INDEX

1 Problem #1: Image Manipulation and Interpolation 4

1.1 Image Resizing via Bilinear Interpolation 4
1.1.1 Motivation 4
1.1.2 Approach and Implementation 4
1.1.3 Results 6
1.1.4 Discussion 7
1.2 Demosaicing of Bayer-Patterned Color Image 8
1.2.1 Motivation 8
1.2.2 Bilinear Demosaicing 8
1.2.2.1 Approach and Implementation 8
1.2.2.2 Results 10
1.2.2.3 Discussion 10
1.2.3 MHC Demosaicing 11
1.2.3.1 Approach and Implementation 11
1.2.3.2 Results 12
1.2.3.3 Discussion 12

2 Problem #2: Histogram Equalization and Image Filtering 13

2.1 Histogram Equalization 13
2.1.1 Motivation 13
2.1.2 Approach and Implementation 13
2.1.2.1 Transfer Function Based Histogram Equalization 15
2.1.2.2 Cumulative – Probability Based Histogram Equalization 17
2.1.3 Results 19
2.1.4 Discussion 20
2.2 Image Filtering – Creating Oil Painting Effect 21
2.2.1 Motivation 21
2.2.2 Approach and Implementation 21
2.2.3 Results 23
2.2.4 Discussion 24

3 Problem #3: Noise Removal 27

3.1 Mixed Noise in Color Image 27
3.1.1 Motivation 27
3.1.2 Approach and Implementation 27
3.1.2.1 Rank Order Based Adaptive Median Filter 29
3.1.2.2 Mean Filter 30
3.1.2.3 Gaussian Filter 30
3.1.3 Results 31
3.1.4 Discussion 32

3.2 Guided Image Filtering 33
3.2.1 Motivation 33
3.2.2 Approach and Implementation 33
3.2.3 Results 35
3.2.4 Discussion 36
3.3 Block Matching and 3-D Transform Filter 37
3.3.1 Motivation 37
3.3.2 Approach and Implementation 37
3.3.3 Results 38
3.3.4 Discussion 39

4 References 40

LIST OF FIGURES

1 Problem #1: Image Manipulation and Interpolation 5
1.1 Bilinear Interpolation 6
1.2 (a)Original Image of size 512 × 512 6
1.2 (b)Image of size 650 × 650 obtained from bilinear Interpolation 6
1.3 Bayer Array 9
1.4 (a)Original Parrot CFA Image 10
1.4 (b)Bilinear Demosaicing 10
1.4 (c)MHC Demosaicing 12

2 Problem #2: Histogram Equalization and Image Filtering 13
2.1 (a) Histogram of Red in original Jet 13
2.1 (b) Histogram of Green in original Jet 14
2.1 (c) Histogram of Blue in original Jet 14
2.2 (a) Histogram of Red in Transfer function based Equalization 16
2.2 (b) Histogram of Green in Transfer function based Equalization 16
2.2 (c) Histogram of Blue in Transfer function based Equalization 16
2.3 (a) Histogram of Red in Cumulative - Probability based Equalization 17
2.3 (b) Histogram of Green in Cumulative - Probability based Equalization 18
2.3 (c) Histogram of Blue in Cumulative - Probability based Equalization 18
2.4 (a) Transfer Function Equalization 19
2.4 (b) Cumulative – Probability Equalization 19
2.5 (a) Quantized Barn (N=5) 23
2.5 (b) Oil Painting Effect 23
2.6 (a) Quantized Coliseum (N=5) 23
2.6 (b) Oil Painting Effect 23
2.7 (a) Oil Painting Effect (N = 9) 24
2.7 (b) Oil Painting Effect (N = 15) 24
2.8 (a) Oil Painting Effect (N = 9) 25
2.8 (b) Oil Painting Effect (N = 15) 25
2.9 (a) Oil Painting Effect with N = 5 and total 512 colors 26
2.9 (b) Oil Painting Effect with N = 5 and total 512 colors 26

3 Problem #3: Noise Removal 27
3.1 (a) Histogram of Red with both salt & pepper and Gaussian noise 27
3.1 (b) Histogram of Green with salt & pepper noise 28
3.1 (c) Histogram of Blue with both salt & pepper and Gaussian noise 28
3.2 Filtered Image (N=5) 31
3.3 Guided Filter Concept 33
3.4 (a) $\varepsilon = 2500$, $r = 2$,N=5 35
3.4 (b) $\varepsilon = 350000$, $r = 2$,N=5 35
3.4 (c) $\varepsilon = 100000$, $r = 2$,N=3 35
3.5 Cartoon Matting after applying Guided Filter 15 times 36
3.6 Image for BM3D Filtering with Standard Deviation of 60 38

Amruta Kulkarni,
USC ID: 6914970827
Email: arkulkar@usc.edu
Date: 20 Sept 2015

EE 569 HOMEWORK #1

---

1. Problem #1: Image Manipulation and Interpolation

1.1 Image Resizing via Bilinear Interpolation

1.1.1 Motivation:

Image Resizing is a very commonly performed operation. We come across situations where we need to increase or decrease the size of an Image as per our requirements. One way of doing that is by using Bilinear Interpolation. In this problem, we increase the size of an Image from 512 × 512 to 650 × 650 by using the same.

1.1.2 Approach and Implementation:

- Initially we have an RGB image of size 512 × 512 and now we want to make it larger so that it has a size of 650 × 650.
- For every location in the new enlarged image, we need to obtain a corresponding scaled location in the original image.
- This can be done by using a scaling factor, which when multiplied to the pixel locations in enlarged image (B) will give us scaled pixel locations in the original image (A).
- Now, for a certain location in B we get a scaled location that lies somewhere in A.
- From this location in A, we can get the intensity value at it by using Bilinear Interpolation i.e. from its nearest 4 pixels.
- The basic idea is to map all pixels of image B to the corresponding scaled locations in pixel A and then acquire intensities at those locations.

- The following figure depicts the above concept:

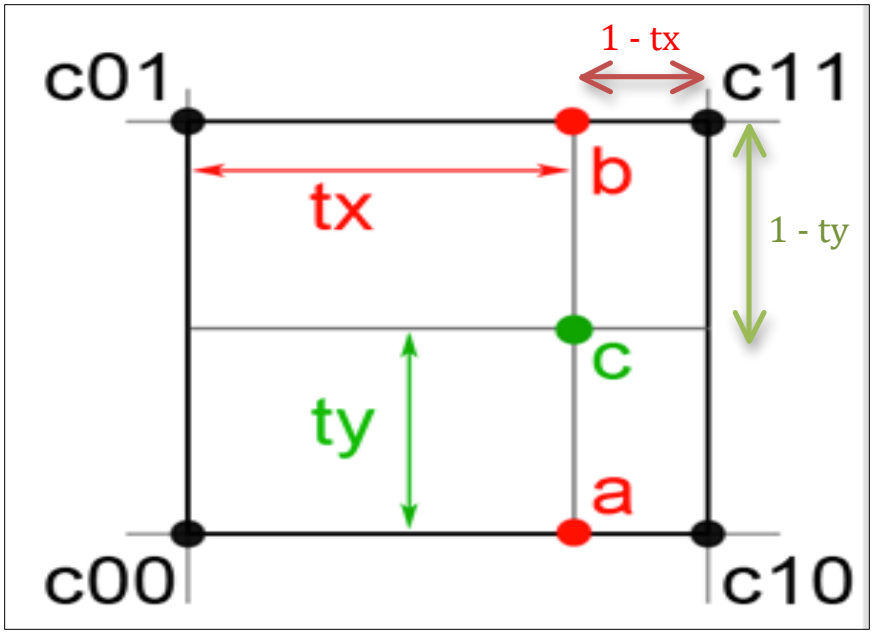

Figure 1.1 Bilinear Interpolation [1]

- Now, we want to obtain the intensity value at 'C' by using the intensity values at its nearest 4 neighbors 'C00', 'C10', ' C11' and 'C01'.
- This is obtained by using the following formula:

$$C = (1-tx)(1-ty)C00 + (1-tx)(ty)C01 + (tx)(1-ty)C10 + (tx)(ty)C11$$

- This newly obtained intensity value is then assigned to the actual location (i.e. the one before scaling) in image B.
- On similar lines, all the pixels are assigned intensity values obtained from the nearest four pixels based on their scaled locations in the original Image A.
- This results in the Image B which has an enlarged size and in our problem it is 650 × 650.
- This approach has been implemented using C++.

1.1.3 Results:

The output image (650 × 650) obtained after implementing Bilinear Interpolation on the image The Starry Night is as shown below:

Figure 2.2(a) Original Image of size 512 × 512

Figure 3.2(b) Image of size 650 × 650 obtained from bilinear Interpolation

1.1.4 Discussion:

The concept of Interpolation is mainly used when we want to obtain new values at locations that are continuous or in other words, non-discrete. Usually, in images, pixel locations are integers i.e. discrete values. But as we saw in the implementation part, we might want to determine intensity values at scaled non-integer locations. In such cases, Bilinear Interpolation comes in handy and it enables us to obtain intensities at any non-integer location in our original image and then this new intensity belongs to some integer locations in the resized image.

1.2 Demosaicing of Bayer-patterned color Image

1.2.1 Motivation:

In cameras like DSLR, the image is captured by an array of sensors. These sensors are ordered in a pattern known as Bayer Pattern where the sensor arrays alternate between Red, Green, Red, Green… and Green, Blue, Green, Blue… So, 50% of the sensors capture green, 25% capture blue and the remaining 25% capture red. Thus, every pixel has only one color amongst red, blue or green. But to create an RGB image, we need three colors to be present at every pixel location. To get the information of the colors that are absent in the pixel, we need to consider neighboring pixels that have those colors present in them. Hence, we are borrowing color information from the neighboring pixels so as to fill in all 3 colors (red, green, & blue) in every pixel. This is known as demosaicing. In this problem, we have considered two types, namely Bilinear Demosaicing and Malvar-He-Cutler(MHC) Demosaicing.

1.2.2 Bilinear Demosaicing:

1.2.2.1 Approach and Implementation:

- We first need to create a data matrix that can store all three colors at every pixel.
- If we want to fill in colors at the pixels located along the corners or the edges, then we need 8 neighboring pixels from which color information can be extracted.
- Therefore, we can simply extend the original image by copying rows and columns in such a way that the alternating array sequence is not disturbed.
- Once we have this extended matrix, we can perform demosaicing.
- If we look at figure 1.3, we can see that the top-left blue pixel can get red color from the 4 pixels along its diagonals, whereas it can get the green color from the 4 pixels along its edges.
- Similarly, for a red colored pixel, blue color can be extracted form the 4 pixels along its diagonals and green color from the pixels along its edges.

- On the other hand to get red and blue colors at a green pixel, we can extract colors only from two neighboring pixels along its edges.

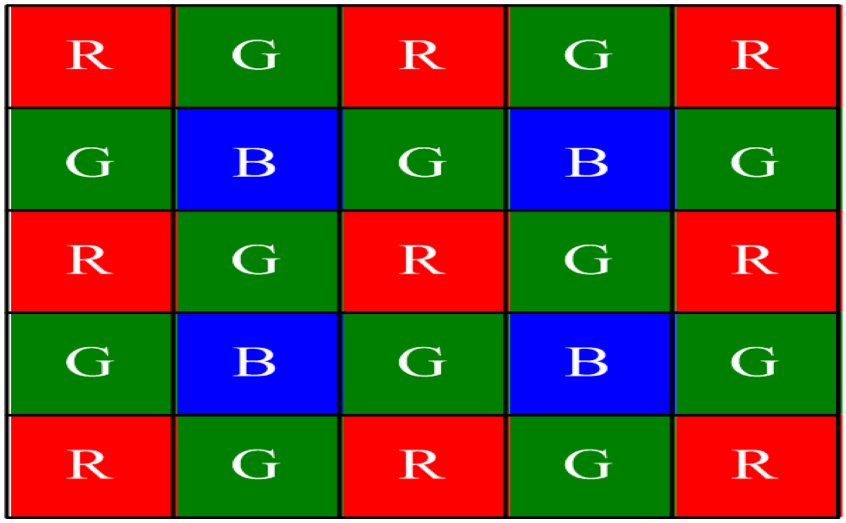

Figure 4.3 Bayer Array

- Thus, every pixel will get the colors that it needs by taking the average of those colors from the neighboring pixels.
- In this manner, we add the remaining two colors to every pixel and this results in an RGB image in which every pixel has three colors in it i.e. we get a 24 bits image from an 8 bits image.
- This approach has been implemented using C++.

1.2.2.2 Results:

The images before and after using Bilinear Interpolation are as shown below:

Figure 5.4(a) Original CFA Parrot Image **Figure 1.4(b) Bilinear Demosaicing**

Thus we can see that a gray scale image, which had only one color information in every pixel, has been converted into a color image where every pixel now has three colors.

1.2.2.3 Discussion:

Bilinear demosaicing is a basic approach whose main strength is that it operates at a fast speed. Moreover, the computational complexity is quite low. But this method has some shortcomings. In the above image, we can see some moiré effect along the tree bark below the parrot. This effect is mainly because there are a set of lines which differ in angles or spacing within themselves. We also see some aliasing effect in the Image that leads to blurring of some parts.

1.2.3 MHC Demosaicing:

1.2.3.1 Approach and Implementation:

- In Bilinear Demosaicing, we were considering a 3 × 3 window so as to cover the 8 neighboring pixels along the edges and diagonals.
- This concept has been further extended in the Malvar-He-Cutler Linear Image Demosaicing where we now use a 5 × 5 window.
- Thus, we again need to do image extension so that when the window is placed along pixels near the corners or edges, then the entire window must contain pixels.
- In this method, different filters have been used for different pixels located in different rows.
- In Fig. 1.3, if we want to calculate the value of green color at the red pixel located at the center, then we take a 5 × 5 window and consider all the green pixels as well as all the red pixels that lie in this window.
- Thus, to get an absent color at a certain pixel, we consider all the pixels that have the color that we need, as well as pixels that have the color that we have.
- We convolve every window of pixels with the appropriate filter to get all the colors at a pixel located at the center of the window.
- The filters have weights at every location according to the color that has to be restored.
- This approach has been implemented using C++.

1.2.3.2 Results:

The 24 bit color image that was obtained using MHC algorithm is as shown below:

Figure 6.4(c) MHC Demosaicing

1.2.3.3 Discussion:

If we compare the background of MHC and Bilinear Images, we can conclude that MHC performs better. The sharpness of the lines along the bark is more in MHC than in bilinear. Also, the background green color looks less blurry than that in Bilinear. Ideally, MHC should be more efficient than bilinear as it uses a larger window for obtaining absent colors in the pixels. But the algorithm that has been implemented has not given expected results perhaps due to some flaw in the source code.

2. Problem #2: Histogram Equalization and Image Filtering

2.1 Histogram Equalization

2.1.1 Motivation:

Histogram Equalization is a technique that is widely used for image enhancement. Suppose we have a range of intensities from 20 to 150 but we want this range to cover intensities from 0 to 255. Then, to achieve this, we need to use histogram equalization which will adjust the intensity range so that the intensities are now well distributed across the image. By doing this, we are increasing the contrast of the image at hand. In this problem, we are considering two types of Histogram Equlization techniques which are Transfer Function based technique and Cumulative-probability based technique.

2.1.2 Approach and Implementation:

- Basically, we want all 256 intensity levels to be present in our Image so that the image has a good contrast.
- Now let us look at the histograms of the three channels shown in Fig. 2.1(a),(b) and (c):

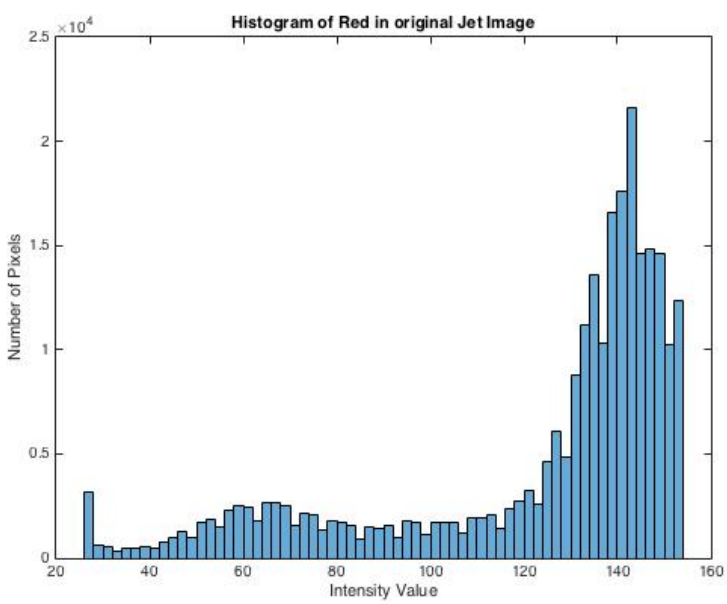

Figure 2.1(a)

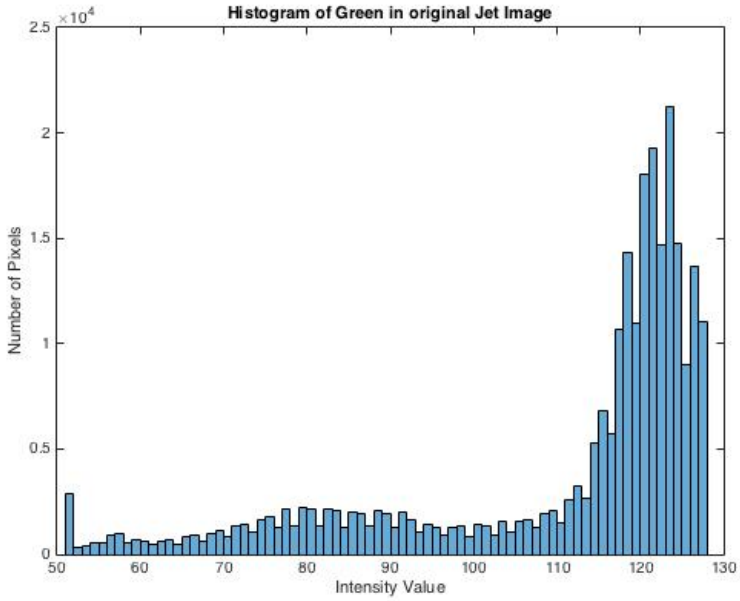

Figure 2.1(b)

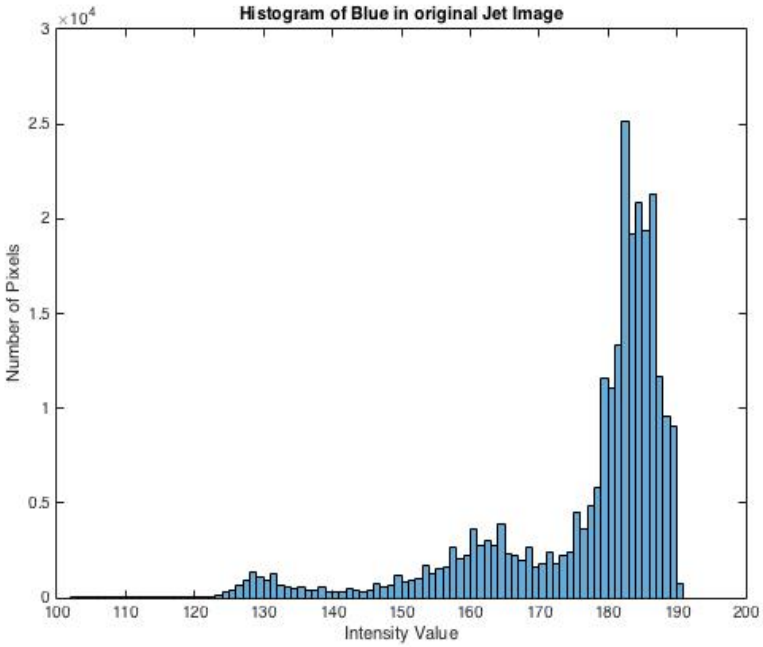

Figure 2.1(c)

- Thus, we can see that the range of intensity values in the three channels is different and none of them have 256 values.
- Thus we need to perform Histogram Equalization on each channel separately, so as to get 256 intensity values and thereby improve the contrast.

2.1.2.1 Transfer Function Based Histogram Equalization

- In this method, we first obtain the frequency of occurrence of every intensity value that is present in a specific channel.
- Once we get all the frequencies, we need to divide every frequency by the total number of pixels in the channel and this will give the probability of each intensity value in that channel.
- After obtaining the probabilities of all the intensity levels, we perform a cumulative addition of all probabilities to get the cumulative distribution value of every intensity level.
- Thus, so far we calculated cumulative probabilities for intensity levels from 0 to 255 giving rise to 256 levels, irrespective of the initial number of levels that were present in each channel individually.
- Now, each of the initial intensity levels are replaced with the product of their cumulative probability and the maximum intensity i.e. 255
- This results in an image with 256 intensity values ranging from 0 to 255.
- After doing the same procedure for all the channels, we obtain an enhanced image with high contrast and clarity.
- The histograms of the three equalized channels are as shown in Fig. 2.2(a), (b) and (c):

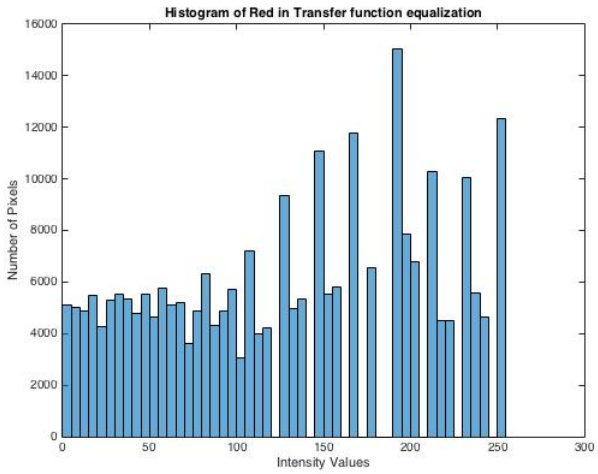

Figure 2.2(a)

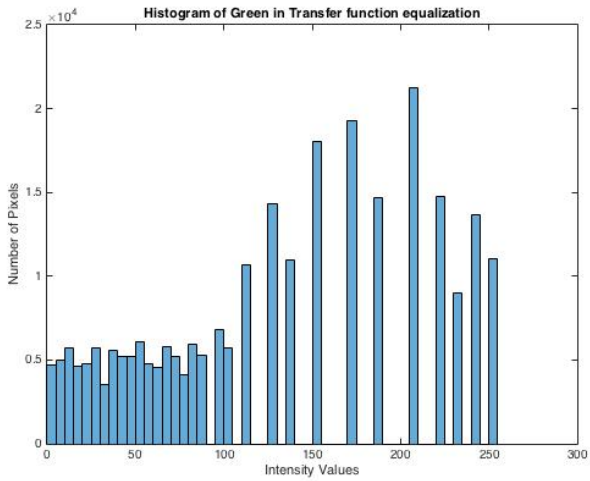

Figure 2.2(b)

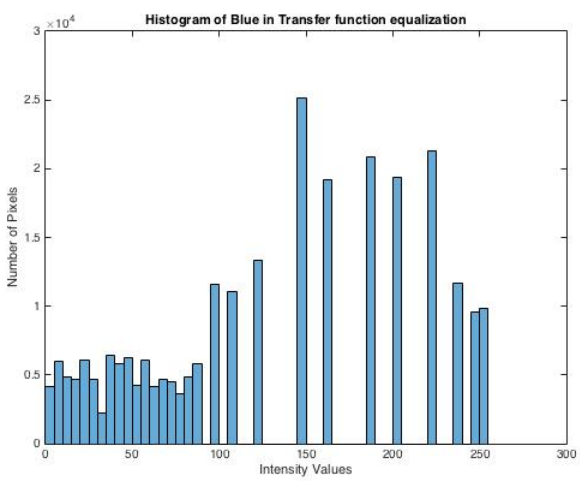

Figure 2.2(c)

2.1.2.2 Cumulative – Probability based Histogram Equalization

- This method is also known as the bucket filling method.
- The main idea is to have 256 intensity levels in such a way that each level will occur in equal number of pixels.
- Now, in our problem we have the jet image which has 512 × 512 pixels and the intensity levels are 256.
- The number of pixels that each level will have are obtained by dividing the total number of pixels by the number of intensity levels.
- Thus, we need 1024 pixels in each of the 256 buckets where every bucket belongs to one of the intensity levels in the range of 0 to 255.
- Now, we need to arrange all the pixels in our original image in ascending order and then put pixels in buckets accordingly.
- Once we have all the pixels in order, all we do is assign 0 to the first 1024 pixels, assign 1 to the next 1024 pixels and so on till we assign 255 to the last bucket of 1024 pixels.
- We have now created an Image with much better contrast than earlier, the reason being that all intensity levels are uniformly distributed throughout the image.
- The histograms obtained for the three channels are as shown below:

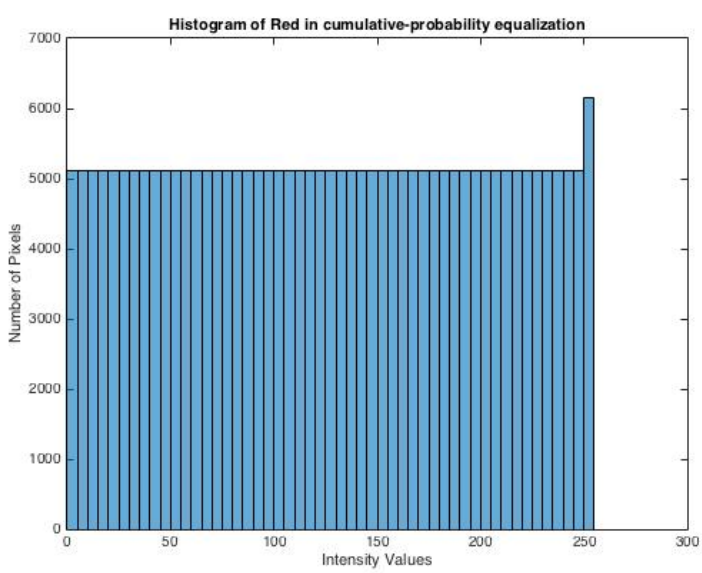

Figure 2.3(a)

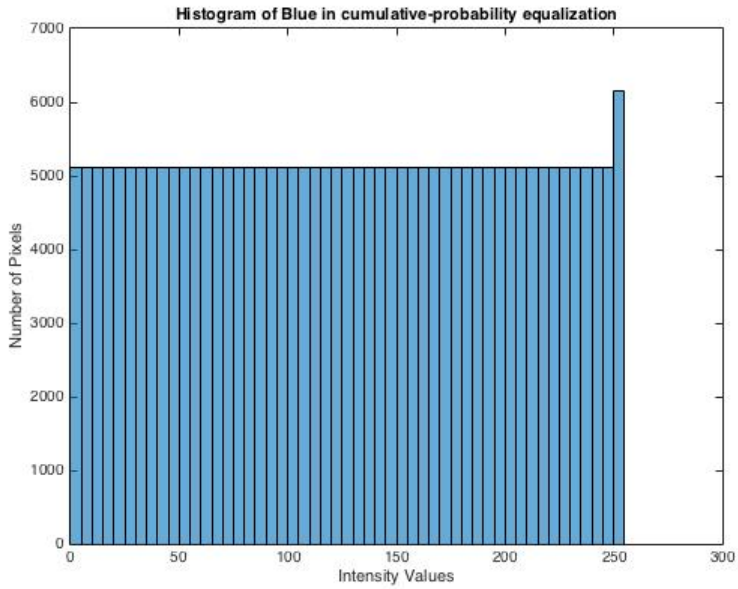

Figure 2.3(b)

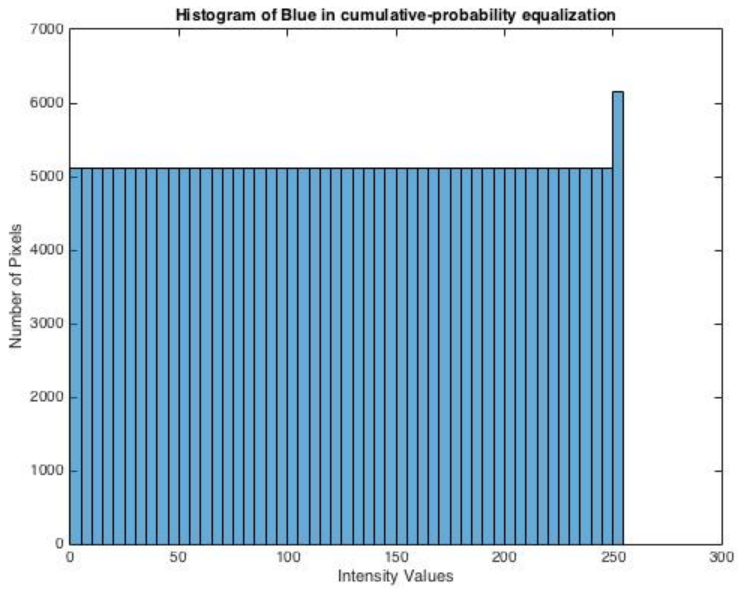

Figure 2.3(c)

2.1.3 Results:

Both the above approaches have been implemented in C++. The images that were obtained by using the above two techniques are as follows:

Figure 2.4(a) Transfer Function Equalization

Figure 2.4(b) Cumulative-Probability Equalization

2.1.4 Discussion:

As we can see in the above images, the contrast of both the images is almost the same. But if we look at the respective histograms, then we can conclude that the intensity distribution across both the images is very different. The enhancement results of both the images look very similar to our eyes. These techniques work well only when pixels are similarly distributed across the image. If there are lighter or darker regions then these techniques might not work very well. One way of improving these results is by adaptive histogram equalization. In this technique, multiple histograms are calculated for every part of the image and then the intensity values are redistributed.

2.2 Image Filtering – Creating Oil Painting Effect

2.2.1 Motivation:

In Images, there are 256 intensity levels associated with every primary color i.e. Red, Green and Blue. This gives us the total number of colors to be 256 × 256 × 256. But if we consider oil paintings, the number of colors is not as large as that of Images. We can reduce the color set of Images in such a way that they contain a fewer number of colors like an oil painting does. This is a great way of converting our Images into realistic oil paintings and this effect can make images to look like pieces of art.

2.2.2 Approach and Implementation

- As we already know, the number of colors in an image is much more than that in an oil painting.
- In a color Image, we have 3 primary colors and each color has 256 different intensities.
- In this problem, we are required to get a color set of 64 colors.
- This implies that every primary color can have only 4 intensity levels and this will give us a total of 64 colors.
- Thus we need to quantize the 256 levels of each color to 4 levels.
- So, we first calculate histograms for Red, Green and Blue.
- Now, we have to rescale the range of intensities of 0 to 255 to another range of 0 to 3.
- The total number of pixels is the product of height and width of the image and now we want 4 intensity levels for all the pixels in every channel.
- Number of pixels in each intensity level = Total pixels / 4 approximately.
- In the Barn image, have 104500 pixels and we need to have roughly 26125 pixels of each of the 4 intensity values.
- From the histogram, we can add the number of pixels until the limit for each intensity level is reached, and then we can set the threshold

values corresponding to the three intensity levels separating the 4 bins.

- Therefore everytime the number of pixels become equal to or greater than 26125, we need to set the intensity (at which the limit has reached) as the threshold.
- Then, start counting the number of pixels from that intensity onward, till the limit is again reached.
- After we do this, we will get 3 threshold intensities.
- These intensities are somewhere between 0 and 255.
- So we now have four bins with different intensity ranges in them.
- Now, we calculate the average of all the intensities in each bin and this average value will be assigned to all the intensities in that bin.
- We assign the average value of first bin to all the intensities that lie within that bin, and similarly for the next three bins.
- In this way, we have a quantized matrix for every channel, and the levels in this matrix are the four averages of each bin respectively.
- The image that gets generated after assigning four levels to all pixels in every channel, is the quantized image.
- Now, once we have the quantized data for all the three channels, we need to create the painting effect.
- We first create a neighborhood window of dimensions N × N. This is the oil-painting filter.
- In our quantized image, every pixel must have an N × N neighborhood. For this, we first perform extension of the rows and columns near the edges.
- Now, for every pixel, we take its neighborhood window, and then we assign the intensity value that has the maximum frequency in that window, to the current pixel.
- After doing this for every pixel, all the new pixel values are such that they create an effect like that of an oil-painting where there are pixel blocks with the same color.
- This approach has been implemented using C++.

2.2.3 Results:

- The quantized and oil-painting images for Barn and Coliseum are as shown below:

Figure 2.5(a) Quantized Barn(N = 5)

Figure 2.5(b) Oil-Painting Effect

Figure 2.6(a) Quantized Coliseum (N = 5)

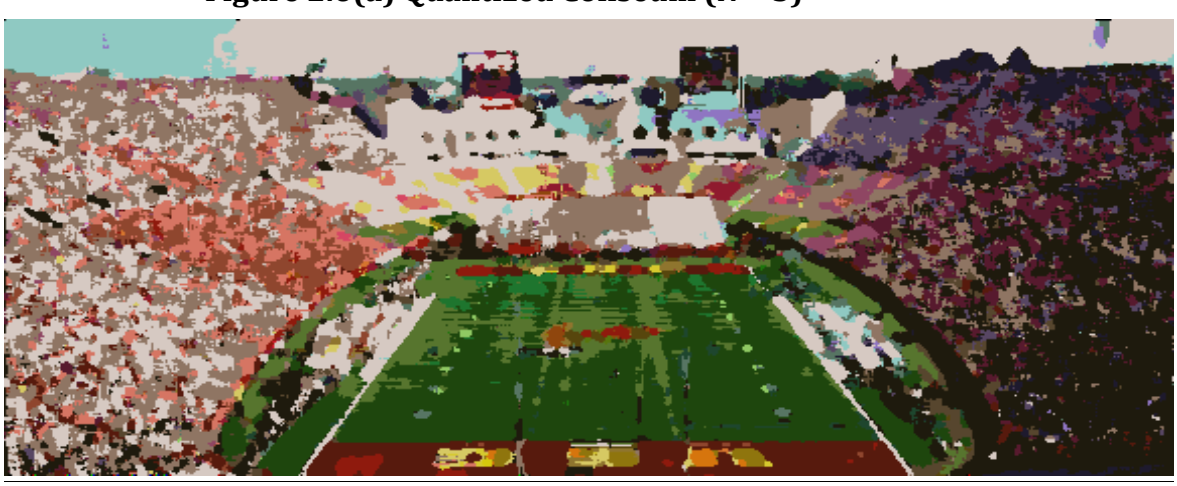

Figure 2.6(b) Oil-Painting Effect

2.2.4 Discussion:

- The oil-painting filter has dimensions of N × N and changing the value of N results in different effects on the image.
- Now lets consider N as 3. So when a window of 3 × 3 is chosen around a pixel, all the pixels in that window might have different values and thus we wont be able to get a value that has the maximum frequency and this might result in the current pixel value being unchanged.
- In such cases, our image will look more like the quantized image itself and less like an oil painting.
- The window size cannot be too large as it will make all the pixels look similar and we might loose the details of the image.
- Thus, we need to choose an appropriate N such that the oil painting filter successfully changes pixel values to give an oil-painting effect to the image.
- The oil-painting effect for barn and coliseum images for different values of N is shown below:

Figure 2.7(a) Oil-Painting Effect(N=9)

Figure 2.7(b) Oil-Painting Effect(N=15)

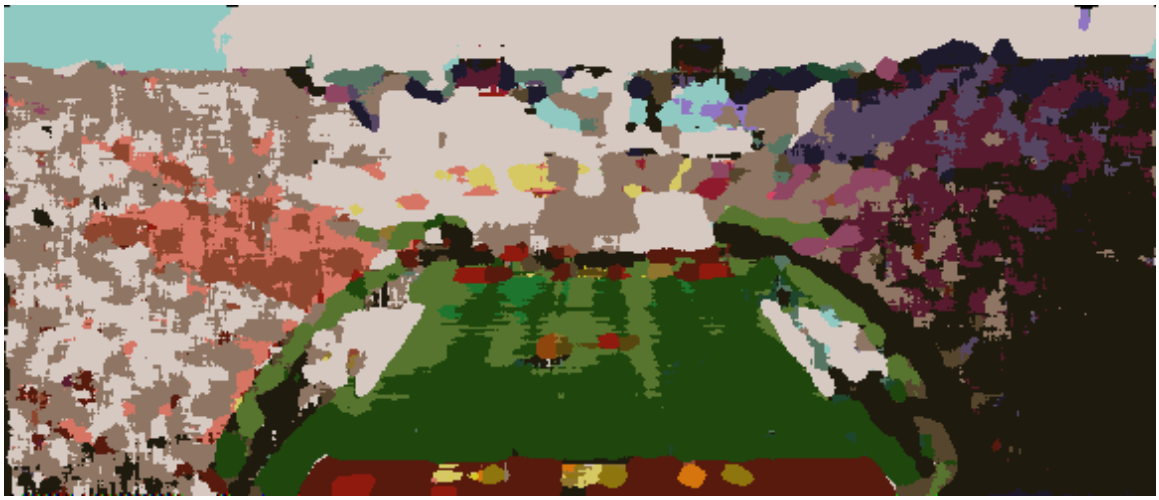

Figure 2.8(a) Oil-Painting Effect(N=9)

Figure 2.8(b) Oil-Painting Effect(N=15)

- Thus we can see that N = 5 (Fig.2.5(b)) gives the best result as the filter is most effective at this value.
- For large values of N, the image details get reduced and just colored block-like shapes are visible.
- Until now, we were considering total 64 colors but if we want 512 colors in an image, then we need to assign 8 intensity levels to every image.
- Thus, we will need to separate intensities into 8 different bins such that the total number of pixels in each bin is roughly the same.
- We then need to calculate the average of these 8 bins and then assign them to the intensities that lie in those respective bins.

- The oil-paintings of Barn and Coliseum with 512 colors are as shown below:

Figure 2.9(a) Oil-Painting Effect (N=5 and total 512 colors)

Figure 2.9(b) Oil-Painting Effect (N=5 and total 512 colors)

- We can see that these images have more number of colors and hence more details are retained.

3. Problem #3: Noise Removal

3.1 Mixed noise in color Image

3.1.1 Motivation:

Images captured by using a camera have a lot of noise that gets picked up from the surroundings and many such factors. Noise deteriorates the quality of images and it might also result in the loss of minute details from an Image. Thus, Noise reduction is a very important concept that can help to improve the quality of images. In our problem, we will be dealing with two types of noise namely salt & pepper and Gaussian noise.

3.1.2 Approach and Implementation:

- Noise affects every channel in a different way.
- Red channel has a different type and amount of noise than the blue or green channel.
- To get rid of the noise from an image, we first need to observe the histograms of every channel.
- The histograms of noisy peppers image for all the three channels are as shown below:

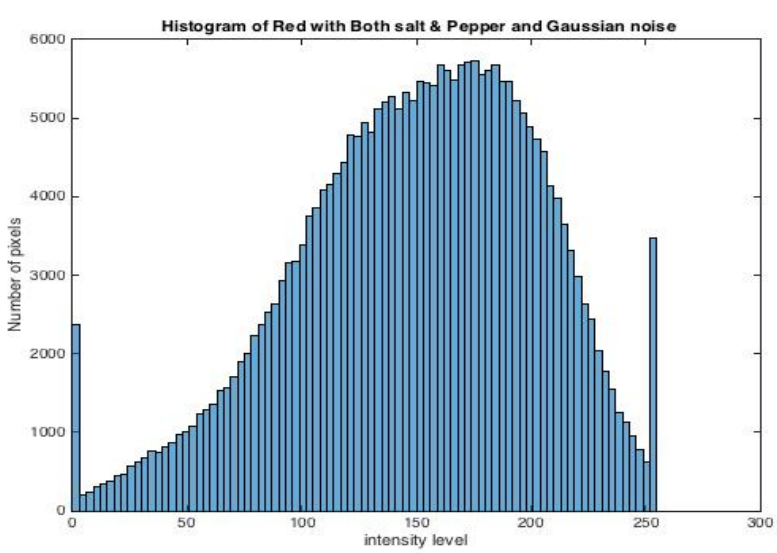

Figure 3.1(a)

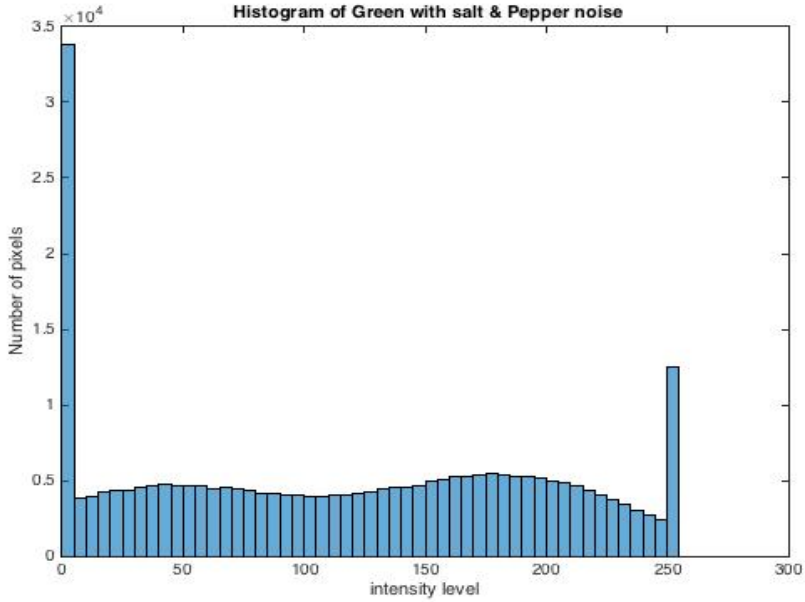

Figure 3.1(b)

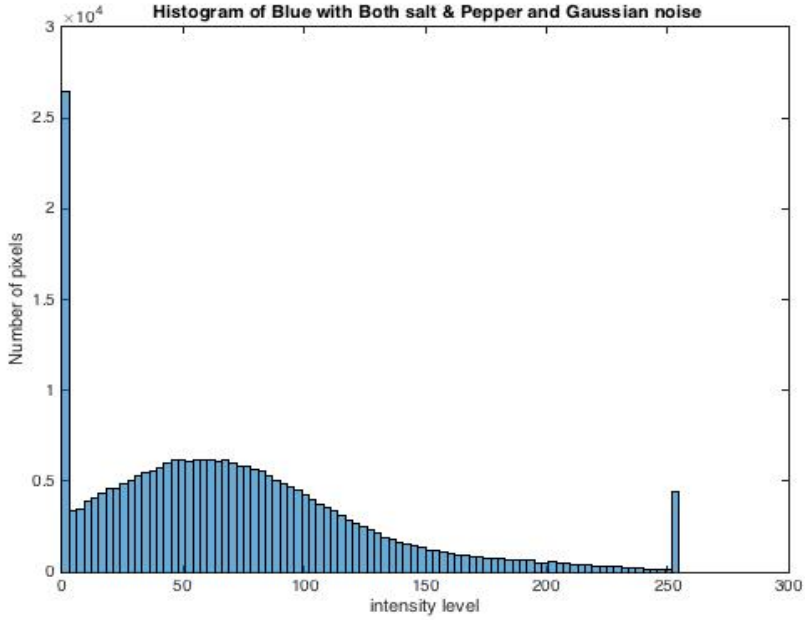

Figure 3.1(c)

- From the above figures, we can see the spikes that are present at the first and last intensity levels in all the three channels.
- These spikes are due to salt and pepper noise and thus all three channels have salt & pepper noise.
- Now in the histogram of red color, we can see a Gaussian shape, and this is because of the Gaussian noise in that channel.
- There is no Gaussian noise in green channel whereas for blue there is some amount of Gaussian noise as we can see a shifted and scaled bell-shaped Gaussian curve.
- Thus, we can conclude that every channel has a different type of noise.
- Thus, we need to perform filtering on individual channels according to the noise they have.
- Ideally, median filters are used to get rid of the salt & pepper noise whereas mean filters or Gaussian filters are used to get rid of the Gaussian noise.
- We will first apply median filter to all the three channels and then apply mean filter.
- Mean filters are used to remove the Gaussian noise because mean of this type of noise is 0.
- Thus if we take the mean of Gaussian noise, it will result in a zero value.
- We first need to remove the salt & pepper noise before applying mean filter, otherwise mean of the noise will not be zero as there is mixed noise.
- Thus the cascading order is median filter first and then mean filter.
- Now, the steps performed in median filtering and mean filtering are as follows:

3.1.2.1 Rank Order Based Adaptive Median Filter [2]

Step 1: Create an N × N sliding window

Step 2: Generate an extended matrix for every channel so that every pixel of the original matrix has pixels in the window

Step 3: Each window when placed on a pixel, will have N × N pixels in it.

Step 4: Arrange the pixels in the window in an ascending order

Step 5: Calculate the minimum, maximum and median intensity values of this window

Step 6: Now, if the median as well as the central pixel lies strictly between the minimum and maximum values, then that implies that the pixel does not have noise. Thus it will remain unchanged.

Step 7: If only the median value lies strictly between the minimum and maximum whereas the central pixel value does not satisfy the criteria of strictly between minimum and maximum, replace the pixel value by the median intensity value.

Step 8: Repeat steps 4 to 7 till the window slides through the entire matrix

We have now eliminated the salt & pepper noise from the image.

3.1.2.2 Mean Filter

Step 1: Create an N × N window
Step 2: Slide this window through the matrix
Step 3: Calculate the mean of all the pixels lying in the window
Step 4: Replace the central pixel by the mean value
Step 5: Repeat steps 3 and 4 for all the pixels

3.1.2.3 Gaussian Filter

Step 1: Create a Gaussian kernel window where every location has a certain weight depending upon the distance from the center of the kernel

Step 2: Convolve this window with the neighborhood matrix of each pixel to obtain an intensity value without Gaussian noise

Step 3: Replace the central pixel in the neighborhood matrix with the new intensity value

3.1.3 Results:

This approach has been implemented using C++. The image obtained after applying median and mean filters to all the channels separately is as shown below:

Figure 3.2 Filtered Image with N = 5

3.1.4 Discussion:

The window size can be changed. Different PSNR values for different window sizes are as shown below:

Color	N = 5	N = 7	N = 11
Red	PSNR = 27.2295	PSNR = 26.3972	PSNR = 24.8282
Green	PSNR = 18.0972	PSNR = 17.3388	PSNR = 16.7782
Blue	PSNR = 20.5933	PSNR = 19.8219	PSNR = 19.2922

Table No. 3.1 PSNR values for different window sizes

Thus, we can say that increasing the value of N reduces the PSNR. The mean and median filters that have been used are very basic filters for noise removal. As we can see from the above images, they cannot give the best results. We can see the loss of information along edges. Thus, the main disadvantage of these filters is that along with noise, they also remove some useful data due to which the edges present in an image become blurred. To overcome this shortcoming, we can use different filters like guided filters or non-Local means filter as well as some transforms like wavelet transforms. These more advanced filters efficiently remove noise and also maintain the required edge sharpness.

3.2 Guided Image Filtering

3.2.1 Motivation:

As we saw in the earlier section, noise removal using the basic filters led to decrease in the level of edge sharpness. Thus we need to use some more advanced filters that preserve required image data and remove the redundant data i.e. the noise from an image. Guided Filter is one such filter that performs better than many low-pass filters. It is a non-linear filter which does not lead to edge degradation.

3.2.2 Approach and Implementation:

- Guided Image filter performs edge-preserving smoothing on images that contain noise.
- The basic idea is to filter the image with the help of another image known as the guidance image, which can be the image itself or a different version of that image.
- In our case, the guidance Image is the same as the actual Image that is the noisy peppers image.

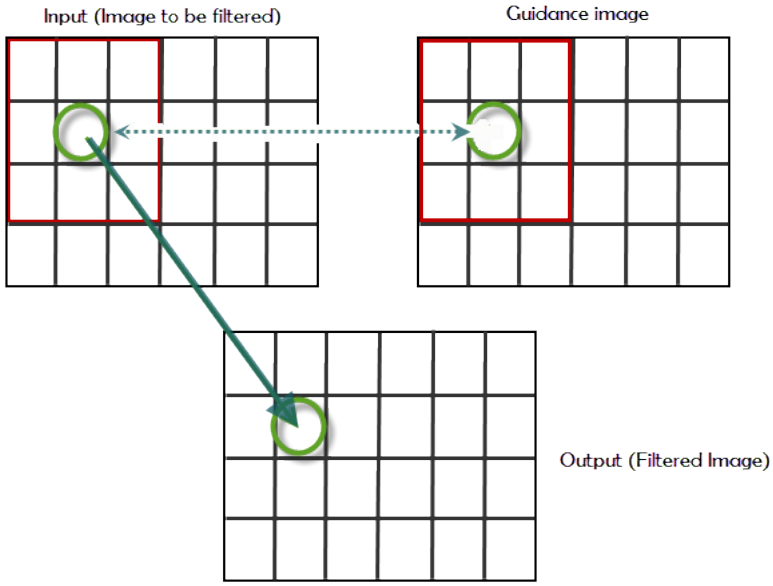

Figure 3.3 Guided Filter Concept [3]

- Now, we first need to create a window with a certain radius and this window will take in different values from the image matrix
- We have to calculate the mean and variance of all the windows of the input image and the guidance image.
- The variance and covariance of a pixel between these two images helps us in determining the value of that pixel in the output image.

$$a_k = \frac{\frac{1}{|\omega|}\sum_{i\in\omega_k} I_i p_i - \mu_k \bar{p}_k}{\sigma_k^2 + \epsilon}$$

$$b_k = \bar{p}_k - a_k \mu_k.$$

$$q_i = \frac{1}{|\omega|}\sum_{k:i\in\omega_k} (a_k I_i + b_k)$$

$$= \bar{a}_i I_i + \bar{b}_i$$

- Using the above set of formulae we can obtain the value of the pixel in the output image.
- The complexity of implementation of this algorithm is that we need to store the 'a' and 'b' values of all the windows and then again retrieve those values when calculating the pixel value which is shared by those corresponding windows
- But the overall efficiency of this algorithm is good and hence it is very beneficial.
- This approach has been implemented using C++.

3.2.3 Results:

The following images have been obtained using guided filtering approach and they have different values of 'ε' and radius 'r'.

Figure 3.4(a) ε = 2500, r = 2,N=5

Figure 3.4(b) ε = 350000, r = 2,N=5

Figure 3.4(c) ε = 100000, r = 1,N=3

3.2.4 Discussion:

- As we can observe from the above images, noise does not get completely eliminated for small values of 'ε'. Also, increasing 'r' to a suitable value results in better noise removal.
- The best results are obtained when ε is around 100000.
- In conclusion it can be said that guided filter works better than linear filters because it efficiently preserves edge sharpness. Moreover it has a simple approach that gives exact results.
- Also, it does not just depend upon the image content of the neighboring pixels, but also upon the content of the guidance image which influences filtering
- The claim of getting a cartoon matting effect by applying a guided filter multiple times can be verified from the images given below:

Figure 3.5 Cartoon Matting Effect after applying Guided Filter 15 times

3.3 Block Matching and 3-D Transform Filter

3.3.1 Motivation:

There are different noise removal filters that are quite efficient in removing the noise. But some of them have drawbacks like halos near edges or inability to be applied to sparse inputs. Thus we can consider a state-of-the-art de-noising algorithm that gives the best possible results.

3.3.2 Approach and Implementation:

- In this algorithm, the image enhancement is performed by using 2D image fragments that are similar which are further arranged in 3D groups.
- First, the noisy image is taken as the input image.
- The adaptive-shape neighborhood is then generated with the use of 8 directional LPA-ICI.
- Now, blocks that are similar are obtained by using block-matching technique.
- Adaptive shape neighborhood is then obtained form each of these blocks by using the shape that was obtained earlier.
- Depending upon the value of
  (number of matched blocks / number of pixels in neighborhood)
  the transform to be applied is chosen.
- If the value is large enough then we conclude that there are enough similar neighborhoods and then we obtain the PCA basis.
- If the value is small, we go for non data-adaptive approach.
- Now a 3D group is formed by stacking the neighborhoods which were obtained earlier.
- The PCA transform is then applied to this group.
- This is followed by inversion of the 3D transform to get estimated values of the grouped neighborhoods and these are then sent back to their original locations.

3.3.3 Results:

The image that was obtained after using the BM3D filter on the noisy peppers image is as shown below:

Figure 3.6 Image from BM3D Filter at Standard Deviation of 60

3.3.4 Discussion:

- The above implementation has been done in MATLAB by using the code provided in [4].
- There are tunable parameters in BM3D. Standard deviation can be changed according to he level of noise in the noisy image.
- For lower values of standard deviation, noise removal is not efficient. Thus larger values result in a sharp and noise-free image
- Also, noise free image can also be given for reference
- Profiles can be normal or fast as per our requirements.
- BM3D is both spatial domain as well as frequency domain filter.
- This is because it carries out PCA transform based on where the pixels are spatially located.
- In the part A of this problem, basic de-noising filters like median and mean filters were used to remove noise but they resulted into image degradation and moderate results.
- In part B, we used guided filter to preserve the edge sharpness and confirmed that it works better than the linear filters. But this filter also has some drawbacks.
- In part C, we implemented the BM3D filter which gave the best de-noising results by completely eliminating the noise from the image and also performing smoothing as well as edge preservation.

4. References

[1] Scratchpixel
[2] Study and Analysis of Impulse Noise reduction filters by Geoffrine Judith.M.C et al.
[3] Mathworks Documentation on Guided Filtering
[4] BM3D Image Denoising with Shape-Adaptive Principal Component Analysis by Kostadin Dabov et al.