The Report for Digital Image Processing Laboratory 6

This report is contributed by HUANG Guanchao, SID 11912309, from SME. The complete resources of this laboratory, including source code, figures and report in both `.md` and `.pdf` format can be retrieved at my GitHub repo

Image Denoising

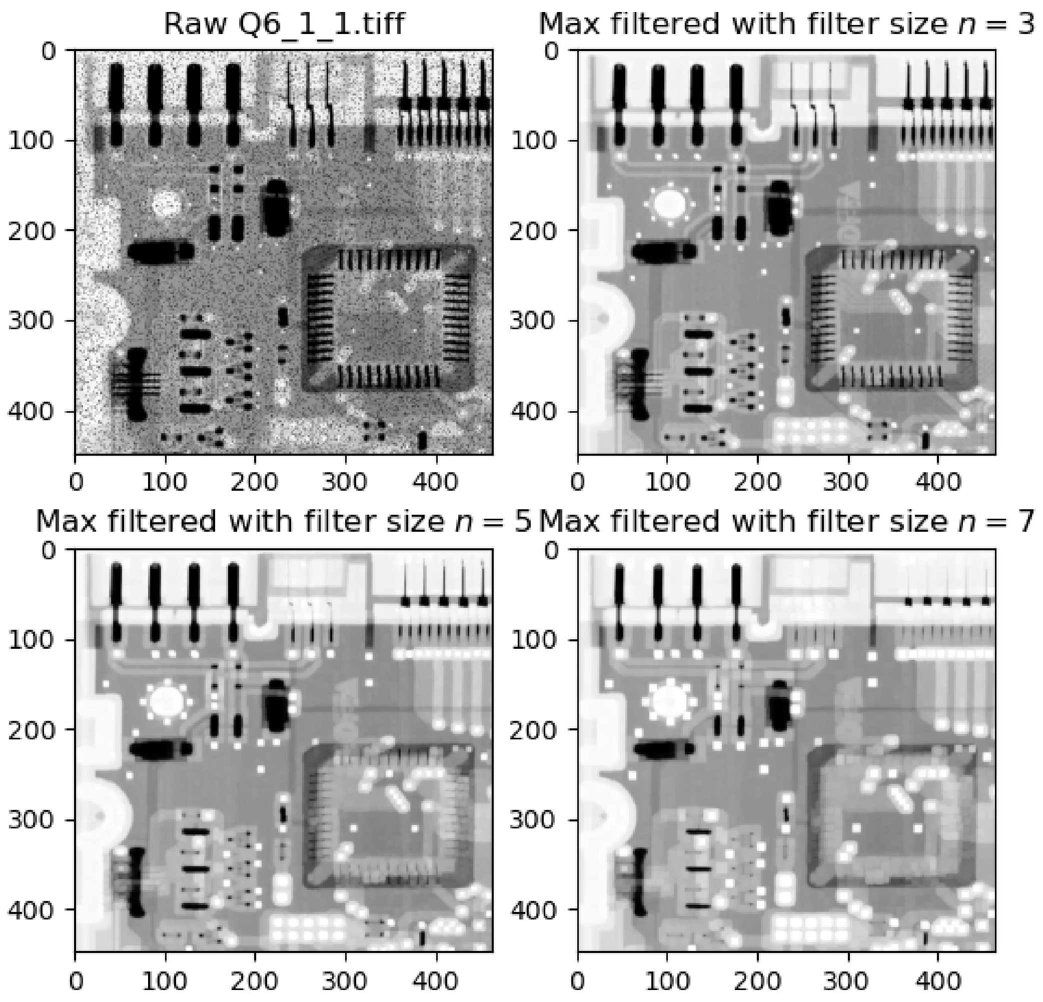
Remove Pepper-Noise for `Q6_1_1.tiff`

It can be easily discovered that, the image is polluted by **pepper noise**, namely many "black" pixels with low intensity. In the presence of pepper noise, **max filtering** may have good effect. The filter is defined mathematically as follows:

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

```python
def max_filter(img_raw, n: int = 3):
    m = (n - 1) // 2
    row, col = img_raw.shape
    img_pad = np.pad(img_raw, m)
    img_out = np.array([img_pad[i:i + n, j:j + n].max()
                        for i in range(row)
                        for j in range(col)]).reshape(row, col)
    return img_out
```

For larger filter size, more details are lost, and thus setting filter size to be `n = 3` is satisfying.


Raw Q6_1_1.tiff
Max filtered with filter size $n = 3$
Max filtered with filter size $n = 5$
Max filtered with filter size $n = 7$
0
100
200
300
400

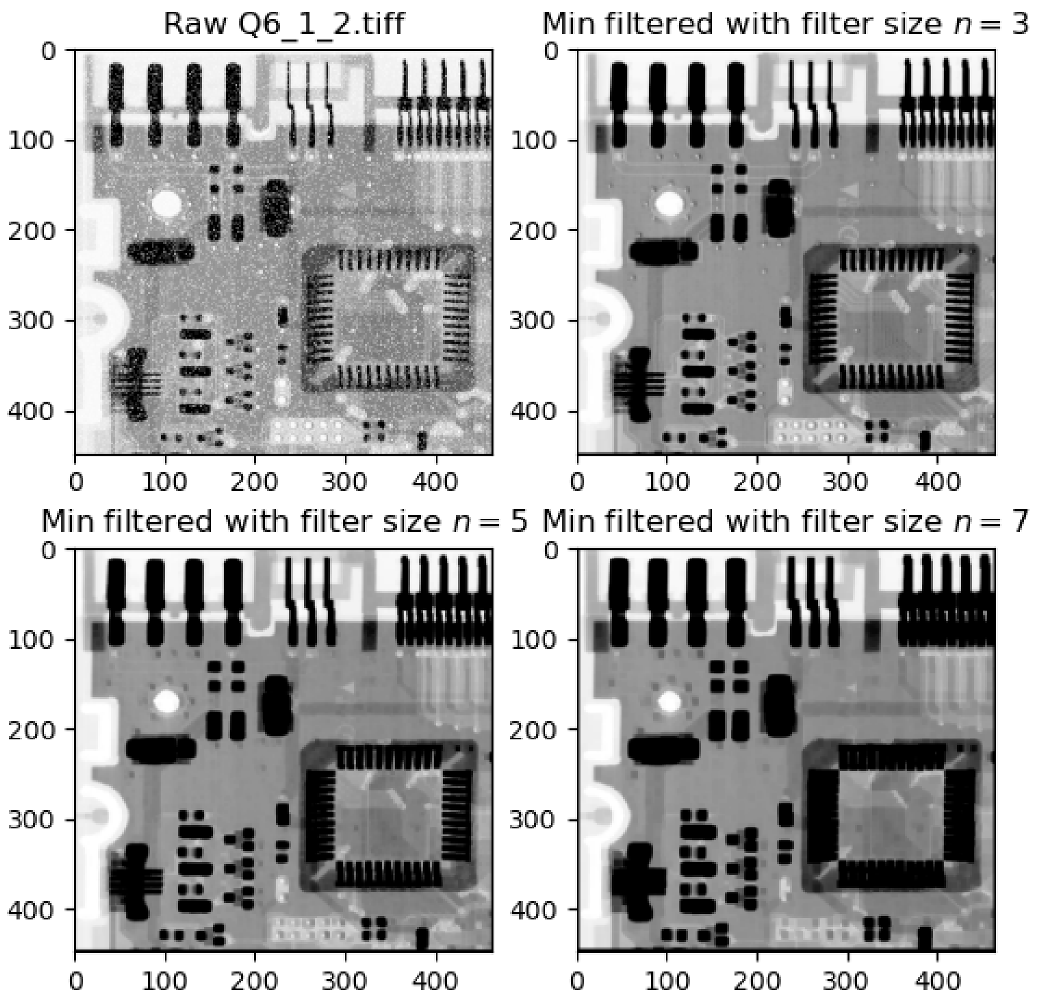
Remove Salt-Noise for `Q6_1_2.tiff`

Similar to `Q6_1_1.tiff`, the image is polluted by **salt noise** instead, namely many "bright" pixels with high intensity. In the presence of salt noise, **min filtering** may have good effect. The filter is defined mathematically as follows:

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

```
def min_filter(img_raw, n: int = 3):
    m = (n - 1) // 2
    row, col = img_raw.shape
    img_pad = np.pad(img_raw, m)
    img_out = np.array([img_pad[i:i + n, j:j + n].min()
                        for i in range(row)
                        for j in range(col)]).reshape(row, col)
    return img_out
```

For larger filter size, more details are lost, and the dark areas become larger due to min filtering. Thus, setting filter size to be `n = 3` is proper.



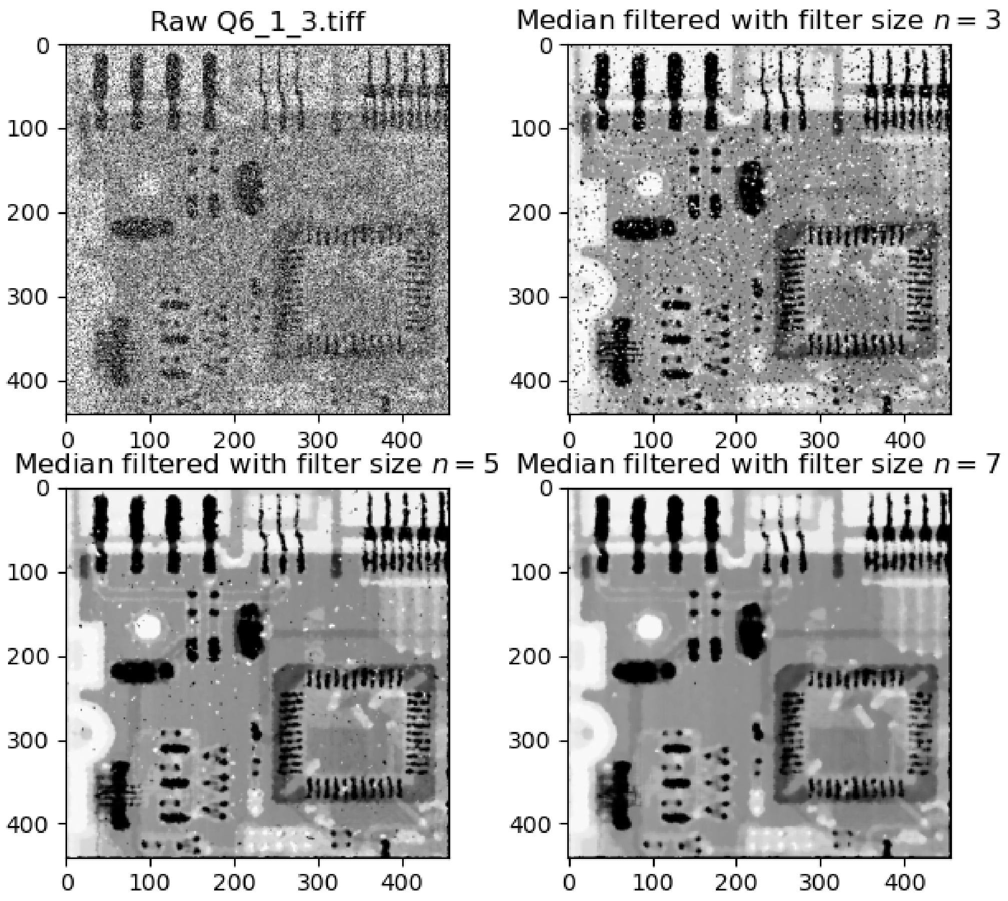
Remove Salt-and-Pepper Noise for `Q6_1_3.tiff`

By observing both the dark and bright regions of the image, we may discover that the image is polluted by **salt-and-pepper noise**. In the presence of which, **median filtering** may have good effect. The filter is defined mathematically as follows:

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

```
def median_filter(img_raw, n: int = 3):
    m = (n - 1) // 2
    row, col = img_raw.shape
    img_pad = np.pad(img_raw, m)
    img_out = np.array([np.median(img_pad[i:i + n, j:j + n])
                        for i in range(row)
                        for j in range(col)]).reshape(row, col)
    return img_out
```

For larger filter size, more details could be lost. The smallest filter size to achieve satisfying filtering result is `n = 7`.


Raw Q6_1_3.tiff
Median filtered with filter size $n = 3$
Median filtered with filter size $n = 5$
Median filtered with filter size $n = 7$

Remove the Combined Noise in `Q6_1_4.tiff`

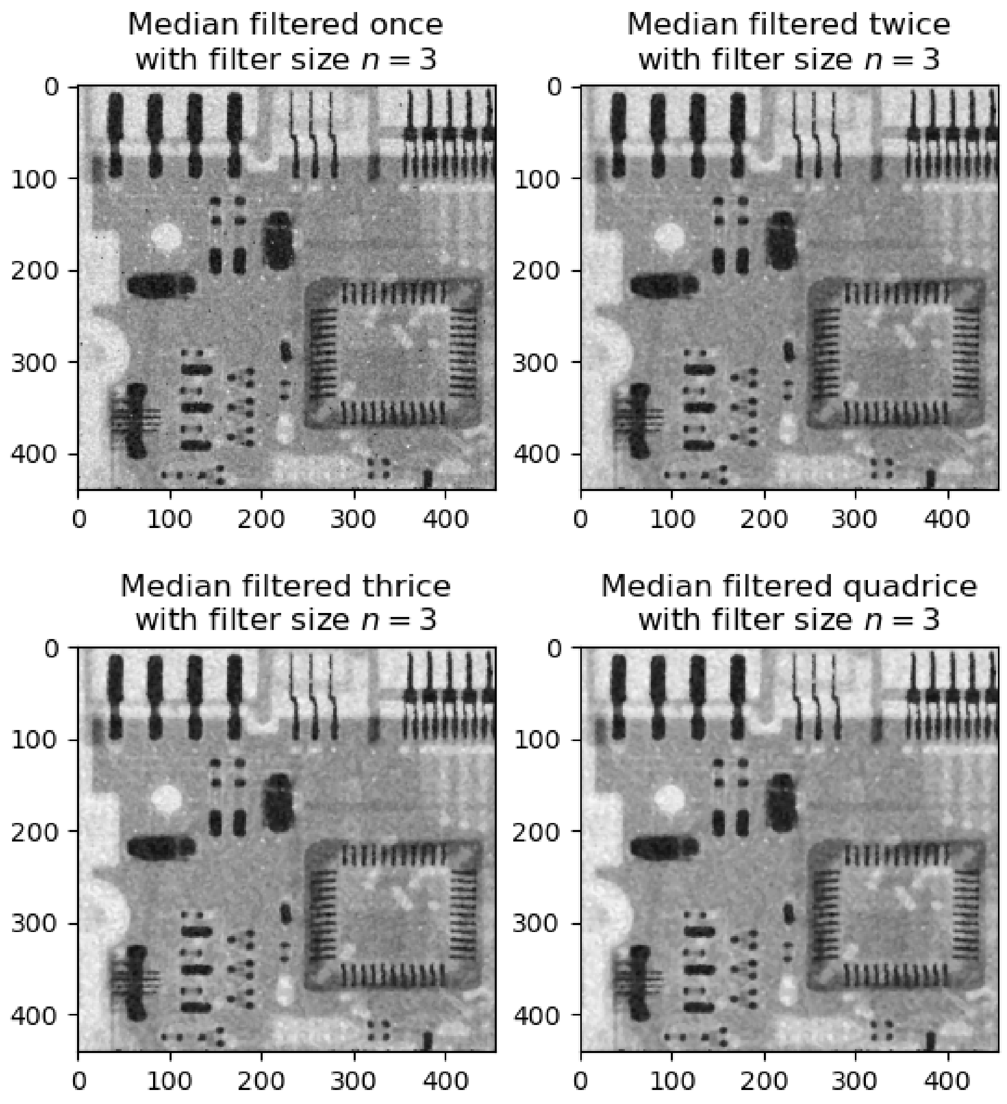
From observation, we may found that the noise in `Q6_1_4.tiff` is the combination of salt-and-paper and other specific type of noise.

Multiple Median Filtering

One possible way for dealing with complex noise is by multiple median filtering.

```
Q6_1_4_1 = median_filter(Q6_1_4)
Q6_1_4_2 = median_filter(Q6_1_4_1)
Q6_1_4_3 = median_filter(Q6_1_4_2)
Q6_1_4_4 = median_filter(Q6_1_4_3)
```

The result for various iterations of median filtering is shown below.


Median filtered once with filter size $n = 3$
Median filtered twice with filter size $n = 3$
Median filtered thrice with filter size $n = 3$
Median filtered quadrice with filter size $n = 3$
0
100
200
300
400

Though the image looks better than its raw, the result is still not satisfying enough.

Alpha-Trimmed Filtering

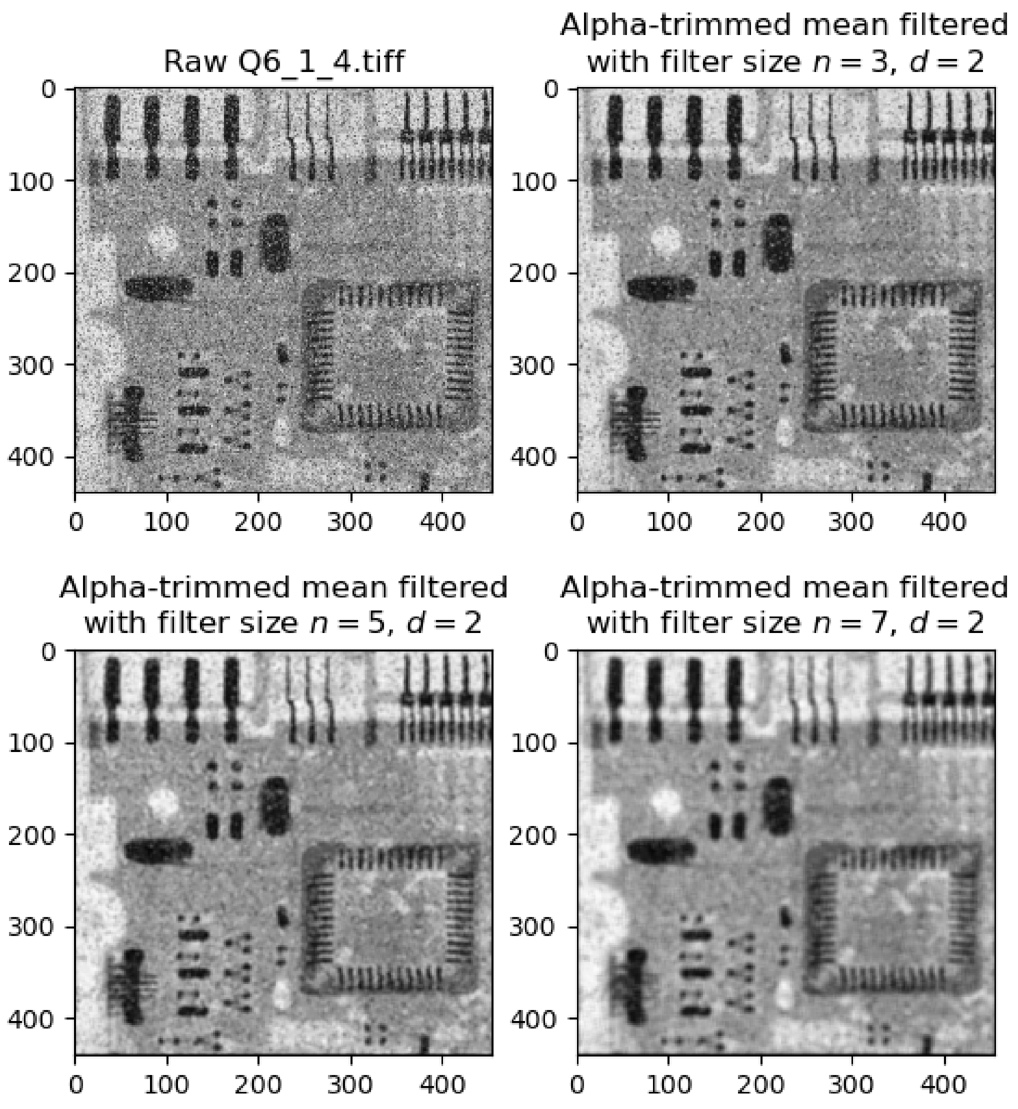
For combined noise, alpha-trimmed filtering is also another option. The filter is mathematically defined as follows.

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

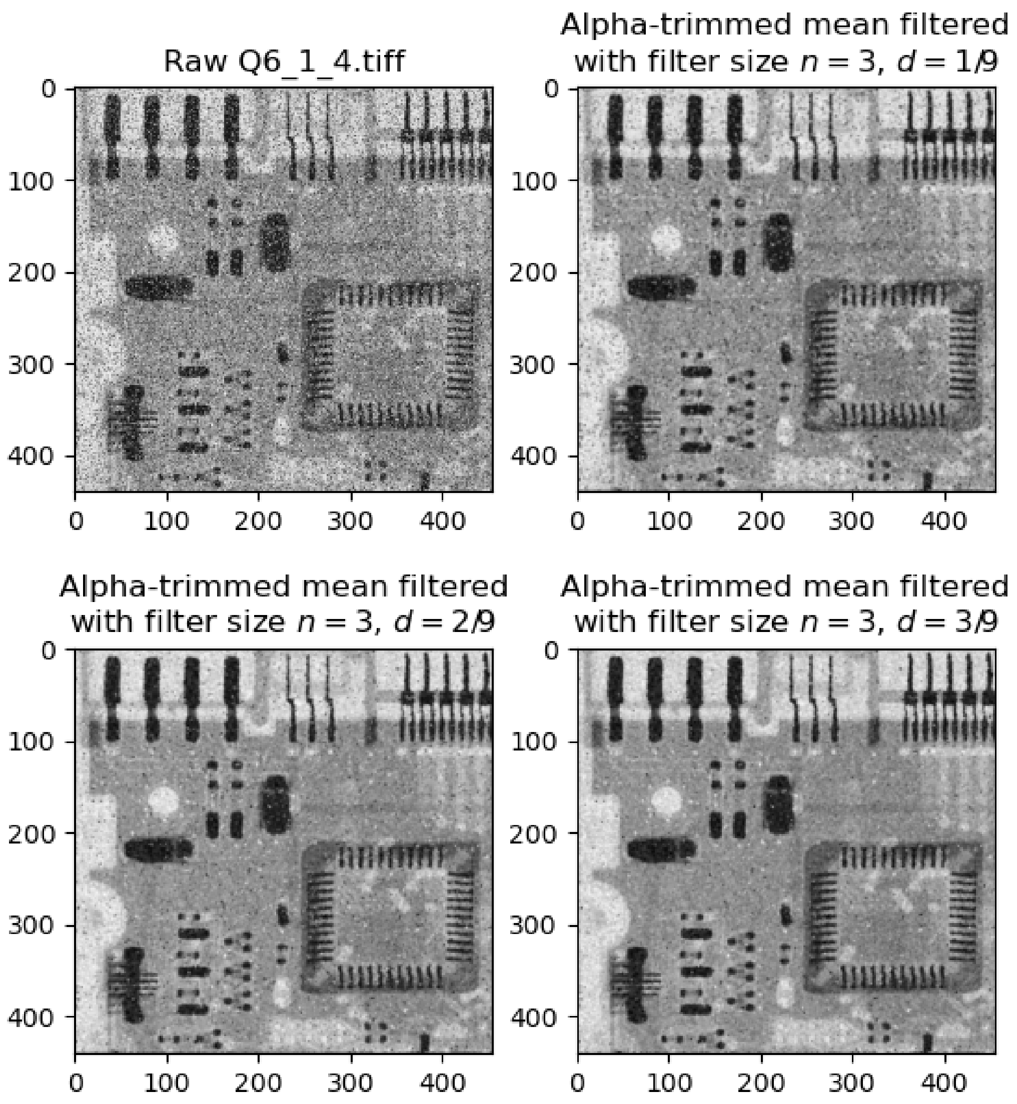
We delete the $d/2$ lowest and the $d/2$ highest intensity values of $(g,\,t)$ in the neighborhood S_{xy}. Let $g_r(s,\,t)$ represent the remaining $mn - d$ pixels.

```python
def alpha_filter(img_raw, n: int = 3, d=0.1):
    m = (n - 1) // 2
    trimmed = max(int(d * n ** 2), 1)
    row, col = img_raw.shape
    img_pad = np.pad(img_raw, m)

    img_out = np.array([np.mean(np.sort(img_pad[i:i + n, j:j + n].flat)
                                [trimmed:-trimmed])
                        for i in range(row)
                        for j in range(col)])

    return img_out.astype(np.uint8).reshape(row, col)
```

First consider different filter size, `d` is by default `0.1`, that is, the maximum and minimum 10% of pixels in the window is excluded from averaging. The result comparison is shown below.


Raw Q6_1_4.tiff
Alpha-trimmed mean filtered with filter size $n = 3$, $d = 2$
Alpha-trimmed mean filtered with filter size $n = 5$, $d = 2$
Alpha-trimmed mean filtered with filter size $n = 7$, $d = 2$
0
100
200
300
400

Similar to other methods used before, larger filter size introduces greater blurriness, only `n=3` is acceptable. Based on this, the comparison of different `d` is shown below.


Raw Q6_1_4.tiff
Alpha-trimmed mean filtered with filter size $n = 3$, $d = 1/9$
Alpha-trimmed mean filtered with filter size $n = 3$, $d = 2/9$
Alpha-trimmed mean filtered with filter size $n = 3$, $d = 3/9$
0
100
200
300
400

Again, though the image looks better than its raw, the result is still not satisfying enough.

Adaptive Median Filtering

Adaptive filtering may have good effect in some cases, and here try using adaptive median filtering, which works in two stages given below.

The behavior of spatial adaptive filter changes based on statistical characteristics of the image inside the filter region defined by the $m \times n$ rectangular window. The performance is superior to common filters discussed before.

Firstly, S_{xy} is the local region, and some notations are predefined.

Notation	Meaning
$z_{\min}$	minimum intensity value in S_{xy}
$z_{\max}$	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}

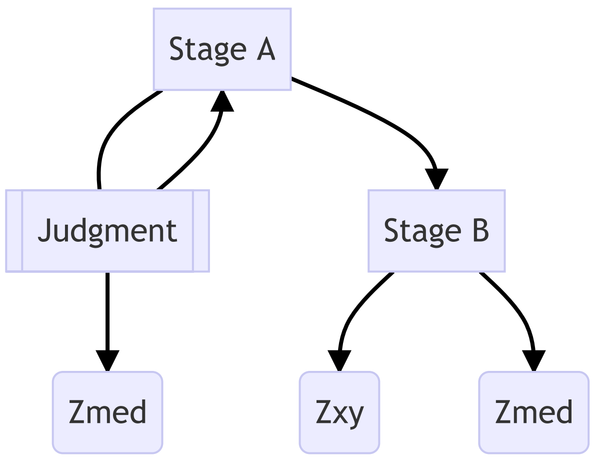
- *Stage A*

  > To determine if the median value is an impulse or not.

  - Define $A_1 = z_{\text{med}} - z_{\min}$, $A_2 = z_{\text{med}} - z_{\max}$.
  - If $A_1 > 0$ and $A_2 < 0$, go to *Stage B*;
  - Else increase the window size.
  - If window size is less or equal to $S_{\max}$, repeat *Stage A*;
  - Else output z_{med}.
- *Stage B*

  > To determine if the processed point is an impulse or not.

  - Define $B_1 = z_{xy} - z_{\min}$, $B_2 = z_{xy} - z_{\max}$.
  - If $B_1 > 0$ and $B_2 < 0$, output z_{xy};
  - Else output z_{med}.



> The `Python` implementation of adaptive filtering is rather more complicated than methods used before, hence the code is concisely.

At the beginning of the function, obtain the dimension of the image, and initialize the window size. For the purpose of convenience, the default window size is set to be `n = 1`, namely a single pixel. Also, instantiate an empty array for storing the result.

```
# start from window size of one, that is the pixel itself
row, col = img_raw.shape
img_out = np.zeros((row, col))
```

A separate function `window()` for obtaining the window is implemented. For invalid window, namely out of bound or exceeding the maximum window size, the function would return `None`.

> Due to the `namespace` concept in `Python`, a function is able to access any variables in its parent `namespace`s, and therefore there is no need for passing other parameters.

```python
def window():
    """
    :return: if out of bound or reaching the , return None, if not return the window
    """
    if i - m >= 0 and i + m <= row and j - m >= 0 and j + m <= col and n <= s:
        return img_raw[i - m:i + m + 1, j - m:j + m + 1]
```

Do filtering within `for` loop.

```python
for i in range(row):
    for j in range(col):
        ...
```

In each iteration, first initialize the window and the value used in Stage A. Functions `np.median()`, `np.min` and `np.max()` are also valid for a single value.

```python
# initialize the window and the values in stage A, by initialization the window is single pixel
n = 1
m = (n - 1) // 2

img_tmp = window()
a1 = np.median(img_tmp) - np.min(img_tmp)
a2 = np.median(img_tmp) - np.max(img_tmp)
```

At the start of each iteration, the window contains only one pixel, and therefore `a1 = a2 = 0`, the algorithm would enter Stage A, a `while` loop here, definitely. In each iteration of the loop, it is first verified that the window is valid, that is, to make sure the window size does not exceed its maximum `s`, and also no out of bound exception occurs. Judgment is also made to make sure that Stage B is not available yet.

> For the purpose of convenience, padding is not implemented here in adaptive filtering.

```python
# while the window is valid and the requirement tof going stage B is not satisfied, keep looping in stage A
while img_tmp is not None and not (a1 > 0 and a2 < 0):
    n += 2
    m = (n - 1) // 2
    img_tmp = window()
```

In each iteration of the `while` loop, namely Stage A, the window size and the window `img_tmp` is updated. Once the prerequisites are no more satisfied, the loop is exited. Further process is still based on valid window, so it is needed to recover the window to its previous status.

```
# when out of bound, or reaching the maximum windows size, or stage B is
available
# recover the window size to the last one
n -= 2
m = (n - 1) // 2
img_tmp = window()  # get the window
```

Since there is more than one case that cause the `while` loop to exit, after the recovery, we need to justify again why the Stage A `while` loop is exited. If the condition for Stage B is satisfied, the exit is due to the jump to Stage B.

```
if a1 > 0 and a2 < 0:  # when stage B is available
    b1 = img_raw[i, j] - np.min(img_tmp)
    b2 = img_raw[i, j] - np.max(img_tmp)
    if b1 > 0 and b2 < 0:
        img_out[i, j] = img_raw[i, j]
```

If not, the exit is due to invalid window size, including out of bound of reaching the maximum size.

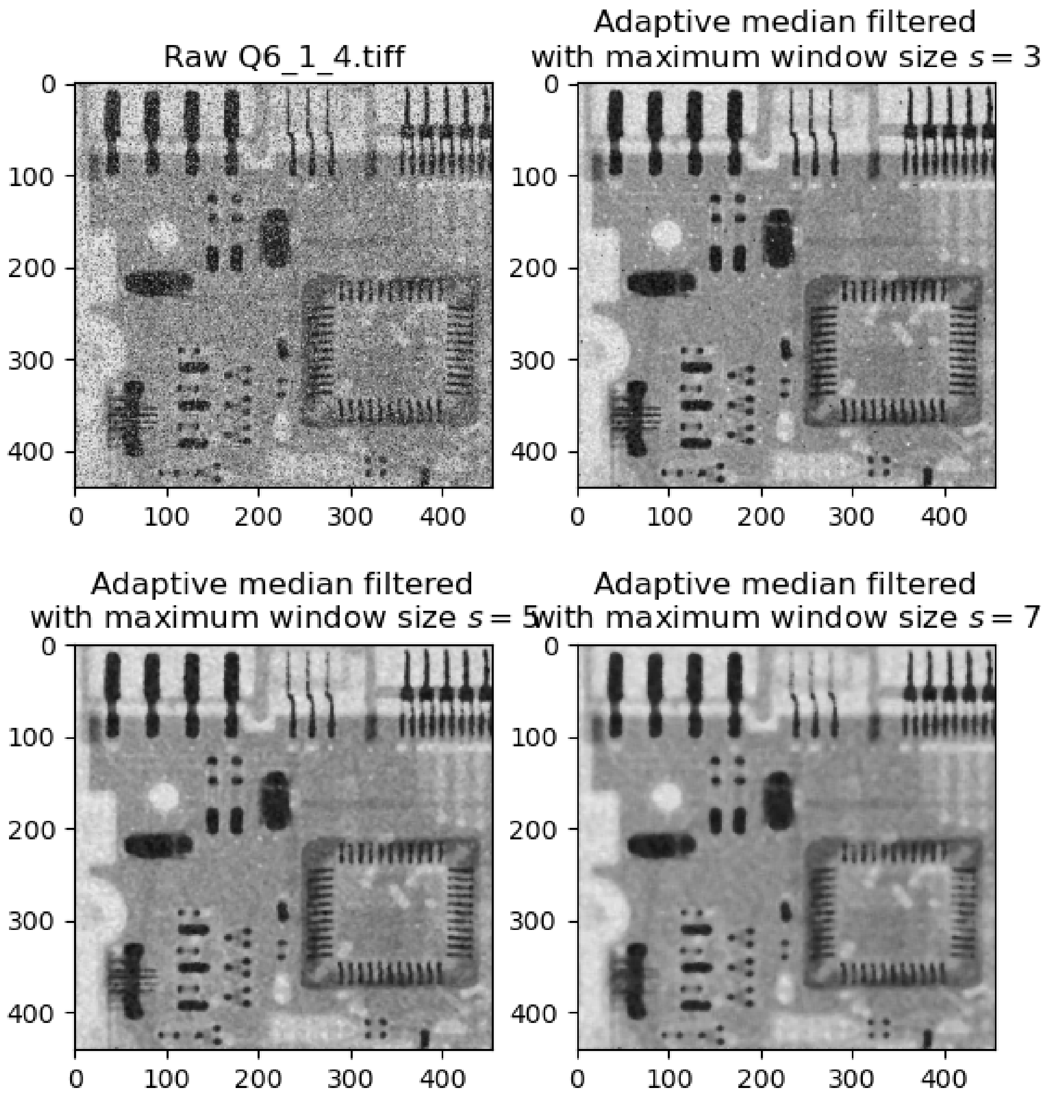
```
# another possible case is combined below
else:  # stage B is not available, the exit of while loop is due to reaching the
max possible window size
    img_out[i, j] = np.median(img_tmp)
```

Note that, the implicit case in Stage B for taking `img_out[i, j] = np.median(img_tmp)` is combined in the final `else` block.

In a new iteration, the window size is reload automatically. Finally, return the result.

```
return img_out.astype(np.uint8)
```

The result of adaptive median filtering is shown below.


Raw Q6_1_4.tiff
Adaptive median filtered with maximum window size $s = 3$
Adaptive median filtered with maximum window size $s = 5$
Adaptive median filtered with maximum window size $s = 7$
0
100
200
300
400

The final result is largely improved with adaptive filtering.

Image Restoration

It is known that image `Q6_2.tif` was degraded from an original image due to the *atmosphere turbulence*, the model for which is given as

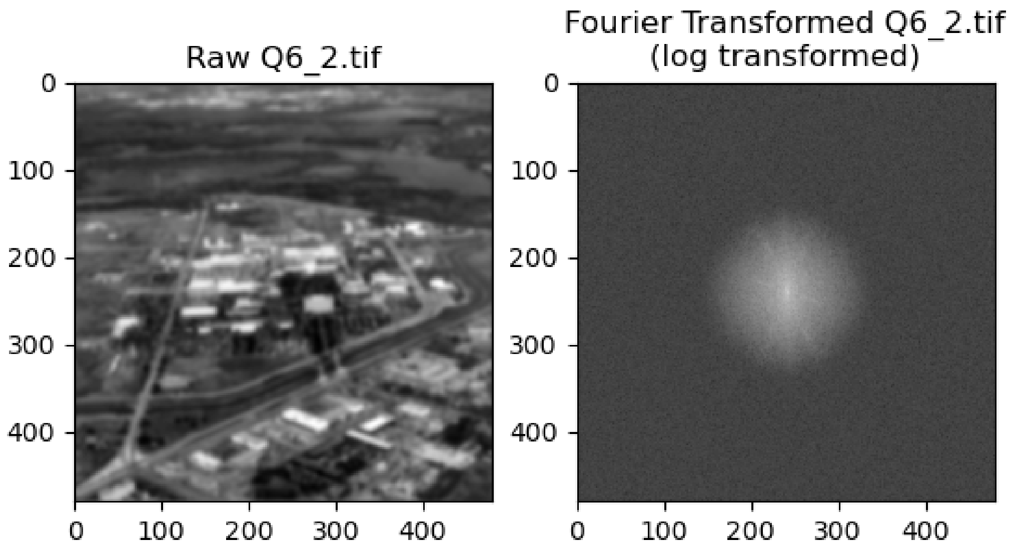
$$H(\mu,\, \nu) = \exp\left[-k\left(\mu^2 + \nu^2\right)^{5/6}\right].$$

In which, k is a constant that depends on the nature of atmospheric turbulence, and is given as $k = 0.0025$ in this lab problem.

Based on the degradation function, different filtering methods are available for restoration.

Firstly, read in the image, and do Fourier transform.

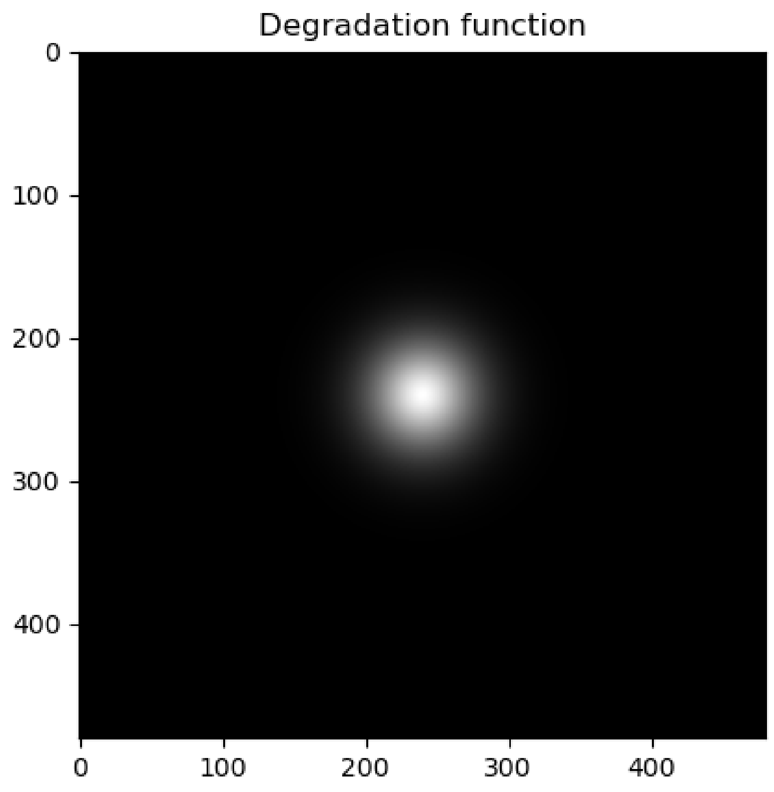
```python
Q6_2 = np.array(Image.open("Q6_2.tif"))
row, col = Q6_2.shape

img_fourier = fftshift(fft2(np.pad(Q6_2, ((0, row), (0, col)))))
img_view = np.log10(np.abs(img_fourier) + 1).astype(np.uint8)
```



> Exceptions occurred in this step, see [Bugs and Fixations](#).

Based on the turbulence model, we may generate the response function in both spatial and frequency domain.

```python
k = 0.0025
func = np.fromfunction(lambda mu, nu: np.exp(-k * (mu ** 2 + nu ** 2) ** (5 /
6)), (row, col))
func_fourier = fftshift(fft2(np.pad(func, ((0, row), (0, col)))))
func_view = np.log10(np.abs(img_fourier) + 1)
```

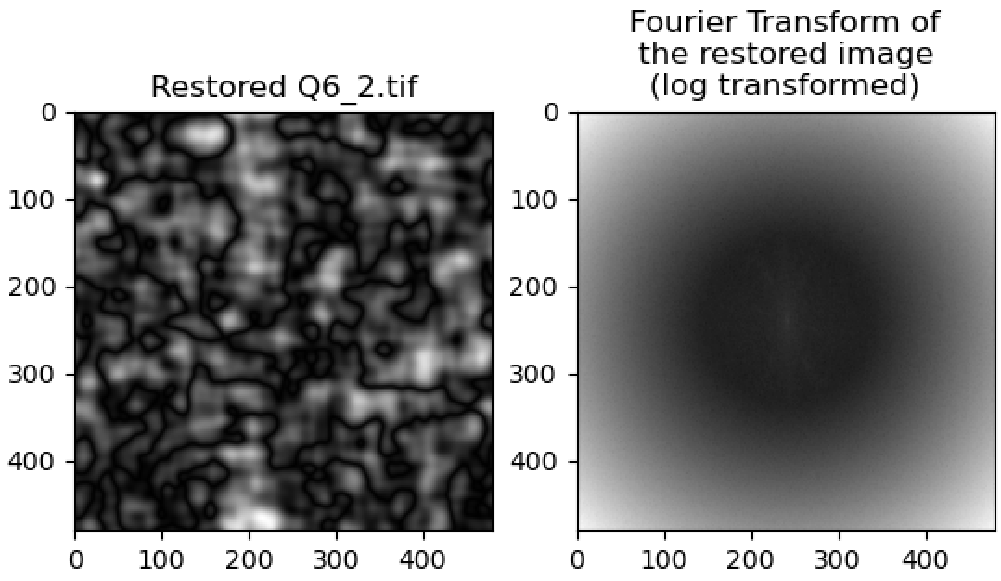


Full Inverse Filtering

An estimate of the transform of the original image is given as

$$\hat{F}(\mu,\,\nu) = \frac{G(\mu,\nu)}{H(\mu,\,\nu)}.$$

```
img_out_fourier = img_fourier / func_fourier
img_out_view = np.log10(np.abs(img_out_fourier) + 1)
img_out = np.real(ifft2(fftshift(img_out_fourier)).astype(np.uint8))
```



We can see that the result is completely mass, which is unsurprising. Full inverse filtering does not take unknown noise signal into consideration, and therefore the restored image is actually

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

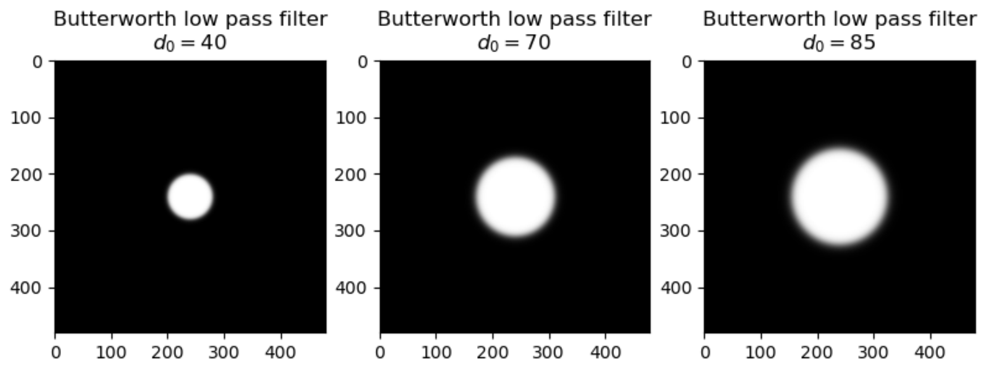
Moreover, if the degradation function has zero or very small values, then the ratio $N(\mu, \nu)/H(\mu, \nu)$ could easily dominate the estimated $\hat{F}(\mu, \nu)$.

Radially Limited Inverse Filtering

To avoid part of the problems in full inverse filtering, one approach is to limit the filter frequencies to values near the origin. Butterworth low pass filter can achieve such goal.

```
def butterworth(d: int = 40):
    n = 10
    def _func(mu, nu):
        return 1 / (1 + ((mu - row / 2) ** 2 + (nu - col / 2) ** 2) ** n /
                    d ** (2 * n))
    return np.fromfunction(_func, (row, col)).astype(float)
```

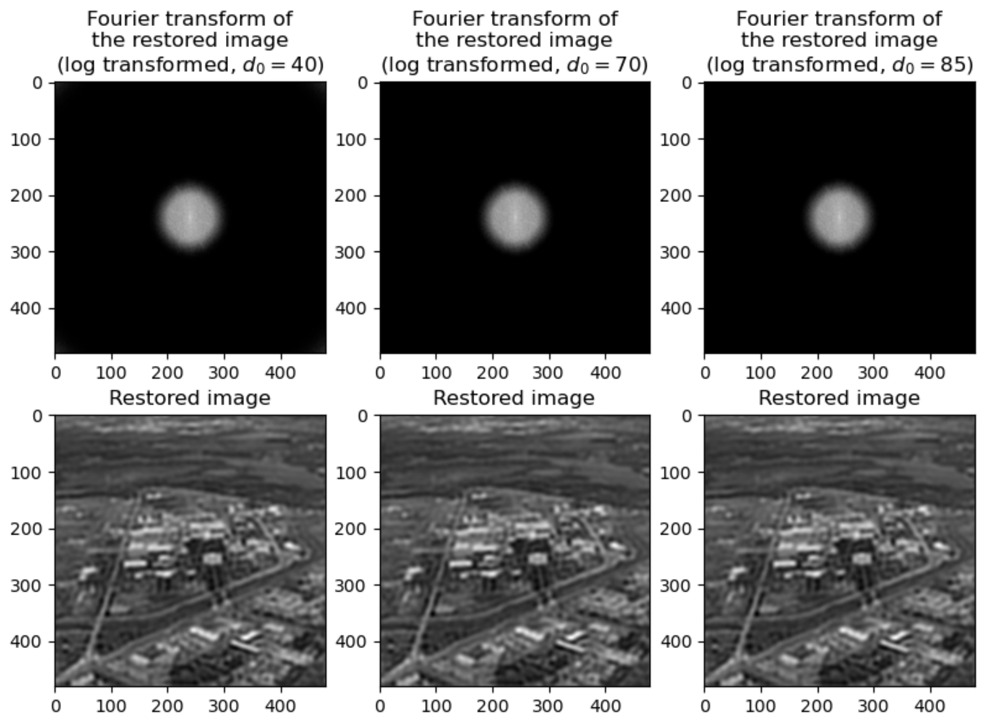
> Note that, the datatype conversion `.astype(float)` is necessary, otherwise the datatype of the resulting filter would be `object`, and therefore cannot be plotted.



The following operations are similar to full inverse filtering, but with applying a low pass filter to the final result to limit its frequency bandwidth.

```
def restore(butter, fourier=img_out_fourier):
    fourier *= butter
    img_out_view = np.log10(np.abs(fourier) + 1)
    img_out = np.abs(ifft2(ifftshift(img_out_fourier)))
    return fourier, img_out_view, img_out
```

The final result is shown below.


Fourier transform of the restored image (log transformed, $d_0 = 40$)
Fourier transform of the restored image (log transformed, $d_0 = 70$)
Fourier transform of the restored image (log transformed, $d_0 = 85$)
Restored image
Restored image
Restored image

Wiener Filtering

Wiener filtering, or *minimum mean square error filtering* is proposed by N. Wiener in 1942. The final object of image restoration is to find an estimate of the uncorrupted image such that the mean square error between them is minimized, which is defined as follows.

$$e^2 = E\left\{\left(f - \hat{f}\right)^2\right\}$$

The minimum of the error function is given in the frequency domain by the expression.

$$\begin{aligned}\hat{F}(\mu,\nu) &= \frac{H^*(\mu,\nu)S_f(\mu,\nu)}{S_f(\mu,\nu)\left|H(\mu,\nu)\right|^2 + S_\eta(\mu,\nu)}G(\mu,\nu) \\ &= \frac{H^*(\mu,\nu)}{\left|H(\mu,\nu)\right|^2 + S_\eta(\mu,\nu)/S_f(\mu,\nu)}G(\mu,\nu) \\ &= \frac{1}{H(\mu,\nu)}\frac{\left|H(\mu,\nu)\right|^2)}{\left|H(\mu,\nu)\right|^2 + S_\eta(\mu,\nu)/S_f(\mu,\nu)}G(\mu,\nu)\end{aligned}$$

The notations are given in the table,

Notation	Meaning
$H(\mu,\nu)$	degradation function
$H^*(\mu,\nu)$	complex conjugate of $H(\mu,\nu)$
$\left|H(\mu,\nu)\right|^2$	$= H^*(\mu,\nu)H(\mu,\nu)$
$S_\eta(\mu,\nu)$	$= \left|N(\mu,\nu)\right|^2$, power spectrum of the noise
$S_f(\mu,\nu)$	$= \left|F(\mu,\nu)\right|^2$, power spectrum of the underaged image

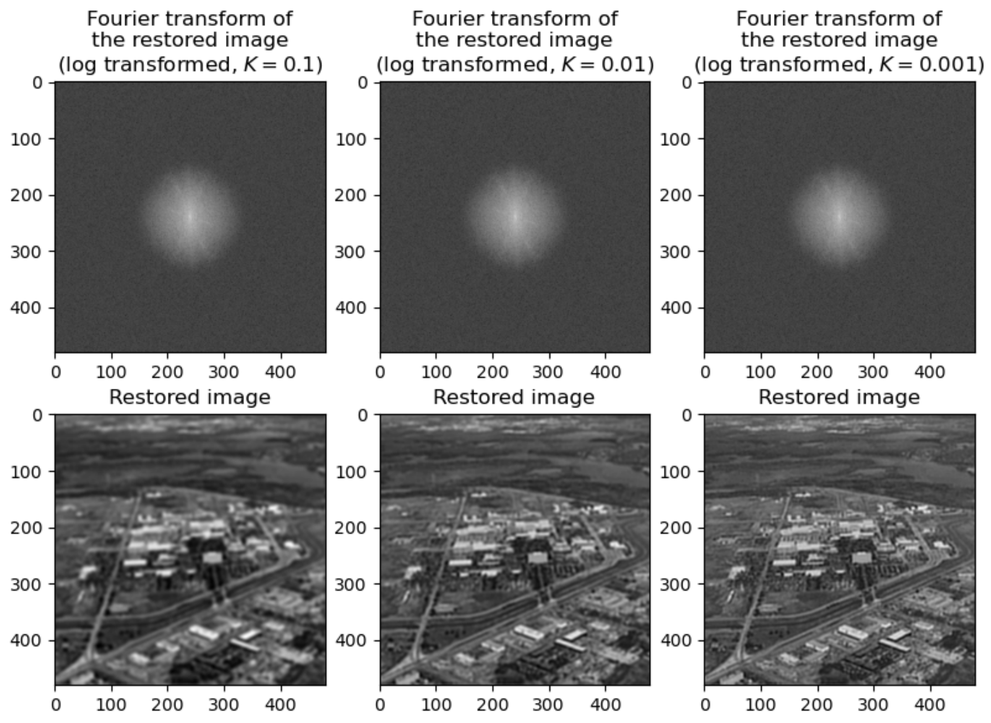
When the power spectrum of $\left|N(\mu,\nu)\right|^2$ and $\left|F(\mu,\nu)\right|^2$ are unknown, the following approximation usually is used.

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

In which, K is a specified constant. Generally the value of K is chosen interactively to yield the best visual results.

The only unique part of Wiener filtering is generating the restored image.

```
img_out_fourier = func ** 2 / func / (func ** 2 + K) * fourier
```


Fourier transform of the restored image (log transformed, $K = 0.1$)
Fourier transform of the restored image (log transformed, $K = 0.01$)
Fourier transform of the restored image (log transformed, $K = 0.001$)
Restored image
Restored image
Restored image
0
100
200
300
400

It is obvious that, the restoration effect is greatly improved with Wiener filtering. Smaller K value, namely larger signal to noise ratio yields sharper images.

Conclusion

- Full inverse filtering is completely not acceptable.
- Radially limited inverse filtering is practical.
- Wiener filtering has the best result.

Bugs and Fixations

Error in Converting `Pillow` Image to `NumPy` Array

`Pillow` worked well in `Q6_1`, however, while dealing with `Q6_2.tif`, errors occurred. Directly reading the image into `numpy.ndarray` would throw error `tempfile.tif: Cannot read TIFF header.`, and the resulting `ndarray` is empty. When trying read the image and convert it into `numpy.ndarray` separately, it is found that the first conversion would fail and throw the same error, and the second conversion would not, but returns a array of all `0` elements.

```
# throws error, and return an empty array
tmp = Image.open("Q6_2.tif")
Q6_2 = np.asarray(tmp)

# throws one error, and return an all 0 array
tmp = Image.open("Q6_2.tif")
Q6_2 = np.asarray(tmp)
Q6_2 = np.asarray(tmp)
```

A natural trial is to update all packages. Run `conda update --all`, it is found that a large amount of packages have available updates. The following two updates need to be focused on.

```
pillow   8.1.1-py38h4fa10fc_0 --> 8.2.0-py38h4fa10fc_0
libtiff      4.1.0-h56a325e_1 --> 4.2.0-hd0e1b90_0
```

After executing installations, run codes again, another error occurred in the conversion, and the process was terminated immediately.

```
Process finished with exit code -1073740791 (0xC0000409)
```

Some posts suggest that, downgrading `libtiff` package could solve this problem. Try return to the previous version.

```
conda install libtiff=4.1.0
...
libtiff  4.2.0-hd0e1b90_0 --> 4.1.0-h56a325e_1
```

Now the image could be read and converted successfully. Notice that, the version of `libtiff` has **NOT** change in the whole time, so the real cause of this exception remains unknown to us. However, it definitely has to do with some unexpected properties in `Q6_2.tif`. The most possible explanation in the environment is the version of `Pillow` caused this exception, and updating `Pillow` from `8.1.1` to `8.2.0` solved this issue.

Fourier Transform for Inverse Filtering

Unlike Fourier transform previous image process in spatial domain, which requires zero-padding to avoid wraparound error, in inverse filtering, padding will lead to completely wrong results. Therefore, zero-padding must not be added before Fourier transform.

Moreover, it is strange that in inverse fourier transform, we must take the *absolute value*, instead of the *real part*, and the reason remains unknown.