EE569 HW1

Problem1 Image Demosaicing and Histogram Manipulation

Motivation:

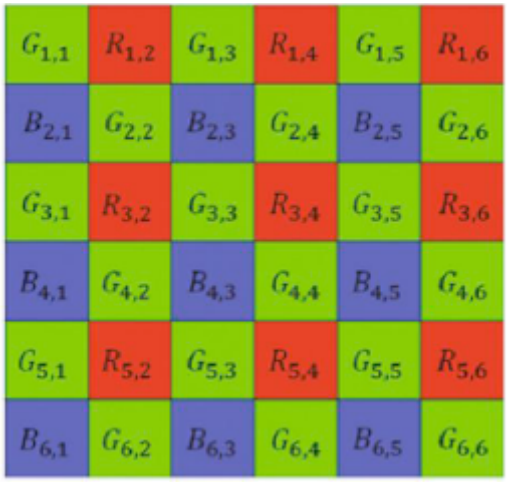
Image demosaicing part introduce us about Bayer sensor, and let us practice how to deal with data from this sensor to show a colored picture.

Histogram manipulation part basically is used for adjusting the contrast. There're two basic methods, transfer function and bucket filling. Beyond that, there's a smarter method called CLAHE that can control the extent of contrast manipulation in different part of the image.

Procedures and results

1.a

In this part I applied bilinear interpolation based on this diagram. Originally there's only one of the three channel information on each pixels. After the bilinear interpolation, each pixel contains all three channel information and can display colour.



The size of the image is built in to the code of each sub-question, because OpenCV is not allowed in this problem. But this is not good for potability, which means you may need to edit the code every time you change a picture.

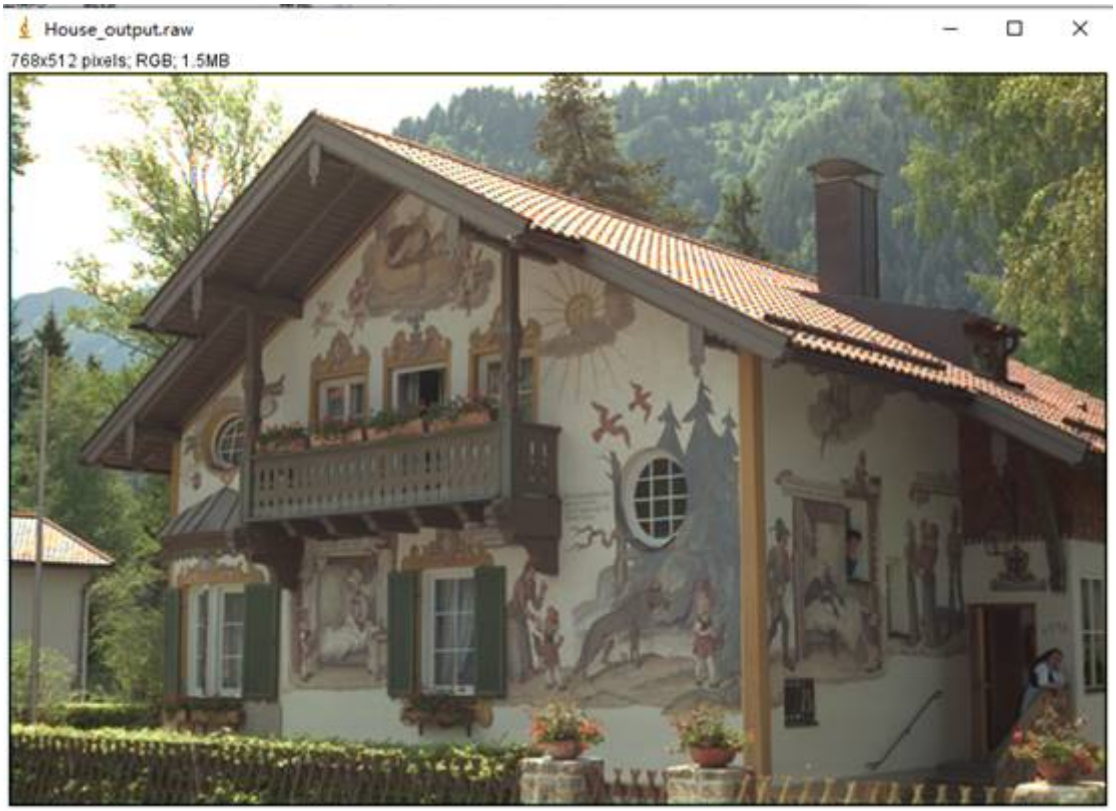
What need to be mentioned is that I did not use bilinear demosaicing on all the pixels on the edges and corners, because they lack some adjacent pixels. That is to say I did not do any padding and just leave the edges unchanged. This rule is applied for all the three problem in this homework.

cammand:

g++ -o test BilinearDemosaicing.cpp

test images/House.raw output/House_output.raw

Output:House_output.raw



1.b

1.b.1

First I visualize the pixel information on a histogram. I output the intensity distribution data to a txt file, and then use python to plot.

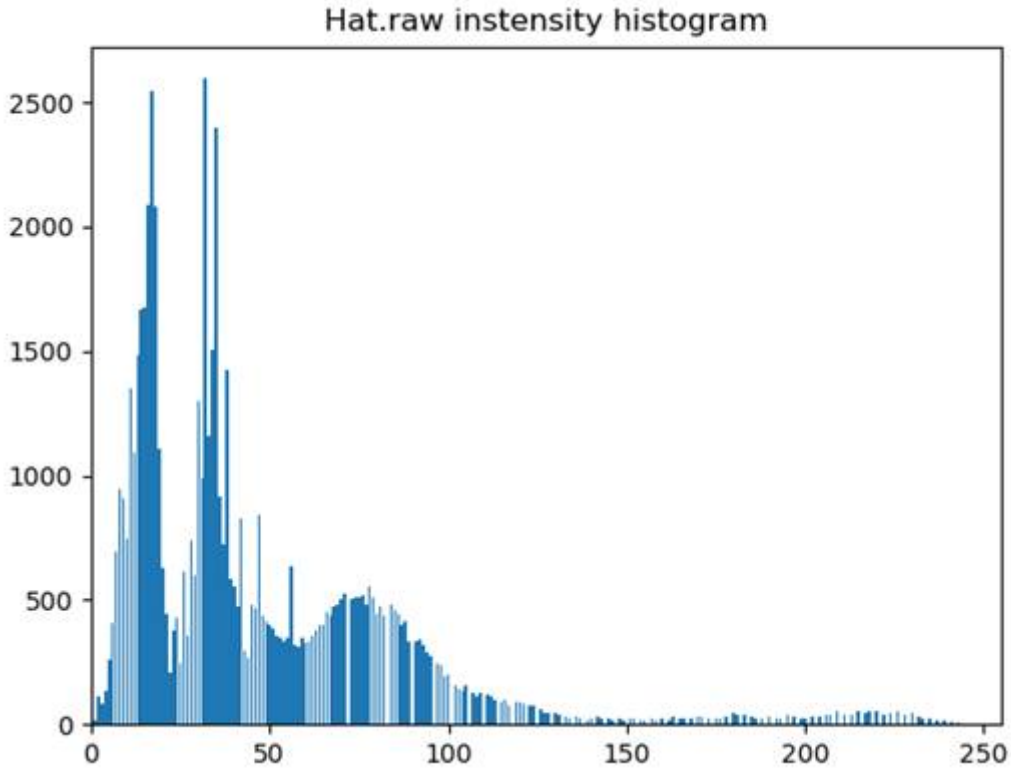
The code is in HistogramMani1.cpp (but has been partially commented for further use) and plot Histogram.py (also has been partially commented)

command:

g++ -o HistogramMani1 HistogramMani1.cpp

python3 plot Histogram.py

There seems some strips on the figure, where somewhere is dense and somewhere is sparse. But it is due to the plotting function, and has nothing to do with the image itself.

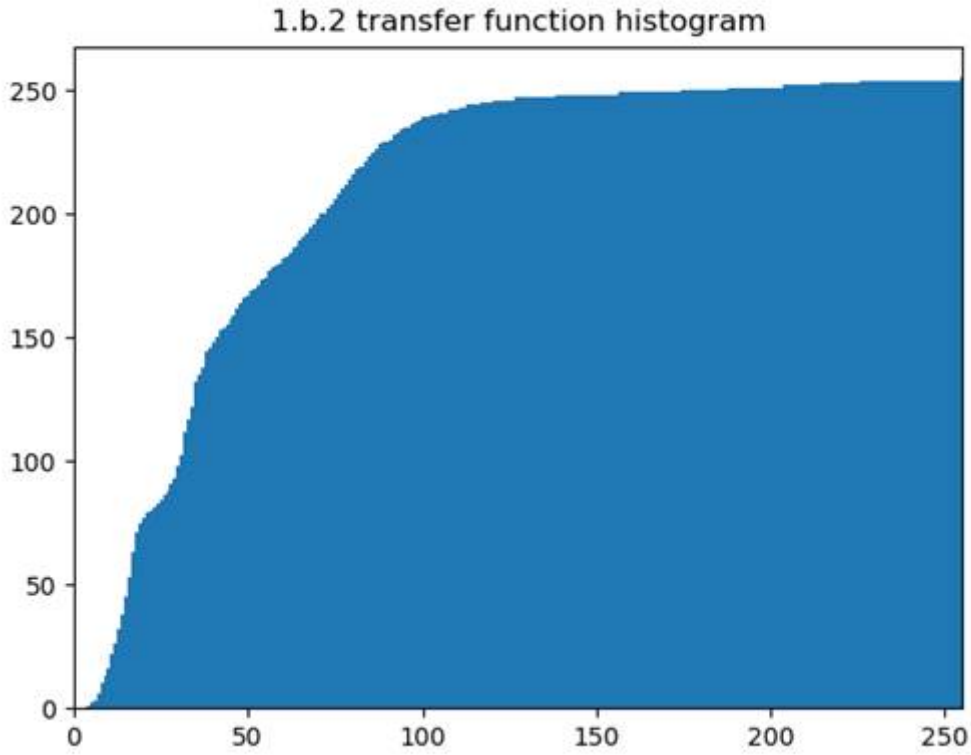


1.b.2

In this part I applied transfer-function-based histogram equalization method.

When doing the transfer function based histogram equalization and mapping original CDF to the new uniform CDF, I do floor rounding. That is, if Prob(x)*255 is not an integer, we round it down.

The transfer function histogram is generated as before (output to a txt and use pyplot). The txt outputing code blocks has been commented.



command:

g++ -o HistogramMani1 HistogramMani1.cpp

HistogramMani1 images/Hat.raw

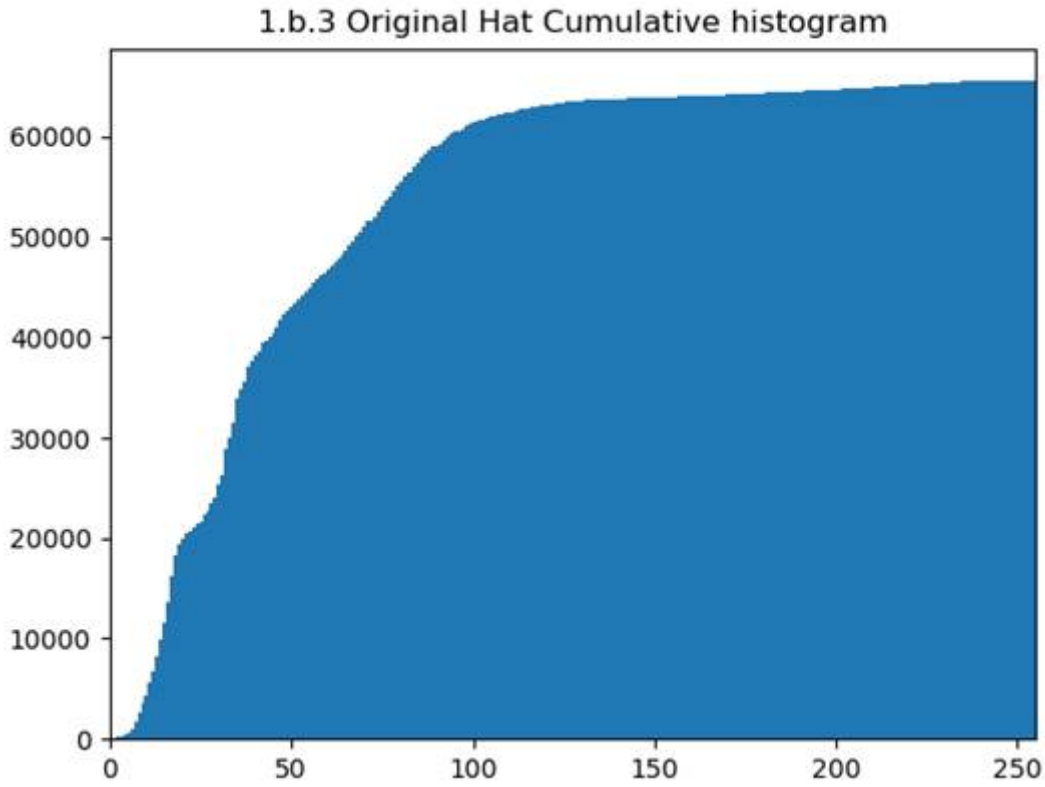
** Output:MethodAEnhanced.raw*

1.b.3

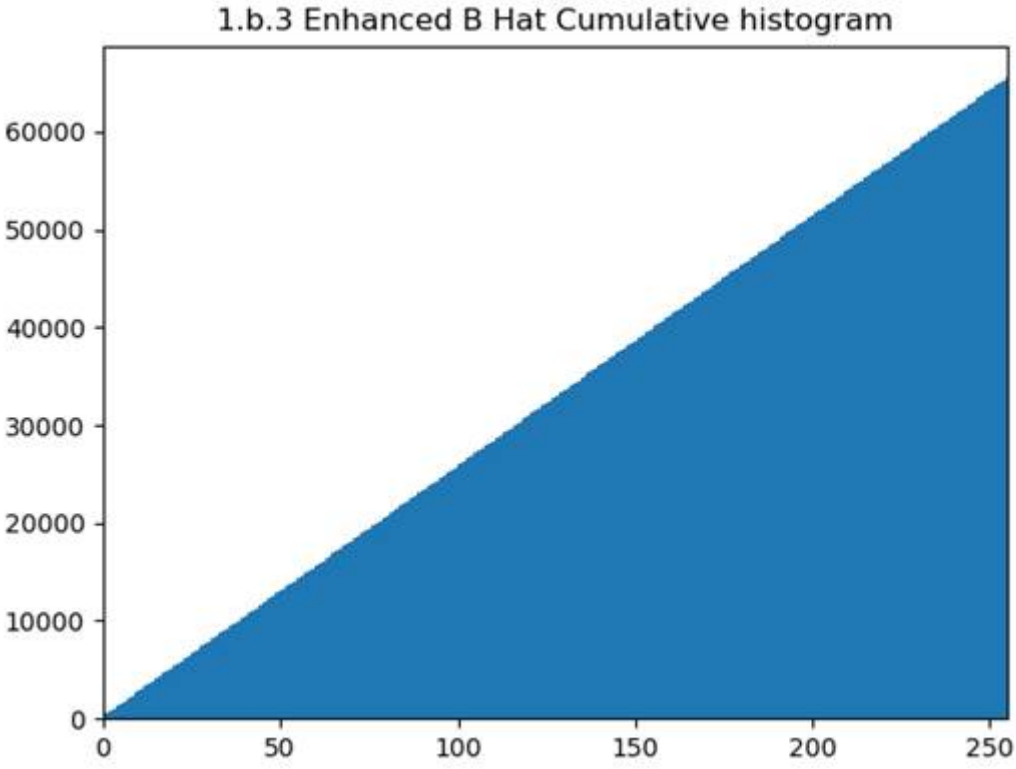
In this part I applied cumulative-probability-based histogram equalization method.

When filling the buckets, pixels values are assigned sequentially, not randomly. But the difference will not be distinguishable to eyes.

The original pixel intensity cumulative distribution seems like this



And after method B enhancement, it becomes exactly uniform.



Command:

g++ -o HistogramMani2 HistogramMani2.cpp

HistogramMani2 images/Hat.raw

Output:MethodBEnhanced.raw

1.b.4

Comparing the two method, I may suppose method B is better, or in other words, better at equalization. For method A manipulated images, there may be 512 pixels with intensity 0, 0 pixels in intensity 1 (be skipped). In contrast, method B manipulated images will have exactly same number of pixels in every intensity value.

1.c

1.c.1

My comprehension about CLAHE:

CLAHE is an advanced edition of AHE. Adaptive histogram equalization does not enhance the global contrast anymore, it only enhance the local contrast like a 8*8 square by default. But if there is a relatively uniform area, it will also be forced to stretch to the whole intensity range, that will enhance the noise. CLAHE limit the local contrast, by setting a threshold for pixels to be equalized. To be intuitive, it only equalized those of pixels that exceed the threshold in the histogram. It is equivalent to limit the slope of transfer function, since the local slope is just the amount of pixel with specific intensity.

1.c.2.1

In this part we first convert image to YUV domain, then apply transfer-function-based histogram equalization method, finally return to RGB domain.

command:

g++ -o clahe CLAHE.cpp

clahe images/Taj_Mahal.raw output/TajMani1.raw

Output:TajMani1.raw

1.c.2.2

This part does the same thing with last one, but we use cumulative-probability-based histogram equalization method.

Since 240000 can not be divide into integer by 256, we round it to the nearest integer, and set it to be the numbers of pixels in each buckets. That maybe 937 or 938

Command:

g++ -o clahe2 CLAHE2.cpp

clahe2 images/Taj_Mahal.raw output/TajMani2.raw

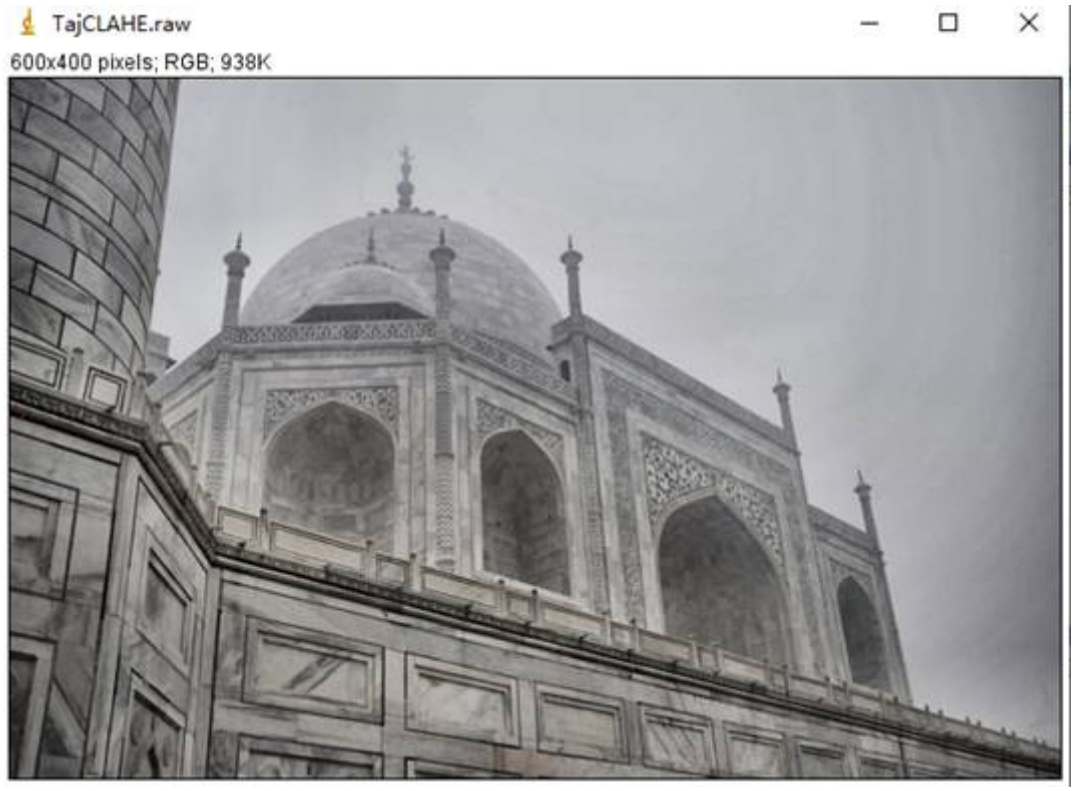
Output: TajMani2.raw

1.c.3

In this part we use CLAHE and see how better it does than HE.

After several attempt, I shall say the best cliplimit is between 2 and 3, so I set 2.5. That means the slope for the transfer function will not exceed 2.5. And for the gridsize, the default value 8*8 seems the best.

My g++ cannot find OpenCV library, so I can only run it in Clion. No recommended command.



1.c.4

Comparing CLAHE and AHE, it's obvious that CLAHE does better than AHE. In AHE-manipulated figures, there're parts where it's too dark or too bright, while CLAHE picture looks uniform and clear everywhere.

Discussion

1.Bayer filter is widely applied in camera, but the pattern used in this homework is not the only pattern. There're many other arrangement for the RGB sensors. So be careful with which channel the raw value belongs to.

2.Typically a C++ program on Windows will provide 2MB stack memory, which is not that much for storing multiple picture. During my implementation, I found the stack will overflow if I read two 768 * 512 * 3 image without dynamic memory allocation. Therefore, in following part, I use 'new' to allocated memory for the three dimensional array.

```
    // Allocate image data array
    unsigned char ***Imagedata;
    Imagedata=new unsigned char **[height];
    int i,j,k;
    int cnt=0;
    for(i=0; i<height; i++)
    {
        Imagedata[i]=new unsigned char *[width];
        for(j=0; j<width; j++) {
            Imagedata[i][j] = new unsigned char[BytesPerPixel];
            for (k = 0; k < BytesPerPixel; k++) {
                file>>noskipws>>Imagedata[i][j][k];
            }
        }
    }

   ////////////////////////////////free the memory
    for(i=0; i<height; i++)
    {
        for(j=0; j<width; j++)
        {
            delete [] Imagedata[i][j];
        }
    }
    for(i=0; i<height; i++)
    {
        delete [] Imagedata[i];
    }
    delete [] Imagedata;
```

A resultant problem caused by dynamic memory is that 'fread' no longer works. Maybe it required a continuous memory as destination, while 'new' doesn't necessarily gives that kind piece of space. Therefore, I switched to fstream. When using fstream, one thing you should pay attention to is, our raw image is binary, so you must use `ios::binary`. Moreover, the fstream flow will basically skip space, whose ascii value is 32. So when doing stream input and output, you must add the keyword `noskipws`.

#####

3.The image is processed row by row, and the number of row is [height]. That means the array should be `Imagedata[height][width][BytesPerPixel]` in stead of `Imagedata[width][height][BytesPerPixel]`

4.When converting from YUV to RGB with the equations given, it may exceed the range [0,255]. For value <0 we turn it to 0, for >255, we turn it to 255. An interesting thing is the equation provided in the homework material is not the only equation. There're some other RGB-YUV convertion equations for the sake of fast computation or other reasons.

Problem 2 Image Denoising

Motivation

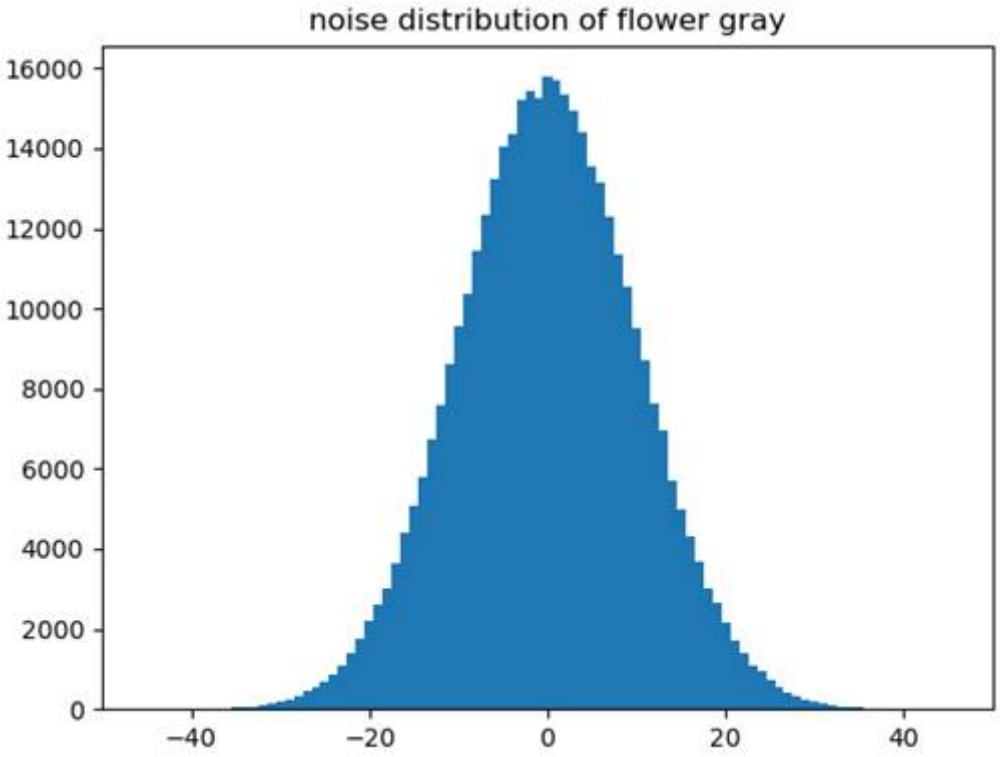
Noise is the most common problem in real world due to the nature, equipment, or data transmission. In this part we learn about two typical noise in digital images: Gaussian and impluse, and how to reduce them. We first deal with Gaussian noise in gray image with gaussian filter, bilateral filter, and non-local mean filter. Then we try to deal with mixed noise in color image.

Procedure and results

2.a

2.a.1

First we try to anaylze what type of noise it is in Flower_gray_noisy.raw. I use the noisy picture to substract source picture, and get the distribution of noise. It's possibly gaussian. And in addition, when looking at the pixels, there's no obvious salt noise or pepper noise, so I suppose there's no impulse noise. The noise in *Flower_gray_noisy.raw* is gaussian



command:

g++ -o ImageDeno ImageDeno.cpp

ImageDeno

Output:*Flower_gray_noise_Histo.txt*

Plot with Python plotHistogram.py

2.a.2

In this part, we applied linear filter(mean and gaussian) to denoise. Also, we compare the results of 3 * 3 and 5* 5 filter size. We evaluated the goodness of the result by peak-signal-to-noise-ratio (PSNR), which represent how much noise there is in a image.

For all the filters, we round the modified value to the closest integer. And with regard to those pixels on the edge, which cannot be set as the center of a filter, we just leave them unchanged.

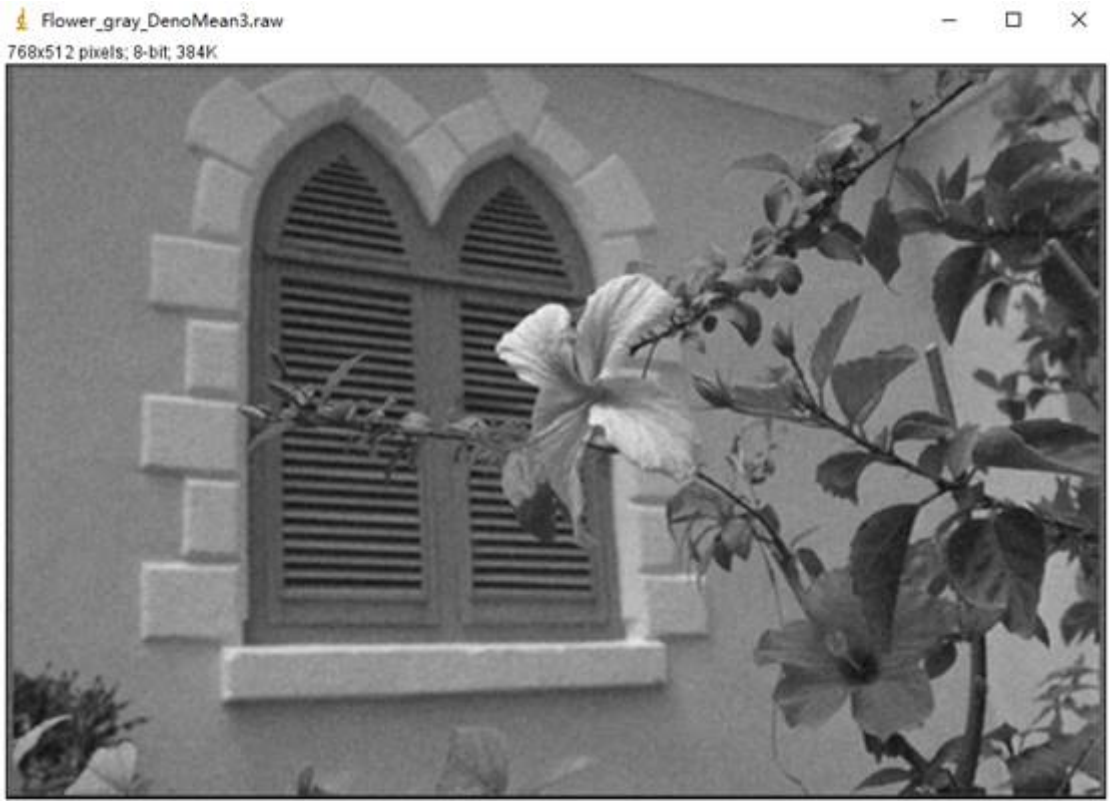
For mean filter, with 3*3 filter size, we got the following output with following command:

g++ -o ImgDenoMean ImgDenoMean.cpp

ImgDenoMean images/Flower_gray_noisy.raw output/Flower_gray_DenoMean3.raw 3

Output(PSNR): 31.3409

Flower_gray_DenoMean3.raw



For mean filter, with 5*5 filter size, we got the following output with following command:

g++ -o ImgDenoMean ImgDenoMean.cpp

ImgDenoMean images/Flower_gray_noisy.raw output/Flower_gray_DenoMean5.raw 5

Output(PSNR): 28.1684

Flower_gray_DenoMean5.raw

For the guassian filters, we assume the standard deviation is 1.

With 3*3 filter size, we got the following output with following command:

g++ -o ImgDenoGauss ImgDenoGauss.cpp

ImgDenoGauss images/Flower_gray_noisy.raw output/Flower_gray_DenoGauss3.raw 3

Output(PSNR): 32.426

Flower_gray_DenoGauss3.raw

This is the applied filter

$$\frac{1}{16} \times \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

With 5*5 filter size, we got the following output with following command:

g++ -o ImgDenoGauss ImgDenoGauss.cpp

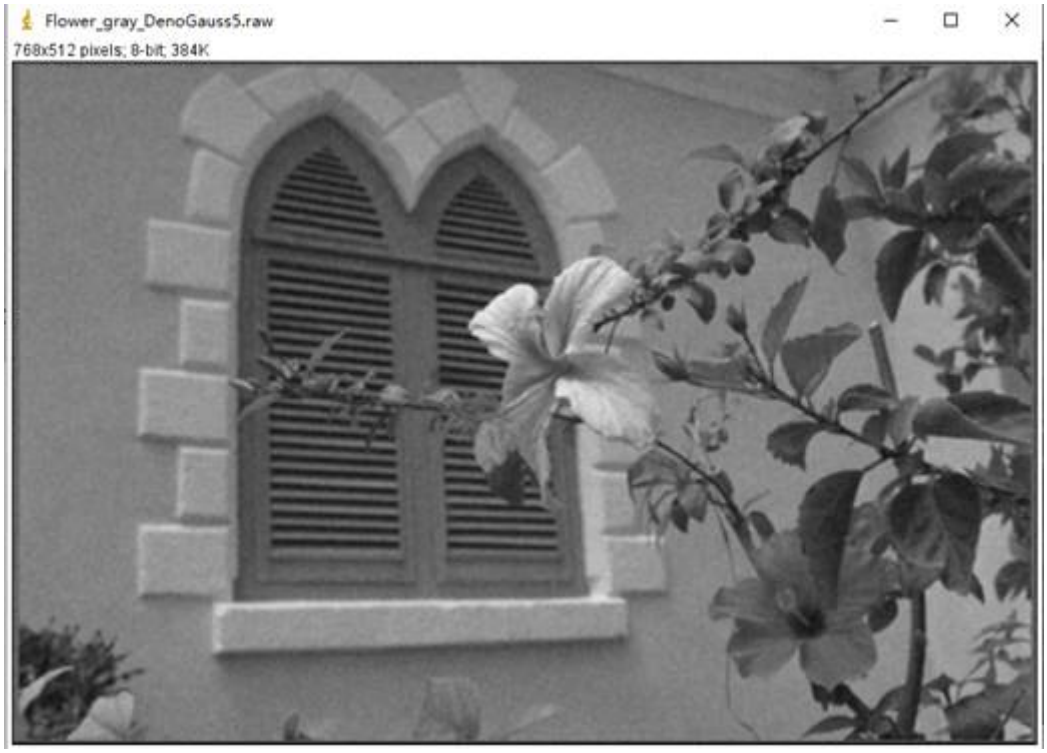
ImgDenoGauss images/Flower_gray_noisy.raw output/Flower_gray_DenoGauss5.raw 5

Output(PSNR): 31.1134

Flower_gray_DenoGauss5.raw

This is the applied filter

$$\frac{1}{273} \times \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$



Based on the PSNR, we can see at same filter size, gaussian filter has better performance than mean filter, and with same filter type, 3 * 3 size has better performance than 5 * 5.

Based on visual inspection, I would also say mean filter will make the image blurrier than Gaussian. The edges are not clear any more. So overall, Gaussian filter performs better.

2.b

2.b.1

Now we try non-linear filter: bilateral filter. It is expected to generate less edge degradation on edges.

This is a bilaterally filtered output with kernel size 3*3 and σc=1, σs=20

g++ -o BilateralFilter BilateralFilter.cpp

BilateralFilter images/Flower_gray_noisy.raw output/Flower_gray_BiFi.raw 3

Output(PSNR): 32.659

Flower_gray_BiFi.raw 3

The σc/σs can only be adjusted within the code blocks.

If we use 5*5 kernel, with same sigma tuning, the PSNR is 32.9485, no significant difference.

2.b.2

There're two parameter we can tune in bilateral filter, which is the standard deviation of two gaussian distribution.

σc determine the weight according to spatial distance, and σs determines the weight according to intensity difference. The larger σc is, the more weight a distant pixel will get. Similarly, the larger σs is, the more weight a pixel with non-similar intensity will get. In addition, because the two gaussian distribution are timed together, so if one goes to zero, the whole term vanish. Therefore both parameter should be properly set.

To investigate the influence of σc and σs, we use the 3*3 kernel and test on different value combination.

σc/σs PSNR	20	30	40	50
1	32.659	33.5743	33.7102	33.5943
2	33.0841	33.7361	33.6515	33.3809
4	33.1446	33.7241	33.5897	33.2867

The best parameter for this picture we can achieve might be around σc=2, σs=30.

2.b.3

We can see in most cases, when parameters are properly tuned, the performance of bilateral filter is slightly better than 3*3 size gaussian filter, and is also better than mean filter. So bilateral filter is better than linear filter in this picture.

2.c

2.c.1

This part we apply NLM with OpenCV, and tune the parameters.

There's two in four parameters mentioned by HW instructions that can be tuned by Opencv, the small neighbor window size and the large search window size. Also, there's an additional parameter h indicating filter strength. According to OpenCV documents, "big h value perfectly removes noise but also removes image details, smaller h value preserves details but also preserves some noise".

This is run with parameter h=10, small window=7, big window=21. The two window sizes are default values. PNSR=34.289

The program can only be run directly in Clion, so no sample command. The parameters are also tuned within code blocks.

To find the best parameter combination, I first look for proper h.

h	3(default)	5	7	10	12	15	20
PSNR	28.1249	28.45	31.6954	34.289	34.2068	32.8919	30.4811

So we let h=10.

Then we try to find best window size

Neighbor/search	11	21	31	41
3	34.2497	34.7967	34.7661	34.6865
5	34.3215	34.7775	34.6901	34.5648
7(default)	33.9902	34.289	34.1523	34.0106
9	33.6833	33.8773	33.7088	33.551

So the best neighbor window/search window size is (3 * 3, 11 * 11)

To theoretically analyze h and a (this h is supposed to be different with OpenCV h, but maybe they are corelated), larger a will give more weight to distant pixels within a neighbor window, while larger h will give more weight to non-similar neighbor blocks.

2.c.2

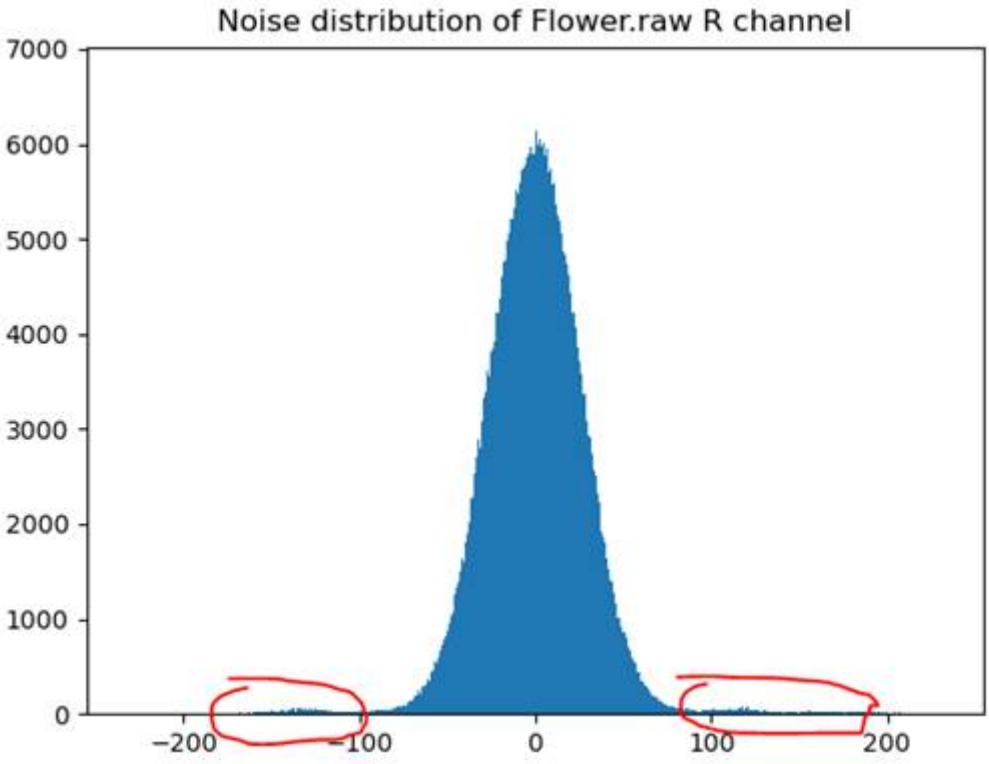
Compare with previous linear filters and bilateral filter, NLM output has slightly higher PSNR, and it seems much more finer and smoother than previous output, especially the wall. You can hardly see any noises on it. But on the other hand, it may have lose some details, and make it kind of unreal.

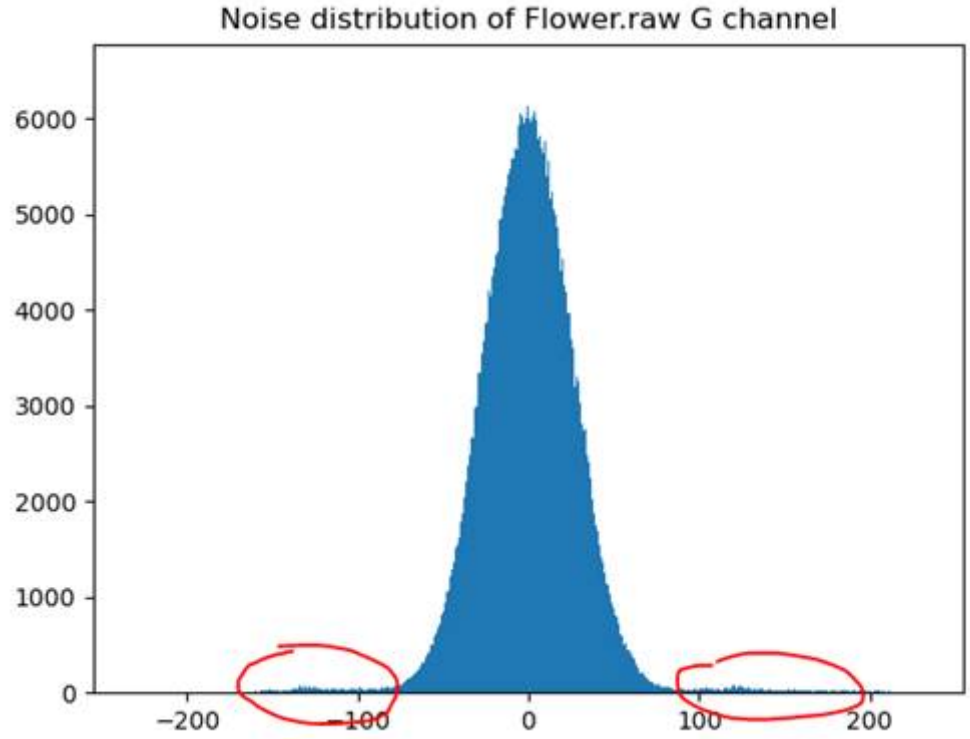
2.d

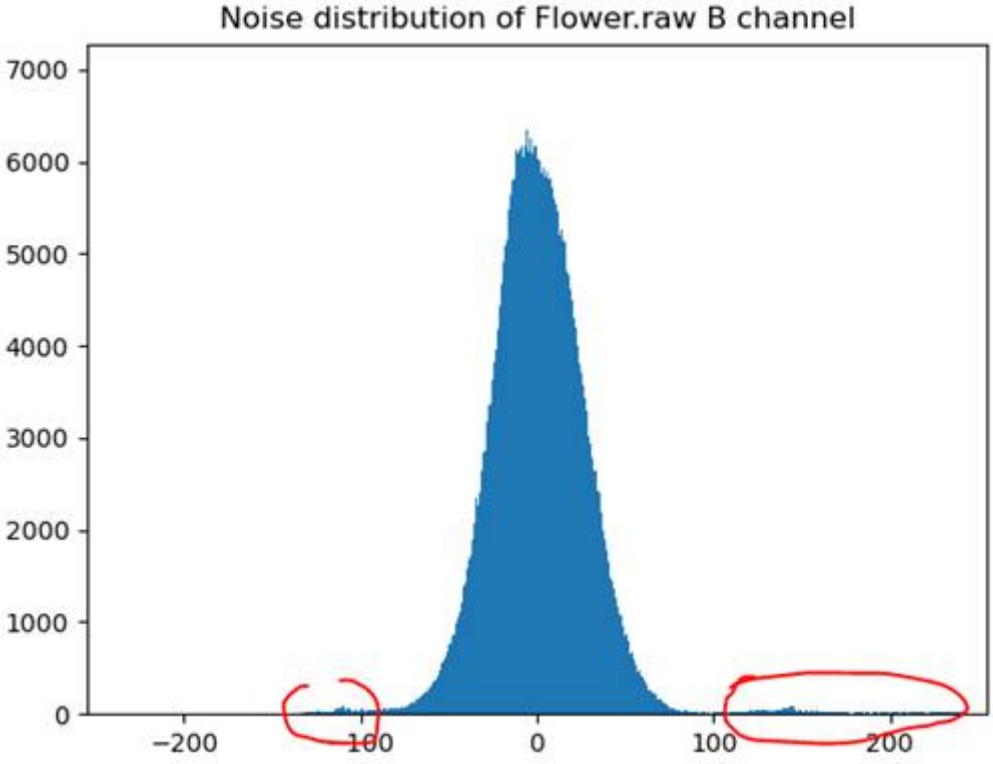
2.d.1

In this part we deal with colored picture instead of grey ones, and it contains mixed noise. So we use combination of different filters.

I first capture the noise distribution of three color channel to analyze noise type, by subtracting the noisy image with the original image.







While majority of the noise follows the Gaussian distribution, there's tiny amount of noise who has huge intensity locate on two sides of the main peak. They are possibly impulse noise. The rest majority are gaussian noise.

2.d.2

We would deal with impulse noise first, because impulse noise are easy to recognized (intensity difference between source image and noisy image goes beyond a threshold). If impulse filter is applied after gaussian noise filter, the bad impulses would already have done unnecessary impact on other good pixels, and the impulses themselves would also be reduced, letting it harder to recognize in the second round.

To deal with impulse noise, I would choose median filter. Basically we have two choice, mean filter and median filter. The reason why median filter is better than mean in terms of impulse removal is that, median value is not sensitive to one or two unusual values. That is to say it can remove salts or peppers without afftecting surrounding good pixels, while mean filter cannot do that.

To deal with Gaussian noise, I would use NLM filter as long as the resource consumption is acceptable, since it has the best performance.

2.d.3

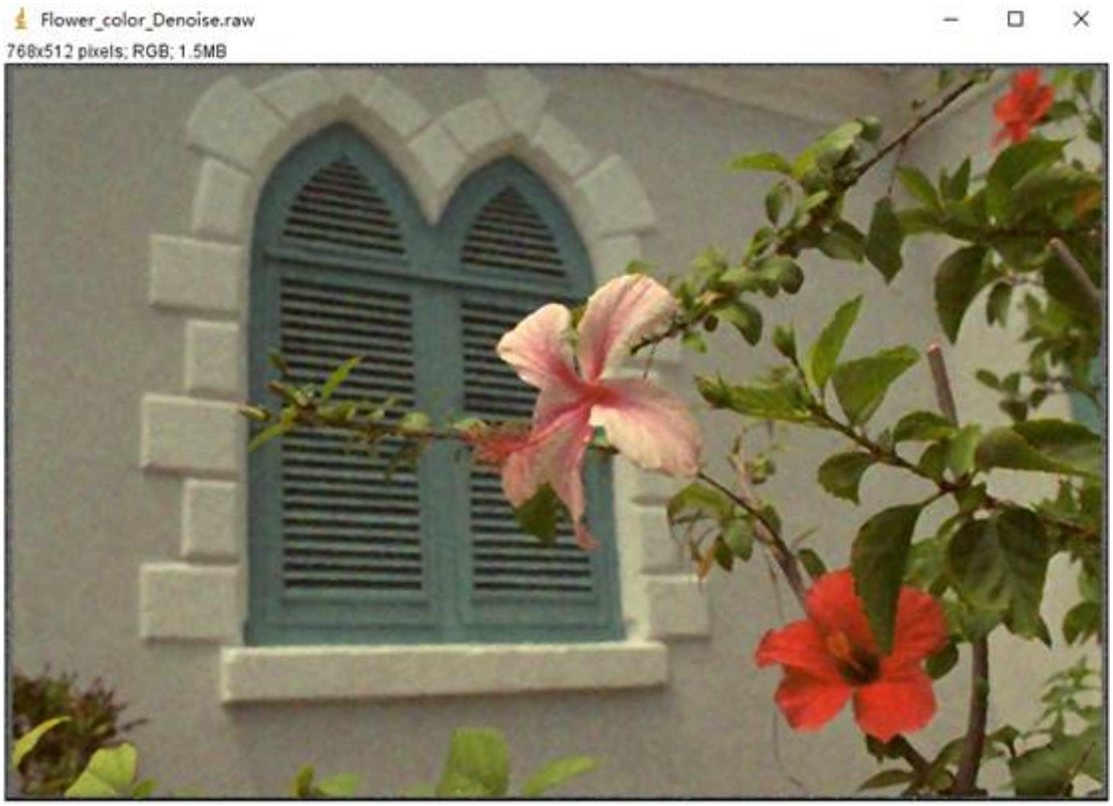
(1) Methods and results

My final method is use a 5*5 median filter first, then apply NLM.

For the median filter, I find out that the best size is 5*5, and the best threshold for abs(Noisy[i][j][k]-Src[i][j][k]) is 10. All pixels that has greater difference than this threshold would undergo the median filter. In addition, because the noise in this picture is huge and dense, I have to set another threshold, that those heavily polluted pixels will not join the calculation of neighborhood median. The setting for “heavily polluted pixels” is having noise intensity greater or equal to 20.

Then I applied NLM with OpenCV *fastNlMeansDenoisingColored()* function. However I did not see any improvement by adjusting its parameter, so I just refer to the default values as well as the best practical value found in previous parts.

The final output is shown below.



(2) Shortcomings

Compare with the noisy image, the algorithm succeeds in removing the impulse. However, we can still see the color on the wall is not uniform. I tried many methods like introducing another Gaussian filter trying to smoothen the wall, but they didn't work. And there're actually a few singular pixels on the edges of some leaves.

(3) Possible improvements

The noise in this colored image may be too heavy to be removed by linear and non-linear filters. But still, I think there'll possibility that the wall can be processed better through learning method. Because it's a large continuous area, and the color varies a little. Maybe we can try to find out the boundary of the walls, and then somehow 'dye' it smoothly.

Discussion

1.Linear filter means the manipulation process can be represented through convolution, all the output pixel is fixed linear combination of input pixels. For example, mean filter and gaussian filter. Bilateral filter is a typical non-linear filter because the linear combination may vary. The weight in the red circle is not a fixed value.

$$Y(i,j) = \frac{\sum_{k,l} I(k,l) w(i,j,k,l)}{\sum_{k,l} w(i,j,k,l)}$$

2.When finding the best filter in practice, we should not only consider the output quality. When the filter size goes larger, for example, or when the NLM search windows sizes goes bigger, it's taking more time to process. Remember it is a single picture. If it's a frame of a video, or one in a huge image data set, it's definitely unacceptable to take several seconds on a single picture.

3.The repaired picture seems still blurry, but I don't think it's the algorithm or the parameter tuning that is to blame. The colored Flower image seems to have much high noise than grey one. One the histogram, you can see the peak is lower and the variance is much bigger, making it difficult to repair. When dealing with grey picture, NLM actually do quite well. The repaired wall seems as smooth as a soap. In color picture, the repair quality is not that pleasing. But given the different noise volumn, and all parameter tuning has been tested, I think it may be the best outcome we can get at this stage.

Problem 3 Special Effect Image Filters: Creating Frosted Glass Effect

Motivation

We not only want to denoise in real life, but also we may prefer the image to look like within certain scene. In this part we try to add special effect to our picture. Here, we create frosted glass filter, and see how it works when noise exist.

Procedure and results

3.1

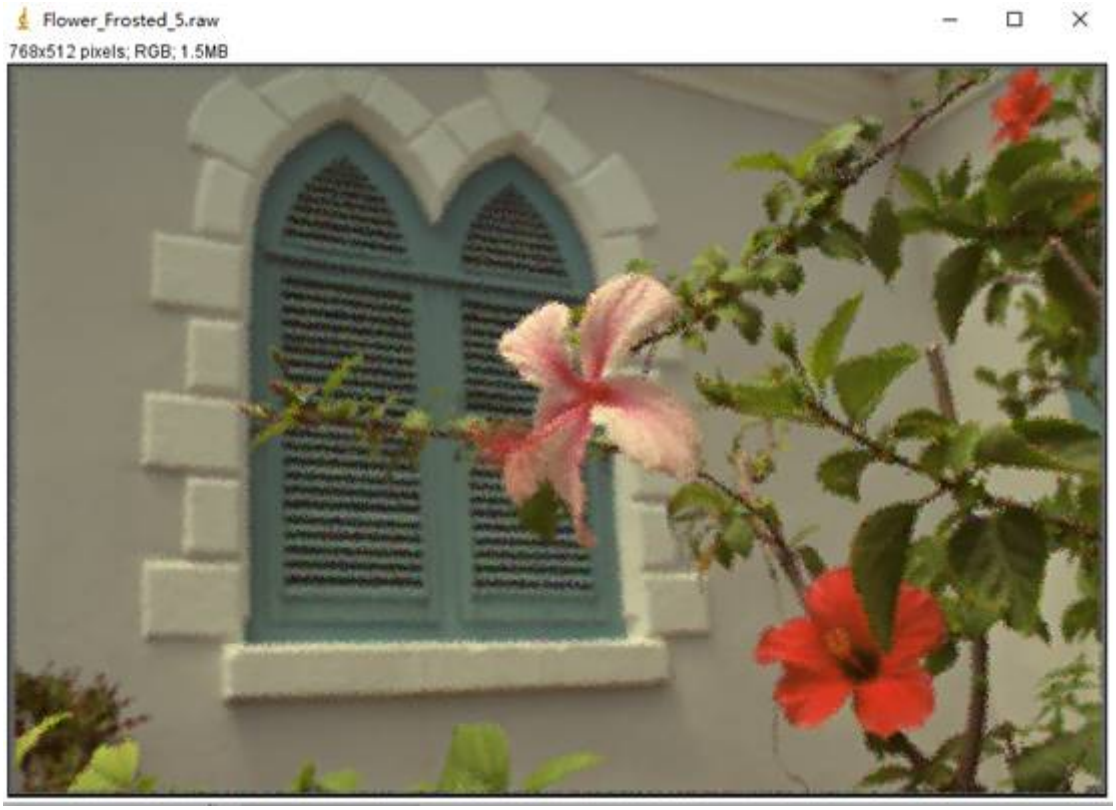
This part we compare different filter size.

For size 5 frosted glass filter,

g++ -o FrostedGlassFilter FrostedGlassFilter.cpp

FrostedGlassFilter images/Flower.raw output/Flower_Frosted_5.raw 3 5

Output: Flower_Frosted_5.raw



For size 7 frosted glass filter,

g++ -o FrostedGlassFilter FrostedGlassFilter.cpp

FrostedGlassFilter images/Flower.raw output/Flower_Frosted_7.raw 3 7

Output: Flower_Frosted_7.raw

3.2

When there're noise in the original image, we want to see how the frosted filter performs.

FrostedGlassFilter images/Flower_noisy.raw output/Flower_noisy_Frosted_5.raw 3 5

Output: Flower_noisy_Frosted_5.raw

FrostedGlassFilter images/Flower_noisy.raw output/Flower_noisy_Frosted_7.raw 3 7

Output: Flower_noisy_Frosted_7.raw

I don’t think the noise leads to any significant change. It seems as noisy as before.

3.3

Here we want to figure out will the order of how filters are applied lead to any performance difference.

If use gaussian denoise first then forsted,

FrostedGlassFilter output/Flower_gray_DenoGauss5.raw output/Flower_gray_noisy_DenoFro_5.raw 1 5

Output: Flower_gray_noisy_DenoFro_5.raw

If we first frosted then denoise:(use 5*5 filter in both step)

FrostedGlassFilter images/Flower_gray_noisy.raw output/Flower_gray_noisy_Frosted_5.raw 1 5

Output: Flower_gray_noisy_Frosted_5.raw

ImgDenoGauss output/Flower_gray_noisy_Frosted_5.raw output/Flower_gray_FroDeno_5.raw 5

Output: Flower_gray_FroDeno_5.raw

For process A (first denoise then apply forsted filter), The additional frosted filter makes smooth area, like the wall, even smoother than when only gaussian filter is applied. But the boundaries like the flower, leaves and window-shade become very ugly.

For process B (first frosted filter then denoise), I think the overall frosted effect seems more uniform than process A.

Discussion

1.When frosted filter is applied on noisy picture, I don't see the noise make the frosted filter work better or worse. But when they're combined and used sequentially, there's difference. I can hardly say which order is better. Personally, I would prefer process B, that is first frosted then denoise. I think in this case frosted effect is applied more globally uniform. Anyway it depends on every image itself.