Programming Assignment #1

Name: 林穎志 Student ID: 411551007

Introduction / Objectives:

This program is to do image processing on several images. The main objective is to make an image better which means is more clearly to see or to make noises to be lesser. The images we prepare in this program are a woman picture on newspaper with noises, a scenery photo in university with low contrast, a gray scale internet news photo and an old photo be taken long time ago respectively.

Method:

Contrast adjustment: Histogram equalization, Alpha/ Beta/ Gamma correction

Noise reduction: Box filter, Gaussian filter, Max filter, Min filter, Median filter, Adaptive median filter

Sharpening: Laplacian filter, Unsharp masking, Sharpen filter

Review of the methods I have implemented:

I. Contrast adjustment

1. Histogram equalization: Let the intensity distribution of an image be uniformed and a better overall of contrast.
2. Alpha/ Beta correction: A linear transform as the formula $g(x)=\alpha f(x)+\beta$.
3. Gamma correction: An intensity transformation as the formula $s=c\cdot r^{\gamma}$.

II. Noise reduction

1. Box filter: A smoothing filter by calculating the mean of center and the pixel in neighborhood.
2. Gaussian filter: A Gaussian has the advantage of being smooth in both the spatial and frequency domains.
3. Max filter/Min filter/Median filter: An order-statistic filter that change the pixel value to its neighborhood max/ min/ median value.
4. Adaptive median filter: A filter designed to handle high density of impulse noise. The value decision is based on whether original pixel value and local median appear to be impulse value.

III. Sharpening

1. Laplacian filter: A filter usually used to detect edge as the definition
$$\nabla^2 f = \partial^2 f/\partial x^2 + \partial^2 f/\partial y^2$$
2. Unsharp masking: Sharpen photographic by operating original negative and blurred positive as the digital form
$$g_{mask}(x,y)=f(x,y)-\bar{f}(x,y)$$
$$g(x,y)=f(x,y)+k*g_{mask}(x,y)$$
3. Sharpen filter: Make the edges of image sharper.

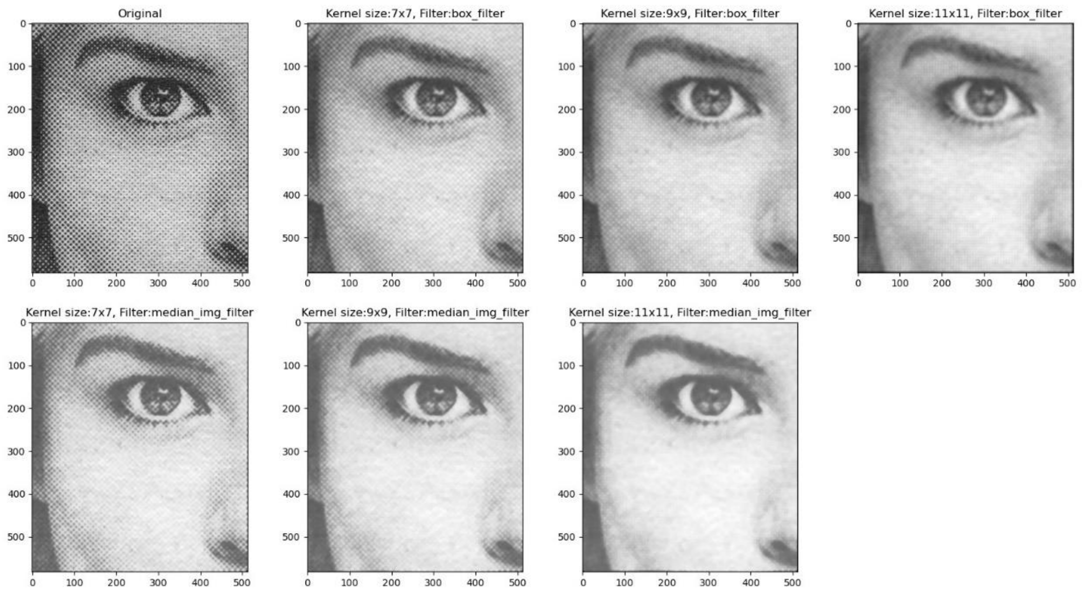
Picture 1:

I. Original Picture:

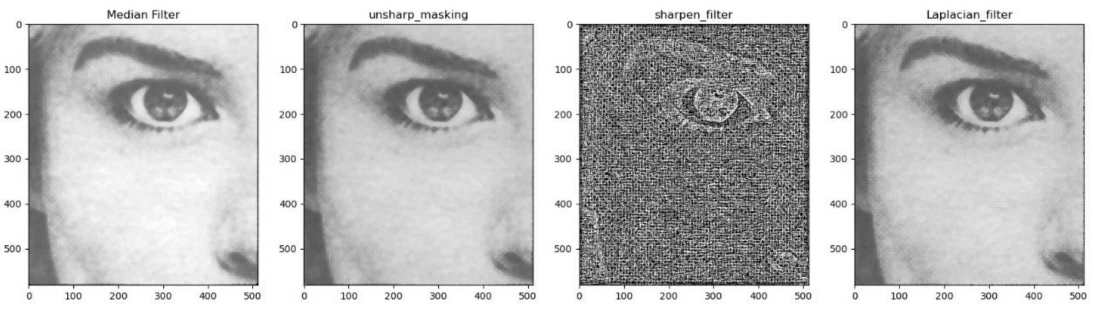
II. Implement Method: Box Filter, Median Filter, Unsharp Masking, Sharpen Filter, Laplacian Filter

III. Experiments & Results:

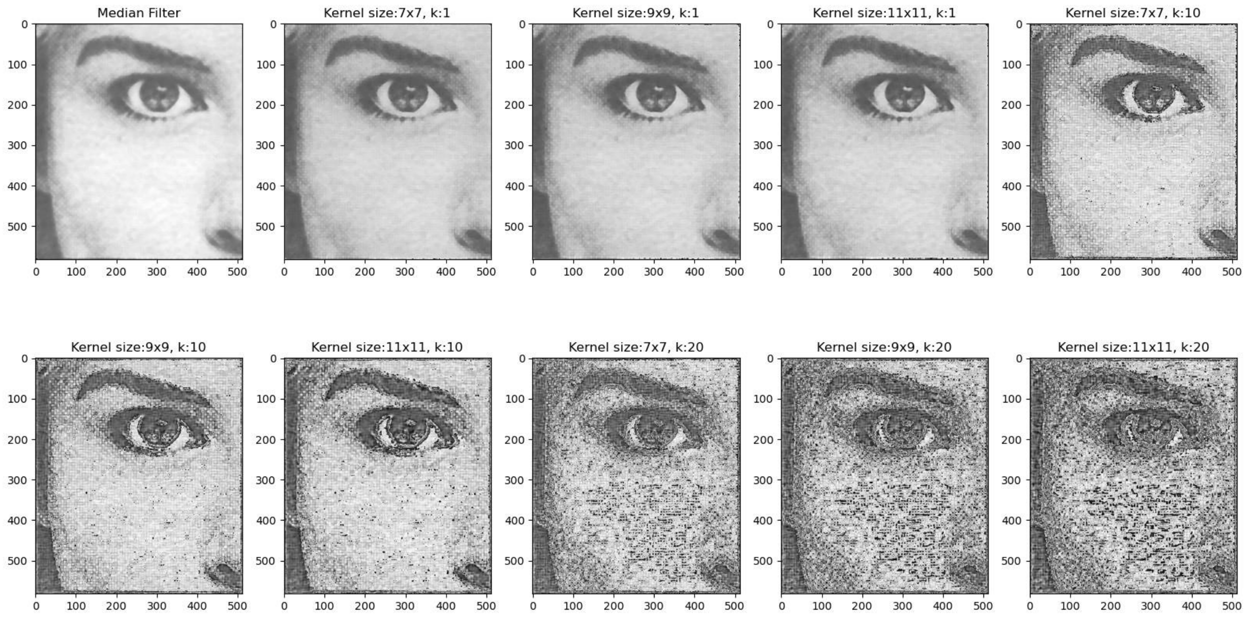
1. Different kernel size of Median Filter and Box Filter


Original
Kernel size:7x7, Filter:box_filter
Kernel size:9x9, Filter:box_filter
Kernel size:11x11, Filter:box_filter
Kernel size:7x7, Filter:median_img_filter
Kernel size:9x9, Filter:median_img_filter
Kernel size:11x11, Filter:median_img_filter

2. Different kinds of filter (Unsharp Masking, Sharpen Filter, Laplacian Filter)



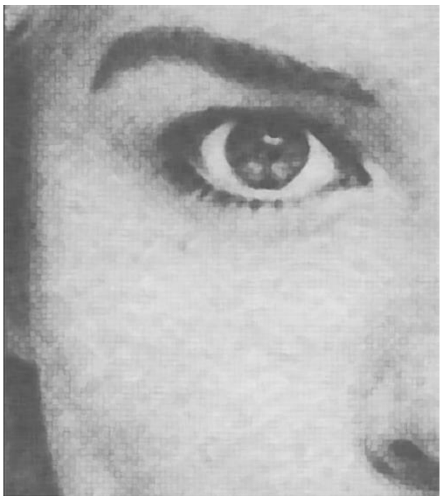
3. Different kernel size and parameter k in unsharp masking



IV. Discussions:

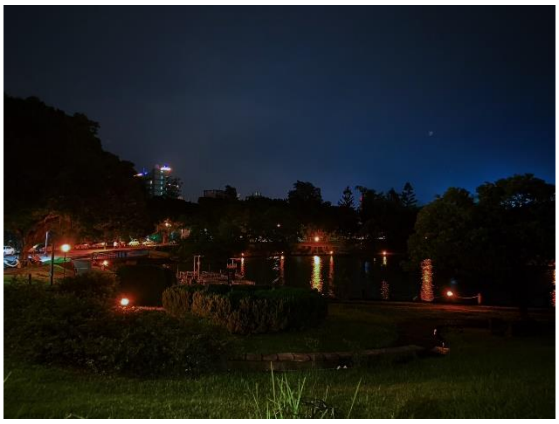
This is a woman’s face image with noises as you can see as above. We can observe that median filter can have a better performance on denoising such kind of noise due to the noise residue on the face while using box filter. Furthermore, we can observer that within a larger kernel size, the effect of denoising can be better. To make the facial features more pronounced, we apply sharpening with different filters. The result shows that Laplacian filter and unsharp masking have a better performance. At last, we choose unsharp masking to do image sharpening and apply different parameters of k and kernel size. We can observe that with the larger k and kernel size, the sharpening effect is more significant.

Ⅴ. Final Output:



Picture 2:

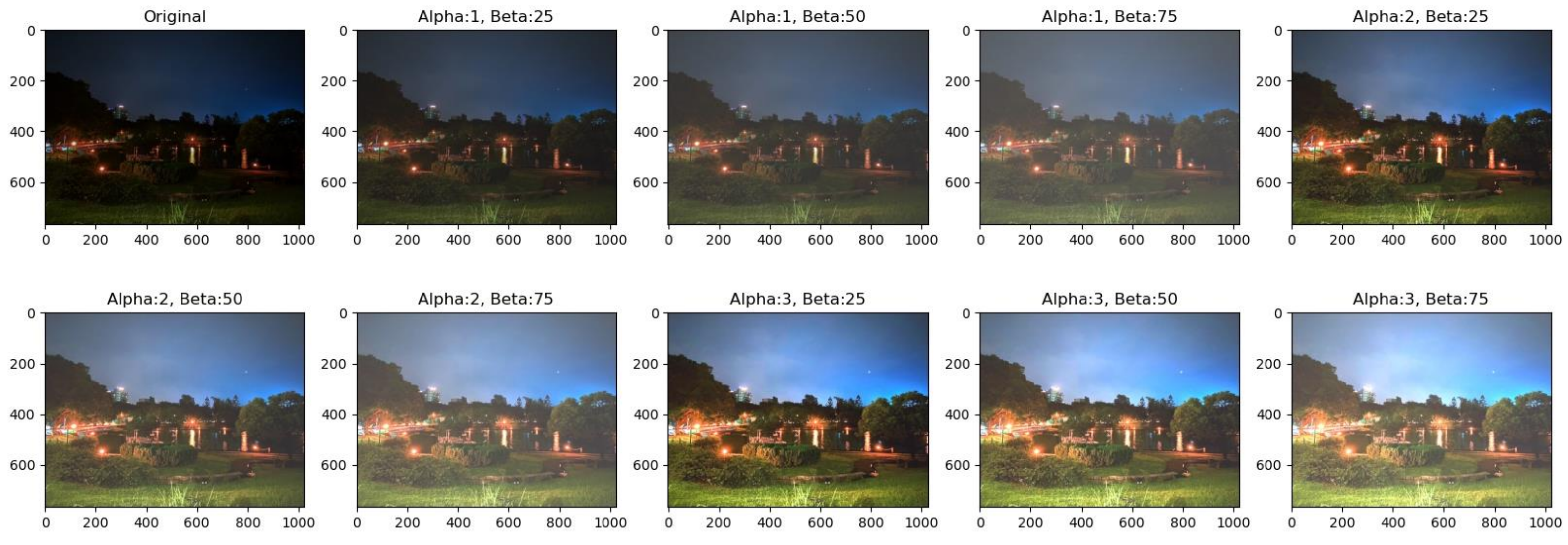
I. Original Picture:



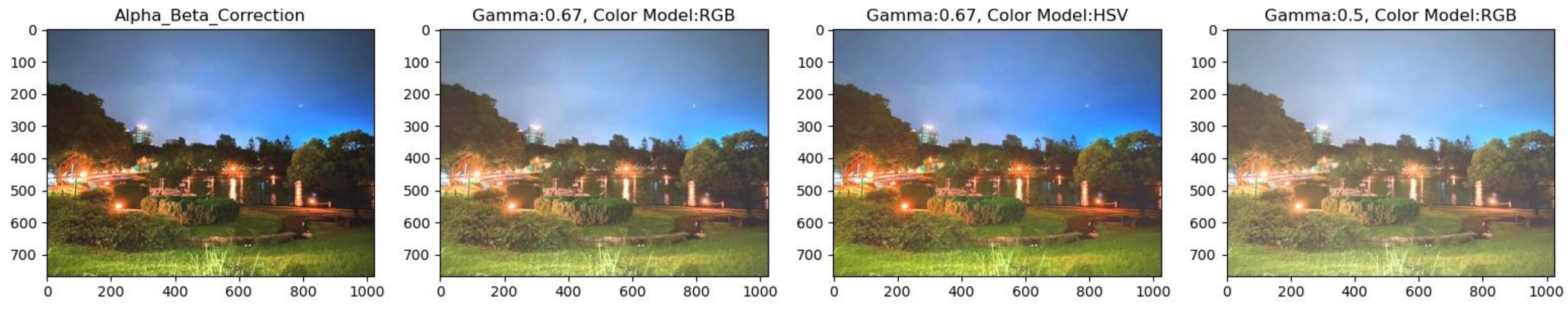
II. Implement Method: Gamma correction, Alpha correction, Beta Correction, RGB & HSV color model processing, Lighting mask

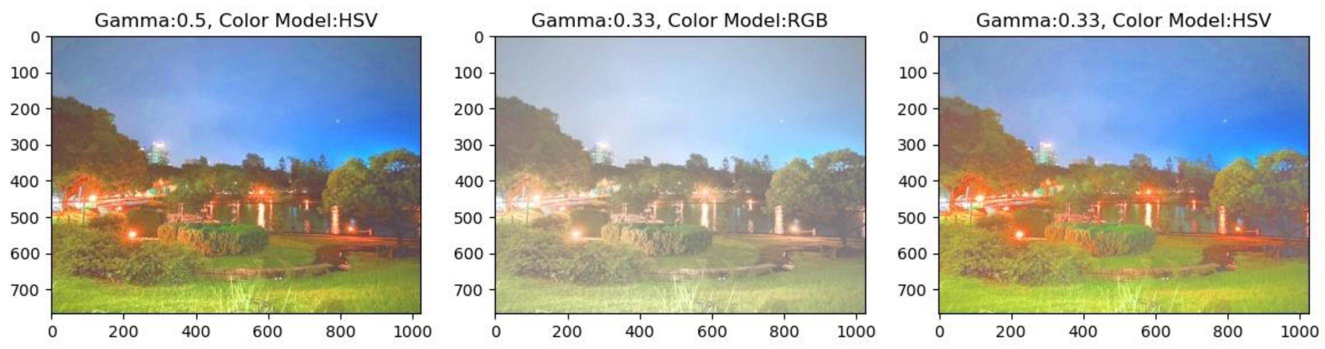
III. Experiments & Results:

1. Experient different settings of Alpha and Beta correction

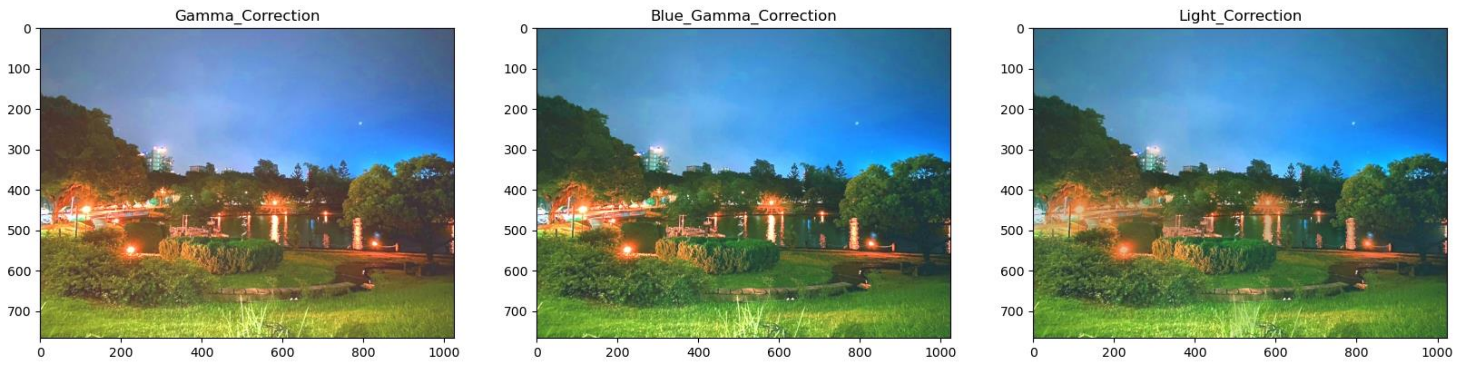


2. Experiment of Gamma correction in different γ and color models (HSV, RGB)


Alpha_Beta_Correction
Gamma:0.67, Color Model:RGB
Gamma:0.67, Color Model:HSV
Gamma:0.5, Color Model:RGB
0
100
200
300
400
500
600
700
800
1000


Gamma:0.5, Color Model:HSV
Gamma:0.33, Color Model:RGB
Gamma:0.33, Color Model:HSV

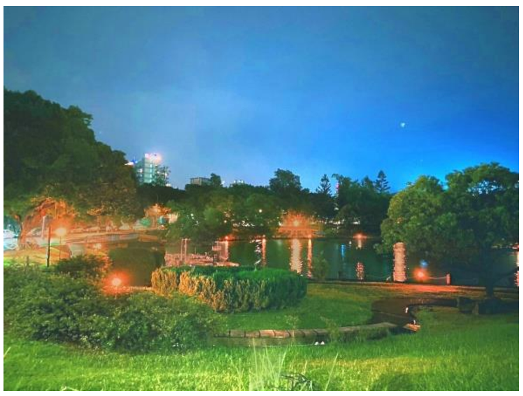
3. Do advance processing (Blue Channel Correction and Light Masking)



IV. Discussions:

As we can see, the original picture is an image with low brightness and low contrast. Hence, we apply alpha and beta correction on the original image. The result shows that the best output image appears at the setting of α= 3 [1.0-3.0], β = 25 [0-100]. The effect of brightness and contrast becomes more significant when alpha and beta go larger. In addition, the effect of applying Gamma Correction in HSV/ RGB color model is showed at experience 2. The exposure becomes higher when the value of γ go larger, the best output image appears at the setting of γ = 0.67 applying in HSV model. Finally, we observe that the red color in the output is a little bit more and the light of the street light is too bright. We apply gamma correction in the blue channel (γ = 1.5) and light correction using mask to let the road light go darker.

Ⅴ. Final Output:



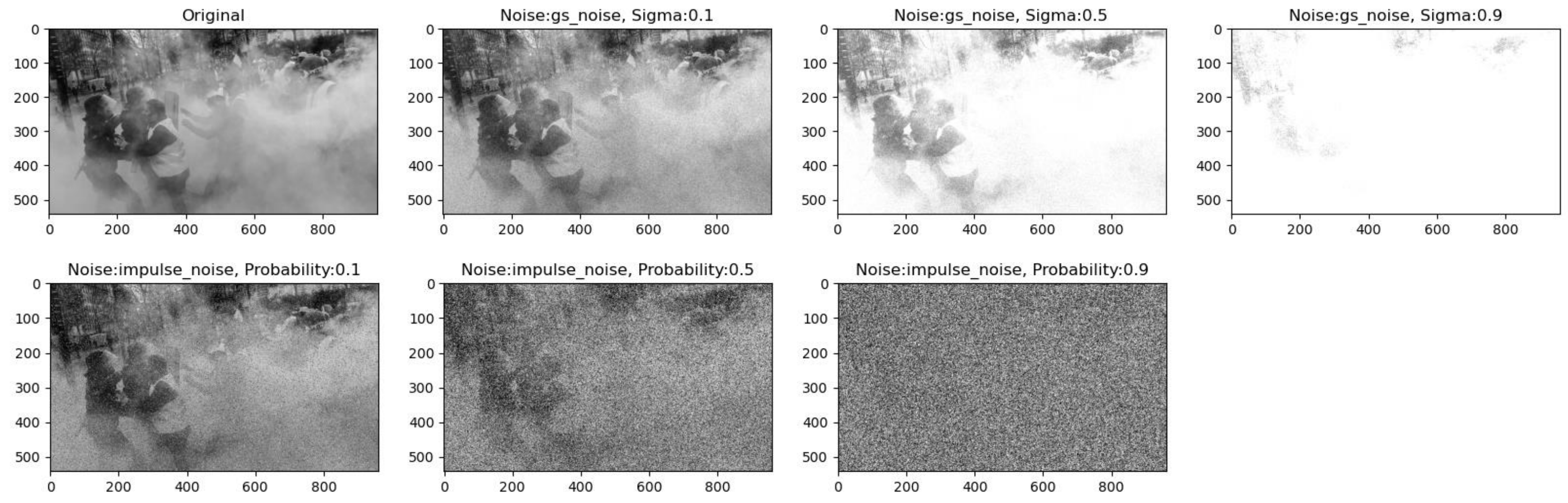
Picture 3:

I. Original Picture:

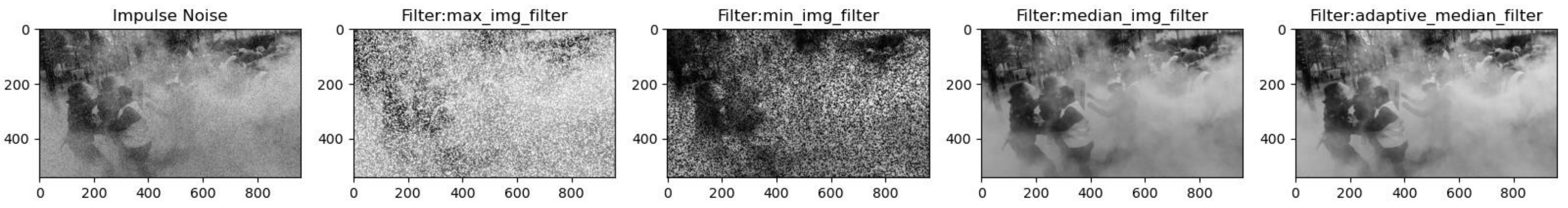
II. Implement Method: Impulse Noise, Gaussian Noise, Max Filter, Median Filter, Min Filter, Adaptive Median Filter

III. Experiments & Results:

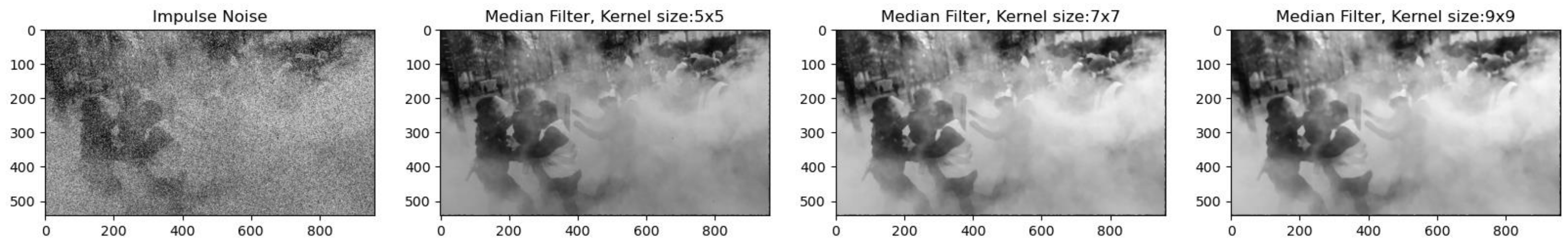
1. Different type of noise with different settings (Gaussian noise, Impulse Noise)

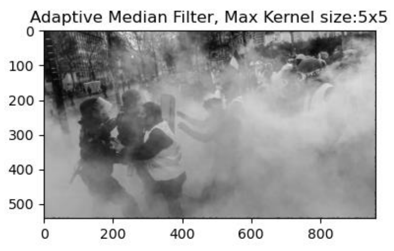

Original
Noise:gs_noise, Sigma:0.1
Noise:gs_noise, Sigma:0.5
Noise:gs_noise, Sigma:0.9
Noise:impulse_noise, Probability:0.1
Noise:impulse_noise, Probability:0.5
Noise:impulse_noise, Probability:0.9

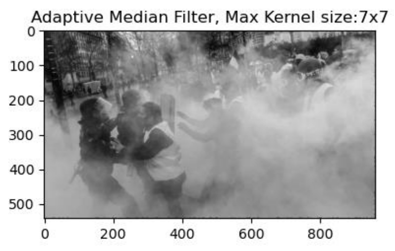
2. Effect of different filters

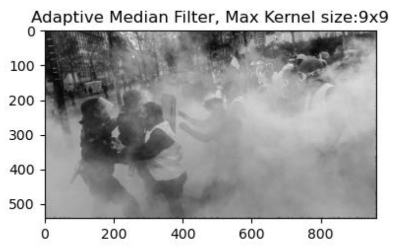


3. Comparison of Median Filter and Adaptive Median Filter


Impulse Noise
Median Filter, Kernel size:5x5
Median Filter, Kernel size:7x7
Median Filter, Kernel size:9x9



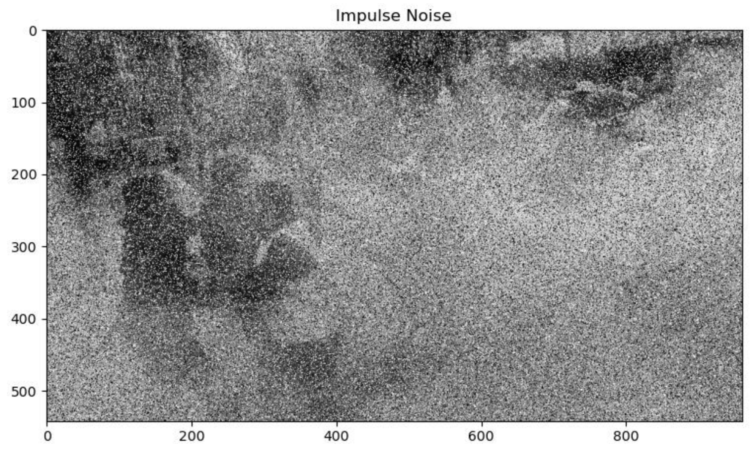


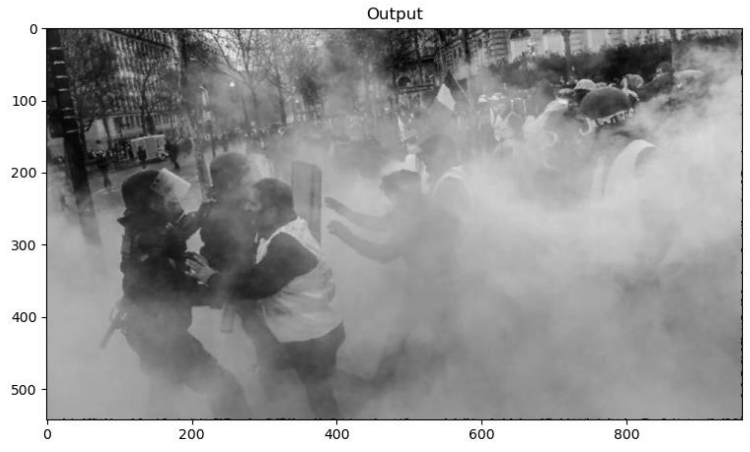


IV. Discussions:

The original picture is an image with gray scale. To test the performance of different filters, we apply gaussian and impulse noise. In experience 1, you can see the different effect of noise by applying invariant settings. The effect of impulse noise is more significant while the probability sets larger. In experience 2, we can observe that median and adaptive median filter have a better result in denoising. Further discussion on both filters, we do experience 3 and apply different settings. The result shows that adaptive median filter is a better approach dealing with the image with high density of impulse noise. The edges of the image after adaptive median filter are more concrete than those in median filter.

V. Final Output:





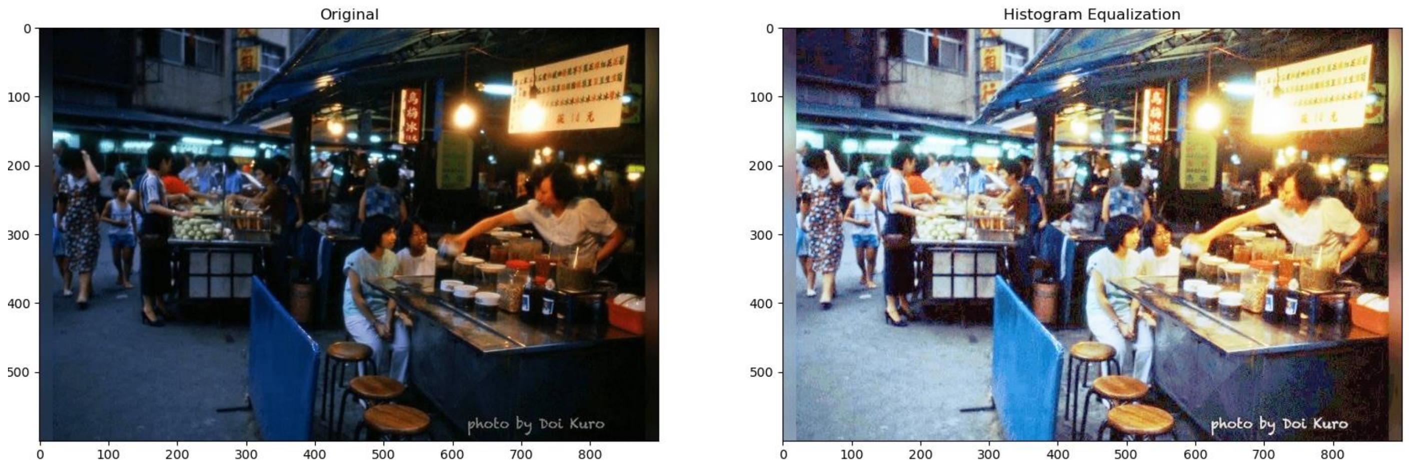
Picture 4:

I. Original Picture:

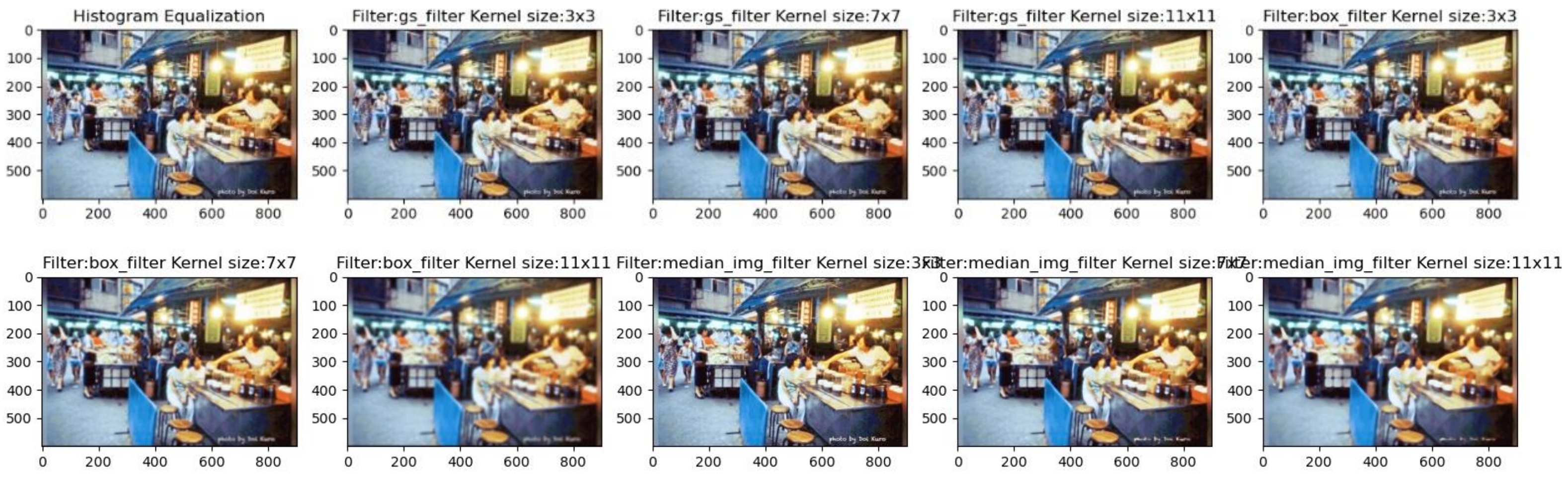
II. Implement Method: Histogram Equalization, Gaussian Filter, Box Filter, Median Filter,

III. Experiments & Results:

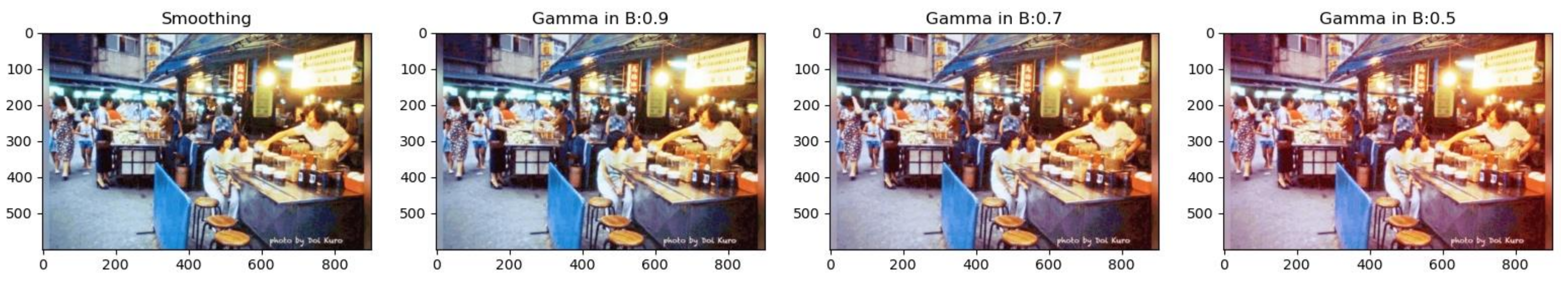
1. Histogram Equalization



2. Smoothing with different filters


Histogram Equalization
Filter:gs_filter Kernel size:3x3
Filter:gs_filter Kernel size:7x7
Filter:gs_filter Kernel size:11x11
Filter:box_filter Kernel size:3x3
Filter:box_filter Kernel size:7x7
Filter:box_filter Kernel size:11x11
Filter:median_img_filter Kernel size:3x3
Filter:median_img_filter Kernel size:7x7
Filter:median_img_filter Kernel size:11x11

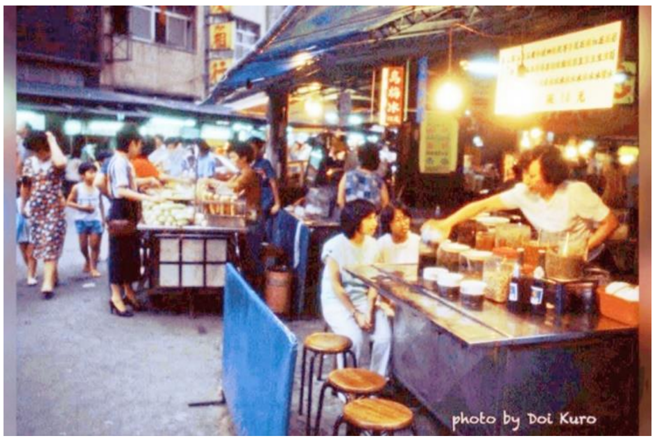
3. Gamma correction in Channel Blue



IV. Discussions:

The original picture is a image that took a long time ago, so you can see that the quality of the image is not so good (including low contrast, blurry image, blue color is too much …). We first apply histogram equalization to let the image distribution be uniformed and get better contrast, as you can see the result in the experience 1. To furthermore denoise the image, we apply smoothing filters in experience 2. The result shows that the image has a better performance on applying Gaussian filter in kernel size of 3*3. The edges become blurred when the kernel size set larger. Finally, we do some advance processing like gamma correction in blue channel to get the output image. The γ value is set as 0.7.

V. Final Output:



Code:

1. preprocessing.py

```
from copy import copy
from pickletools import uint8
import cv2
import argparse
import random
import numpy as geek
import skimage.util.noise as noise
import numpy as np
import matplotlib.pyplot as plt
import math

#Resize
def img_resize(image, resize_height, resize_width):

    image_shape=np.shape(image)
    height=image_shape[0]
    width=image_shape[1]

    if (resize_height is None) and (resize_width is None):
        return image

    #Resize height
    if resize_height is not None:
        resize_width=int(width* (resize_height/height) )
    #Resize width
    elif resize_width is not None:
        resize_height=int(height* (resize_width/width) )

    img = cv2.resize(image, dsize=(resize_width, resize_height))

    return img

#RGB -> Gray
def RGB2GRAY(image):
    return cv2.cvtColor(image , cv2.COLOR_BGR2GRAY)
```

```
# RGB -> HSV
def RGB2HSV(image):
    return cv2.cvtColor(image, cv2.COLOR_BGR2HSV)



#Noise
#Salt and pepper Noise
def impulse_noise(image, prob = 0.5):
    output = image.copy()
    h, w = image.shape[:2] #height, weight
    ps = 1 - prob

    for i in range(h):
        for j in range(w):
            rand_prob = random.random()
            if rand_prob > 0 and rand_prob <= prob/2:      #rdn < prob
                output[i, j] = 0
            elif rand_prob > prob/2 and rand_prob <= prob:        #rand_prob > ps
                output[i, j] = 255
            else:
                output[i, j] = image[i, j]
    return output

#Noise
#Gaussian Noise
def gs_noise(image, mean=0, sigma=0.1):

    # Standardize
    output = image / 255.0
    # gs_noise
    noise = np.random.normal(mean, sigma, image.shape)
    output = output + noise
    # clip 0~1
    output = np.clip(output, 0, 1)
    # float -> int (0~1 -> 0~255)
    output = np.uint8(output*255)

    return output
```

```
#Padding
def padding(image , kernel_size):

    padding_size = (kernel_size[0] // 2, kernel_size[1] // 2)
    height = image.shape[0]
    width = image.shape[1]
    output_h = height + padding_size[0] * 2
    output_w = width + padding_size[1] * 2
    output = np.zeros( (output_h , output_w), dtype="uint8")

    for i in range(height):
        for j in range(width):
            output[i + padding_size[0]][j + padding_size[1]] = image[i][j]

    return output

#Convolution
def conv(image, kernel):

    kernel_size = kernel.shape
    output = np.zeros(image.shape, dtype = "uint8")
    pad_img = padding(image , kernel_size)

    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            temp = []
            result = 0
            temp.append( pad_img[ i:i+kernel_size[0], j:j+kernel_size[1] ] )
            image_data = np.row_stack(temp).flatten()
            kernel_data = np.row_stack(kernel).flatten()

            #Calculate
            for k in range(len(kernel_data)):
                result += image_data[k] * kernel_data[k]
            output[i][j] = result

    return output
```

```
# Sharpen
def Laplacian_filter(image):

    kernel = np.array(
        [[0, -1, 0],
        [-1, 5, -1],
        [0, -1, 0]])

    if len(image.shape) == 2:
        output = conv(image, kernel)
    else :
        image_r = image[:, :, 0]
        image_g = image[:, :, 1]
        image_b = image[:, :, 2]

        output_r = conv(image_r , kernel)
        output_g = conv(image_g , kernel)
        output_b = conv(image_b , kernel)
        output = np.dstack((output_r, output_g, output_b))

    return output

# Sharpen
def sharpen_filter(image):

    kernel = np.array(
        [[-1, -1, -1],
        [-1, 8, -1],
        [-1, -1, -1]])

    if len(image.shape) == 2:
        output = conv(image,kernel)
    else :
        image_r = image[:, :, 0]
        image_g = image[:, :, 1]
        image_b = image[:, :, 2]
        output_r = conv(image_r , kernel)
        output_g = conv(image_g , kernel)
        output_b = conv(image_b , kernel)
        output = np.dstack((output_r, output_g, output_b))

    return output

# Sharpen
def unsharp_masking(image, kernel_size = (3, 3), k = 1):

    mask = image - box_filter(np.copy(image), kernel_size)
    output = image + k*mask

    return output

#Smoothing
#Average
def box_filter(image, kernel_size = (3, 3)):

    all = kernel_size[0]* kernel_size[1]
    kernel = np.array([1]*all).reshape(kernel_size)

    avg_kernel = (1/ all)*kernel

    if len(image.shape) == 2:
        output = conv(image, avg_kernel)
    else :
        image_r = image[:, :, 0]
        image_g = image[:, :, 1]
        image_b = image[:, :, 2]

        output_r = conv(image_r , avg_kernel)
        output_g = conv(image_g , avg_kernel)
        output_b = conv(image_b , avg_kernel)
        output = np.dstack((output_r, output_g, output_b))

    return output
```

```
#Gaussian Filter
def gs_filter(image, kernel_size = (3, 3)):

    #Gassian filter
    x1 = 1 - math.ceil(kernel_size[0] / 2)
    x2 = math.ceil(kernel_size[0] / 2)
    y1 = 1 - math.ceil(kernel_size[1] / 2)
    y2 = math.ceil(kernel_size[1] / 2)

    x, y = np.mgrid[x1:x2, y1:y2]

    gaussian_kernel = np.exp(-(x**2+y**2))

    #Normalization
    gaussian_kernel = gaussian_kernel / gaussian_kernel.sum()

    if len(image.shape) == 2:
        output = conv(image, gaussian_kernel)
    else :
        image_r = image[:, :, 0]
        image_g = image[:, :, 1]
        image_b = image[:, :, 2]

        output_r = conv(image_r , gaussian_kernel)
        output_g = conv(image_g , gaussian_kernel)
        output_b = conv(image_b , gaussian_kernel)
        output = np.dstack((output_r, output_g, output_b))

    return output


#Median filter
def median_img_filter(image, kernel_size = (3, 3)):

    def median_filter(image, kernel_size):
        height = image.shape[0]
        width = image.shape[1]
        output = np.zeros((height, width), dtype = "uint8")

        #Padding
        padding_image = padding(image , kernel_size)

        #Find Median
        for i in range(height):
            for j in range(width):
                temp = []
                temp.append(          padding_image[          i:i+kernel_size[0],
j:j+kernel_size[1] ] )
                #Median
                median = np.median(np.row_stack(temp).flatten())
                output[i][j] = median
        return output

    if len(image.shape) == 2:
        output = median_filter(image, kernel_size)
    else:
        image_r = image[:, :, 0]
        image_g = image[:, :, 1]
        image_b = image[:, :, 2]

        output_r = median_filter(image_r , kernel_size )
        output_g = median_filter(image_g , kernel_size )
        output_b = median_filter(image_b , kernel_size )
        output = np.dstack((output_r, output_g, output_b))

    return output

#Max filter
def max_img_filter(image , kernel_size = (3, 3)):

    def max_filter(image , kernel_size):

        image = np.copy(image)
        o_height = image.shape[0]
        o_width = image.shape[1]
        #Padding
        image = padding(image , kernel_size)

        #Max
        result = np.zeros((o_height, o_width), dtype=np.uint8)
        for i in range(image.shape[0] - kernel_size[0]+1):
            for j in range(image.shape[1] - kernel_size[1]+1):
                result[i, j] = np.max(image[ i:i+kernel_size[0], j:j+kernel_size[1] ])

        return result

    if len(image.shape) == 2:
        output = max_filter(image, kernel_size)
    else:
        image_r = image[:, :, 0]
        image_g = image[:, :, 1]
        image_b = image[:, :, 2]

        output_r = max_filter(image_r , kernel_size )
        output_g = max_filter(image_g , kernel_size )
        output_b = max_filter(image_b , kernel_size )
        output = np.dstack((output_r, output_g, output_b))

    return output

#Min filter
def min_img_filter(image , kernel_size = (3, 3)):

    def min_filter(image , kernel_size):

        image = np.copy(image)
        o_height = image.shape[0]
        o_width = image.shape[1]

        #Padding
        image = padding(image , kernel_size)
        #Min
        result = np.zeros((o_height, o_width), dtype=np.uint8)
        for i in range(image.shape[0] - kernel_size[0]+1):
            for j in range(image.shape[1] - kernel_size[1]+1):
                result[i, j] = np.min(image[ i:i+kernel_size[0], j:j+kernel_size[1] ])

        return result

    if len(image.shape) == 2:
        output = min_filter(image, kernel_size)
    else:
        image_r = image[:, :, 0]
        image_g = image[:, :, 1]
        image_b = image[:, :, 2]

        output_r = min_filter(image_r , kernel_size )
        output_g = min_filter(image_g , kernel_size )
        output_b = min_filter(image_b , kernel_size )
        output = np.dstack((output_r, output_g, output_b))

    return output

# Histogram Equalization
def hist_equa_img(image):

    def hist_equa(image):
        data = np.zeros(256).astype(np.int64)
        image_f = image.flatten()
        for i in image_f:
            data[int(i)] += 1

        p = data/image_f.size
        p_sum = geek.cumsum(p)
        equal = np.around(p_sum * 255).astype('uint8')
        output = equal[image_f].reshape(image.shape)

        return equal[image]
    if len(image.shape) == 2:
        output = hist_equa(image)
    else:
        image_r = image[:, :, 0]
        image_g = image[:, :, 1]
        image_b = image[:, :, 2]

        output_r = hist_equa(image_r )
        output_g = hist_equa(image_g )
        output_b = hist_equa(image_b )
        output = np.dstack((output_r, output_g, output_b))

        return output



# Gamma correction
def gamma_correction(image, r, c=1):
    output = image.copy()
    output = output/ 255
    output = (1/c * output) ** r

    output *= 255
    output = output.astype(np.uint8)

    return output

#RGB Gamma correction
def RGB_gamma_correction(image, r, channel):

    image_r = image[:, :, 0]
    image_g = image[:, :, 1]
    image_b = image[:, :, 2]

    if channel == 'R':
        output_r = gamma_correction(image_r, r)
        output = np.dstack((output_r, image_g, image_b))
    elif channel == 'G':
        output_g = gamma_correction(image_g, r)
        output = np.dstack((image_r, output_g, image_b))
    else:
        output_b = gamma_correction(image_b, r)
        output = np.dstack((image_r, image_g, output_b))

    return output


# Alpha and Beta correction
def alpha_beta_correction(image, a, b):
    output = np.zeros(image.shape, image.dtype)

    # Initialize values
    for y in range(image.shape[0]):
        for x in range(image.shape[1]):
            for c in range(image.shape[2]):
                output[y,x,c] = np.clip(a*image[y,x,c] + b, 0, 255)

    return output


def reduce_highlights(img, criteria = 200, alpha = 0.1, beta = 0.1):

    image_r = img[:, :, 0]
    image_g = img[:, :, 1]
    image_b = img[:, :, 2]

    # img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    ret, thresh = cv2.threshold(image_b, criteria, 255, 0)
    contours, hierarchy    = cv2.findContours(thresh.copy(),cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
    img_zero = np.zeros(img.shape, dtype=np.uint8)

    for contour in contours:
        x, y, w, h = cv2.boundingRect(contour)
        img_zero[y:y+h, x:x+w] = 255
        mask = img_zero
    result = cv2.illuminationChange(img, mask, alpha=alpha, beta=beta)

    return result


def adaptive_median_filter(image, max_size: int=7):

    def zero_padding(src, padding_left: int, padding_right: int, padding_top: int,
padding_bottom: int):

        height, width = src.shape
        # Vertical Zero padding for source
        boundary_top = np.zeros((padding_top, width))
        boundary_bottom = np.zeros((padding_bottom, width))
        result = np.vstack((boundary_top, src, boundary_bottom))
        # Horizontal Zero padding for source
        boundary_left = np.zeros((height+padding_top+padding_bottom,
padding_left))
        boundary_right = np.zeros((height+padding_top+padding_bottom,
padding_right))
        result = np.hstack((boundary_left, result, boundary_right))

        return result

    # For Gray image
    assert max_size % 2 ==1, 'kernel size must be odd.'
    image = np.copy(image)
    height, width = image.shape
    kernel_h, kernel_w = (max_size-1)//2, (max_size-1)//2

    image = zero_padding(image, kernel_w, kernel_w, kernel_h, kernel_h)

    #Padding
    # kernel_size = (kernel_h, kernel_w)
    # image = padding(image , kernel_size)

    filter_size = 1
    result = np.zeros(image.shape, dtype=np.uint8)
    for i in range(kernel_h, height+kernel_h):
        for j in range(kernel_w, width+kernel_w):
            filter_size = 1
            while filter_size <= kernel_w:
                local_med = np.median(image[i-filter_size:i+filter_size+1, j-filter_size:j+filter_size+1])
                local_max = np.max(image[i-filter_size:i+filter_size+1, j-filter_size:j+filter_size+1])
                local_min = np.min(image[i-filter_size:i+filter_size+1, j-filter_size:j+filter_size+1])

                if local_med==local_max or local_med==local_min:
                    result[i, j] = local_med
                    filter_size += 1
                elif image[i, j]==local_max or image[i, j]==local_min:
                        result[i, j] = local_med
                        break
                else:
                    result[i, j] = image[i, j]
                    break

    result = result[kernel_h:height+kernel_h, kernel_w:width+kernel_w]
    return result
```

2. HW1.py

```
from concurrent.futures.process import _MAX_WINDOWS_WORKERS
from re import I
from PIL import Image
from PIL.Image import core as _imaging
import os
import sys
import time
import numpy as np
import cv2
import sys
import math
from copy import copy

import random
import skimage.util.noise as noise
import matplotlib.pyplot as plt
from matplotlib.cbook import get_sample_data
import matplotlib.pyplot as plt
from sklearn.cluster import k_means

from os import listdir
from os.path import isfile, join
import numpy
from processing import*

def show_img(img):
        plt.figure(figsize=(15,15))
        image_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        plt.imshow(image_rgb)
        plt.show()

# def show_all_img(result, columns = None, rows = 1):

#       if columns is None:
#               columns = len(result)
#       fig = plt.figure(figsize=(20, 20))
#        # fig.suptitle('Median Filter with different kernel sizes')
#        # plt.xlabel('Median Filter', fontweight='bold')
#        for i, img_n in enumerate(result):
#            ax = fig.add_subplot(rows, columns, i+1)
#            ax.title.set_text(img_n)
#            # ax.set_title(i)
#            img = result[img_n]
#            if len(img.shape) == 2:
#                plt.imshow(img, cmap='gray')
#            else:
#                img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
#                plt.imshow(img)
#        plt.show()

def show_all_img(result, columns = None, rows = 1):

    if columns is None:
        columns = len(result)
    fig = plt.figure(figsize=(20, 20))

    for i, img_n in enumerate(result):
        ax = fig.add_subplot(rows, columns, i+1)
        ax.title.set_text(img_n)
        # ax.set_title(i)
        img = result[img_n]
        if len(img.shape) == 2:
            plt.imshow(img, cmap='gray')
        else:
            img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
            plt.imshow(img)
    plt.show()



#All files
img_file = 'example'
onlyfiles = [f for f in listdir(img_file) if isfile(join(img_file, f))]
print(onlyfiles)
```

```
# Read Image
img1 = cv2.imread(os.path.join(img_file, onlyfiles[0]))
img2 = cv2.imread(os.path.join(img_file, onlyfiles[1]))
img3 = cv2.imread(os.path.join(img_file, onlyfiles[2]))
img4 = cv2.imread(os.path.join(img_file, onlyfiles[3]))
img5 = cv2.imread(os.path.join(img_file, onlyfiles[4]))
img6 = cv2.imread(os.path.join(img_file, onlyfiles[5]))

#Image 1 - Newspaper woman
input = img5
print(input.shape)
input = img_resize(input, None, 512)

# output
o_input = input
g_input = RGB2GRAY(input)

#Processing
result    = {}
result['Original'] = g_input

#test 0
filters = [box_filter, median_img_filter]
k_sizes = (7, 9, 11)

for filter in filters:
    for k_size in k_sizes:
        kernel_size = (k_size, k_size)
        print('filter:', filter, ' kernel size: ', kernel_size)
        img_sm = filter(g_input, kernel_size=kernel_size)
        filter_name = str(filter).split()[1]
        name = 'Kernel size:{}x{}, Filter:{}'.format(k_size, k_size, filter_name)
        result[name] = img_sm

show_all_img(result, columns = 4, rows = 2)

#test 1
kernel_test = (3, 5, 7, 9, 11)
```

```
for k in kernel_test:
    kernel_size = (k, k)
    img_ft = median_img_filter(g_input, kernel_size=kernel_size)
    result['Kernel size:{}x{}'.format(k, k)] = img_ft

show_all_img(result)

#test 2
result2 = {}
sharpen_filters = [unsharp_masking, sharpen_filter, Laplacian_filter]
img_ft = median_img_filter(g_input, kernel_size = (11, 11))
result2['Median Filter'] = img_ft
for m in sharpen_filters:
    img_sp = m(img_ft)
    result2[str(m).split()[1]] = img_sp

show_all_img(result2)

#test 3
result3 = {}
ks = [1, 10, 20]
k_sizes = [7, 9, 11]
img_ft = median_img_filter(g_input, kernel_size = (11, 11))
result3['Median Filter'] = img_ft
for k in ks:
    for k_size in k_sizes:
        kernel_size = (k_size, k_size)
        print('k:', k, ' kernel size: ', kernel_size)
        img_sp = unsharp_masking(img_ft, kernel_size=(kernel_size), k = k)
        name = 'Kernel size:{}x{}, k:{}'.format(k_size, k_size, k)
        result3[name] = img_sp

show_all_img(result3, 5, 2)
```

Final of P1

```
final_result = {}
img_ft = median_img_filter(g_input, kernel_size = (11, 11))
img_sp = unsharp_masking(img_ft, kernel_size=(7, 7), k = 1)
```

```
final_result['result'] = img_sp
show_all_img(final_result)

##################################################
#Image 2 - NCTU
input = img2
print(input.shape)
input = img_resize(input, None, 1024)

# output
o_input = input
# convert img to gray
g_input = RGB2GRAY(input)

#Processing
result    = {}
result['Original'] = o_input

#test 1
r = 0.5
model = 'HSV'
hue, sat, val = cv2.split(RGB2HSV(image=o_input))
val_gamma = gamma_correction(val, 0.5)
hsv_gamma = cv2.merge([hue, sat, val_gamma])
img_r = cv2.cvtColor(hsv_gamma, cv2.COLOR_HSV2BGR)
result2['Gamma:{}, Color Model:{}'.format(r, model)] = img_r

alpha = [1, 2, 3]
beta = [25, 50, 75]
for a in alpha:
        for b in beta:
                print('Alpha: ', a, 'Beta: ', b)
                img_ab = alpha_beta_correction(o_input, a, b)
                result['Alpha:{}, Beta:{}'.format(a, b)] = img_ab

show_all_img(result, 5, 2)

#test 2    Gamma, Color model
```

```
result2 = {}
#alpha 3, beta 25
#r = 0.67, model = HSV
a = 3
b = 25
img_ab = alpha_beta_correction(o_input, a, b)
result2["Alpha_Beta_Correction"] = img_ab
rs = [0.67, 0.5, 0.33]
models = ['RGB', 'HSV']

for r in rs:
    for model in models:
        if model == 'RGB':
            img_r = gamma_correction(img_ab, r)
            result2['Gamma:{}, Color Model:{}'.format(r, model)] = img_r
        else:
            hue, sat, val = cv2.split(RGB2HSV(image=img_ab))
            val_gamma = gamma_correction(val, r)
            hsv_gamma = cv2.merge([hue, sat, val_gamma])
            img_r = cv2.cvtColor(hsv_gamma, cv2.COLOR_HSV2BGR)
            result2['Gamma:{}, Color Model:{}'.format(r, model)] = img_r

show_all_img(result2, 4, 2)

#test 3
result3 = {}
#alpha 3, beta 25
#r = 0.67, model = HSV
a = 3
b = 25
r = 0.67
img_ab = alpha_beta_correction(o_input, a, b)
hue, sat, val = cv2.split(RGB2HSV(image=img_ab))
val_gamma = gamma_correction(val, r)
hsv_gamma = cv2.merge([hue, sat, val_gamma])
img_r = cv2.cvtColor(hsv_gamma, cv2.COLOR_HSV2BGR)

img_br = RGB_gamma_correction(img_r, r = 1.5, channel = 'B')
```

```
img_light = reduce_highlights(img_br, criteria = 253, alpha=0.12, beta = 0.12)


#result
result3["Gamma_Correction"] = img_r
result3["Blue_Gamma_Correction"] = img_br
result3["Light_Correction"] = img_light

show_all_img(result3)

#Final of P1
final_result = {}
result3 = {}
#alpha 3, beta 25
#r = 0.67, model = HSV
a = 3
b = 25
r = 0.67
img_ab = alpha_beta_correction(o_input, a, b)
hue, sat, val = cv2.split(RGB2HSV(image=img_ab))
val_gamma = gamma_correction(val, r)
hsv_gamma = cv2.merge([hue, sat, val_gamma])
img_r = cv2.cvtColor(hsv_gamma, cv2.COLOR_HSV2BGR)

img_br = RGB_gamma_correction(img_r, r = 1.5, channel = 'B')
img_light = reduce_highlights(img_br, criteria = 253, alpha=0.12, beta = 0.12)

final_result['result'] = img_light
show_all_img(final_result)

##################################################
#Image 3 -
input = img5
print(input.shape)
input = img_resize(input, None, 960)

# output
o_input = input
```

```
# convert img to gray
g_input = RGB2GRAY(input)

#Processing
result    = {}
result['Original'] = g_input

#test 1
noises = [gs_noise, impulse_noise]

for n in noises:
    if n == gs_noise:
        sigma = [0.1, 0.5, 0.9]
        for s in sigma:
            img_n = n(g_input, s)
            noise_name = str(n).split()[1]
            name = 'Noise:{}, Sigma:{}'.format(noise_name, s)
            result[name] = img_n
    else:
        prob = [0.1, 0.5, 0.9]
        for p in prob:
            img_n = n(g_input, p)
            noise_name = str(n).split()[1]
            name = 'Noise:{}, Probability:{}'.format(noise_name, p)
            result[name] = img_n

show_all_img(result, 4, 2)

#test 2
result2 = {}
img_n = impulse_noise(g_input, 0.1)
result2['Impulse Noise'] = img_n

filters = [max_img_filter, min_img_filter, median_img_filter, adaptive_median_filter]

for filter in filters:
    img_filter = filter(img_n)
    filter_name = str(filter).split()[1]
    name = 'Filter:{}'.format(filter_name)
    result2[name] = img_filter

show_all_img(result2, 5, 1)

#test 3
result3 = {}
img_n = impulse_noise(g_input, 0.3)
result3['Impulse Noise'] = img_n

kernel_size = [5, 7, 9]

#Median
for k in kernel_size:
    print('filter: Median Filter, kernel size: ', k)
    k_size = (k, k)
    img_ft = median_img_filter(img_n, k_size)
    name = 'Median Filter, Kernel size:{}x{}'.format(k, k)
    result3[name] = img_ft

#Adaptive Median
for k in kernel_size:
    print('Adaptive Median Filter, kernel size: ', k)
    k_size = 2*k+1
    img_ft = adaptive_median_filter(img_n, k_size)
    name = 'Adaptive Median Filter, Max Kernel size:{}x{}'.format(k, k)
    result3[name] = img_ft

show_all_img(result3, 4, 2)

#final
final_result = {}
img_n = impulse_noise(g_input, 0.3)
final_result['Impulse Noise'] = img_n

kernel_size = [5, 7, 9]

#Adaptive Median
```

```
k = 7
k_size = 2*k+1
img_ft = adaptive_median_filter(img_n, k_size)
name = 'Output'
final_result[name] = img_ft

show_all_img(final_result)

#################################################
#Image 4 -
input = img3
print(input.shape)
input = img_resize(input, None, 900)

# output
o_input = input
# convert img to gray
g_input = RGB2GRAY(input)

#Processing
result    = {}
result['Original'] = o_input

#test 1 histogram
img_his = hist_equa_img(o_input)
result['Histogram Equalization'] = img_his

show_all_img(result)

#test 2
result2 = {}
result2['Histogram Equalization'] = img_his
filters = [gs_filter, box_filter, median_img_filter]
kernel_size = [3, 5, 7]

for filter in filters:
        for k in kernel_size:
                print(filter, k)
        k_size = (k, k)
        img_ft = filter(img_his, k_size)
        filter_name = str(filter).split()[1]
        name = 'Filter:{} Kernel size:{}x{}'.format(filter_name, k, k)
        result2[name] = img_ft

show_all_img(result2, 5, 2)

#test 3
result3 = {}
sharpen_filters = [unsharp_masking, sharpen_filter, Laplacian_filter]
img_ft = gs_filter(img_his)
result3['Smoothing'] = img_ft
for m in sharpen_filters:
    print(m)
    img_sp = m(img_ft)
    result3[str(m).split()[1]] = img_sp

show_all_img(result3)

#test 4
result4 = {}
#input
result4['Smoothing'] = img_ft
rs = [0.9, 0.7, 0.5]
for r in rs:
    img_br = RGB_gamma_correction(img_ft, r = r, channel = 'B')
    result4['Gamma in B:{}'.format(r)] = img_br

show_all_img(result4)

#Final
final_result = {}
img_his = hist_equa_img(o_input)
img_ft = gs_filter(img_his)
img_br = RGB_gamma_correction(img_ft, r = 0.7, channel = 'B')
final_result['output'] = img_br
```

```
show_all_img(final_result)
```