in the current industry.

Keywords: DIP

Contents

EE326 DIP (2021)

DOI: 10.1017/pan.xxxx.xx

Corresponding author
Qingyuan Fan

Advised by
Yajun Yu

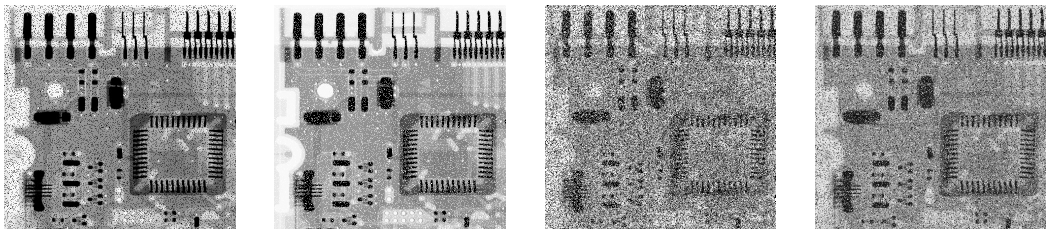
1 Introduction

Image restoration has been widely used in aviation, criminal investigation and other fields. The goal of the image restoration is to remove the noise of an image or restore the image from distortion like blur and atmosphere turbulence. In the first section of the lab, we will implement several spacial domain filter techniques and using adaptive median filter to remove the noise on the image. In the second and third section, we will attempt to use radially limited inverse filter and wiener filter to deal with the degradation models like turbulence and linear motion.

2 Task 1

2.1 Analysis

Remove the noise from the input images Q6_1_1.tif, Q6_1_2.tif, Q6_1_3.tif and Q6_1_4.tif. Explain your observation and the method used to each of the images, and why such methods are used.



(a) Q6_1_1 (b) Q6_1_2 (c) Q6_1_3 (d) Q6_1_4

Figure 1. Figure provided by Task1

From the figure provided by the question, we can intuitively see that the noise on the image are mainly salt and pepper noise. In the lab before, we used median filter to remove such noise.

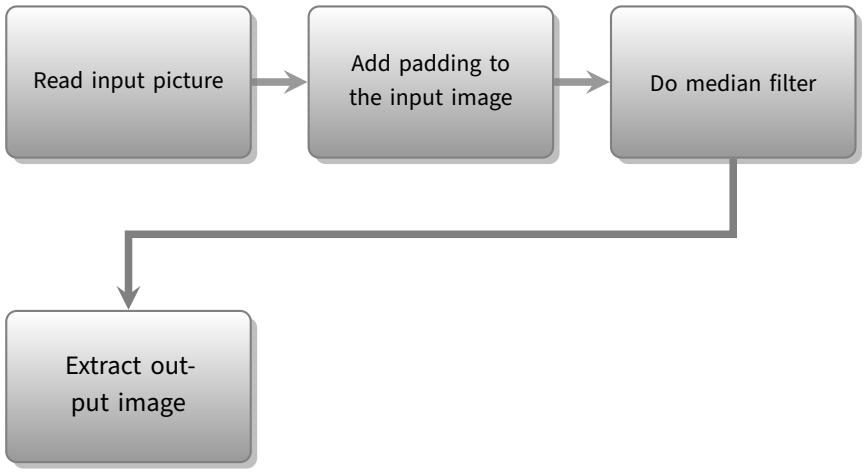


Figure 2. Workflow of Median Filter

adaptive median filter

However, the conventional filter with still filter window on image with excessive noise intensity is not very satisfactory. In order to solve this problem, we apply the filter whose window size can be changed according to the noise of the image called adaptive filter to the input image.

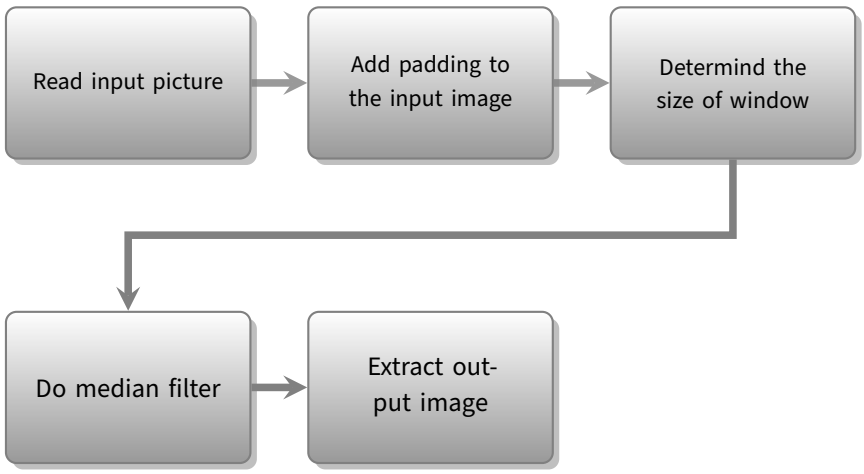


Figure 3. Workflow of Adaptive Median Filter

In the adaptive median filter, we firstly extract the following value from the image:
S_{xy}: The domain we apply the filter
z_{min}: The minimum intensity value in S_{xy}
z_{max}: The maximum intensity value in S_{xy}
z_{med}: Median intensity value in S_{xy}
z_{xy}: Intensity value at coordinates (x, y)
S_{max}: Maximum allowed size of S_{xy}

Then, we split the adaptive median filter into two stages listed as follows:

Stage A:
$A1 = z_{med} - z_{min}$; $A2 = z_{med} - z_{max}$
if $A1 > 0$ and $A2 < 0$, go to stage B
else increase the size of window
if window size <= S_{max}, repeat stage A
else output z_{med}

Stage B:
$B1 = z_{xy} - z_{min}$; $A2 = z_{xy} - z_{max}$
if $B1 > 0$ and $B2 < 0$, return z_{xy}
else output z_{med}

There are mainly three purpose to implement such filter:
1. Remove the salt-and-pepper (impulse) noise.
2. Provide smoothing of other noise that may not be impulsive.
3. Reduce distortion like excessive thinning or thickening of object boundaries.

```
@njit
def padding(img, pad):
    padded_img = np.zeros((img.shape[0] + 2 * pad, img.shape[1] + 2 *
    ↪  pad))
    padded_img[pad:-pad, pad:-pad] = img
    return padded_img

```

```python
@njit
def stage_A(mat, x, y, s, sMax):
    window = mat[x - (s // 2):x + (s // 2) + 1, y - (s // 2):y + (s // 2) + 1]
    Zmin = np.min(window)
    Zmed = np.median(window)
    Zmax = np.max(window)

    A1 = Zmed - Zmin
    A2 = Zmed - Zmax

    if A1 > 0 and A2 < 0:
        return stage_B(window, Zmin, Zmed, Zmax)
    else:
        s += 2
        if s <= sMax:
            return stage_A(mat, x, y, s, sMax)
        else:
            return Zmed


@njit
def stage_B(window, Zmin, Zmed, Zmax):
    h, w = window.shape

    Zxy = window[h // 2, w // 2]
    B1 = Zxy - Zmin
    B2 = Zxy - Zmax

    if B1 > 0 and B2 < 0:
        return Zxy
    else:
        return Zmed

@njit(parallel=True)
def AdaptiveMedianFilter(img, s=3, sMax=21):
    if len(img.shape) == 3:
        raise Exception("Single channel image only")

    row, col = img.shape
    a = sMax // 2
    padded_img = padding(img, a)

    f_img = np.zeros(padded_img.shape)

    for i in prange(a, row + a + 1):
        for j in prange(a, col + a + 1):
            value = stage_A(padded_img, i, j, s, sMax)
            f_img[i, j] = value
```

```
55
56      return f_img[a:-a, a:-a]
```

Contraharmonic Median Filter

The contraharmonic filter is an advanced version of mean filter and it is well suited for reducing or virtually eliminating the effects of salt-and-pepper noise. After extract the window from the image, the filter divide the sum of different orders of the image data to return the result, whose formula can be written as follows;

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

Specifically, when $Q > 0$, the filter eliminates pepper noise. When $Q < 0$, the filter eliminates salt noise.

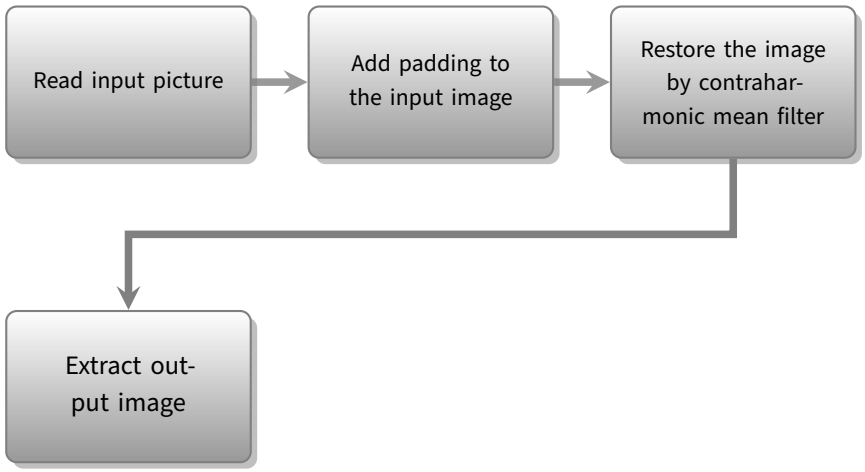


Figure 4. Workflow of Contraharmonic Median Filter

```
1   @njit
2   def contrahamonic_filter(i, j, image_after_padding, n_size, Q):
3       # im_arr = image_after_padding[(i - int((n_size-1)/2)):(i +
        ↪  int((n_size-1)/2)), (j - int((n_size-1)/2)):(j +
        ↪  int((n_size+1)/2))].flatten()
4       im_arr = image_after_padding[(i):(i + int((n_size))), (j):(j +
        ↪  int((n_size)))].flatten()
5       im_arr_Q = np.power(im_arr, Q)
6       im_arr_QP1 = np.power(im_arr, Q + 1)
7       out = 0
8       # print(np.sum(im_arr_Q))
9       if (np.sum(im_arr_QP1) != 0 and np.sum(im_arr_Q) != 0):
10          out = np.sum(im_arr_QP1) / np.sum(im_arr_Q)
11      # if (out == inf )
12      return out
13
14  @njit(parallel=True)
```

```
def contrahamonic_mean_filter(input_img, n_size, Q):
    row, col = input_img.shape
    input_img_padding = padding_jit(input_img, int((n_size - 1) / 2))
    output_img = np.zeros((row, col))

    for i in prange(row):
        for j in prange(col):
            output_img[i][j] = contrahamonic_filter(i, j,
            ↪  input_img_padding, n_size, Q)
    output_img = np.where(np.isnan(output_img), 0, output_img)
    return output_img
```

Alpha Trimmed Median Filter

Besides the pepper and salt noise, the image also contains Gaussian noise. The Alpha-trimmed mean filter removes the $d/2$ highest and $d/2$ lowest intensity values. The average of these remaining $mn - d$ values is the output of every pixel, which is also be illustrated in the formula below:

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

When $d = 0$, the alpha-trimmed filter will be reduces to the arithmetic mean filter discussed in the previous section. If we choose $d = mn - d$, the filter becomes a median filter. For other values of d, the alpha-trimmed filter is useful in situations involving multiple types of noise, such as a combination of salt-and-pepper and Gaussian noise.

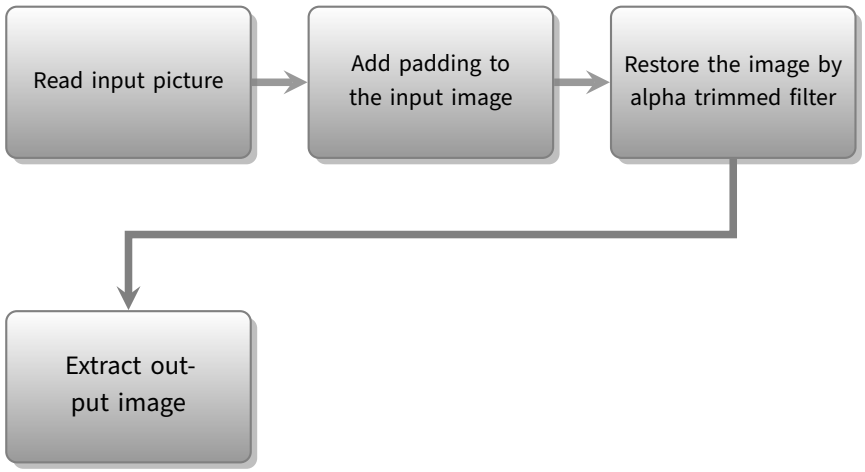


Figure 5. Workflow of Alpha Trimmed Median Filter

```
@njit
def alpha_trimmed_filter(i, j, image_after_padding, n_size, d):
    # im_arr = image_after_padding[(i - int((n_size-1)/2)):(i +
    ↪  int((n_size-1)/2)), (j - int((n_size-1)/2)):(j +
    ↪  int((n_size+1)/2))].flatten()
    im_arr = image_after_padding[(i):(i + int((n_size))), (j):(j +
    ↪  int((n_size)))].flatten()
```

```
    im_arr = im_arr[np.floor(d / 2):np.floor(n_size * n_size - d / 2)]
    return np.sum(im_arr / (n_size ** 2 - d))

@njit(parallel=True)
def alpha_trimmed_mean_filter(input_img, n_size, d):
    row, col = input_img.shape
    # print(input_img.shape)
    # input_img_padding = np.pad(input_img, n_size - 2, pad_with,
    ↪  padder=0)
    input_img_padding = padding_jit(input_img, int((n_size - 1) / 2))
    output_img = np.zeros((row, col))
    # print(input_img_padding.shape)

    for i in prange(row):
        for j in prange(col):
            output_img[i][j] = alpha_trimmed_filter(i, j,
            ↪  input_img_padding, n_size, d)

    # output_img = np.reshape(par_output, input_img.shape)
    return output_img
```

2.2 Result

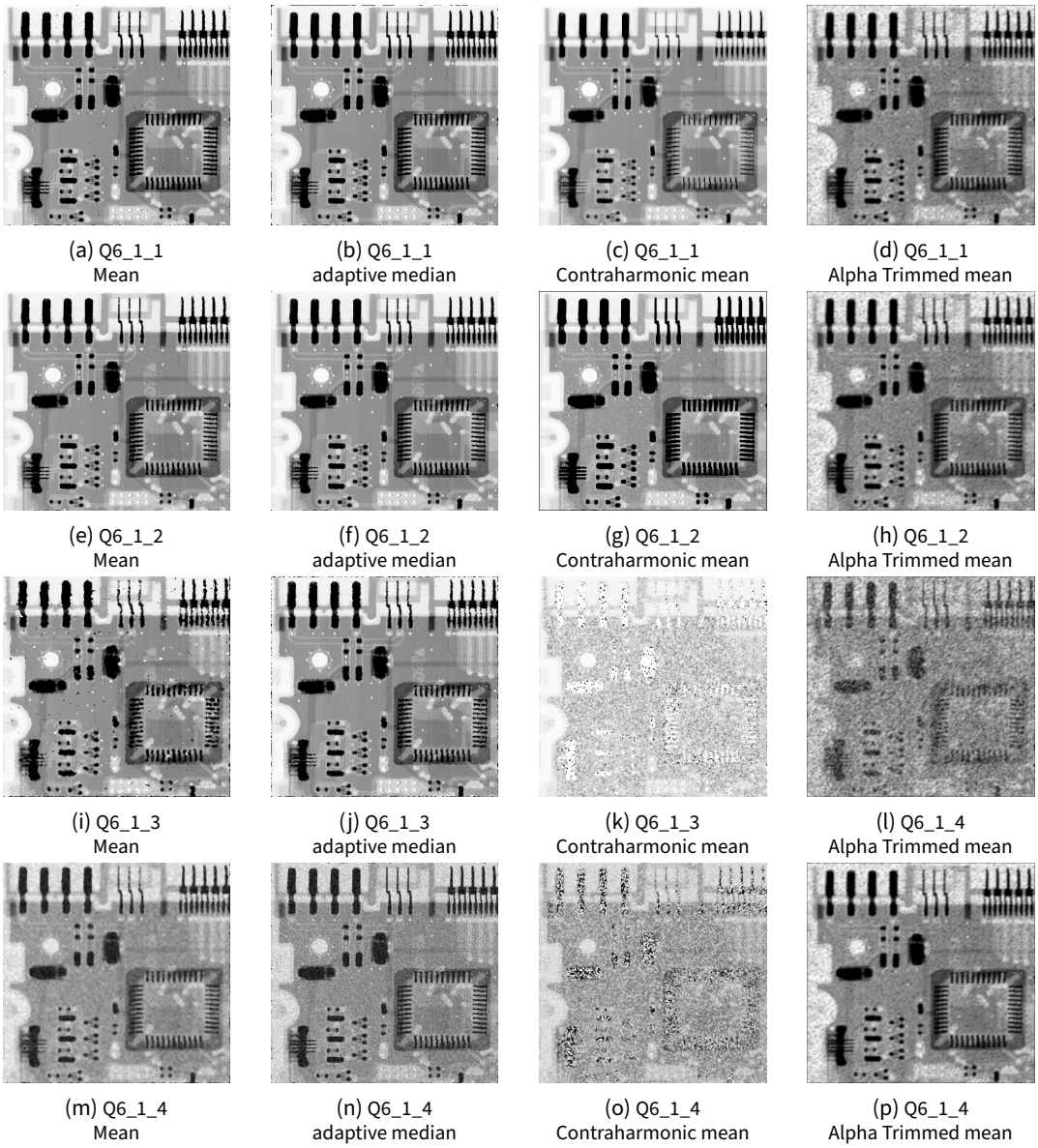


Figure 6. Result different filtering algorithm methods mentioned above.

From the result, we could find that the adaptive median filter has a certain noise reduction effect for each picture (noise type). Except Figure Q6_1_4, it is also the best one of the four noise reduction methods. The standard mean filter also works well in the first two figures as it only has salt or pepper noise, due to the complex noise environment of the latter two pictures, the effect of this filter will be slightly worse.

On the contrary, as the contraharmonic median filter can remove either pepper or salt noise from the image. Therefore, its use is more limited, and it can only get better results in the first two pictures. In the last two pictures, the composition of noise becomes more complicated, so this filter will bring negative effects to the picture instead.

3 Task 2: Remove Atmosphere Turbulence

3.1 Analysis

Image Q6_2.tif was degraded from an original image due to the atmosphere turbulence given on slide 65 with $k = 0.0025$. Restore the original image from the input Q6_2.tif by using full inverse filtering, radially limited inverse filtering and Wiener filtering. Discuss how the parameters, if any, are determined, and the different effects by using the different algorithms.

Figure 7. Figure provided by Task2

The degradation model aimed to restore the image from atmosphere turbulence was proposed by Hufnagel and Stanley Hufnagel and Stanley 1964. The model can be expressed by the formula below:

$$H(u,v) = e^{-k(u^2+v^2)^{5/6}} \tag{3}$$

In formula 3, k is a constant that depends on the nature of the atmosphere turbulence. The value of k is positively correlated with the severity of atmospheric turbulence. In this experiment, we tried k between 10e-5 and 10e-2.

The input image is processed by the inverse filter deployed on the frequency domain. The transformation of the image is listed in formula 4

$$\widehat{F}(u,v) = [\frac{1}{H(u,v)}\frac{|H(u,v)|^2}{|H(u,v)|^2 + S_\eta(u,v)/S_f(u,v)}]G(u,v) \tag{4}$$

As we can't determine the signal noise ration of the image, $S_\eta(u,v)/S_f(u,v)$ directly, so we just substitute it by K and trying different K in the experiment below to determine the closest SNR value. After the substitution, the formula 4 becomes:

$$\widehat{F}(u,v) = [\frac{1}{H(u,v)}\frac{|H(u,v)|^2}{|H(u,v)|^2 + K}]G(u,v) \tag{5}$$

Where $H(u,v)$ is the degradation function mentioned above.

3.2 Workflow

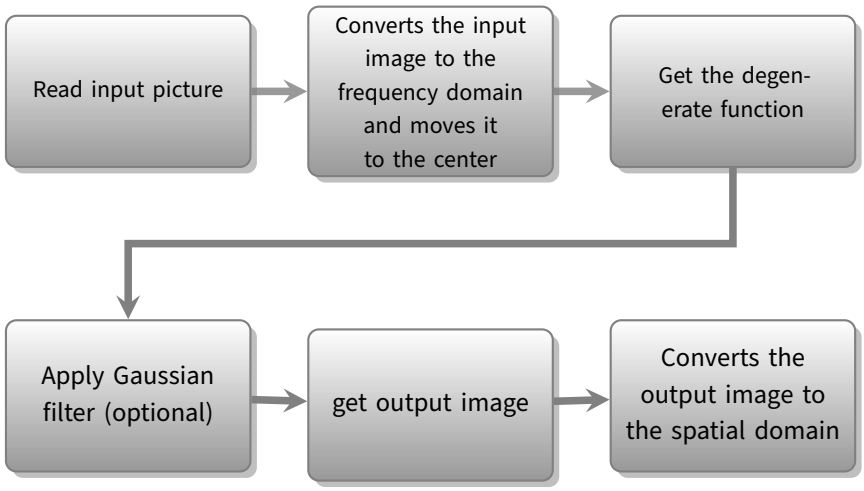


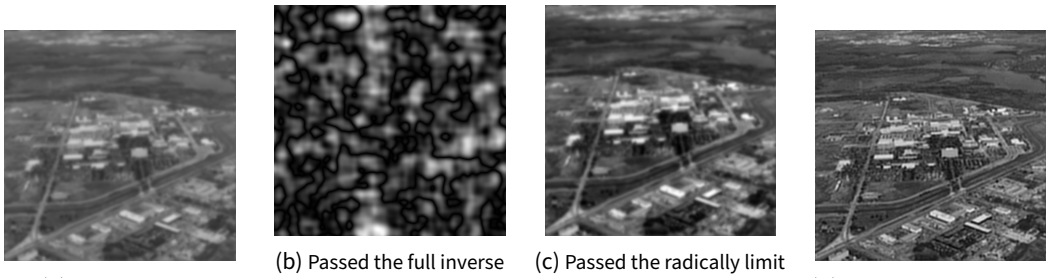
Figure 8. Workflow of removing the Atmosphere Turbulence from image.

3.3 Python Code

```
def fft2d(input_img):
    # row, col = input_img.shape
    # input_img = FFT_zero_padding(input_img)
    # input_img = multiply_center(input_img)
    input_img = np.fft.fft2(input_img)
    input_img = np.fft.fftshift(input_img)

    return input_img

def ifft2d(input_img):
    output_img = np.fft.ifft2(np.fft.ifftshift(input_img))
    # output_img = transform_centering(output_img)
    return np.abs(output_img)

@njit
    def normalize(input_img):
        return (input_img - np.min(input_img)) / (np.max(input_img) -
        ↪  np.min(input_img))

def remove_air_turbulence(input_img, mode, K, gaussian, sigma):
    input_img_after_fft = fft2d(input_img)
    k = 0.0025

    row, col = input_img_after_fft.shape
    u, v = np.meshgrid(np.linspace(0, row - 1, row), np.linspace(0, col -
    ↪  1, col))
    u = u - row / 2
    v = v - col / 2
    d = np.power(u, 2) + np.power(v, 2)
```

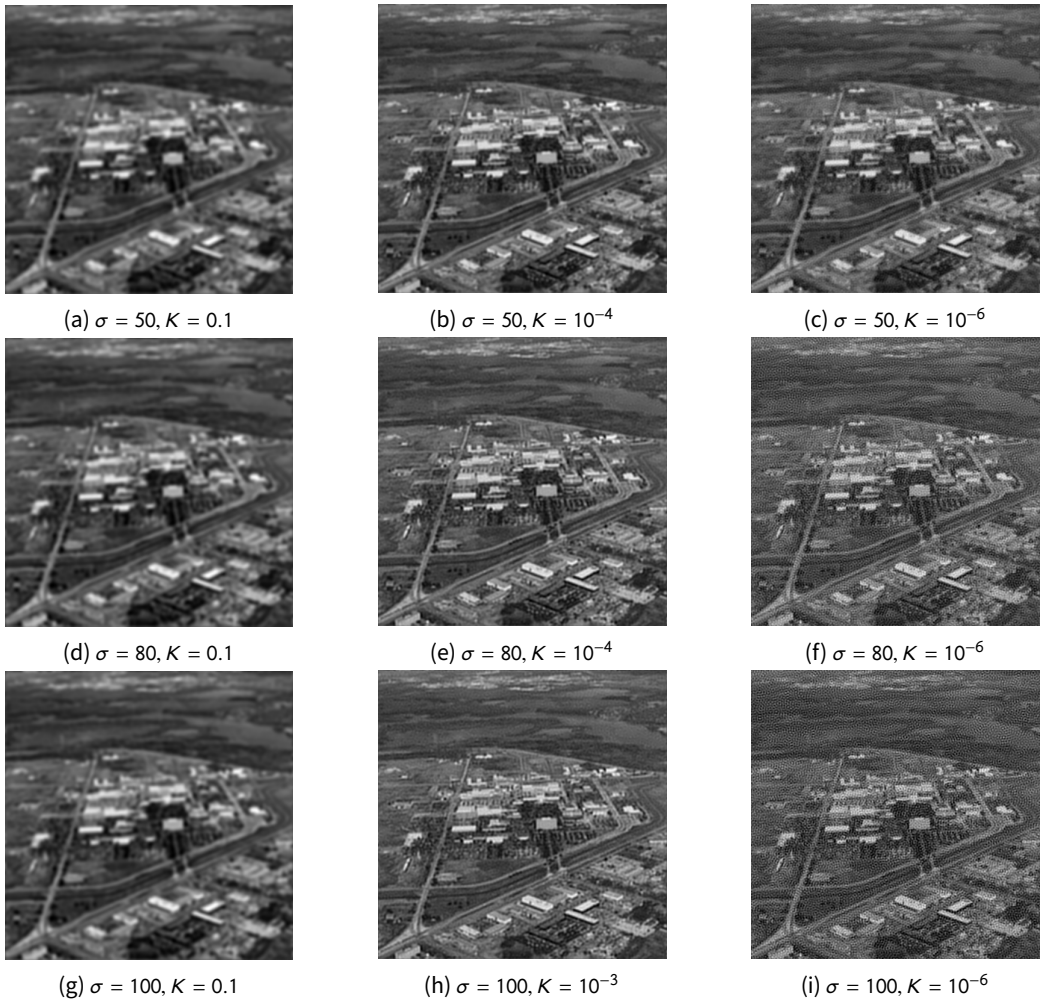
```
    H = np.exp(-(k * (np.power(d, 5 / 6))))

    if mode == 1:
        # full inverse filter
        output_img = input_img_after_fft / H

    if mode == 2:
        output_img = input_img_after_fft
        # Limit inverse
        for i in range(1, row + 1):
            for j in range(1, col + 1):
                if ((i - row / 2) ** 2 + (j - col / 2) ** 2) < 70:
                    output_img[i - 1, j - 1] = input_img_after_fft[i - 1,
                     ↪  j - 1] / H[i - 1, j - 1]

    if mode == 3:
        # Wiener filter
        buf = np.power(H, 2)
        k2 = K
        output_img = input_img_after_fft * buf / (H * (buf + k2))
        if gaussian == 1:
            output_img = gaussian_LP_freq_filter(output_img, sigma)

    output_img = normalize(ifft2d(output_img)) * 255
    return output_img

```

3.4 Result



(a) original image (b) Passed the full inverse filter (c) Passed the radically limit inverse filter (d) Passed the wiener filter

Figure 9. Figure provided by Task1



(a) $\sigma = 50, K = 0.1$ (b) $\sigma = 50, K = 10^{-4}$ (c) $\sigma = 50, K = 10^{-6}$

(d) $\sigma = 80, K = 0.1$ (e) $\sigma = 80, K = 10^{-4}$ (f) $\sigma = 80, K = 10^{-6}$

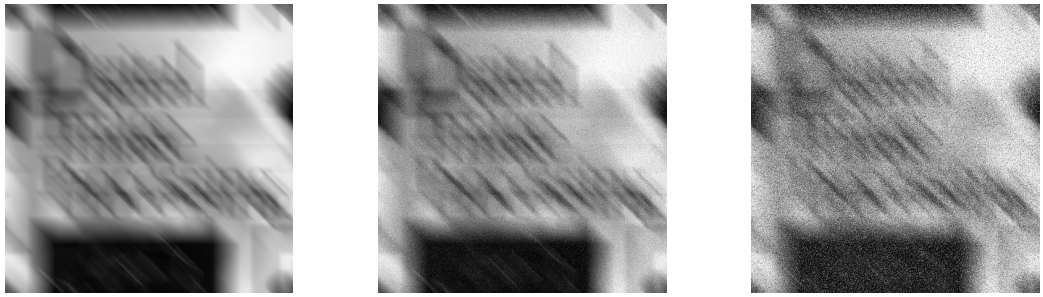
(g) $\sigma = 100, K = 0.1$ (h) $\sigma = 100, K = 10^{-3}$ (i) $\sigma = 100, K = 10^{-6}$

Figure 10. Image fitered by wiener filter with different K and σ

From the result, we could intuitively see that the result provided by the fully inverse filter is completely unusable, as the filter itself doesn't cut off any high frequency component in the frequency domain. The radically limited inverse filter can restore the picture relatively better, as it mainly preserve the frequency component in the low-frequency area, where the effective signal mainly concentrated. We've also tried to use the wiener filter and Gaussian together in the last image, which has a sharper result compared to the image restores by radially limit inverse filter. At the same time, we also need to pay attention that filtering can achieve the best effect only when the value of K and the value of SNR match.

4 Task3: Motion Deblurring

Restore the original images from the inputs Q6_3_1.tif, Q6_3_2.tif and Q6_3_3. Explain your observation and the method used.



(a) Q6_3_1 (b) Q6_3_2 (c) Q6_3_3

Figure 11. Figures with motion blur provided by the problem

4.1 Analysis

According to the textbook, the transformation of the motion blur can also be regarded as a degradation function:

$$\begin{aligned} H(u,v) &= \int_0^T e^{-j2\pi[ux_0(t)+vy_0(t)]}dt \\ * \quad &= \int_0^T e^{-j2\pi[ua+vb]t/T}dt \\ &= \frac{T}{\pi(ua+vb)}\sin[\pi(ua+vb)]e^{-j\pi(ua+vb)} \end{aligned} \tag{6}$$

Based on the results of the previous task, we directly apply the Wiener filter to this question.

4.2 Workflow

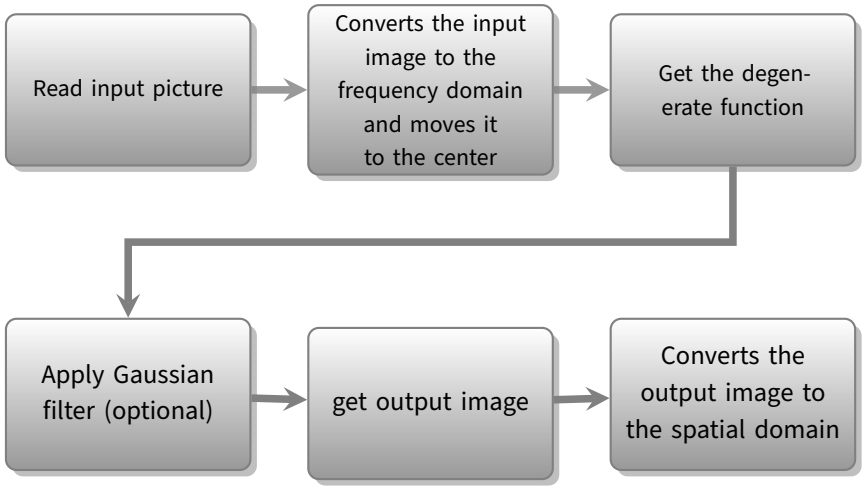


Figure 12. Workflow of removing the motion blur from image.

4.3 Python Code

```
1  def fft2d(input_img):
2      # row, col = input_img.shape
3      # input_img = FFT_zero_padding(input_img)
4      # input_img = multiply_center(input_img)
5      input_img = np.fft.fft2(input_img)
```

```
    input_img = np.fft.fftshift(input_img)

    return input_img

def ifft2d_real(input_img):
    output_img = np.fft.ifft2(np.fft.ifftshift(input_img))
    # output_img = transform_centering(output_img)
    return np.real(output_img)

def normalize_pst(input_img):
    # find percentage
    p95 = np.percentile(input_img, 95)
    p5 = np.percentile(input_img, 5)
    input_img = np.clip(input_img, p5, p95)
    return (input_img - np.min(input_img)) / (np.max(input_img) -
    ↪  np.min(input_img))

def restore_planar_motion(input_img, a, b, T, k, LP, sigma):
    input_img_after_fft = fft2d(input_img)

    row, col = input_img.shape
    u, v = np.meshgrid(np.linspace(1, row, row), np.linspace(1, col, col))
    A = a * u + b * v

    # print(A.shape)
    T_arr = np.ones([row, col]) * T
    H = (T_arr / (np.pi * A)) * np.sin(A * np.pi) * np.exp(-1j * np.pi *
    ↪  A)
    # condition = np.array((A == 0))
    # H = H + condition
    buf = H * np.conj(H)
    # print(buf + k)
    F = input_img_after_fft * buf / (H * (buf + k))

    # gaussian
    if LP == 1:
        F = F * generate_gaussian(row, col, sigma)

    output_img = normalize_pst(ifft2d_real(F)) * 255

    return output_img
```

4.4 Result for Q6_3_1

The background noise of this picture is quite small, or even negligible, so we can directly use Wiener filtering to filter it.

(a) $K = 1e-1$ (b) $K = 1e-2$ (c) $K = 1e-3$

(d) $K = 1e-4$ (e) $K = 1e-5$ (f) $K = 1e-6$

(g) $K = 1e-7$ (h) $K = 1e-8$ (i) $K = 1e-9$

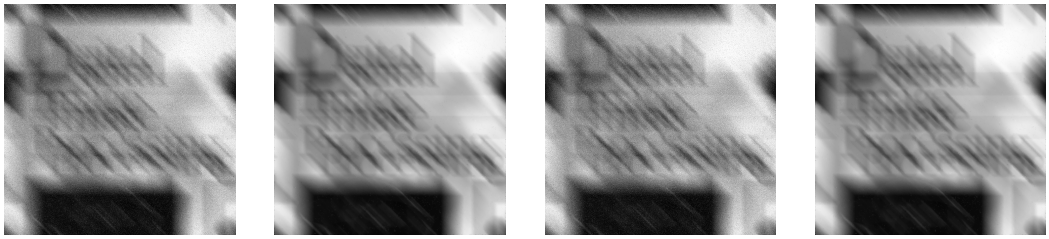
(j) $K = 1e-12$ (k) $K = 1e-15$ (l) $K = 1e-20$

Figure 13. Fitered by wiener filter

The result shows that the k between $1e-10$ and $1e-15$ can get acceptable result. However, excessive k may bring negative effects shown in figure 13l.

4.5 Result for Q6_3_2 and Q6_3_3

Comparing to Q_3_1, Q6_3_2 and Q6_3_3 has many noise on the image. Before applying the wiener filter, we may need to eliminate the noise of the image. Based on task1, we adapt the adaptive median filter (5 times for Q6_3_2, 10 times for Q6_3_3) on the image for multiple times to get an clean result. We also use the function np.percentage to drop the pixel with abnormal high or low brightness for further noise reduction.



(a) Q6_3_2 (b) Q6_3_2 after denoised (c) Q6_3_3 (d) Q6_3_3 after denoised

Figure 14. original image and image after adaptive median filter

This time, we add an extra Gaussian filter after the wiener filter to remove the potential noise that still contained in the image after noise reduction.

(a) $\sigma = 10, K = 2.5e-5$ (b) $\sigma = 50, K = 2.5e-5$ (c) $\sigma = 100, K = 2.5e-5$

(d) $\sigma = 10, K = 2.5e-8$ (e) $\sigma = 50, K = 2.5e-8$ (f) $\sigma = 100, K = 2.5e-8$

(g) $\sigma = 10, K = 2.5e-11$ (h) $\sigma = 50, K = 2.5e-11$ (i) $\sigma = 100, K = 2.5e-11$

(j) $\sigma = 10, K = 2.5e-15$ (k) $\sigma = 50, K = 2.5e-15$ (l) $\sigma = 100, K = 2.5e-15$

(m) $\sigma = 10, K = 2.5e-18$ (n) $\sigma = 50, K = 2.5e-18$ (o) $\sigma = 100, K = 2.5e-18$

Figure 15. Restoration result of Q6_3_2

(a) $\sigma = 10, K = 2.5e-5$ (b) $\sigma = 50, K = 2.5e-5$ (c) $\sigma = 100, K = 2.5e-5$

(d) $\sigma = 10, K = 2.5e-8$ (e) $\sigma = 50, K = 2.5e-8$ (f) $\sigma = 100, K = 2.5e-8$

(g) $\sigma = 10, K = 2.5e-11$ (h) $\sigma = 50, K = 2.5e-11$ (i) $\sigma = 100, K = 2.5e-11$

(j) $\sigma = 10, K = 2.5e-15$ (k) $\sigma = 50, K = 2.5e-15$ (l) $\sigma = 100, K = 2.5e-15$

(m) $\sigma = 10, K = 2.5e-18$ (n) $\sigma = 50, K = 2.5e-18$ (o) $\sigma = 100, K = 2.5e-18$

Figure 16. Restoration result of Q6_3_3

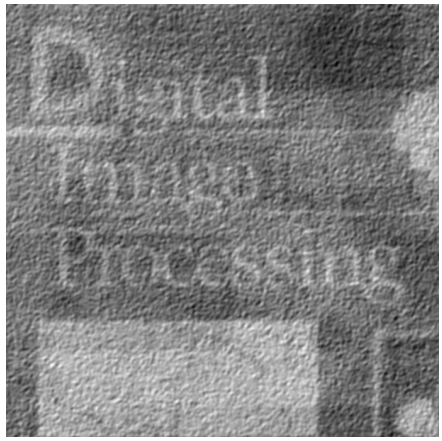
From the two sets of results, we can conclude that with the rise of the σ, the image become

sharper, but sharper image will contains more high frequency noise like the ”mesh” pattern, which has negative effect to the viewer. Conclusively, we could regard $k = 2.5e - 11$ as a proper choice to restore such type of image.

However, we can also find that as the noise function is highly random and there is no inverse function of it, the performance of recovering the image from high intensity noise is still unacceptable.

4.6 Comparison with the image of my friends

(a) My Result with $k = 2.5e - 11, \sigma = 100$, Q6_3_2



(b) Yuan Tong's Result with $k = 2.5e - 5, \sigma = 50$, Q6_3_2

Figure 17. Comparison between my result and Yuan Tong's result.

Comparing the result of mine and my friend Yuan Tong's. I found that the Yuan Tong's result has smoother surface, while my result has higher contrast. Conclusively, when we need to preserve image or plane from the blurred image, Yuan's method could be better, when we need to extract the text from the image, my filter could be better.

5 Conclusion

In this lab, we practiced a variety of noise reduction models both on spacial domain and frequency domain, we also use the concept of degradation model to solve the problem in the real world. We also realized that reducing high-frequency noise and making the image clear are incompatible, so we need to find a balance between them.

Besides, some profiling method on the python program itself including numba JIT (Just-In-Time) have been apply to the spacial filter in task 1, which leads to 99% reduction in program execution time after compilation.

Supplementary Material

The source code of this lab can be retrieved from https://github.com/sparkcyf/SUSTech_EE326_Digital_image_Processing/ or https://mirrors.sustech.edu.cn/git/fanqy2018/SUSTech_EE326_Digital_image_Processing

References

Hufnagel, R., and N. Stanley. 1964. “Modulation transfer function associated with image transmission through turbulent media.” *JOSA* 54 (1): 52–61.

6 Appendix: Complete Code

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


def FFT_zero_padding(input_img):
    row, col = input_img.shape
    output_image = np.zeros([2 * row, 2 * col])
    output_image[0:row, 0:col] = input_img
    return output_image


def FFT_extract_padding(input_img):
    row, col = input_img.shape
    output_image = input_img[int(row / 2):row, int(col / 2):col]
    return output_image


def multiply_center(input_img):
    row, col = input_img.shape
    I, J = np.ogrid[:row, :col]
    mask = np.full((row, col), -1)
    mask[(I + J) % 2 == 0] = 1
    return mask * input_img


def transform_centering(input_img):
    row, col = input_img.shape
    return input_img[0:int(row / 2), 0:int(col / 2)]


def DFT_kernel(input_image_padded, kernel):
    sz = (input_image_padded.shape[0] - kernel.shape[0],
     ↪  input_image_padded.shape[1] - kernel.shape[1])
    DFT_kernel_1 = np.pad(kernel, (((sz[0] + 1) // 2, sz[0] // 2), ((sz[1]
     ↪  + 1) // 2, sz[1] // 2)), 'constant')
    DFT_kernel_1 = multiply_center(DFT_kernel_1)
    DFT_kernel_1_fft = np.fft.fft2(DFT_kernel_1)
    return DFT_kernel_1_fft


def generate_gaussian(a, b, sigma):
    x, y = np.meshgrid(np.linspace(0, a - 1, a), np.linspace(0, b - 1, b))
    x = x - a / 2
```

```
46      y = y - b / 2
47      d = x * x + y * y
48      g = np.exp(-(d / (2.0 * sigma ** 2)))
49      # g = g/np.sum(g)
50      return g
51
52
53  def generate_butterworth(row, col, n, sigma, cr, cc):
54      x, y = np.meshgrid(np.linspace(0, col - 1, col), np.linspace(0, row -
        ↪ 1, row))
55      x = x - cr
56      y = y - cc
57      d = np.sqrt(x * x + y * y)
58      h = 1 / ((1 + (d / sigma)) ** (2 * n))
59      return h
60
61
62  @njit
63  def normalize(input_img):
64      return (input_img - np.min(input_img)) / (np.max(input_img) -
        ↪ np.min(input_img))
65
66
67  @njit
68  def padding_jit(img, pad):
69      padded_img = np.zeros((img.shape[0] + 2 * pad, img.shape[1] + 2 *
        ↪ pad))
70      padded_img[pad:-pad, pad:-pad] = img
71      return padded_img
72
73
74  def normalize_pst(input_img):
75      # find percentage
76      p95 = np.percentile(input_img, 95)
77      p5 = np.percentile(input_img, 5)
78      input_img = np.clip(input_img, p5, p95)
79      return (input_img - np.min(input_img)) / (np.max(input_img) -
        ↪ np.min(input_img))
80
81
82  def pad_with(vector, pad_width, iaxis, kwargs):
83      pad_value = kwargs.get('padder', 0)
84      vector[:pad_width[0]] = pad_value
85      vector[-pad_width[1]:] = pad_value
86
87
88  def arithmetic_filter(i, j, image_after_padding, n_size):
```

```
89      # im_arr = image_after_padding[(i - int((n_size-1)/2)):(i +
        ↪  int((n_size-1)/2)), (j - int((n_size-1)/2)):(j +
        ↪  int((n_size+1)/2))].flatten()
90      im_arr = image_after_padding[(i):(i + int((n_size))), (j):(j +
        ↪  int((n_size)))].flatten()
91
92      return np.sum(im_arr / (n_size ** 2))
93
94
95  @njit
96  def contrahamonic_filter(i, j, image_after_padding, n_size, Q):
97      # im_arr = image_after_padding[(i - int((n_size-1)/2)):(i +
        ↪  int((n_size-1)/2)), (j - int((n_size-1)/2)):(j +
        ↪  int((n_size+1)/2))].flatten()
98      im_arr = image_after_padding[(i):(i + int((n_size))), (j):(j +
        ↪  int((n_size)))].flatten()
99      im_arr_Q = np.power(im_arr, Q)
100     im_arr_QP1 = np.power(im_arr, Q + 1)
101     out = 0
102     # print(np.sum(im_arr_Q))
103     if (np.sum(im_arr_QP1) != 0 and np.sum(im_arr_Q) != 0):
104         out = np.sum(im_arr_QP1) / np.sum(im_arr_Q)
105     # if (out == inf )
106     return out
107
108
109 @njit
110 def alpha_trimmed_filter(i, j, image_after_padding, n_size, d):
111     # im_arr = image_after_padding[(i - int((n_size-1)/2)):(i +
        ↪  int((n_size-1)/2)), (j - int((n_size-1)/2)):(j +
        ↪  int((n_size+1)/2))].flatten()
112     im_arr = image_after_padding[(i):(i + int((n_size))), (j):(j +
        ↪  int((n_size)))].flatten()
113     im_arr = im_arr[np.floor(d / 2):np.floor(n_size * n_size - d / 2)]
114     return np.sum(im_arr / (n_size ** 2 - d))
115
116
117 def geometric_filter(i, j, image_after_padding, n_size):
118     # im_arr = image_after_padding[(i - int((n_size-1)/2)):(i +
        ↪  int((n_size-1)/2)), (j - int((n_size-1)/2)):(j +
        ↪  int((n_size+1)/2))].flatten()
119     im_arr = image_after_padding[(i):(i + int((n_size))), (j):(j +
        ↪  int((n_size)))].flatten()
120     # print(im_arr)
121     geo_product = np.prod(im_arr, where=im_arr > 0, dtype=np.float64)
122
123     # zero_count = np.count_nonzero(im_arr)
124     #
125     # if(zero_count==0):
```

```
126         #     return 0
127         # # else:
128         # print(str(np.power(geo_product,1/8))+"  "+str(np.mean(im_arr)))
129         return np.power(geo_product, (1 / (n_size ** 2 - 1)))
130
131
132     # ContraharmonicMean
133     @njit(parallel=True)
134     def contrahamonic_mean_filter(input_img, n_size, Q):
135         row, col = input_img.shape
136         input_img_padding = padding_jit(input_img, int((n_size - 1) / 2))
137         output_img = np.zeros((row, col))
138
139         for i in prange(row):
140             for j in prange(col):
141                 output_img[i][j] = contrahamonic_filter(i, j,
                        ↪  input_img_padding, n_size, Q)
142         output_img = np.where(np.isnan(output_img), 0, output_img)
143         return output_img
144
145
146     @njit(parallel=True)
147     def alpha_trimmed_mean_filter(input_img, n_size, d):
148         row, col = input_img.shape
149         # print(input_img.shape)
150         # input_img_padding = np.pad(input_img, n_size - 2, pad_with,
            ↪  padder=0)
151         input_img_padding = padding_jit(input_img, int((n_size - 1) / 2))
152         output_img = np.zeros((row, col))
153         # print(input_img_padding.shape)
154
155         for i in prange(row):
156             for j in prange(col):
157                 output_img[i][j] = alpha_trimmed_filter(i, j,
                        ↪  input_img_padding, n_size, d)
158
159         # output_img = np.reshape(par_output, input_img.shape)
160         return output_img
161
162
163     def arithmetic_mean_filter(input_img, n_size):
164         row, col = input_img.shape
165         input_img_padding = np.pad(input_img, n_size - 2, pad_with, padder=0)
166         print(input_img_padding)
167
168         # par_output =
            ↪  np.array(Parallel(n_jobs=6)(delayed(arithmetic_filter)(i,
            ↪  j,input_img_padding,n_size)
169         #                                         for i in range(1, row + 1)
```

```
    #                                           for j in range(1, col + 1)
    #                                           ))
    output_img = np.zeros([row, col])

    for i in range(row):
        for j in range(col):
            output_img[i][j] = arithmetic_filter(i, j, input_img_padding,
             ↪  n_size)

    # output_img = np.reshape(par_output, input_img.shape)
    return output_img


def geometric_mean_filter(input_img, n_size):
    row, col = input_img.shape
    input_img_padding = np.pad(input_img, n_size - 2, pad_with, padder=0)
    # print(input_img_padding)

    # par_output =
    ↪  np.array(Parallel(n_jobs=6)(delayed(arithmetic_filter)(i,
    ↪  j,input_img_padding,n_size)
    #                                           for i in range(1, row + 1)
    #                                           for j in range(1, col + 1)
    #                                           ))
    output_img = np.zeros([row, col])

    for i in range(row):
        for j in range(col):
            output_img[i][j] = geometric_filter(i, j, input_img_padding,
             ↪  n_size)

    # output_img = np.reshape(par_output, input_img.shape)
    # print(normalize(output_img)*255)
    # output_img = normalize(output_img)
    return output_img


# REMOVE AIR TURBULENCE


def fft2d(input_img):
    # row, col = input_img.shape
    # input_img = FFT_zero_padding(input_img)
    # input_img = multiply_center(input_img)
    input_img = np.fft.fft2(input_img)
    input_img = np.fft.fftshift(input_img)

    return input_img

```

```
215
216 def ifft2d(input_img):
217     output_img = np.fft.ifft2(np.fft.ifftshift(input_img))
218     # output_img = transform_centering(output_img)
219     return np.abs(output_img)
220
221
222 def ifft2d_real(input_img):
223     output_img = np.fft.ifft2(np.fft.ifftshift(input_img))
224     # output_img = transform_centering(output_img)
225     return np.real(output_img)
226
227
228 def remove_air_turbulence(input_img, mode, K, gaussian, sigma):
229     input_img_after_fft = fft2d(input_img)
230     k = 0.0025
231
232     row, col = input_img_after_fft.shape
233     u, v = np.meshgrid(np.linspace(0, row - 1, row), np.linspace(0, col -
        ↪  1, col))
234     u = u - row / 2
235     v = v - col / 2
236     d = np.power(u, 2) + np.power(v, 2)
237     H = np.exp(-(k * (np.power(d, 5 / 6))))
238
239     if mode == 1:
240         # full inverse filter
241         output_img = input_img_after_fft / H
242
243     if mode == 2:
244         output_img = input_img_after_fft
245         # Limit inverse
246         for i in range(1, row + 1):
247             for j in range(1, col + 1):
248                 if ((i - row / 2) ** 2 + (j - col / 2) ** 2) < 70:
249                     output_img[i - 1, j - 1] = input_img_after_fft[i - 1,
                        ↪  j - 1] / H[i - 1, j - 1]
250
251     if mode == 3:
252         # Wiener filter
253         buf = np.power(H, 2)
254         k2 = K
255         output_img = input_img_after_fft * buf / (H * (buf + k2))
256         if gaussian == 1:
257             output_img = gaussian_LP_freq_filter(output_img, sigma)
258
259     output_img = normalize(ifft2d(output_img)) * 255
260     return output_img
261
```

```
262
263   def gaussian_LP_freq_filter(input_img, sigma):
264       row, col = input_img.shape
265       gaussian_filter_LP = generate_gaussian(row, col, sigma)
266       filtered_img_LP = np.multiply(input_img, gaussian_filter_LP)
267       return filtered_img_LP
268
269
270   def restore_planar_motion(input_img, a, b, T, k, LP, sigma):
271       input_img_after_fft = fft2d(input_img)
272
273       row, col = input_img.shape
274       u, v = np.meshgrid(np.linspace(1, row, row), np.linspace(1, col, col))
275       A = a * u + b * v
276
277       # print(A.shape)
278       T_arr = np.ones([row, col]) * T
279       H = (T_arr / (np.pi * A)) * np.sin(A * np.pi) * np.exp(-1j * np.pi *
          ↪  A)
280       # condition = np.array((A == 0))
281       # H = H + condition
282       buf = H * np.conj(H)
283       # print(buf + k)
284       F = input_img_after_fft * buf / (H * (buf + k))
285
286       # gaussian
287       if LP == 1:
288           F = F * generate_gaussian(row, col, sigma)
289
290       output_img = normalize_pst(ifft2d_real(F)) * 255
291
292       return output_img
293
294 jit
295 f padding(img, pad):
296   padded_img = np.zeros((img.shape[0] + 2 * pad, img.shape[1] + 2 * pad))
297   padded_img[pad:-pad, pad:-pad] = img
298   return padded_img
299
300
301 jit(parallel=True)
302 f AdaptiveMedianFilter(img, s=3, sMax=21):
303   if len(img.shape) == 3:
304       raise Exception("Single channel image only")
305
306   row, col = img.shape
307   a = sMax // 2
308   padded_img = padding(img, a)
309
```

```
    f_img = np.zeros(padded_img.shape)

    for i in prange(a, row + a + 1):
        for j in prange(a, col + a + 1):
            value = stage_A(padded_img, i, j, s, sMax)
            f_img[i, j] = value

    return f_img[a:-a, a:-a]


    @njit
    def stage_A(mat, x, y, s, sMax):
        window = mat[x - (s // 2):x + (s // 2) + 1, y - (s // 2):y + (s // 2)
        ↪  + 1]
        Zmin = np.min(window)
        Zmed = np.median(window)
        Zmax = np.max(window)

        A1 = Zmed - Zmin
        A2 = Zmed - Zmax

        if A1 > 0 and A2 < 0:
            return stage_B(window, Zmin, Zmed, Zmax)
        else:
            s += 2
            if s <= sMax:
                return stage_A(mat, x, y, s, sMax)
            else:
                return Zmed


    @njit
    def stage_B(window, Zmin, Zmed, Zmax):
        h, w = window.shape

        Zxy = window[h // 2, w // 2]
        B1 = Zxy - Zmin
        B2 = Zxy - Zmax

        if B1 > 0 and B2 < 0:
            return Zxy
        else:
            return Zmed
```