EE 569: Digital Image Processing
Homework #1

By
Brinal Chimanlal Bheda
USC ID - 8470655102
bbheda@usc.edu
Spring 2019

Issue date: 01/07/2019
Submission Date: 01/22/2019

Problem 1: Image Demosaicing and Histogram Manipulation

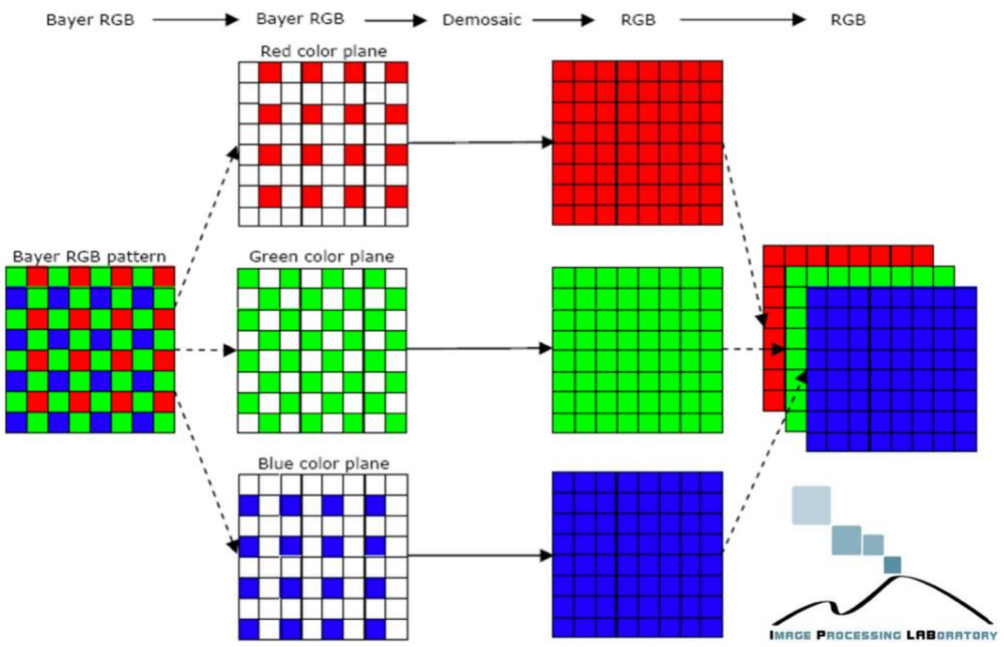
(a) Bilinear Demosaicing

I. Abstract and Motivation

Now-a-days, we capture digital images by using the camera phones. However, these camera phones have CMOS sensors which capture only one color (R/G/B) per sensor/pixel. The other two colors have to be reconstructed based on their neighbor pixel values to obtain the full color. So we need to perform interleaving using Bayes pattern, we have to do interpolation to get the real color images. This is called Demosaicing, to get the RGB channels in all the pixels.

II. Approach and Procedure

In the Bayer pattern, green color is present 50% and red/blue color is 25%. This is because the human eye is more sensitive to the green color. To capture color images, digital camera sensors are usually arranged in form of a color filter array (CFA) called Bayer array. Demosaicing is the process of translating this Bayer array of primary colors having only one-color channel present at each pixel location into the color image that contains R, G, B color values at each pixel.
(Source: http://www.dmi.unict.it/~battiato/mm1112/Parte%207.3%20%20-%20Demosaicing.pdf)



In this method, the missing color value at each pixel is approximated by bilinear interpolation using the average of its two or four adjacent pixels of the same color. To give an example, the missing blue and green values at pixel R3,4 are estimated as:

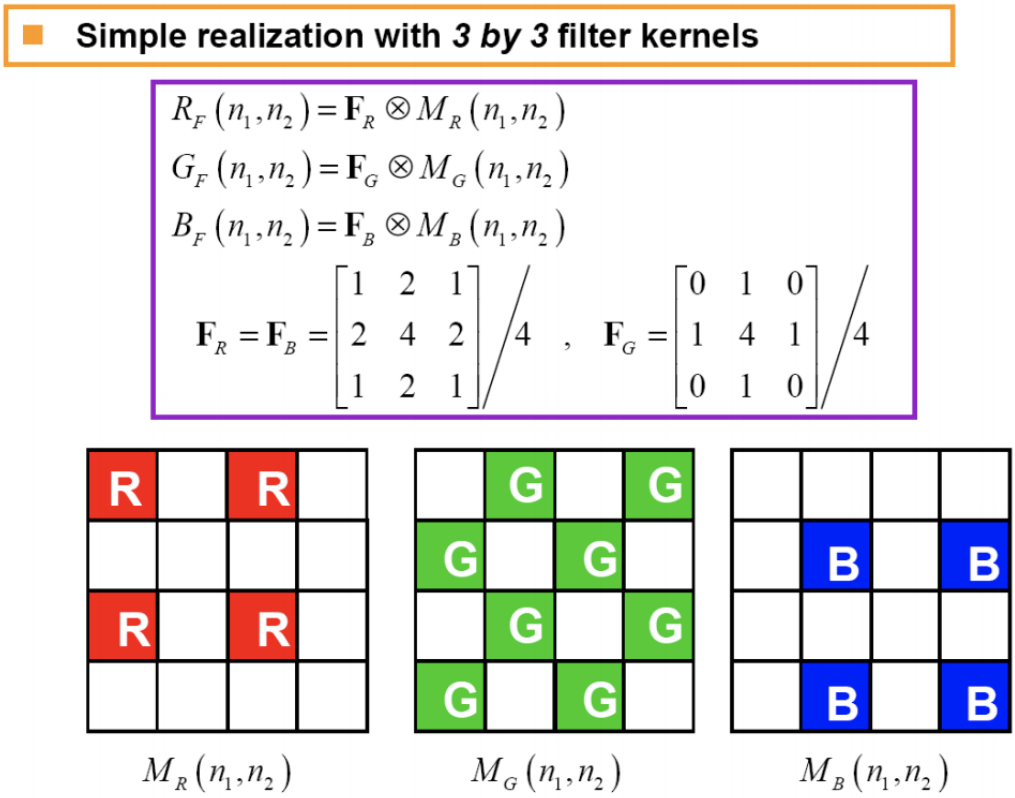
$$\hat{B}_{3,4} = \frac{1}{4}(B_{2,3} + B_{2,5} + B_{4,3} + B_{4,5})$$
$$\hat{G}_{3,4} = \frac{1}{4}(G_{3,3} + G_{2,4} + G_{3,5} + G_{4,4})$$

As for pixel G3,3, the blue and red values are calculated as:

$$\hat{R}_{3,3} = \frac{1}{2}(R_{3,2} + R_{3,4})$$
$$\hat{B}_{3,3} = \frac{1}{2}(B_{2,3} + B_{4,3})$$

Before the above, we use boundary extension to deal with the boundary effect. We use the mirror reflecting algorithm considering the boundary line pixel values as the center and reflecting the other lines inside the image to the boundary on all the sides of the image. This is mainly done to avoid errors in the computation when a filter is being used.

Algorithm implemented in C++ :



- Read cat.raw image which is a gray scale image using fread() function
- Define 3 color channels: channel 0 → Red, channel 1 → Green, channel 2 → Blue
- Add one boundary line to the input image by boundary extension using mirror reflecting on all sides of the image border
- Apply demosaicing by using the 3x3 convolution filter for the bilinear interpolation
- Adding the remaining two missing green/red/blue channel values to the R/G/B pixel location by using the formula mentioned above
- Using two sets of functions for green pixel location for odd and even values
- Crop the boundary extended image to the original image size after the computation is done
- Write the computed image data array on output.raw file using the fwrite() function

III. Experimental Results

Figure 1: The cat image with CFA sensor input

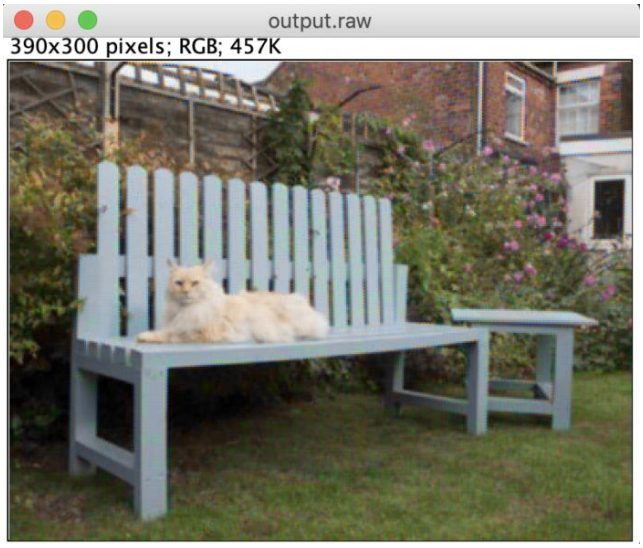


Figure 2: The output cat image after bilinear demosaicing

IV. Discussion

The result after applying bilinear demosaicing to the cat image is shown above. Yes, there are artifacts observed in the output image. Most of the artifacts appear at edges and areas of high frequency. The shortcomings on the output image are: Zipper effect causing color changes at sharp edges like the fence at the top and the bench, False color effect causing artificial/error colors like on the grass, Blurring effect like on the flowers and the grass in the cat output image, Aliasing when any pixel frequency goes above the Nyquist frequency. The images below show these artifacts present in the output image compared to the original image.

Figure 3: The original cat image

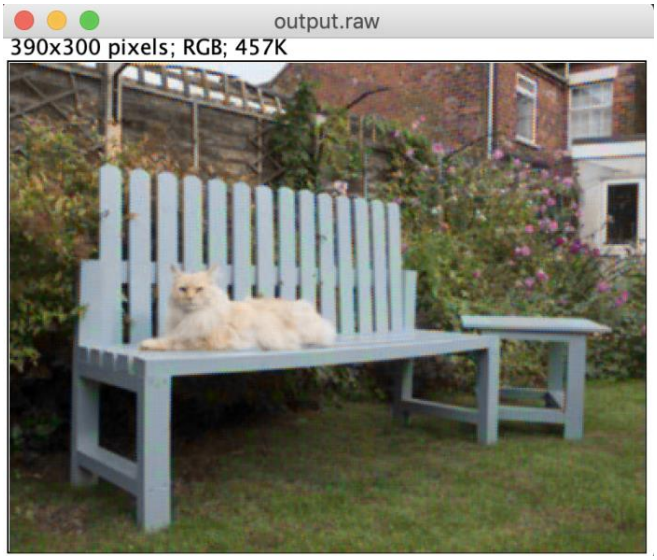


Figure 2: The output cat image after bilinear demosaicing

The cause of these artifacts is as follows:
There is an error in the calculation of the bilinear interpolated red, green and blue plane. Also, in the green channel plane this error is deeper and causes more artifacts. Here, we can see that the original grey image is having only two colors, one lighter and other darker. We try to demosaicing figure b and get results in c. Now, if we separate this RGB image into 3 different RGB planes, we see there is color error in the columns for red and blue; for green image plane there is some deeper effect so the color changes alternatively.

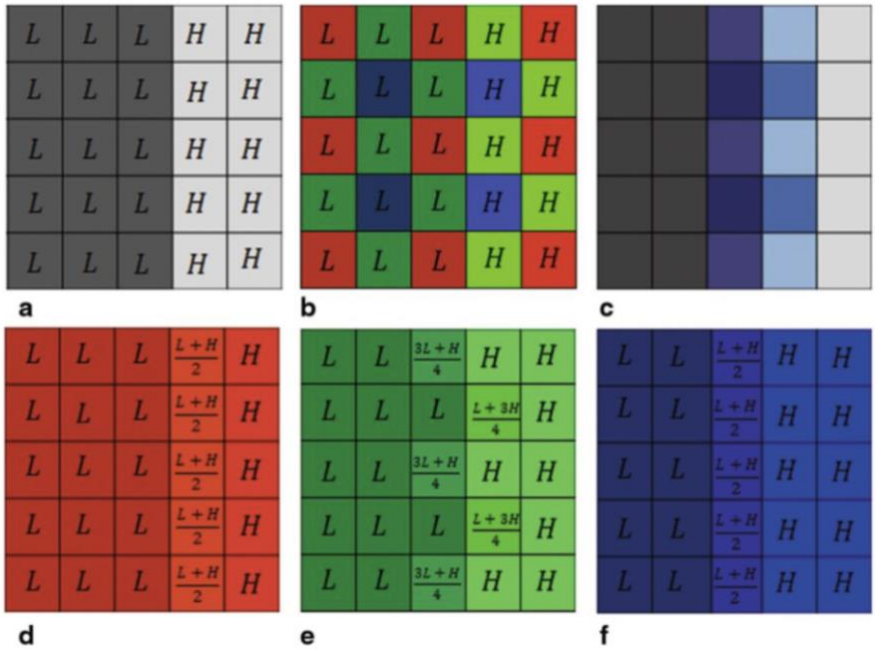


Fig. 2.4 **a** Synthesized gray image, **b** CFA samples of **a**, **c** Bilinear interpolation result, **d** Bilinear interpolated red plane, **e** Bilinear interpolated green plane, **f** Bilinear interpolated blue plane

To improve the demosaicing performance, we use the Malvar-He-Cutler (MHC) demosaicing technique which is discussed in (b). This model is valid and good to use for smooth surfaces. One of the applications of bilinear interpolation is image resizing.

(b) Malvar-He-Cutler (MHC) Demosaicing

I. Abstract and Motivation

Now-a-days, we capture digital images by using the camera phones. However, these camera phones have CMOS sensors which capture only one color (R/G/B) per sensor/pixel. The other two colors have to be reconstructed based on their neighbor pixel values to obtain the full color. So we need to perform interleaving using Bayes pattern, we have to do interpolation to get the real color images. This is called Demosaicing, to get the RGB channels in all the pixels. The MHC technique is an improvised technique and gives better results as compared to the Bilinear demosaicing.

II. Approach and Procedure

MHC is an improved linear interpolation demosaicing algorithm. It yields a higher quality demosaicing result by adding a second order cross channel correction term to the basic bilinear demosaicing result.
(Source: EE 569 – Discussion 1)

The MHC algorithm is stated below.
To estimate a green component at a red pixel location, we have

$$\hat{G}(i,j) = \hat{G}^{bl}(i,j) + \quad _R(i,j) \tag{1}$$

where the 1st term at the right-hand-side (RHS) is the bilinear interpolation result given in (1) and the 2nd term is a correction term. For the 2nd term, alpha is a weight factor, and Δ_R is the discrete 5-point Laplacian of the red channel:

$$_R(i,j) = R(i,j) \quad \frac{1}{4}\big(R(i \quad 2,j) + R(i+2,j) + R(i,j \quad 2) + R(i,j+2)\big) \tag{2}$$

To estimate a red component at a green pixel location, we have

$$\hat{R}(i,j) = \hat{R}^{bl}(i,j) + \quad _G(i,j) \tag{3}$$

where Δ_G is a discrete 9-point Laplacian of the green channel.

To estimate a red component at a blue pixel location,

$$\hat{R}(i,j) = \hat{R}^{bl}(i,j) + \quad _B(i,j) \tag{4}$$

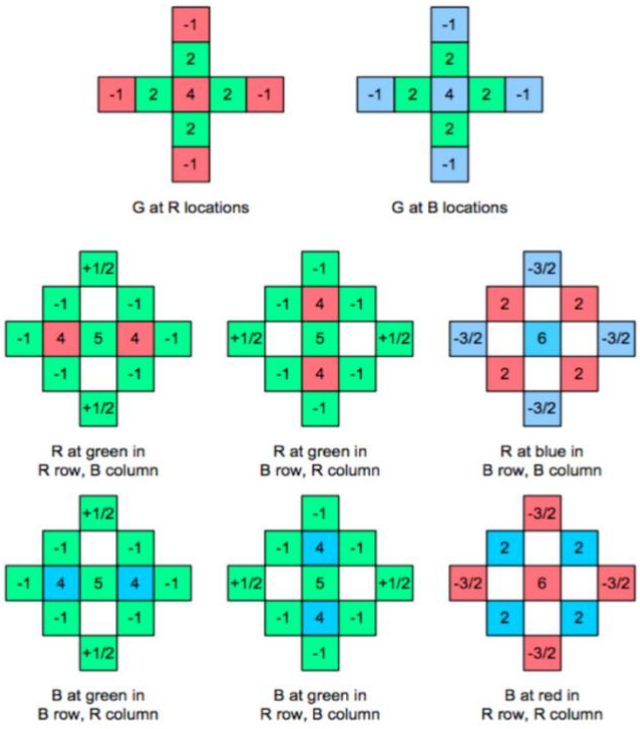
where Δ_B is a discrete 9-point Laplacian of the blue channel. The weights , , control how much correction is applied, and their default values are:

$$= \frac{1}{2}, \quad = \frac{5}{8}, \quad = \frac{3}{4} \tag{5}$$

The procedure for MHC demosaicing is complicated. It can be implemented by convolution with a set of linear filters. Say, for example there are eight different filters for interpolating the different color components at different pixel locations. This is shown in the figure below.

For example: For G3,3, the values can be derived using the filters below.

$$\hat{R}(i,j) = \frac{1}{8}\Big(\begin{array}{ccccc} & & \frac{1}{2}G(i,j-2) & & \\ & -G(i-1,j-1) & & -G(i+1,j-1) & \\ -G(i-2,j) + & 4R(i-1,j) & + \quad 5G(i,j) \quad + & 4R(i+1,j) & + \quad -G(i+2,j) \\ & -G(i-1,j+1) & & -G(i+1,j+1) & \\ & & + \frac{1}{2}G(i,j+2) & & \end{array} \Big).$$



This pattern is used to do the average on the surrounding pixels by using the formula above. So we take the average on the neighbor surrounding pixels and use the weight. We calculate the weighted average of the center pixel by using the surrounding pixels in the corresponding rows and columns. Repeat this procedure for all the pixel locations to add the other two color values at each pixel location.

Before the above, we use boundary extension to deal with the boundary effect. We use the mirror reflecting algorithm considering the boundary line pixel values as the center and reflecting the other lines inside the image to the boundary on all the sides of the image. This is mainly done to avoid errors in the computation when a filter is being used.

Algorithm implemented in C++ :

- Read cat.raw image which is a gray scale image using fread() function
- Define 3 color channels: channel 0 → Red, channel 1 → Green, channel 2 → Blue
- Add two boundary lines to the input image by boundary extension using mirror reflecting on all sides of the image border
- Apply demosaicing by using the 5x5 convolution filter for the bilinear interpolation

- Adding the remaining two missing green/red/blue channel values to the R/G/B pixel location by using the formula mentioned above by the combination of 8 different filters
- Using two sets of functions for green pixel location for odd and even values
- Crop the boundary extended image to the original image size after the computation is done
- Write the computed image data array on output.raw file using the fwrite() function

III. Experimental Results

Figure 1: The cat image with CFA sensor input

Figure 4: The output cat image after MHC demosaicing

IV. Discussion

The result after applying MHC linear demosaicing algorithm to the cat image is shown above. We can see that the errors in the image resulted by the bilinear demosaicing method have almost disappeared when implementing the MHC demosaicing algorithm. We can compare the cat output image results by the bilinear and MHC demosaicing below.

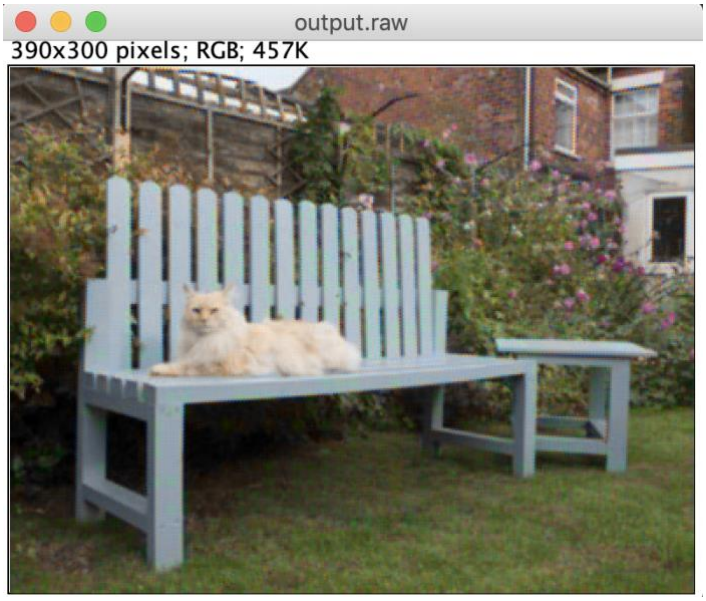


Figure 2: Bilinear demosaicing result

Figure 4: MHC demosaicing result

For the comparison between the bilinear and MHC demosaicing:
Bilinear demosaicing is easy to implement as it has lesser computations. It is good at previewing image which has simple, smooth or less various color. However, this method performs badly at the edges or details in the image and this causes various artifacts like Zipper effect, false-color effect, aliasing, blurring. In the figures above, we see fig 2 is more blur than fig 4.
MHC linear demosaicing performs better at edges or details in the image compared to bilinear demosaicing. We can see this performance difference on various aspects in the image like the cat, bench, flowers, grass. The image is better in colors and also less blurry. Since MHC uses second order correction term to the bilinear method, it helps to preserve the true color especially in the regions where colors change rapidly. Also, while applying MHC, since the correction terms use plus and minus coefficients, it will make some values go above 255 or below 0. Such values result into wrong color in the image. So to deal with it, we assign 255 to values above 255 and 0 to values below 0.

(c) Histogram Manipulation Equalization

I. Abstract and Motivation

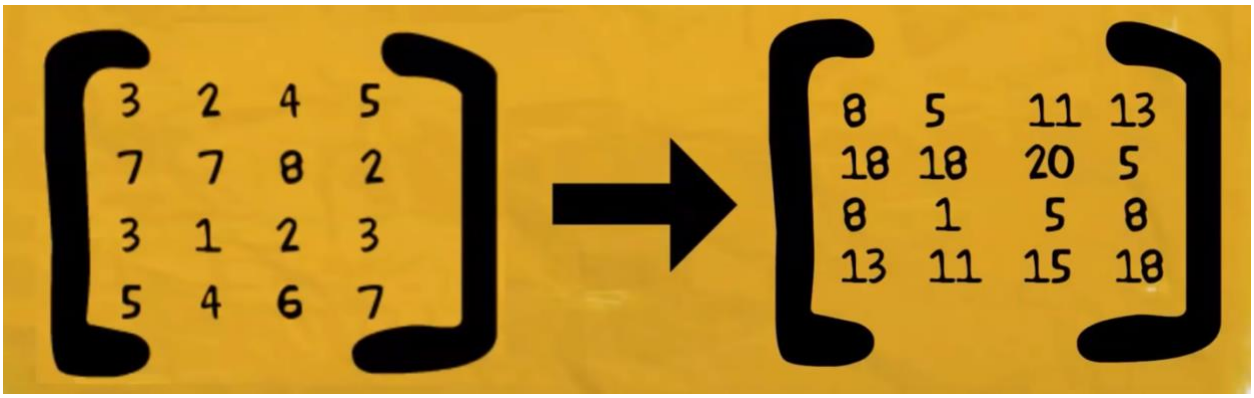
Histogram equalization technique deals with intensity values and contrast of an image. This is a technique which improves the image contrast by changing the image/pixel intensities. Contrast adjustment is needed so as to make the image more viewable as it equalizes the dark and bright components, making the luminance distribution uniform; thus making the image more pleasant to the human eye. Histogram is the count of pixel values (0-255) in an image, it shows the frequency of each pixel at different intensities or probability of pixels. Histogram of an image can be drawn plotting pixel intensities vs frequency or probability of pixel intensities. There are two techniques used for histogram equalization: transfer function based and cumulative probability based. The transfer function is the integration of the input histogram. The cumulative probability is the discrete version of transfer function where it uses the probability density array partitioning that is partition the long string NxN into 256 segments of the same length.

II. Approach and Procedure

Method A: the transfer function based histogram equalization method
(Source: https://www.youtube.com/watch?v=PD5d7EKYLcA)

Take a grayscale image in matrix form. Let each element be a pixel of an image and values of the elements represent intensities of the pixels.
First step is to obtain the histogram - to count the total number of pixels associated with each pixel intensity (0-255). Second step is to calculate the normalized probability histogram - to calculate probability of each pixel intensity in the image matrix. Probability is the number of pixels divided by the total number of pixels. Third step is to calculate cumulative probability. Now we round up the decimal values obtained to the lower integer values (floor rounding). Forth step is create mapping table ie x to CDF(x)*255. Hence, the original image has been transformed to the equalized image with different intensity on each pixel and is shown below.

Sample image pixel intensity values and the contrast enhanced image is given below:

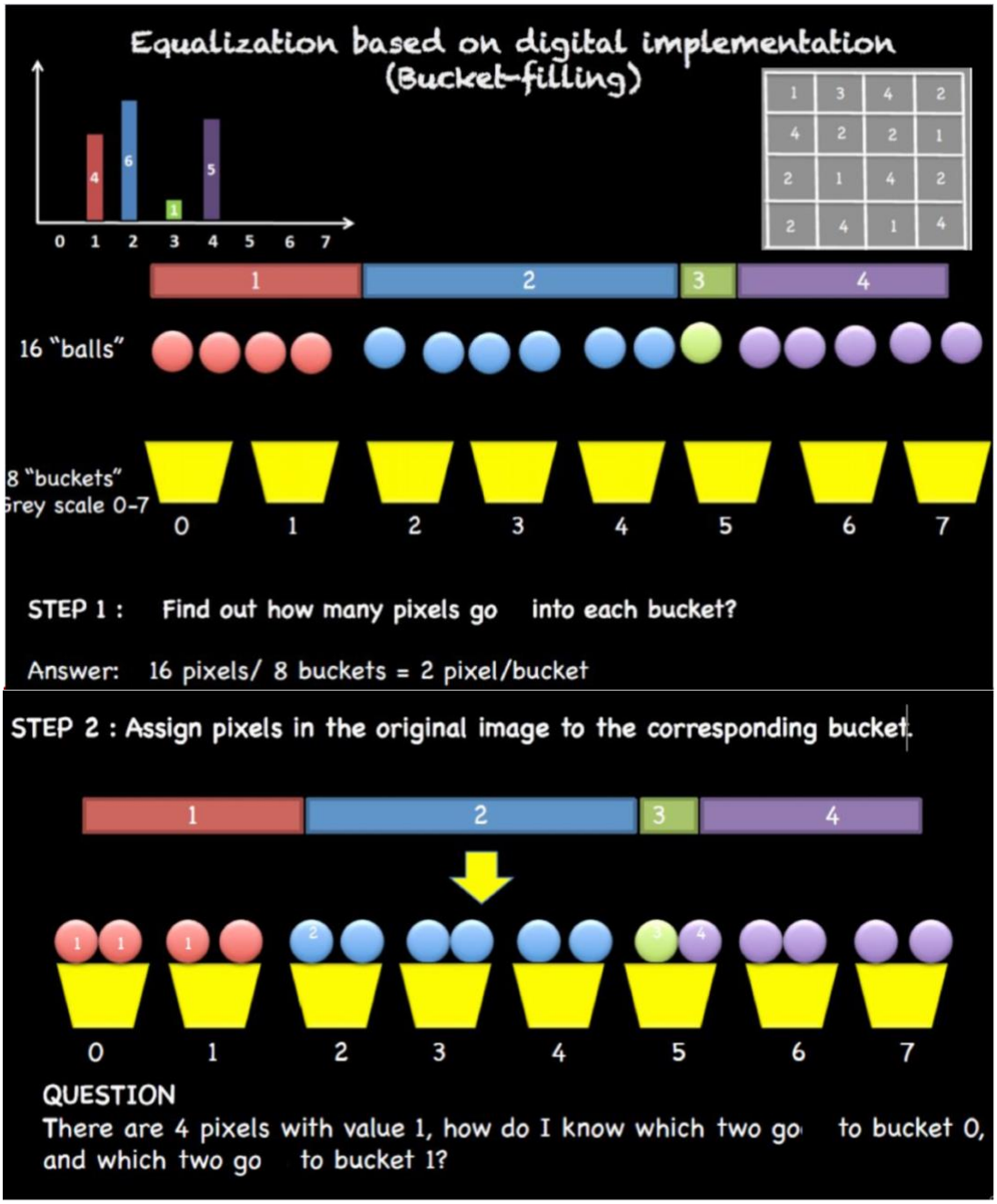


Pixel Intensity	1	2	3	4	5	6	7	8	9	10
No. of pixels	1	3	3	2	2	1	3	1	0	0
Probability	.0625	.1875	.1875	.125	.125	.0625	.1875	.0625	0	0
Cumulative probability	.0625	.25	.4375	.5625	.6875	.75	.9375	1	1	1
C.P * 20	1.25	5	8.75	11.25	13.75	15	18.75	20	20	20
Floor Rounding	1	5	8	11	13	15	18	20	20	20

Algorithm implemented in C++ :

- Read rose_dark.raw and rose_bright.raw input images using fread() function and get height, width and bytesperpixel values
- Run 3 nested for loops for height (400), width (400) and bytesperpixel (=1) to access each pixel in the original image
- Define the histograms of original image, equalized image and transfer function as unsigned char
- Find the count of 0 to 255 pixels in the image
- Normalize this count to get probability density function (pdf) of pixel values
- Now add the pdf cumulatively as shown in the table above to get cdf of pixel values
- Multiply the cdf by 255
- Get the floor rounded values for every 0 to 255 pixels
- Replace the original image pixel values with the new corresponding rounded pixel values which are calculated
- Write the computed contrast enhanced image on output.raw file using the fwrite() function
- Write the histograms to text file

Method B: the cumulative probability based histogram equalization method (bucket filling method)
(Source: EE 569 – Discussion 1)



Each pixel location corresponds to the bucket and each different color corresponds to different pixel value. First, we need to decide how many pixels go in each bucket. Second step is to assign pixels in the original image to corresponding bucket.
The figure above shows that we have to arrange the pixels from 0 to 255 and then assign equal number of pixels to each bucket. The bucket size is defined as the total number of pixels in the image divided by 256 (say, image is of 400x400 size, divide by 256 = 625). After equal number of pixels are assigned in a bucket, the original

image is replaced with pixels that are assigned for each bucket. Thus, this method ensures that equal number of pixels are present in every 0 to 255 buckets.

Algorithm implemented in C++ :

- Read rose_dark.raw and rose_bright.raw input images using fread() function and get height, width and bytesperpixel values
- Run 3 nested for loops for height (400), width (400) and bytesperpixel (=1) to access each pixel in the original image
- Define the histograms of original image, equalized image and transfer function as unsigned char
- Initialize 3 different 1D arrays to store values for the row index, column index and pixel value
- Sort these pixels in the order of 0 to 255
- Store the corresponding row index and column index for each pixel in the 1D arrays defined above
- Change the corresponding pixel values at the given locations according to the bucket size such that each bucket will contain equal number of pixels
- Replace the pixel values in the input image to new pixel values calculated by the bucket filling method and use the row and column index tracked in 1D arrays to put the new values of pixels
- Write the CDF based histogram equalized contrast enhanced image on output.raw file using the fwrite() function
- Write the histograms to text file

III. Experimental Results

Figure 5: rose_dark input image

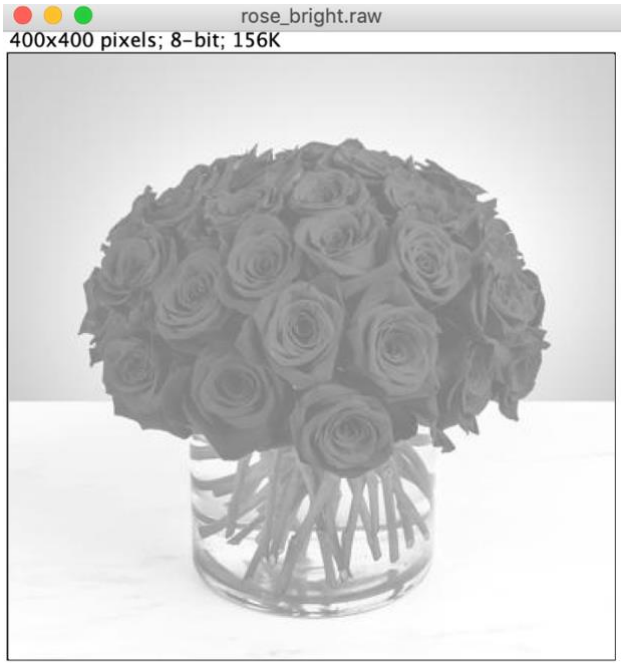


Figure 6: rose_bright input image

Figure 7: enhanced image of rose_dark image using Method A

Figure 8: enhanced image of rose_bright image using Method A

Figure 9: enhanced image of rose_dark image using Method B

Figure 10: enhanced image of rose_bright image using Method B

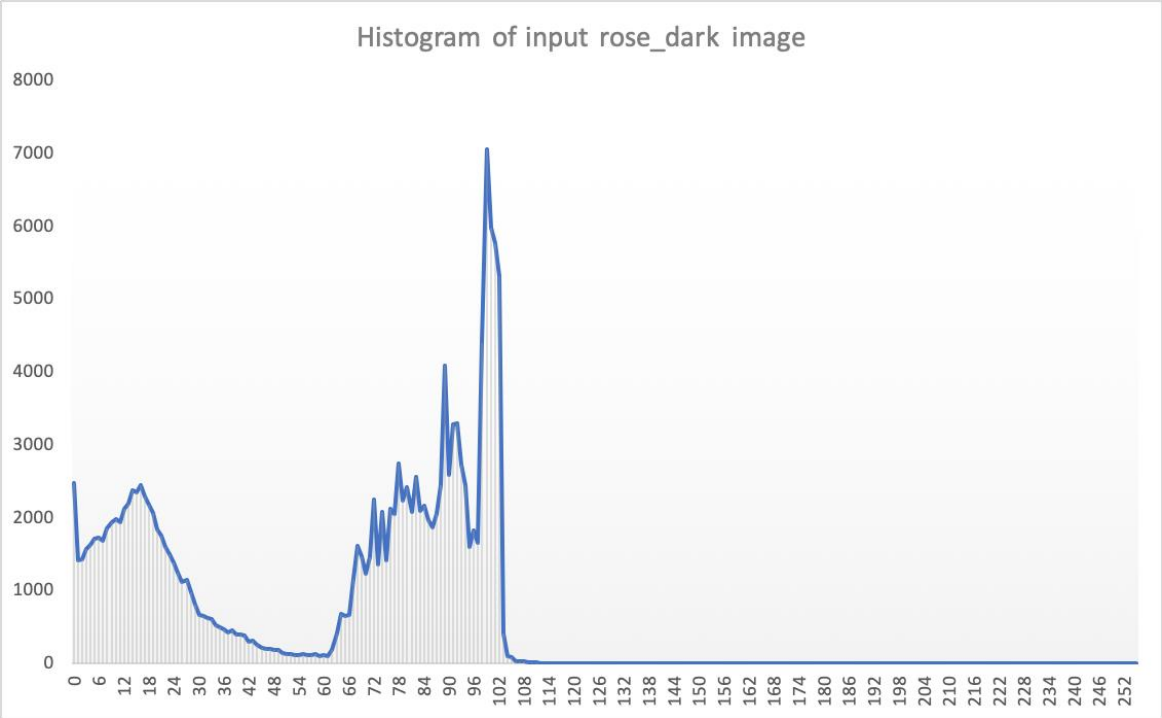


Figure 11: histogram of input rose_dark image

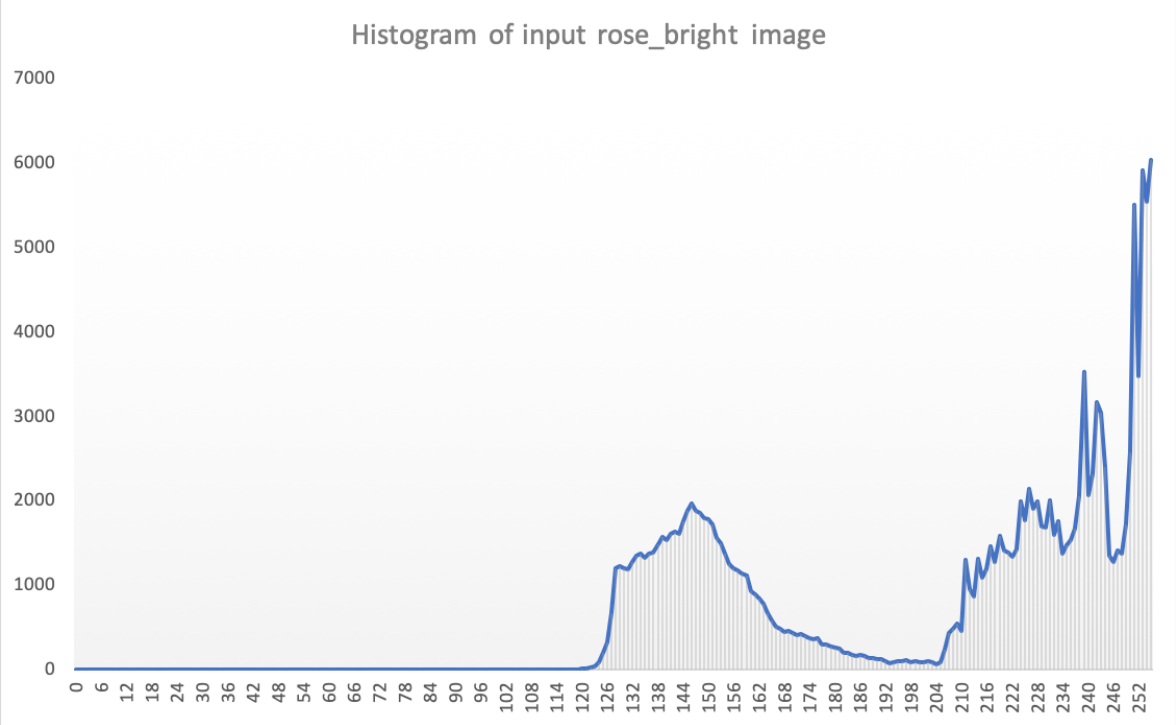


Figure 12: histogram of input rose_bright image

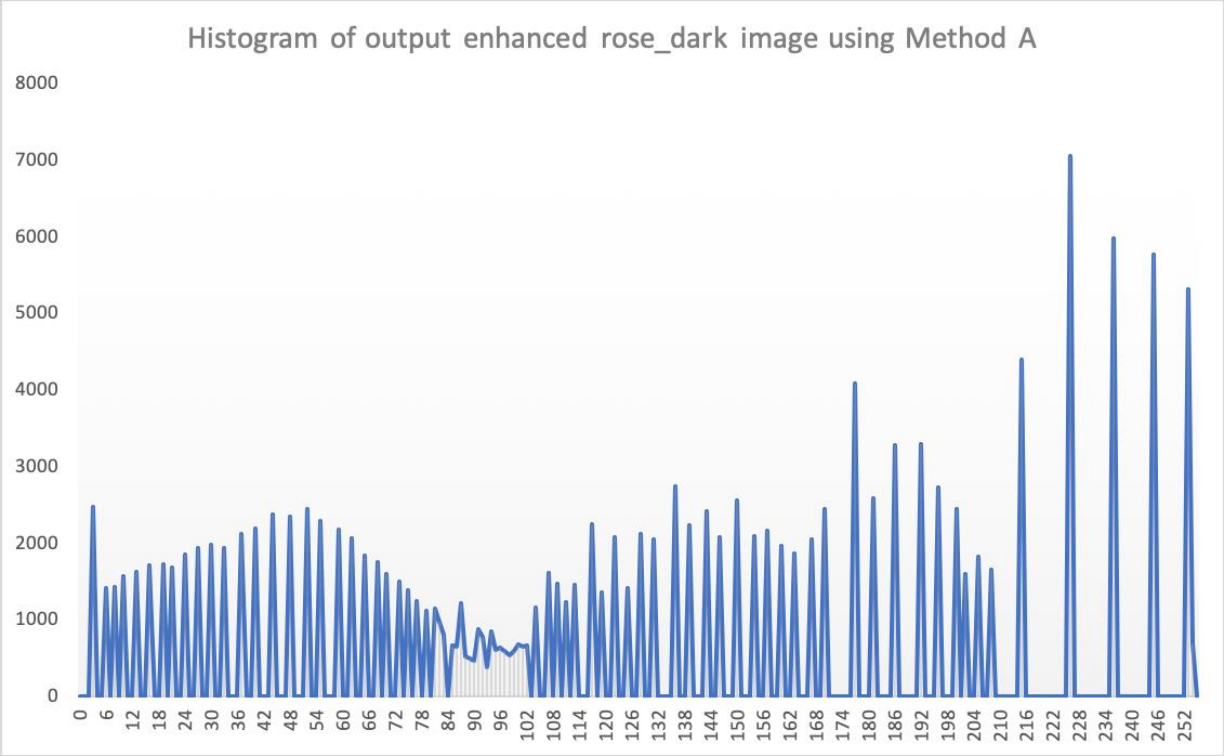


Figure 13: histogram of output enhanced rose_dark image using Method A

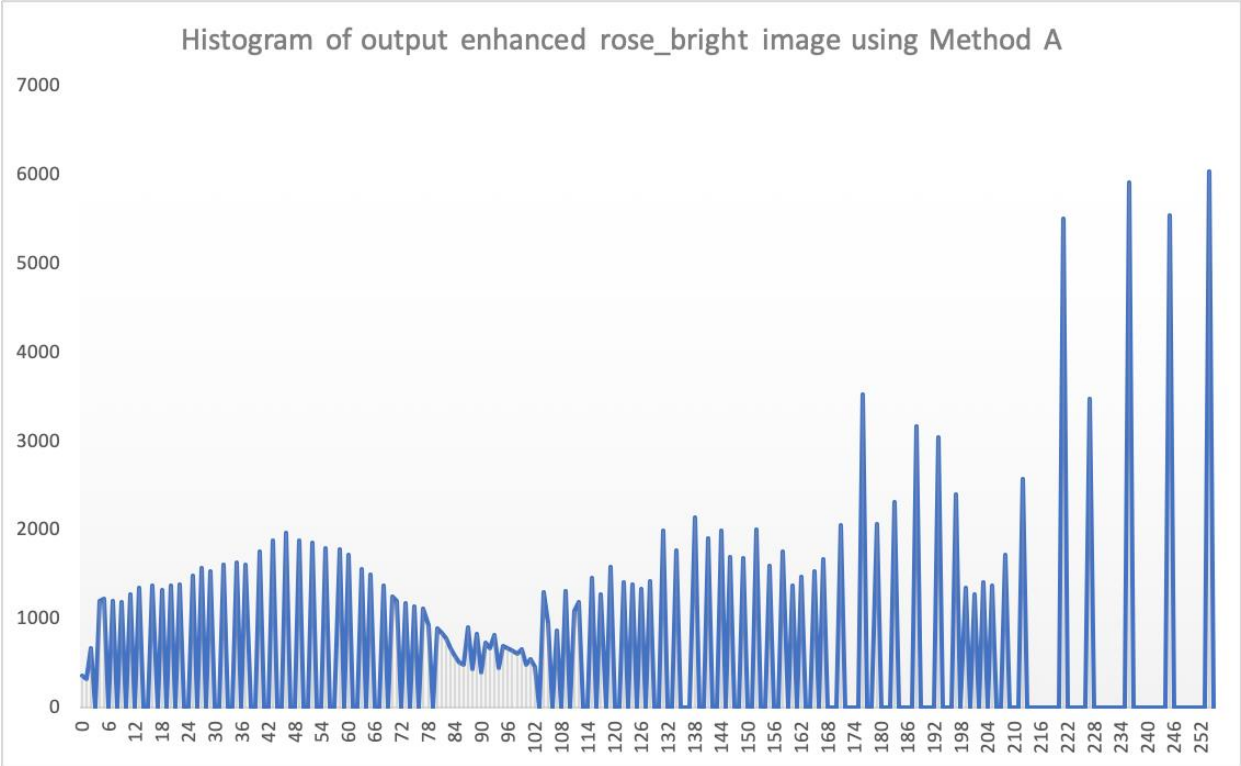


Figure 14: histogram of output enhanced rose_bright image using Method A

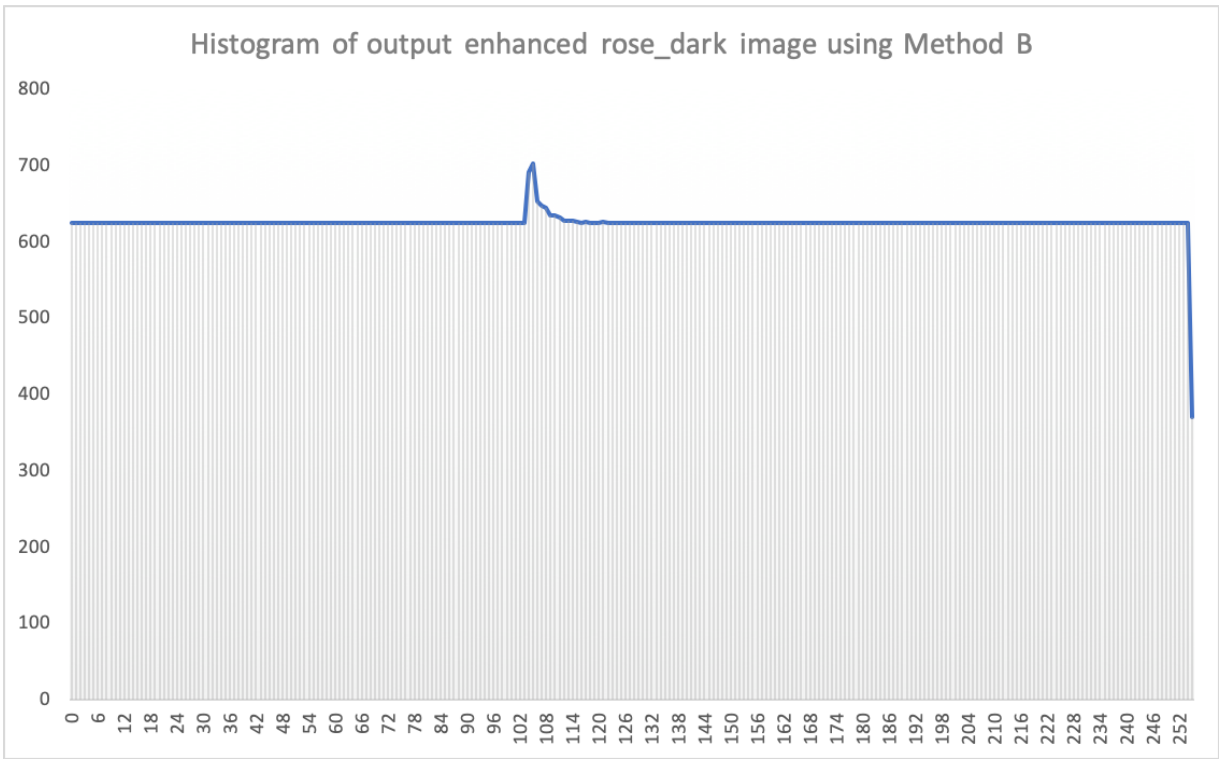


Figure 15: histogram of output enhanced rose_dark image using Method B

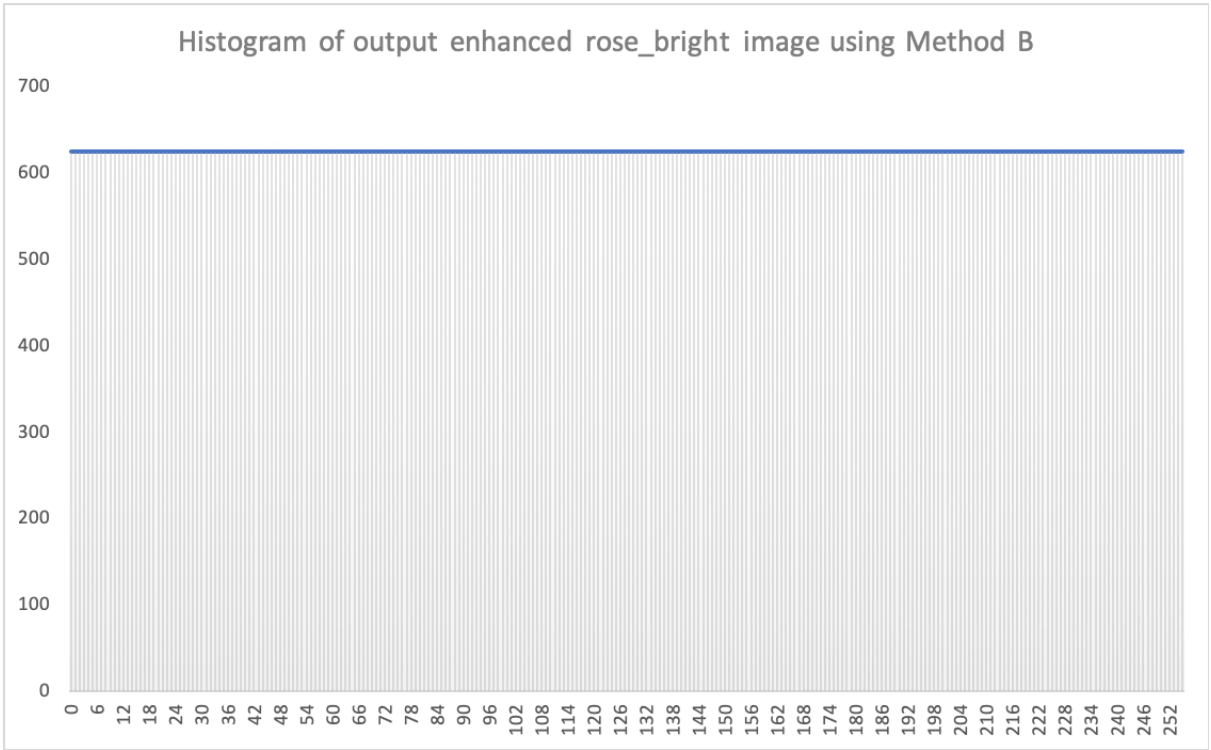


Figure 16: histogram of output enhanced rose_bright image using Method B

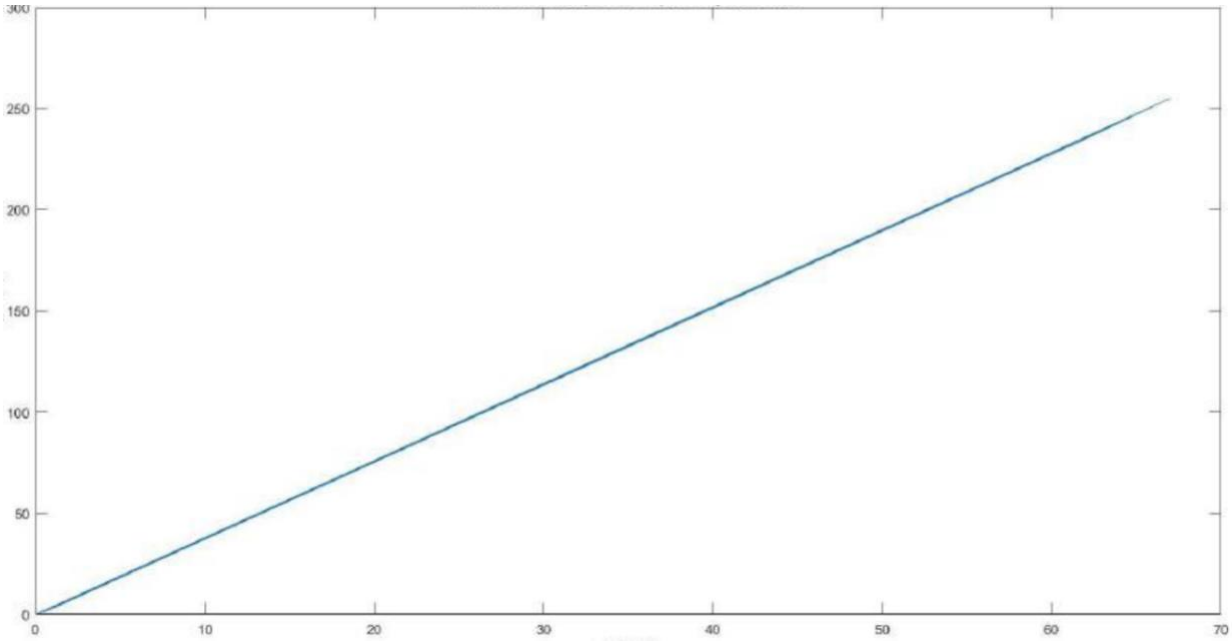


Figure 17: Transfer function of rose_dark using Method A

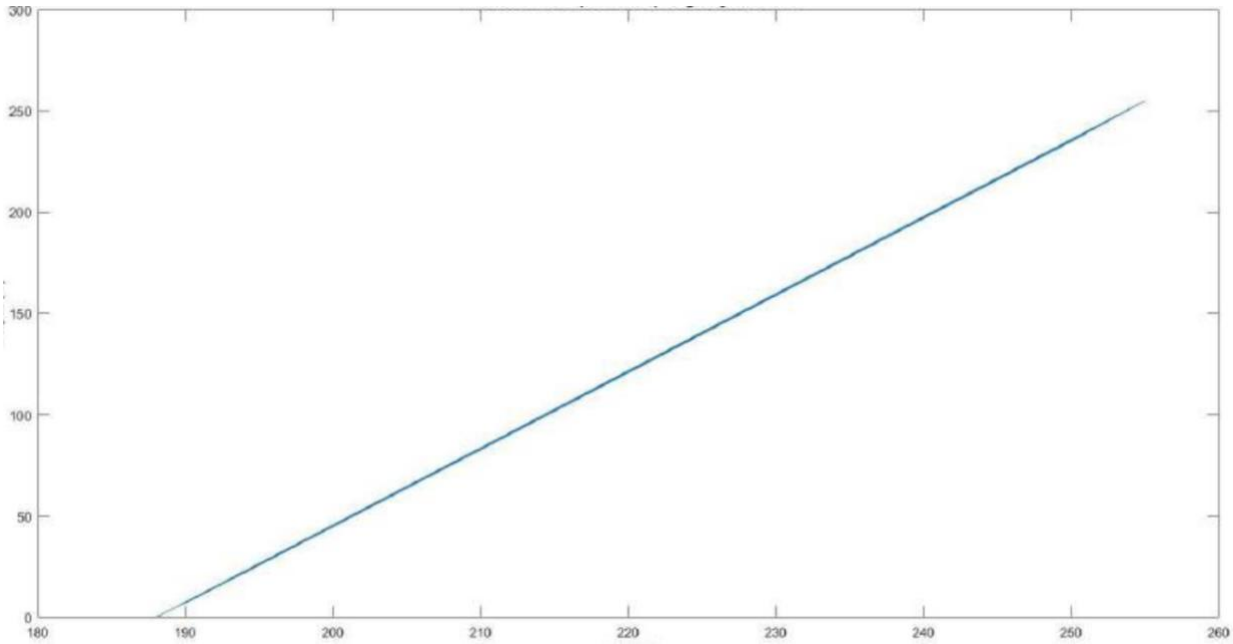


Figure 18: Transfer function of rose_bright using Method A

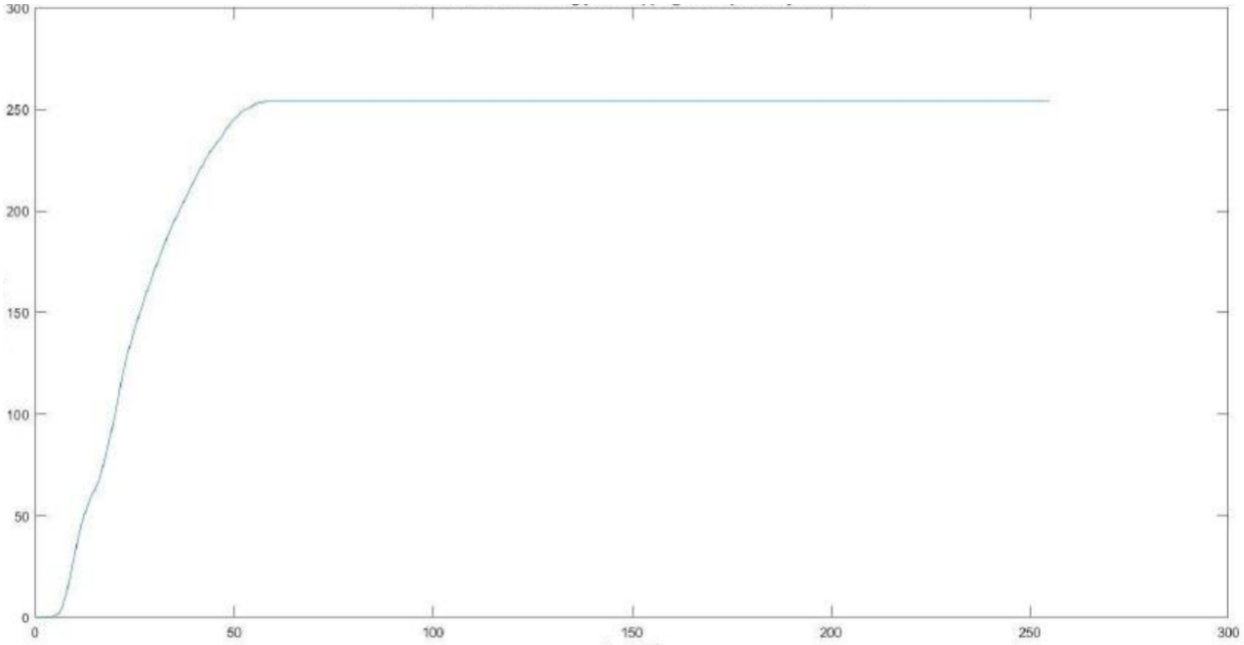


Figure 19: Transfer function of rose_dark using Method B

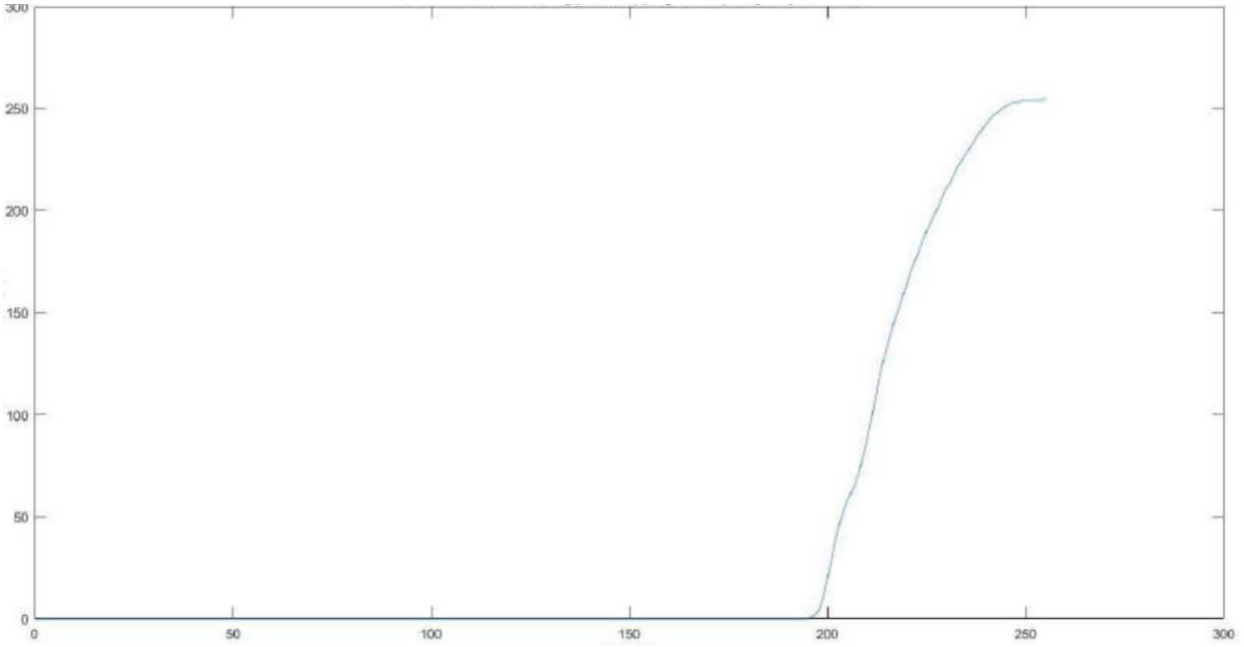


Figure 20: Transfer function of rose_bright using Method B

Figure 21: rose_mix input image

Figure 22: enhanced image of rose_mix image using Method A

Figure 23: enhanced image of rose_mix image using Method B

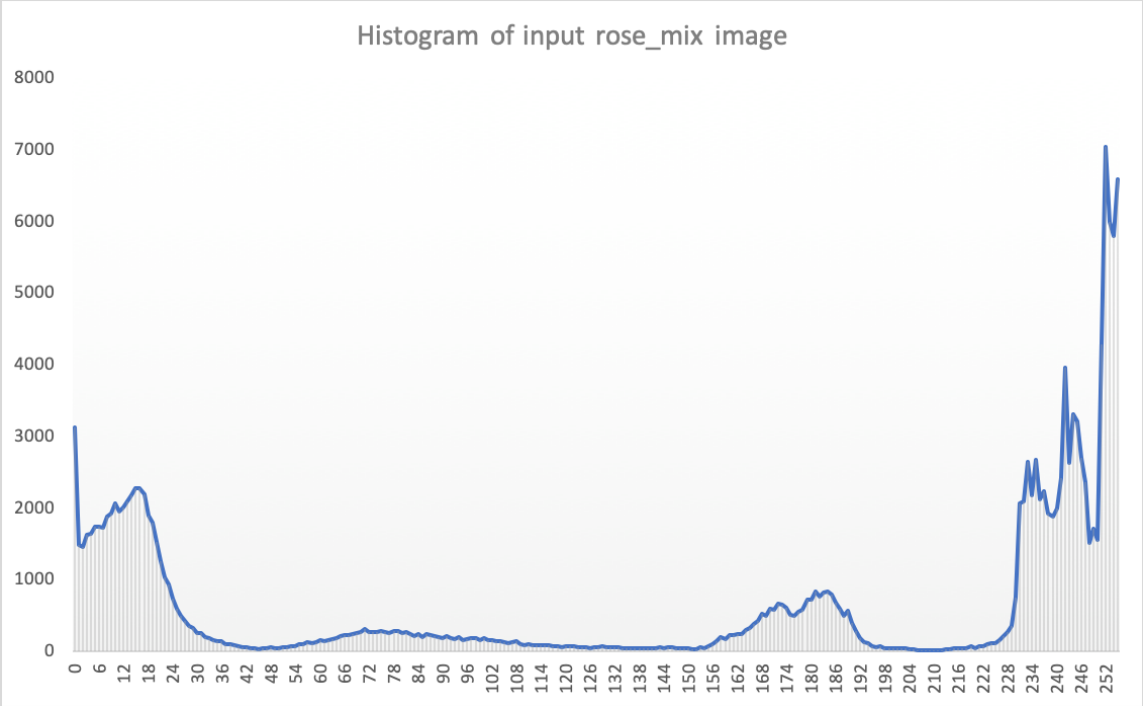


Figure 24: histogram of input rose_mix image

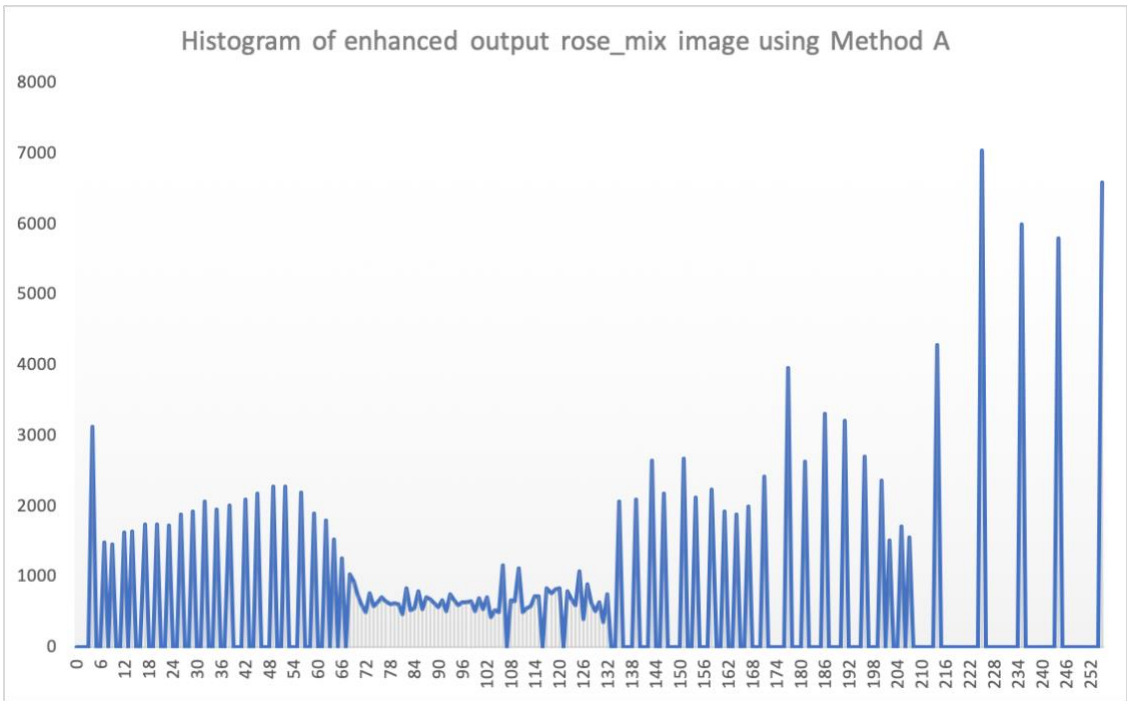


Figure 25: histogram of enhanced output rose_mix using Method A

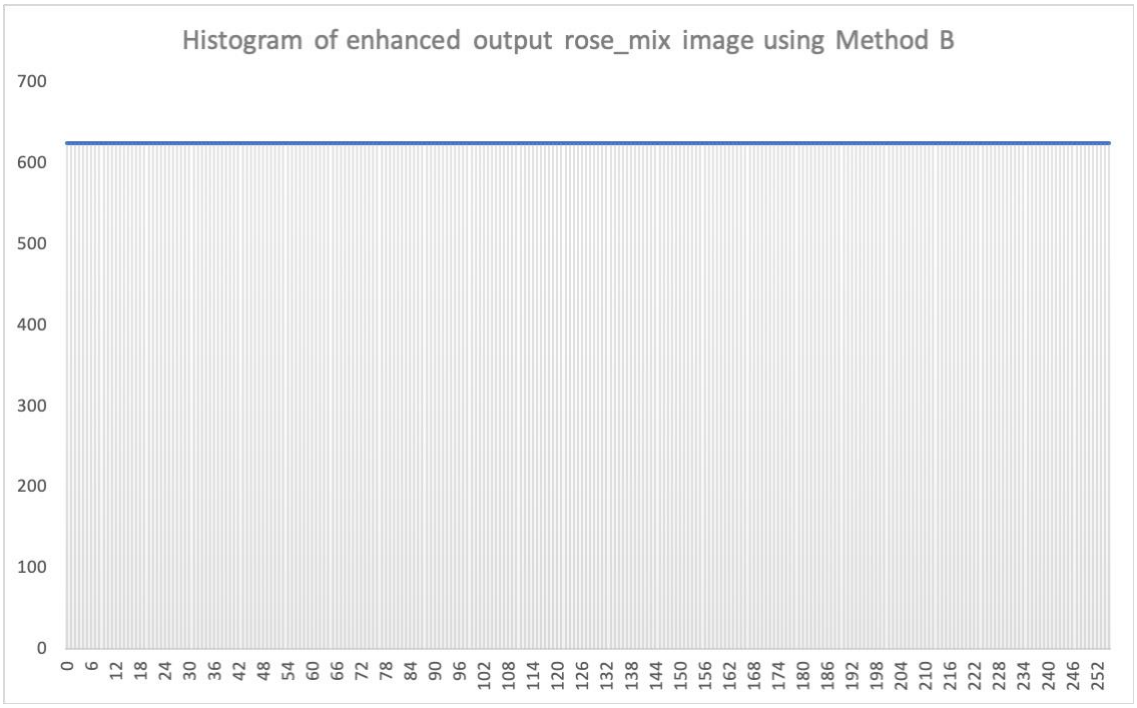


Figure 26: histogram of enhanced output rose_mix using Method B

IV. Discussion

The histogram of all the images is plotted above. The figures have intensity value as the x axis and the number of pixels as the y axis.

The contrast enhanced images of rose_dark and rose_bright by using method A are shown above. The transfer function for each testing image is plotted.

The contrast enhanced images of rose_dark and rose_bright by using method B are shown above. The cumulative histogram for each testing image is plotted.

From the two enhancement results, the following is observed:
Method A is like expanding the histogram. It is simple and enhances the contrast of the image. This method worked in our input image because the histogram was concentrated towards the center. However, if the pixel values were far apart from each other then this method would fail to provide a good contrast enhancement. This is because the entire range of 0 to 255 is not utilized in this method. Also, the brightness of the image seems altered.

Method B perfectly equalizes the histogram. It creates a well enhanced image. The number of pixels in every bucket is equal as we can see in the histogram above. It has better performance as compared to the Method A in contrast enhancement. The output image is very pleasing in terms of contrast to the human eye. It produces lesser loss of quality. But since every pixel has to be checked and put into the bucket, the process becomes tedious and computationally expensive for larger images.

Both these methods have their own pros and cons, we can also use other histogram equalization methods to give even better results such that the computational process will be easier than Method B and also equalize the histogram over the entire dynamic range in a more spreaded manner than Method A. There is a variety of other methods which we can choose to improve the results further like Bi-histogram equalization, Par sectioning, adaptive histogram equalization.

The results of rose_mix by using the implemented Method A and B are shown above. Method A improves the dynamic range of the image as seen in the histogram above, however there are pixel values at the extremes on both sides. Hence there is not much change observed on the equalization histogram overall. The image appears too bright on some parts and too dark on some. If I had a chance to implement my part on it, I would separate the two regions having bright and dark components together and equalize them separately and combine them. This will cause better enhancement as the histogram will be spread equally and create better contrast.

Method B did a very good job on the contrast enhancement and as we see from the histogram, it is equally distributed pixel intensities throughout the image which caused a very good output. Method B is a better method to implement than Method A for this case.

Problem 2: Image Denoising

(a) Gray-level Image

I. Abstract and Motivation

Image Denoising is a very important topic in the industry. In real world, noise is present in all kinds of digital images (gray-scale and color) we capture. Different types of noises observed in the images are Uniform noise, Gaussian noise, Salt and Pepper noise. This causes distortion in the image and does not give a clear output for the humans to observe and infer. Hence, denoising is very crucial to make the image pure. Different types of denoising algorithms are used like Mean filter, Low pass filter, Gaussian filter, Median filter, Non-local means filter, BM3D transform, Partial differential equation filter.

Noise is usually high frequency and image is low frequency, hence we can use low pass filter (averaging) also called as mean filter to kill the noise. However, as edges are also high frequency, LPF causes blurred edges.

Performance of each of these filters can be compared based on PSNR values for the input-output images. PSNR stands for peak signal-to-noise ratio and MSE denotes mean square error which is calculated by the formula below.

$$\text{PSNR(dB)} = 10\log_{10}\left(\frac{\text{Max}^2}{\text{MSE}}\right)$$

$$\text{MSE} = \frac{1}{NM}\sum_{i=1}^{N}\sum_{j=1}^{M}\left(Y(i,j) - I(i,j)\right)^2$$

Two types of noises – Uniform and Impulse noise:
(Source: EE 569 – Discussion 2)

Uniform noise is also known as white Gaussian noise. Noise value is some kind of random variables following Gaussian distribution. We use the mean filter to handle the uniform noise. This filter calculates the mean for the pixel values, using 3x3 mean kernel using the formula below. Also, the weight is calculated as below. The value sigma is the standard deviation of Gaussian distribution and is varied for different values to observe the PSNR. The window size is also varied so as to get

different outputs and compare the results. This filter contributes to the denoising effect.

- Image denoising:
  - uniform: low pass filter (uniform)

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

w_1

1	1	1
1	1	1
1	1	1

3x3 Mean kernel

$$w(i,j,k,l) = \frac{1}{w_1 \times w_2}$$

where (k,l) is the neighboring pixel location within the window of size w_1 x w_2 centered around (i,j), I is the noisy image

Another low pass filter used to remove uniform noise is Gaussian filter. It gets the weighted average of some local neighborhood for certain pixels, using 5x5 Gaussian kernel using the formula below. Also, the weight is calculated as below. The value sigma is the standard deviation of Gaussian distribution and is varied for different values to observe the PSNR. The window size is also varied so as to get different outputs and compare the results. This filter contributes to the denoising effect.

- Image denoising:
  - uniform: low pass filter (Gaussian)

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

w_1

$\frac{1}{273}$

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

$$w(i,j,k,l) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(k-i)^2 + (l-j)^2}{2\sigma^2}\right)$$

where σ is the standard deviation of Gaussian distribution

Impulse noise results due to dead sensor resulting in black output (0) and saturated sensor resulting in white output (255). Thus, the image contains black and white dots on it, also called as salt and pepper noise. We use the median filter which is non-linear to handle the impulse noise by sorting the elements and finding the middle value while applying the filter. Here, we use the window size to be 3x3. The window size is also varied so as to get different outputs and compare the results. Below are the steps to use this filter.

- Image denoising:
  - impulse: median filter (non-linear)
    1. consider each pixel in the image
    2. sort the neighboring pixels into order based upon their pixel values
    3. replace the centered pixel value with the median value from the list

One more denoising method is using Bilateral filters. The above filters may result in blur and hence to retain the sharp edges, this filter is used. Weights depend not only on the Euclidean distance of pixels but also on the difference on the pixel values. The equation for the filter is provided below.

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

$$w(i,j,k,l) = exp\left(-\frac{(i-k)^2 + (j-l)^2}{2\sigma_c^2} - \frac{\|I(i,j) - I(k,l)\|^2}{2\sigma_s^2}\right)$$

where σ_c and σ_s are the spread parameters of your choice

Another method is the Nonlocal Mean filter. We can remove noise by creating multiple copies of noisy images, adding them and using ensemble mean. Get multiple samples from the spatial domain in the same location, from the ones having similar pattern/neighborhood. This is the non-local mean (NLM) algorithm. This is the weighted sum of the similar points in the neighborhood depending on the distance between the points. Correlation in the NLM for points p and q is better/stronger than correlation in LPF for points p and p+d, as in NLM, p and q have more similar neighborhood.

$$Y(i,j) = \frac{\sum_{k=1}^{N'} \sum_{l=1}^{M'} I(k,l) f(i,j,k,l)}{\sum_{k=1}^{N'} \sum_{l=1}^{M'} f(i,j,k,l)}$$

$$f(i,j,k,l) = \exp\left(-\frac{\left\| I(N_{i,j}) - I(N_{k,l}) \right\|_{2,a}^2}{h^2} \right)$$

$$\left\| I(N_{i,j}) - I(N_{k,l}) \right\|_{2,a}^2 = \sum_{n_1,n_2 \in N} G_a(n_1,n_2)(I(i-n_1, j-n_2) - I(k-n_1, l-n_2))^2$$

and

$$G_a(n_1,n_2) = \frac{1}{\sqrt{2\pi}a} \exp\left(-\frac{n_1^2+n_2^2}{2a^2}\right)$$

Below are the steps to use this filter.

- Non-local mean filter:

1. takes Gaussian weighted Euclidean distance between the block centered the target pixel and the neighboring block
2. computationally intensive
3. To speed up, choose a smaller window size rather than the whole image

II. Approach and Procedure

We apply the Mean filter, Median filter, Gaussian filter, Bilateral filter and NLM filter to the pepper_noise.raw image and get the denoised output for various window sizes (NxN) and also calculate the PSNR for each of the N and compare the performance of the filters. The output image for each of these parameters is shown below.

Algorithm implemented in C++ :

- Read rose_color_noise.raw input images using fread() function and get height, width and bytesperpixel values
- Run 3 nested for loops for height (256), width (256) and bytesperpixel (=3) to access each pixel in the original image
- Provide the required NxN window size

- Apply the Mean/Median/Gaussian/NLM/Bilateral filter using the algorithm mentioned below
- Create a 3D array to store the denoised output
- Calculate the PSNR to compare the performance
- Write the computed image output data array on denoised_rose_color.raw file using the fwrite() function

Mean (Low Pass Filter) algorithm in C++:

- For the index of rows and columns of filter, if index < 0 then assign 0
- If row index greater than row, assign row value and if column index greater than column, assign column value
- This is done for boundary pixels to avoid boundary effect as there is no extension in boundary
- Now store NxN neighborhood pixels in 1D array
- Find the average of the array
- The denoised image pixel value is calculated using this average value

Median algorithm in C++:

- For the index of rows and columns of filter, if index < 0 then assign 0
- If row index greater than row, assign row value and if column index greater than column, assign column value
- This is done for boundary pixels to avoid boundary effect as there is no extension in boundary
- Now store NxN neighborhood pixels in 1D array
- Sort the array using the bubble sort algorithm
- Find the middle value of this sorted array
- The denoised image pixel value is calculated using this middle value

Gaussian algorithm in C++:

- Calculate the Gaussian NxN window using formula and normalize it
- For the index of rows and columns of filter window, if index < 0 then assign 0
- If row index greater than row, assign row value and if column index greater than column, assign column value

- This is done for boundary pixels to avoid boundary effect as there is no extension in boundary
- Now store NxN neighborhood pixels in 1D array
- Multiply the corresponding pixel values with Gaussian window
- The denoised image pixel value is calculated using this value

Bilateral algorithm in C++:

- For the index of rows and columns of filter, if index < 0 then assign 0
- If row index greater than row, assign row value and if column index greater than column, assign column value
- This is done for boundary pixels to avoid boundary effect as there is no extension in boundary
- Calculate weight by denoting the sigma parameters
- Apply the equation and find Y
- The denoised image pixel value is calculated using this value

III. Experimental Results

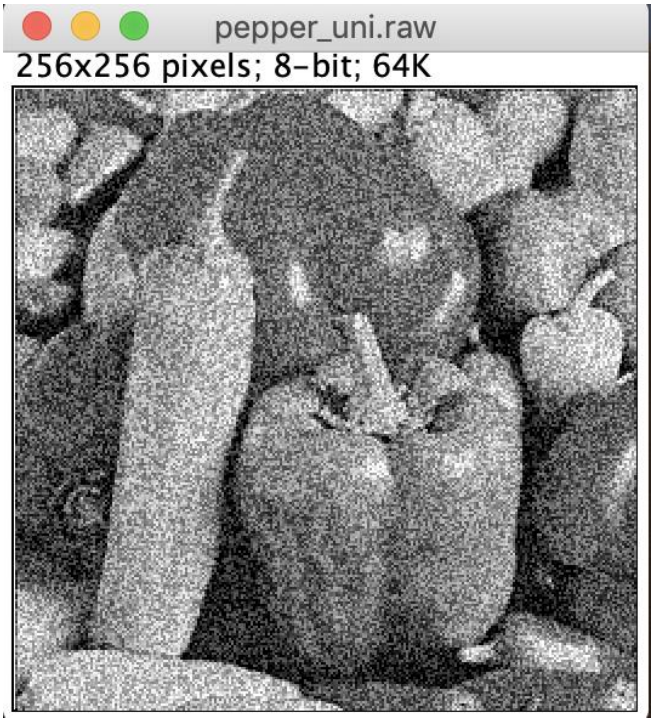


Figure 27: noisy pepper_uni.raw input image

Using Mean Filter (low pass filter):

Window size = 3x3
PSNR = 17.2793 dB

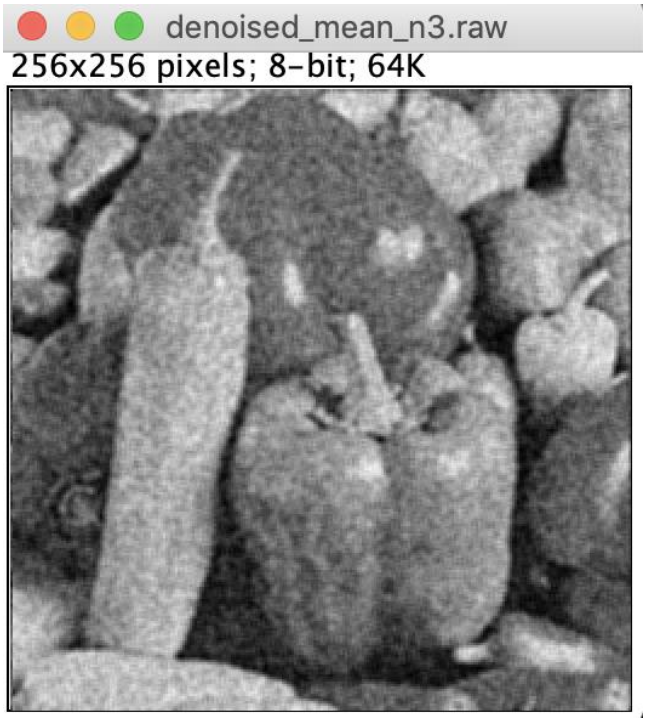


Figure 28: Denoised output image using Mean filter for window size 3x3

Window size = 5x5
PSNR = 16.6608 dB

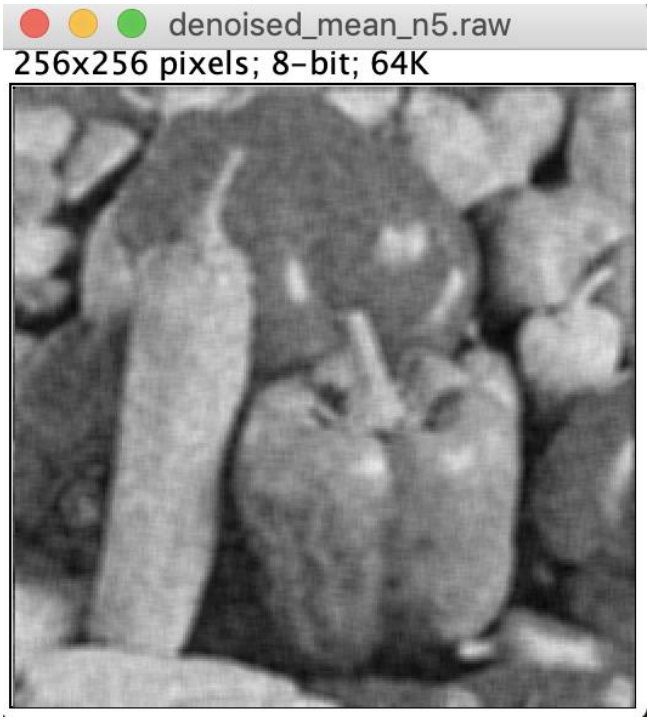


Figure 29: Denoised output image using Mean filter for window size 5x5

Window size = 7x7
PSNR = 16.3342 dB

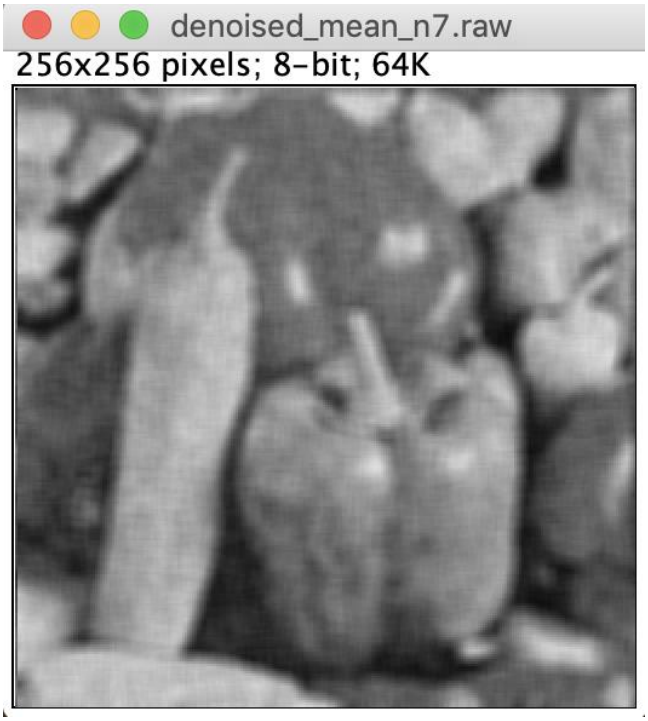


Figure 30: Denoised output image using Mean filter for window size 7x7

Window size = 9x9
PSNR = 16.0769 dB

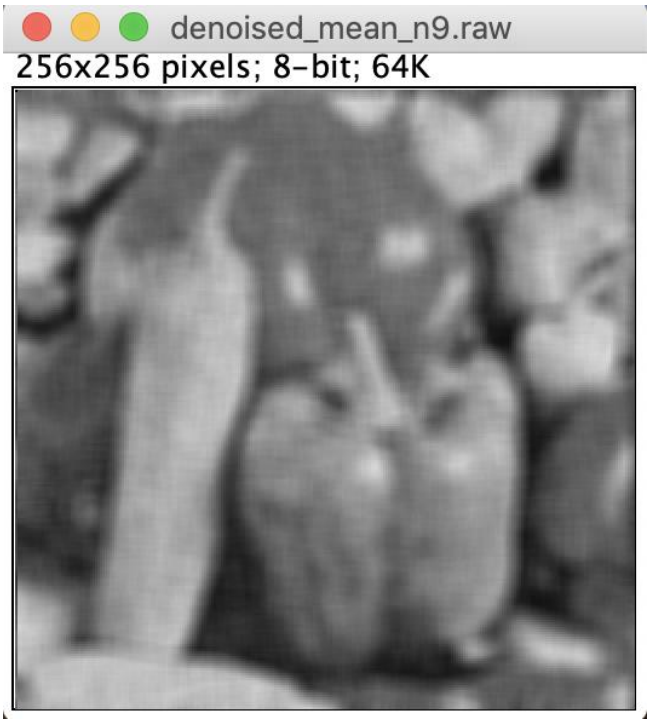


Figure 31: Denoised output image using Mean filter for window size 9x9

Using Median Filter:

Window size = 3x3
PSNR = 16.9789 dB

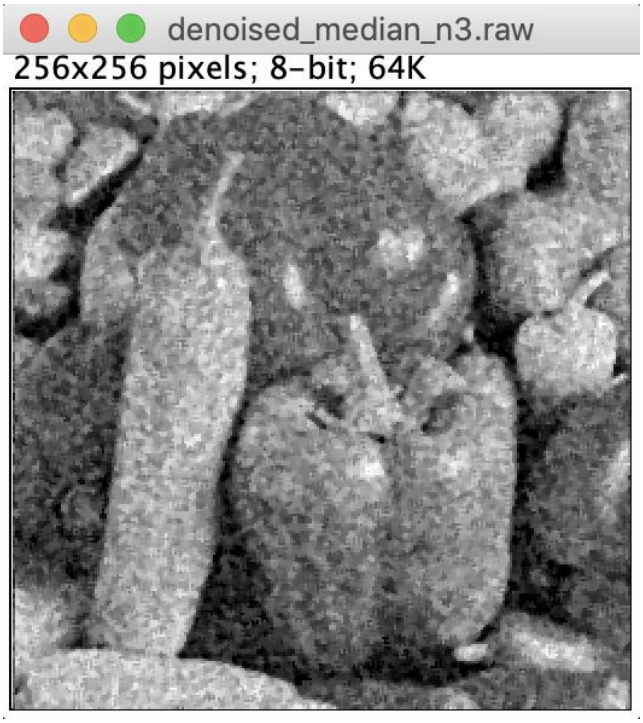


Figure 32: Denoised output image using Median filter for window size 3x3

Window size = 5x5
PSNR = 16.5936 dB

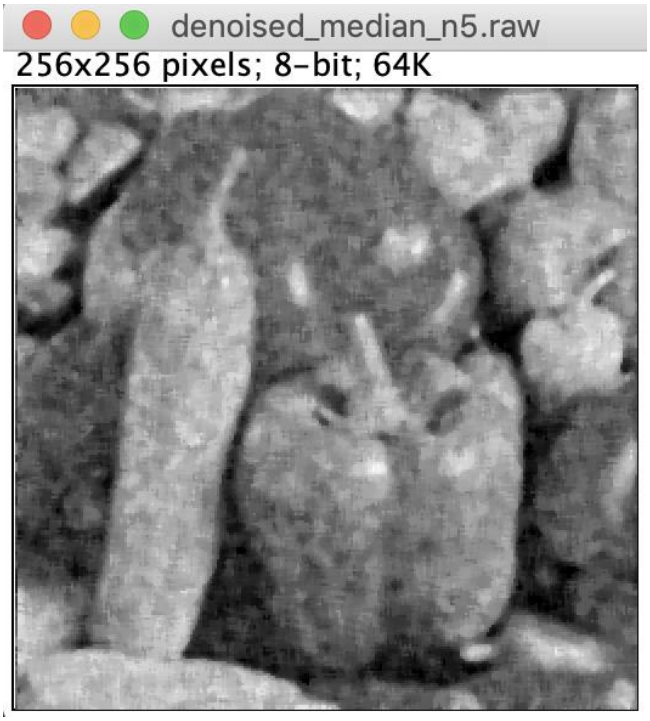


Figure 33: Denoised output image using Median filter for window size 5x5

Window size = 7x7
PSNR = 16.3559 dB

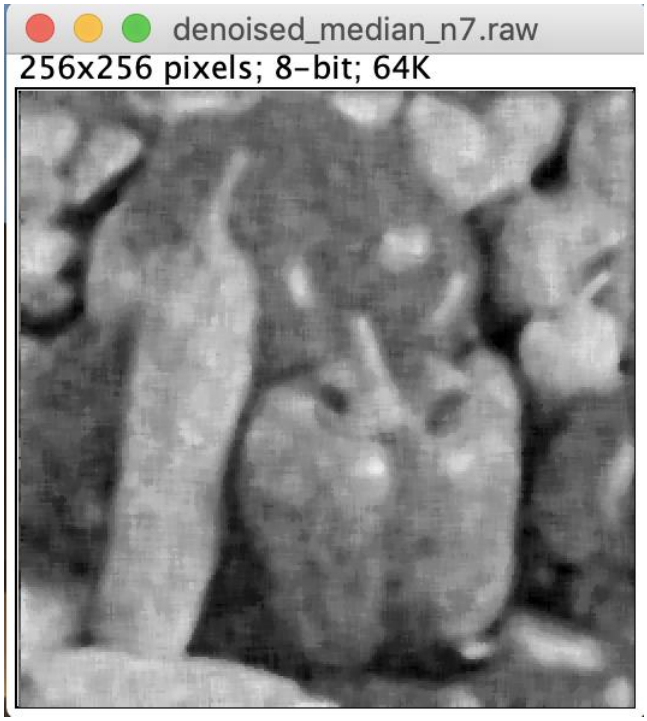


Figure 34: Denoised output image using Median filter for window size 7x7

Window size = 9x9
PSNR = 16.1444 dB

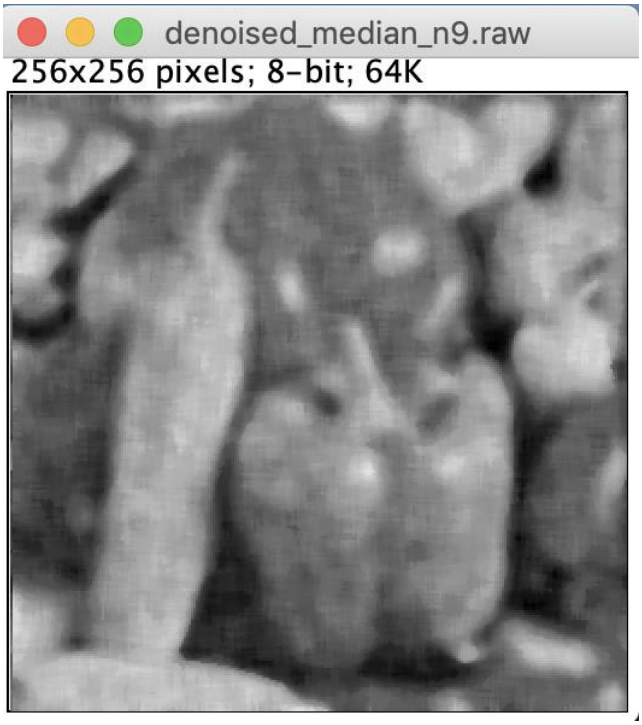


Figure 35: Denoised output image using Median filter for window size 9x9

Using Gaussian Filter:

Window size = 3x3
PSNR = 16.7566 dB

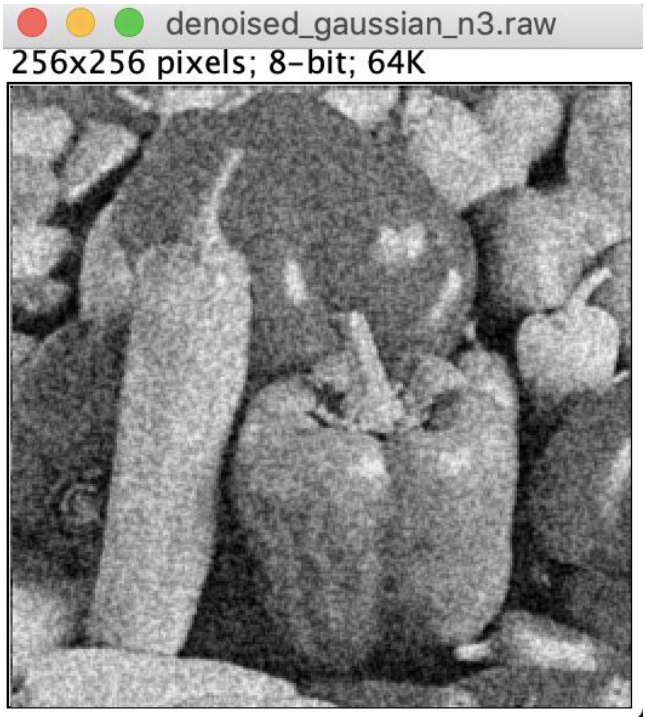


Figure 36: Denoised output image using Gaussian filter for window size 3x3

Window size = 5x5
PSNR = 14.9684 dB

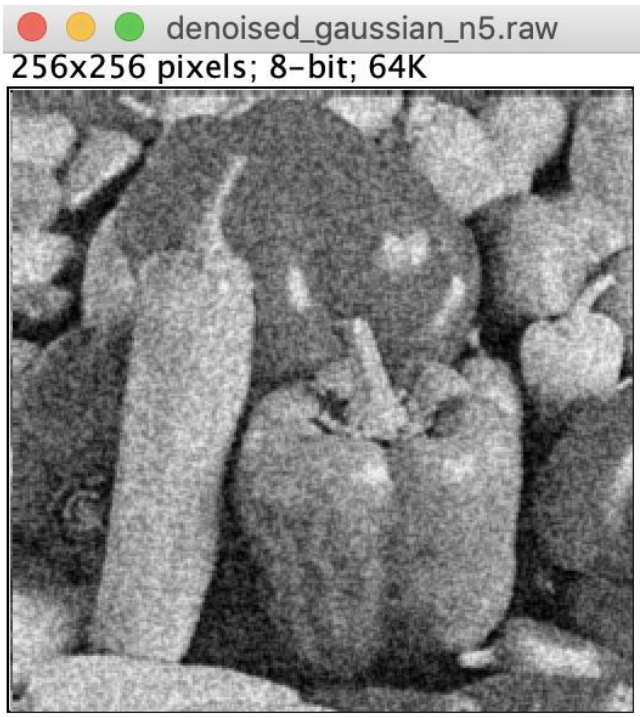


Figure 37: Denoised output image using Gaussian filter for window size 5x5

Window size = 7x7
PSNR = 14.1171 dB

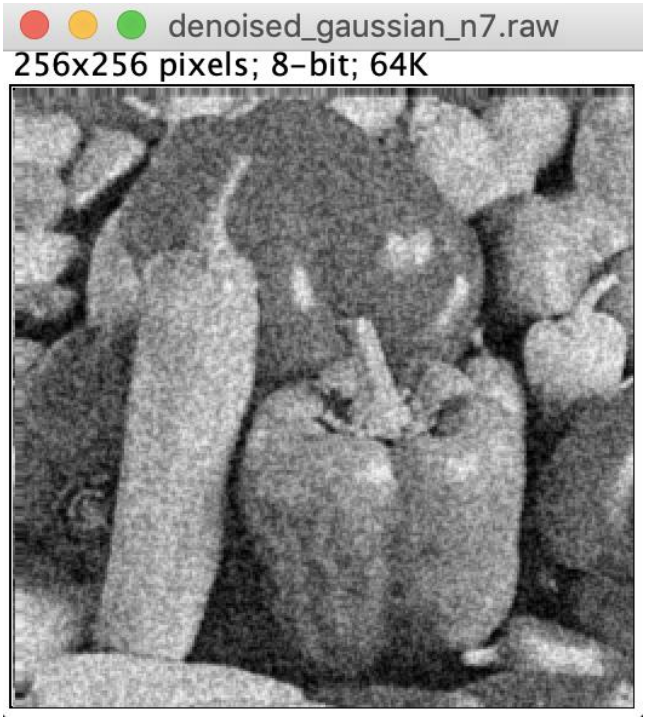


Figure 38: Denoised output image using Gaussian filter for window size 7x7

Window size = 9x9
PSNR = 13.5083 dB

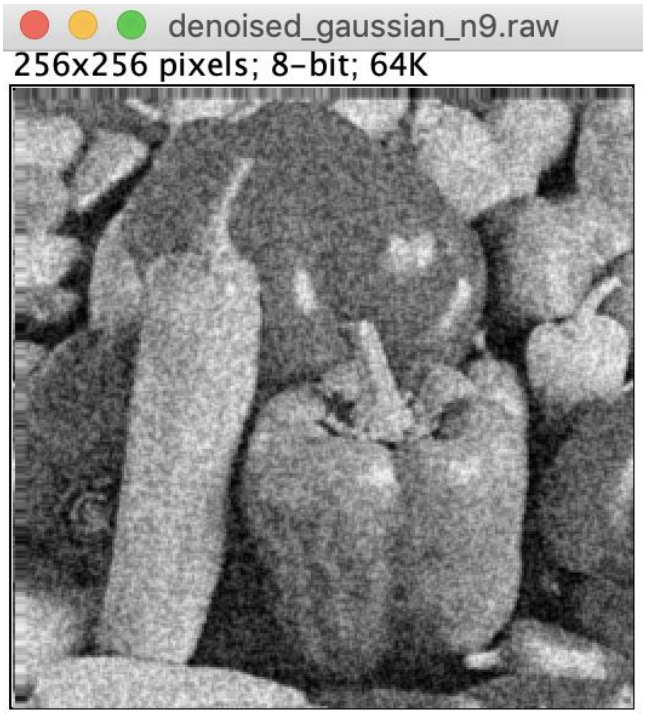


Figure 39: Denoised output image using Gaussian filter for window size 9x9

Using Bilateral Filter:

Window size = 3x3
PSNR = 68.4584 dB

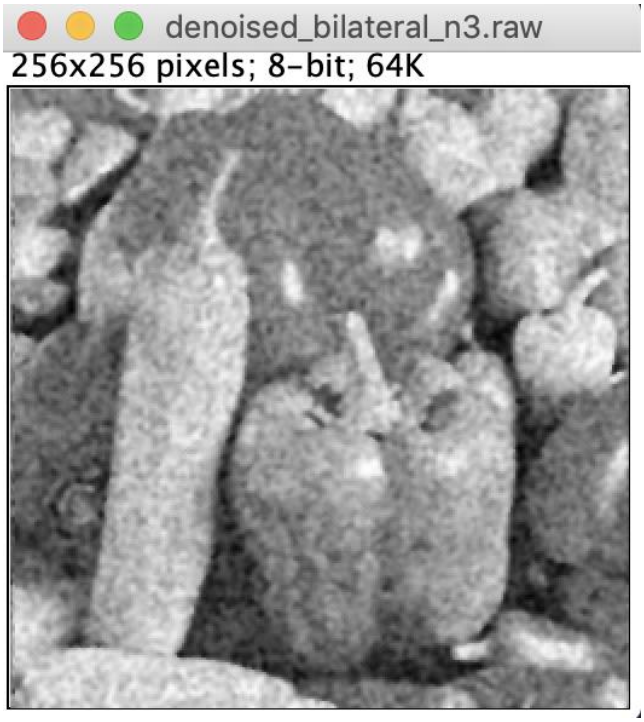


Figure 40: Denoised output image using Bilateral filter for window size 3x3

Window size = 5x5
PSNR = 66.9437 dB

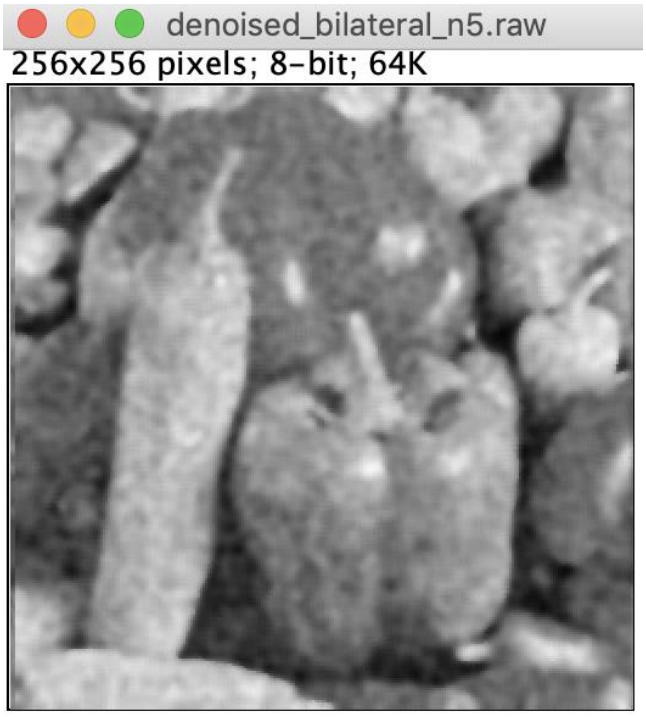


Figure 42: Denoised output image using Bilateral filter for window size 5x5

Window size = 7x7

PSNR = 65.8323 dB

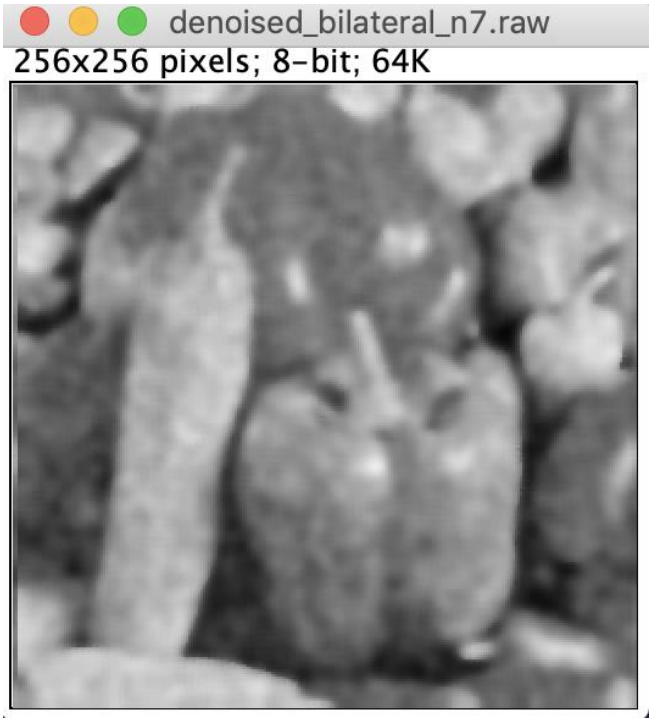


Figure 43: Denoised output image using Bilateral filter for window size 7x7

Window size = 9x9
PSNR = 65.7941 dB

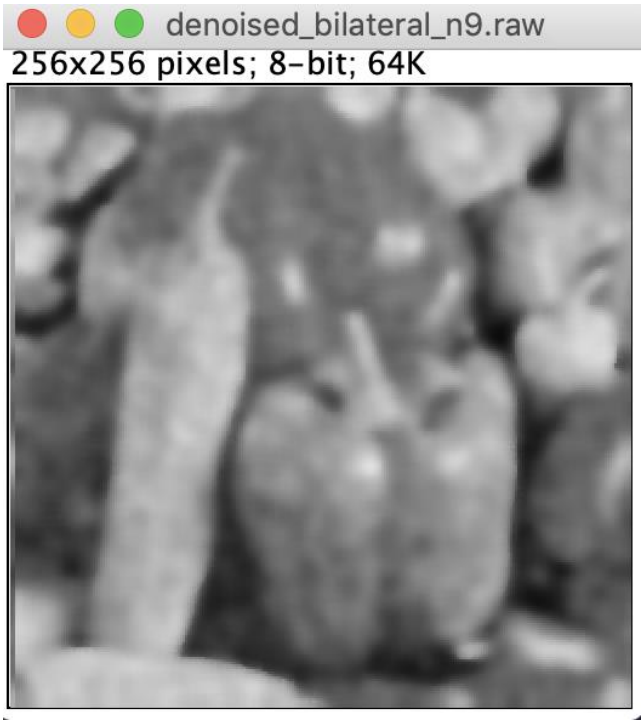


Figure 44: Denoised output image using Bilateral filter for window size 9x9

IV. Discussion

The type of embedded noise can be found out by calculating the difference between the noisy pepper image and the original pepper image. Hence, we can find out that the embedded noise is of uniform type of noise.

The linear filter of window size NxN is applied where N can take odd values – 3,5,7,9.., N if small isn't so powerful but if N is large, it causes blur.

The results showing PSNR values for different window sizes for different types of filters and the output denoised images is shown above.

The performance of all the filters can be compared based on their PSNR values, effect of the window size (NxN). By looking above and comparing, we can say that the PSNR goes on decreasing when increasing the window size and also the output becomes blurry.

For removing uniform noise, we can use Mean (LPF) and Gaussian filter. For removing impulse noise, we can use Median, Bilateral and NLM filter. For removing the uniform noise, Gaussian filter and low pass filter gives a better output than median filter. We also notice that PSNR for Mean filter is higher as compared to Gaussian filter for different window sizes. The Gaussian filter PSNR decreases rapidly as the window size is increased gradually whereas for the Mean filter the PSNR drops by a lesser value as the window size increases.

In the above images, we noticed blur around the edges at high frequency components of pixels. Thus, there is a degradation of edges, however, we can use some kind of filters to preserve these edges. Bilateral filter is one among them.

NLM filter gives the best results as it connects to more similar neighboring pixels and also uses for parameters to get a better output.

(b) Color Image

I. Abstract and Motivation

Performance of each of these filters can be compared based on PSNR values for the input-output images. PSNR stands for peak signal-to-noise ratio and MSE denotes mean square error which is calculated by the formula below.

$$\text{PSNR(dB)} = 10\log_{10}\left(\frac{\text{Max}^2}{\text{MSE}}\right)$$

$$\text{MSE} = \frac{1}{NM}\sum_{i=1}^{N}\sum_{j=1}^{M}(Y(i,j) - I(i,j))^2$$

Two types of noises – Uniform and Impulse noise:
(Source: EE 569 – Discussion 2)

Uniform noise is also known as white Gaussian noise. Noise value is some kind of random variables following Gaussian distribution. We use the mean filter to handle the uniform noise. This filter calculates the mean for the pixel values, using 3x3 mean kernel using the formula below. Also, the weight is calculated as below. The value sigma is the standard deviation of Gaussian distribution and is varied for different values to observe the PSNR. The window size is also varied so as to get different outputs and compare the results. This filter contributes to the denoising effect.

- Image denoising:
  - uniform: low pass filter (uniform)

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

$$w(i,j,k,l) = \frac{1}{w_1 \times w_2}$$

1	1	1
1	1	1
1	1	1

3x3 Mean kernel

where (k,l) is the neighboring pixel location within the window of size w_1 x w_2 centered around (i,j), I is the noisy image

Another low pass filter used to remove uniform noise is Gaussian filter. It gets the weighted average of some local neighborhood for certain pixels, using 5x5 Gaussian

kernel using the formula below. Also, the weight is calculated as below. The value sigma is the standard deviation of Gaussian distribution and is varied for different values to observe the PSNR. The window size is also varied so as to get different outputs and compare the results. This filter contributes to the denoising effect.

- Image denoising:
  - uniform: low pass filter (Gaussian)

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

w_1

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

$$w(i,j,k,l) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(k-i)^2 + (l-j)^2}{2\sigma^2}\right)$$

where σ is the standard deviation of Gaussian distribution

Impulse noise results due to dead sensor resulting in black output (0) and saturated sensor resulting in white output (255). Thus, the image contains black and white dots on it, also called as salt and pepper noise. We use the median filter which is non linear to handle the impulse noise by sorting the elements and finding the middle value while applying the filter. Here, we use the window size to be 3x3. The window size is also varied so as to get different outputs and compare the results. Below are the steps to use this filter.

- Image denoising:
  - impulse: median filter (non-linear)
    1. consider each pixel in the image
    2. sort the neighboring pixels into order based upon their pixel values
    3. replace the centered pixel value with the median value from the list

One more denoising method is using Bilateral filters. The above filters may result in blur and hence to retain the sharp edges, this filter is used. Weights depend not only on the Euclidean distance of pixels but also on the difference on the pixel values. The equation for the filter is provided below.

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

$$w(i,j,k,l) = exp\left(-\frac{(i-k)^2 + (j-l)^2}{2\sigma_c^2} - \frac{\|I(i,j) - I(k,l)\|^2}{2\sigma_s^2}\right)$$

where σ_c and σ_s are the spread parameters of your choice

Another method is the Nonlocal Mean filter. We can remove noise by creating multiple copies of noisy images, adding them and using ensemble mean. Get multiple samples from the spatial domain in the same location, from the ones having similar pattern/neighborhood. This is the non-local mean (NLM) algorithm. This is the weighted sum of the similar points in the neighborhood depending on the distance between the points. Correlation in the NLM for points p and q is better/stronger than correlation in LPF for points p and p+d, as in NLM, p and q have more similar neighborhood.

$$Y(i,j) = \frac{\sum_{k=1}^{N'} \sum_{l=1}^{M'} I(k,l) f(i,j,k,l)}{\sum_{k=1}^{N'} \sum_{l=1}^{M'} f(i,j,k,l)}$$

$$f(i,j,k,l) = \exp\left(-\frac{\left\|I(N_{i,j}) - I(N_{k,l})\right\|_{2,a}^2}{h^2}\right)$$

$$\left\|I(N_{i,j}) - I(N_{k,l})\right\|_{2,a}^2 = \sum_{n_1,n_2 \in N} G_a(n_1,n_2)(I(i-n_1, j-n_2) - I(k-n_1, l-n_2))^2$$

and

$$G_a(n_1,n_2) = \frac{1}{\sqrt{2\pi}a} \exp\left(-\frac{n_1^2+n_2^2}{2a^2}\right)$$

Below are the steps to use this filter.

- Non-local mean filter:

1. takes Gaussian weighted Euclidean distance between the block centered the target pixel and the neighboring block
2. computationally intensive
3. To speed up, choose a smaller window size rather than the whole image

II. Approach and Procedure

Here, since input rose image is a color image, we have to deal with all 3 channels individually. Hence to remove these noises, we will require a set of filters applied to each color channels individually.

To remove the uniform noise, we can use Low Pass Filter or Gaussian filter. To remove the impulse noise, we can use Median filter or Bilateral or NLM filter. These filters are used in cascading.

We apply the Median filter and then Gaussian filter so as to get a better result. Also, we can compare the different combinations of cascade. We can apply the NLM filter and then Gaussian or Bilateral filter then Gaussian.

Algorithm implemented in C++ :

- Read rose_color_noise.raw input images using fread() function and get height, width and bytesperpixel values
- Run 3 nested for loops for height (256), width (256) and bytesperpixel (=3) to access each pixel in the original image
- Provide the required NxN window size
- Separate RGB channels : R → 0, G → 1, B → 2
- Apply the Median/NLM/Bilateral filter to each of these channels individually using the algorithm mentioned above
- Apply the Gaussian/Low Pass filter to each of these channels individually using the algorithm mentioned above
- Create a 3D array to store the denoised output
- Calculate the PSNR to compare the performance
- Write the computed image output data array on denoised_rose_color.raw file using the fwrite() function

III. Experimental Results

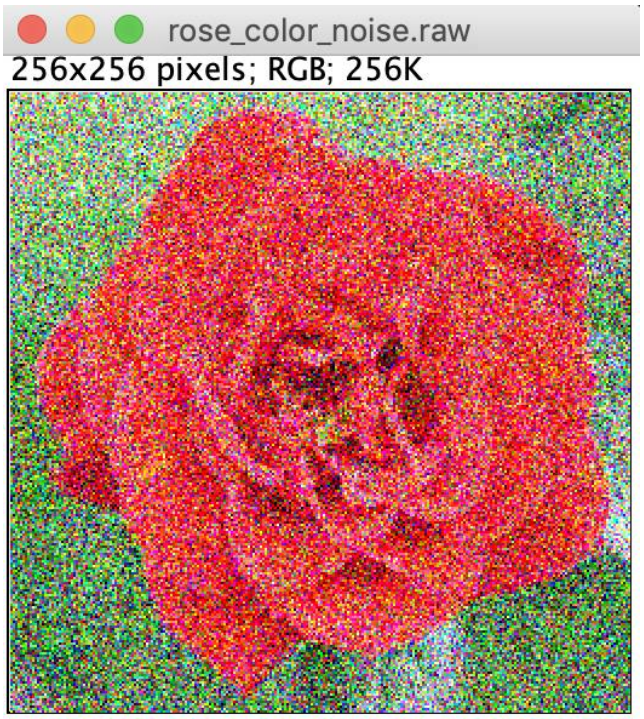


Figure 45: noisy rose_color.raw input image

Using Median Filter first and then Gaussian filter:

Window size = 3x3
PSNR = 23.9139 dB

Figure 46: Denoised output image using Median then Gaussian filter for window size 3x3

Window size = 5x5
PSNR = 19.4591 dB

Figure 47: Denoised output image using Median then Gaussian filter for window size 5x5

Window size = 7x7
PSNR = 15.6893 dB

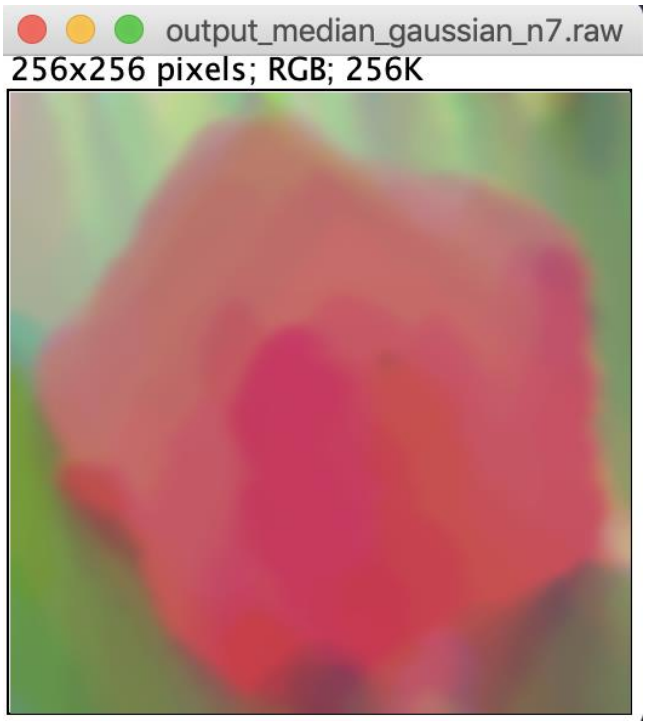


Figure 48: Denoised output image using Median then Gaussian filter for window size 7x7

Using Median Filter first and then Gaussian filter:

Window size = 3x3
PSNR = 6.0827 dB

Figure 49: Denoised output image using Gaussian then Median filter for window size 3x3

Window size = 5x5
PSNR = 6.9457 dB

Figure 50: Denoised output image using Gaussian then Median filter for window size 5x5

Window size = 7x7
PSNR = 6.9192 dB

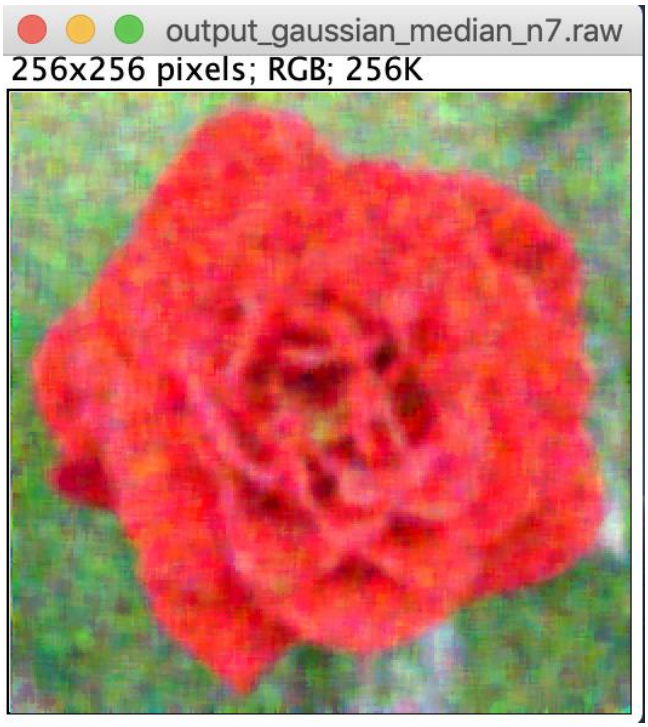


Figure 51: Denoised output image using Gaussian then Median filter for window size 7x7

Window size = 9x9
PSNR = 6.8965 dB

Figure 52: Denoised output image using Gaussian then Median filter for window size 9x9

IV. Discussion

The results showing PSNR values and the output denoised images is shown above.

Mixed rose noisy color image has 3 channels – R, G, B which are each corrupted by impulse and uniform noise. Hence to remove these noises, we will require a set of filters applied to each color channels individually.

Filtering should be performed on individual channels separately for both noise types. This is because all channels have mixed noises present and applying filters to the whole would not result in a clean image as compared to filters applied to each channel individually. Red channel has gaussian and salt noise, Blue channel has gaussian and pepper noise, Green channel has gaussian and salt and pepper noise present. Hence, denoising is done by applying filters to individual channels for getting better results soothing to the human eye.

As the nature of noise is mixed noise, we need to use both linear and non-linear filters to remove uniform and impulse noise. Hence, we need to cascade two kind of filters so as to remove both types of noise.

To remove the uniform noise, we can use Low Pass Filter or Gaussian filter. To remove the impulse noise, we can use Median filter or Bilateral or NLM filter. However, after trying all the above combinations, I came to the conclusion that the better results came when I used Median filter first, and then used Gaussian filter. Also, there were good results obtained when I used NLM or Bilateral filter along with Gaussian filter. By using combinations of these filters in cascade, I was able to get denoised color output. The reason why we shouldn't cascade in the order of Gaussian first and then Median filter is that the salt and pepper noise present in the noisy image will be averaged throughout the image and become more effective and hence it will become very difficult to remove the impulse noise by using Gaussian filter.

Also, along with the above points, I observed that the window size should be small, preferably 3x3. This is because when I observed the output for 5x5 and 7x7 window sizes, the denoised output image appeared quite blurry and also the PSNR values went on decreasing for a larger window size.

However, these parameters all depend upon the type of input image, at times the window size should be medium large so as to get higher PSNR. Also, for some images Gaussian filter should be used before Median filter and it gives a better output. It is relative and very much dependent on the input image and type of noise present or distributed in it, on which kind of filter or filters in cascade should be used to get the best output pleasing to the human eye.

It is difficult to remove uniform noise satisfactorily. Filters may blur the object's edge when smoothing noise. A successful design of a uniform noise filter depends on its ability to distinguish the pattern between the noise and the edges. An alternative to the low pass filter with Gaussian weight function which can perform better is Bilateral filter, NLM (Nonlocal Mean) filter, Wavelet transforms, min-max filter, mid-point filter and other statistical methods based on machine learning and state of the art denoising technique. In the low pass filters, we see degradation of edges, but using the Bilateral filter can help preserve these edges. We can implement some better CNN algorithms to make sure that the high frequency components at the edges are not altered or blurred in the image.

(c) Shot Noise

I. Abstract and Motivation

One more type of noise is Shot noise. Electronic camera image sensor, especially the CMOS sensor typically has noise in the dark parts of the captured image. Such noise is called shot noise, which is caused by statistical quantum (the photon) fluctuations. Impulse and uniform noise are independent with the input images whereas shot noise is dependent on the pixel value of the input images. Noise per pixel is dependent on the original pixel value itself and each pixel z is Poisson distribution shown below.

$$P(z) = \frac{\lambda^z e^{-\lambda}}{z!}$$

The steps to deal with the shot noise are:

$$D = f(z) = 2\sqrt{z + \frac{3}{8}}$$

Use this Anscombe forward transform to convert the Poisson distributed pixel values to Gaussian distributed with unit variance. In the output images here, the shot noise will become uniform/Gaussian noise. Thus, we can use any methods above for the filters in transform domain for denoising. After we remove noise in the transform domain, do the inverse transform – using biased or unbiased function given below.

biased:

$$z' = f^{-1}(D) = \left(\frac{D}{2}\right)^2 - \frac{3}{8}$$

unbiased:

$$z' = f^{-1}(D) = \left(\frac{D}{2}\right)^2 - \frac{1}{8}$$

Unbiased function gives a better result.

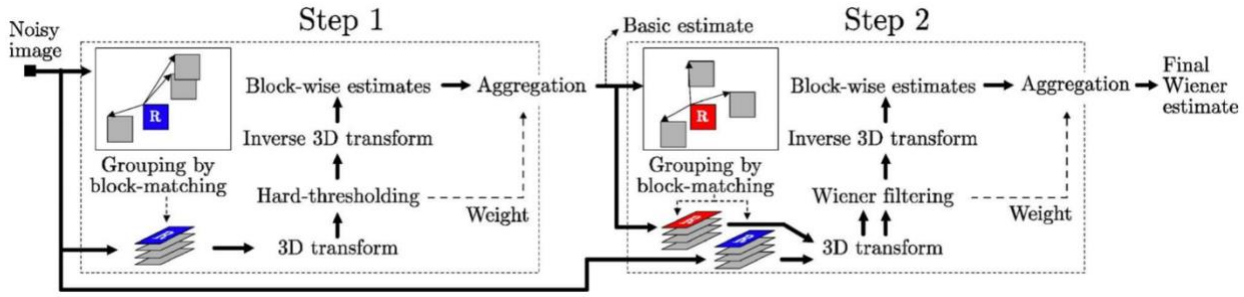
(Source: http://www.cs.tut.fi/~foi/GCF-BM3D/index.html#ref_results)

BM3D is block based denoising tool, which locates similar blocks in an overlapping manner, just as in NLM, similar points are located. Joining these set of blocks, we get a 3D cube and take the 3D transform on the spatial domain and do the filtering in the frequency domain, cutting the high frequency. Take the inverse 3D transform and get the cleaned 3D cube and replace each block back and do the averaging. BM3D tries to handle the uniform and pepper and salt noise.

The three successive steps are: 3D transformation of 3D group, shrinkage of transform spectrum, and inverse 3D transformation. The result is a 3D estimate that consists of the jointly filtered grouped image blocks. By attenuating the noise, the collaborative filtering reveals the finest details shared by grouped blocks and at the same time it preserves the essential unique features of each individual block. The filtered blocks are then returned to their original positions. Because these blocks are overlapping, for each pixel we obtain many different estimates which need to be combined. Aggregation is a particular averaging procedure which is exploited to take advantage of this redundancy.

II. Approach and Procedure

The overview architecture of BM3D method is given below:



Given a noisy image, we first try to do block matching, for example we have some target pixel (i,j) and we set some patch size around (i,j) say 5x5. Then we search the whole image for similar 5x5 patches with given patch (i.j) like say the blue one in the above figure. After, we find all the similar patches, we stack them as a 3D block and we do some 3D transform like Fourier or DCT transform on this 3D block. Then we need some thresholding in the transform domain, so as to ignore some high frequency components. In this way, we do denoising here. After we ignore some high frequency components, we can use inverse 3D transform to go back to the pixel

domain. Thus, we get a stack of patches – one is the target blue one and some similar patches to the target patch (i,j). We can get some activation of denoised patches. After we collect a lot of patches, all these patches will be overlapping with other. So for the final output, we try to handle this overlapping by using weighted average of all the overlapping patches. And aggregate the final denoised images. This gives the output for the first step. In the second step, we need to collect the patches on the denoised images and also combine with the patches in the original noisy images. Now we have two stack of original images -blue and red. We mix them and do some 3D transform to go to the transform domain and use filters for denoising. Then do the inverse transform to go back to image domain. Repeat the steps above for removing the overlapping patches and aggregate all the patches, thus we finally get the final denoised output images. BM3D gives the best results compared to all the denoising methods mentioned above.

Here, the input image contains shot noise. Unlike uniform noise, shot noise is Poisson distributed. First, we apply Anscombe root transformation on each pixel, to the input image, resulting an image with additive Gaussian noise of unit variance. Then, a conventional denoising filter for Gaussian noise is used. The denoised image is finally inverse transformed.

III. Experimental Results

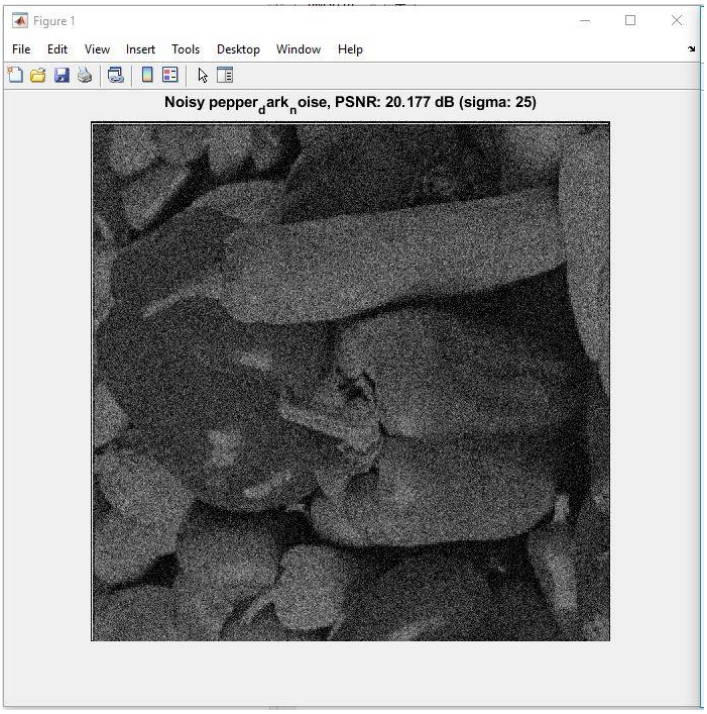


Figure 53: the pepper image with shot noise

Figure 54: the pepper image after Anscombe root transformation

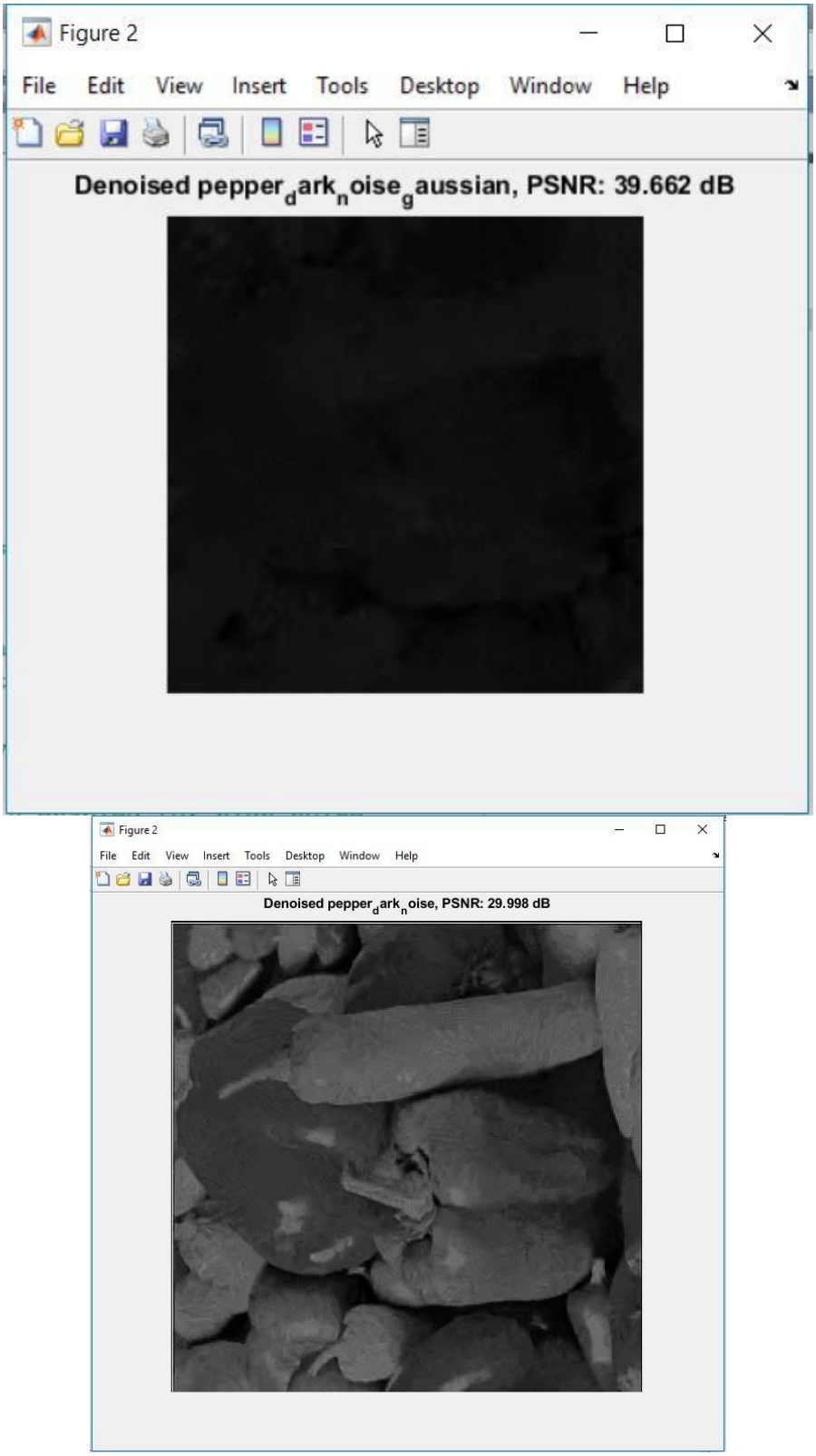
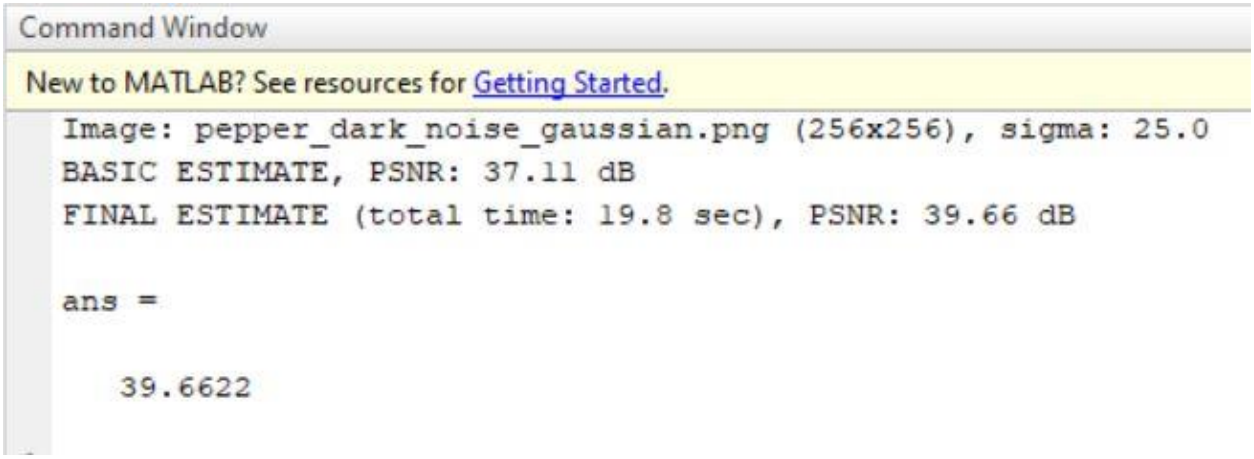
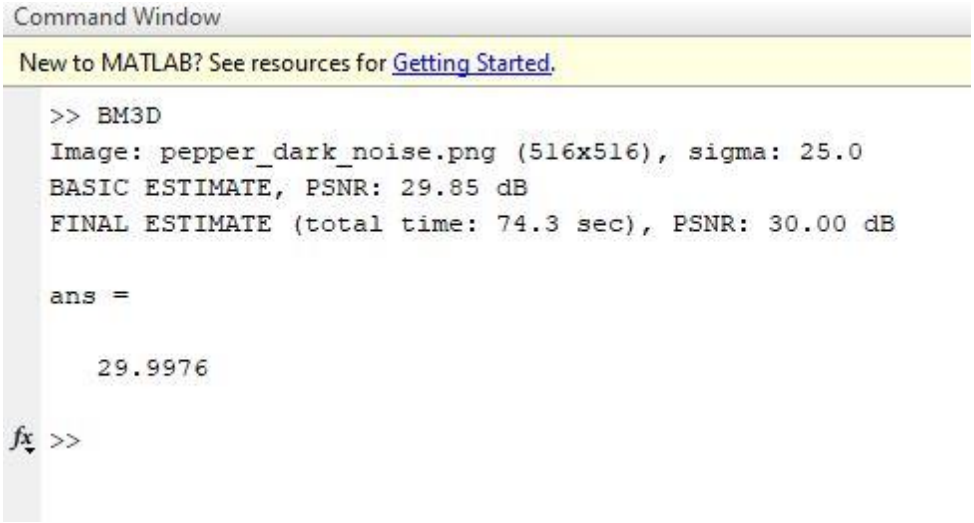


Figure 55: the denoised pepper image after using Anscombe transform, BM3D and inverse transform

Figure 56: the denoised pepper image using Anscombe transform, Gaussian low pass filter and inverse transform

IV. Discussion





To remove the additive Gaussian noise, there are two methods:

- Use Gaussian low pass filter
- Use MATLAB code for block-matching and BM3D transform

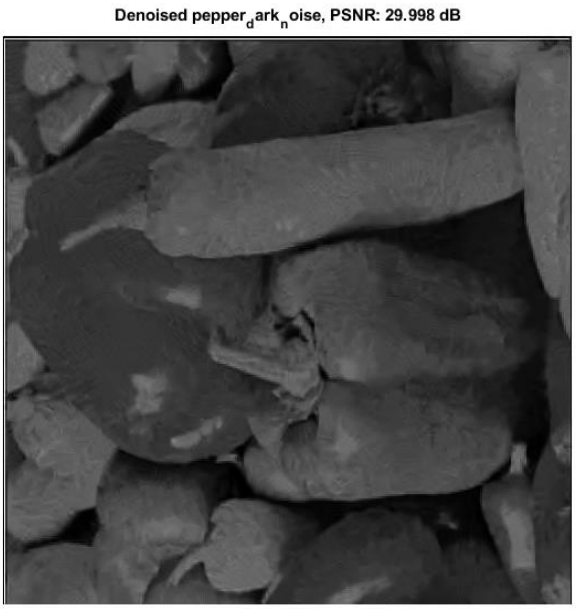
The resulting denoised output images of the pepper_dark_noise.raw input image are obtained using these two methods. The final noise-removed pepper images are shown below:

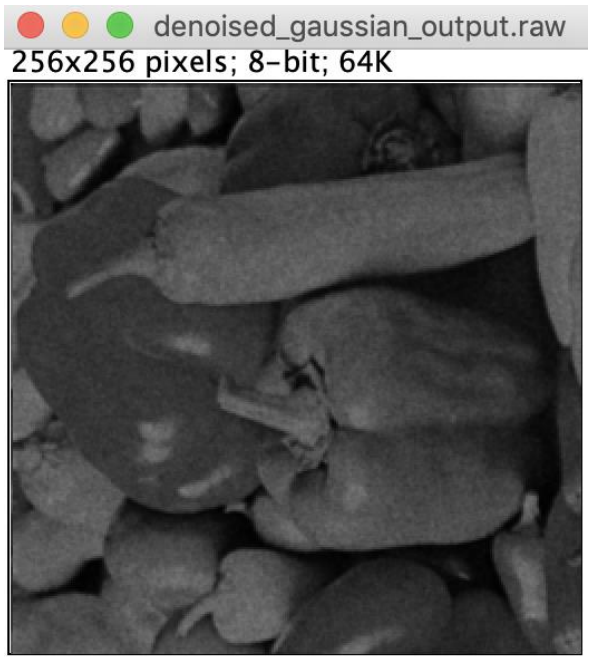
Higher PSNR indicates a better denoised image as more noise is removed. Thus, we can compare the PSNR values for Gaussian and BM3D to find out which filter is better.
By keeping the window size = 3, the PSNR for the Gaussian filter is 28.6762 dB.
By varying the various parameters like beta = 4, N step is 2, N2 = 64, N step Weiner = 2, in the BM3D method, we get PSNR = 39.62 dB.

In the Gaussian filter, the PSNR is much less compared to BM3D.

Gaussian filter is used as a noise removal smoothing filter. It loses the information at sharp edges in the image.
However, in BM3D we can see from the output that the image is denoised and also has sharp edges helping us to distinguish in the bell peppers. It gives a much better visual output as compared to Gaussian low pass filter.

Hence, we can say BM3D is a better method for denoising than Gaussian filter.





Conclusion for denoising:

At the end, I would like to say that I believe performance comparison for all these filters is dependent on a number of parameters like the PSNR value, type of image given as input, visual quality by the human eye. Also, each algorithm varies its performance depending upon the type of image it is dealing with. As sir mentioned in the lecture, “denoising an image is an art” and I truly believe so.