EEE 569 Digital Image Processing
Homework #1

Abinaya Manimaran
manimara@usc.edu

Spring 2018

Submitted on 02/04/2018

Problem 1
Basic Image Manipulation

a) COLOR SPACE TRANSFORMATION

1. COLOR-TO-GRAYSCALE CONVERSION

➢ ABSTRACT AND MOTIVATION:

Images are represented using pixels. The intensity of each pixel is considered to be variable. Each color pixel in an image is represented by R, G and B intensity components/channels. Each R, G and B channel is represented by 1 byte (8 bits). Hence, a color image has 3 bytes per pixel (24 bits). An image is considered to be in grayscale if each pixel is represented by 1 byte. There are various ways to convert a color image to grayscale (i.e) conversion of 3 bytes per pixel to 1 byte per pixel. This problem will help us understand three different ways of converting a color image to grayscale image. Also, we will get to see which method of conversion is better for converting an RGB image to grayscale image.

➢ APPROACH AND PROCEDURES:

The RGB to grayscale image conversion can be done by the following given three formulae. They are conversion by lightness, average and luminosity. Note that each formula reduces the 3 bytes per pixel to 1 byte per pixel. Hence the number of bits that can be used to represent an image in grayscale lies in the range of 0 to 255 (2^8). In such a 8 bit representation of the image, 0 represents black (dark) and 255 represents white (brightness). The intermediate intensity values are shades of gray representation.

Theoretical Approach:

Approach 1:

Lightness = (max(R,G,B) + min(R,G,B)) / 2

Approach 2:

Average = (R+G+B) / 2

Approach 3:

Luminosity = 0.21R + 0.72G + 0.07B

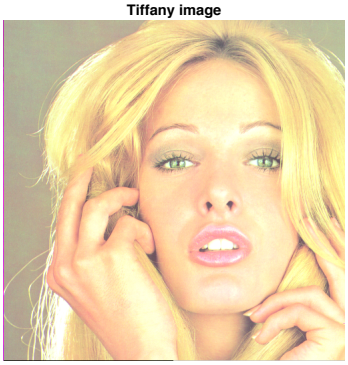
Algorithm Implemented (C++):

main() function:

- Read *Tiffany.raw* image using *fileRead()* function
- Convert 1D image to 3D using *image1Dto3D()* function
- Allocate 2D image pointer arrays for output Grayscale images using *allocMemory2D()*
- Traverse through the image using 2 nested for loops and obtain R, G and B intensity values for every pixel
    - Get max and min value out of R, G and B pixel intensity values
    - Calculate lightness, average and luminosity using the above theoretical formulas.
    - Store each of them at 3 different 2D-image arrays
- Convert all the three 2D image arrays to 1D using *image2Dto1D()*
- Write them to 3 different output raw files using *fileWrite()*
- Deallocate all the memories using *delete*, *freeMemory2D()* and *freeMemory3D()*

➢ EXPERIMENTAL RESULTS:

Tiffany image



Tiffany image - Grayscale Conversion by Average

Tiffany image - Grayscale Conversion by Lightness

Tiffany image - Grayscale Conversion by Luminosity

➢ DISCUSSION:

- From the above results, the lightness method performs poorly. The contrast of the image has significantly reduced when compared to the original RGB color image.
- The Average method performs second best, better than Lightness method.
- The Luminosity method is the best performance one. The contrast and sharpness look preserved much better than the other two methods.
- Many Image Processing tools perform RGB to Grayscale conversion using Luminosity method.

2. CMY(K) COLOR SPACE

➢ ABSTRACT AND MOTIVATION:

There are different color spaces for image representation. Like RGB, CMY is also a color space that is used in real-world applications. One important use of CMY color space conversion is that they are most often used in printing technology. This question will help us understand the RGB to CMY conversion.

➢ APPROACH AND PROCEDURES:

The CMY components of the color space are calculated as given below:

- C = 1 – R or 255 - R
- M = 1 – G or 255 - G

- Y = 1 – B or 255 - B

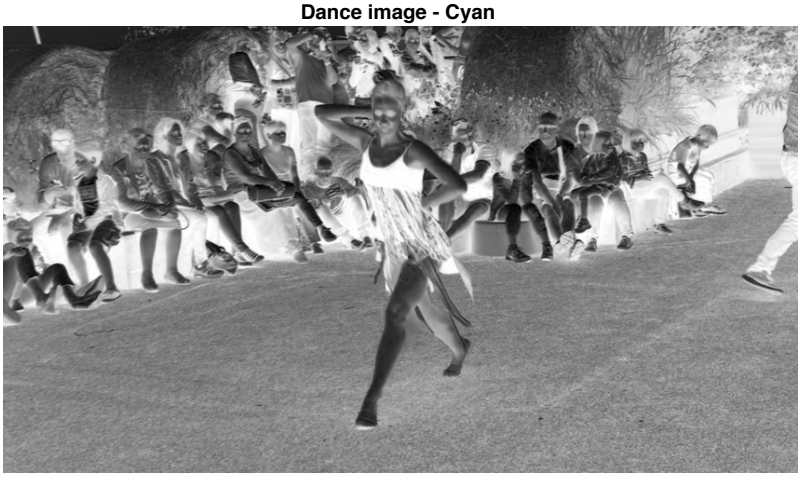
Algorithm Implemented (C++) :

- Read *Bear.raw* image using *fileRead()* function
- Convert 1D image to 3D using *image1Dto3D()* function
- Allocate 2D image pointer arrays for output CMY images using *allocMemory2D()*
- Allocate 3D image pointer array for the final CMY image using *allocMemory3D()*
- Traverse through the image using 2 nested for loops and obtain R, G and B intensity values for every pixel
    - Calculate C, M and Y values from R, G and B values for every pixel
    - Store them to the output 2D arrays initialized
- Convert all the output 2D and 3D image arrays to 1D using *image2Dto1D() and image3Dto1D()*
- Write them to 3 different channel output raw files and combined CMY image using *fileWrite()*
- Deallocate all the memories using *delete*, *freeMemory2D()* and *freeMemory3D()*
- Repeat the same procedure for *Dance.raw* input image

➢ EXPERIMENTAL RESULTS:

Dance image - Input

Dance image - Cyan



Dance image - Magenta

Dance image - Yellow

Dance image - CMY

Bear image - Input

Bear image - Cyan

Bear image - Magenta

Bear image - Yellow

Bear image - CMY

➢ DISCUSSION:

Thus, CMY representation of the RGB color images given were seen. CMY color space can be seen as the color inversion of RGB color space. The darker colors in RGB becomes lighter in the CMY color space. We can observe this very clearly from the body parts of the girl in Dance.raw image. The dress in the Dance.raw image is dark compared to the body parts. But in the CMY image, the colors are inverted.

B) IMAGE RESIZING VIA BILINEAR INTERPOLATION

➢ ABSTRACT AND MOTIVATION:

Often in real-world, the images we capture are not of desired size. Hence to resize an image is a common technique to convert an image of the given size to the desired size. The image can be either resized to a higher size than the given image size or resized to a lower size. Image resizing is done by interpolation of the unknown pixel values using the known pixel values. The technique of interpolation will help us get the image of desired size without much loss of information. This problem will help us understand how image resizing can be done using bilinear interpolation. The given image is of size 512*512. We will resize the image to 650*650 using bilinear interpolation.

➢ APPROACH AND PROCEDURES:

The algorithmic approach of bilinear interpolation is given below:
(Source: https://goo.gl/82Wqod)

Let $\mathbf{I}$ be an $R \times C$ image.
We want to resize $\mathbf{I}$ to $R' \times C'$.
Call the new image $\mathbf{J}$.

Let $s_R = R / R'$ and $s_C = C / C'$.

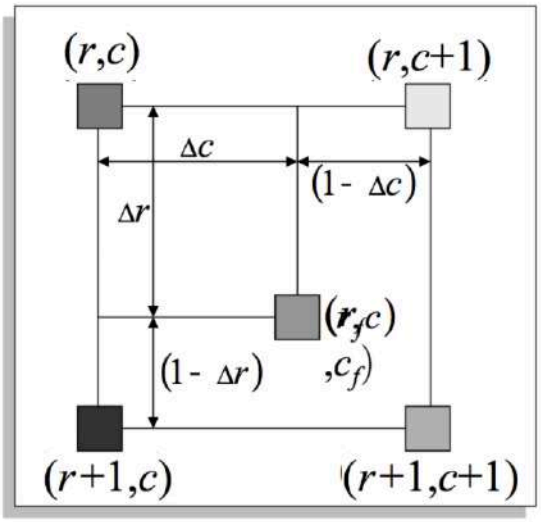
Let $r_f = r' \cdot s_R$ for $r' = 1, \ldots, R'$

and $c_f = c' \cdot s_C$ for $c' = 1, \ldots, C'$.

Let $r = \lfloor r_f \rfloor$ and $c = \lfloor c_f \rfloor$.

Let $\Delta r = r_f - r$ and $\Delta c = c_f - c$.

$$\begin{aligned}\text{Then } \mathbf{J}(r',c') = {} & \mathbf{I}(r,c)\cdot(1-\Delta r)\cdot(1-\Delta c) \\ & + \mathbf{I}(r+1,c)\cdot\Delta r\cdot(1-\Delta c) \\ & + \mathbf{I}(r,c+1)\cdot(1-\Delta r)\cdot\Delta c \\ & + \mathbf{I}(r+1,c+1)\cdot\Delta r\cdot\Delta c.\end{aligned}$$



The above algorithm describes resizing image of size R*C to R'*C'. For every r' and c' in the desired image, the intermediate points in the old image r_f and c_f are calculated. Then, using the 4 boundary points estimated, interpolate the desired intensity value. Since we linearly interpolate along both the rows and columns of the image, this method is called Bilinear Interpolation.

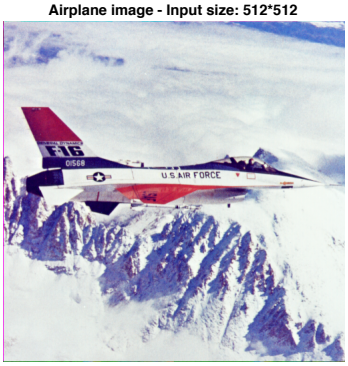
Algorithm Implemented (C++):

- Read *Airplane.raw* image using *fileRead()* function
- Convert 1D image to 3D using *image1Dto3D()* function
- Allocate 3D image pointer array for the final resized image using *allocMemory3D()*
- Obtain row ratio and column ratio using the old image size and desired image size
- Traverse through the desired resized image using 3 nested for loops
    - Get mapped row index and column index in the old image
    - Get reference row and column index by flooring the mapped index values
    - Get delta row and column by subtracting the above values

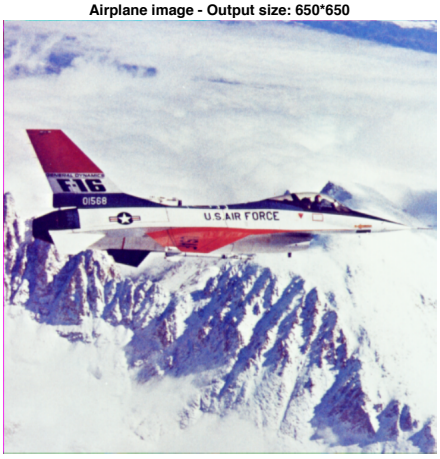
    - Index boundary overflow is taken care by resigning the index to the final row and column index
    - Calculate bilinear interpolated new pixel intensity value using the above formula
- Convert the output resized 3D image to 1D using *image2Dto1D() and image3Dto1D()*
- Write resized output raw image using *fileWrite()*
- Deallocate all the memories using *delete*, *freeMemory2D()* and *freeMemory3D()*

➢ EXPERIMENTAL RESULTS:

Airplane image - Input size: 512*512



Airplane image - Output size: 650*650



➢ DISCUSSION:

Bilinear Interpolation, as above shown helps in obtaining image of the desired size using the pixel intensities of given image. Since it is just an approximation, and the values interpolated gives a slightly distorted image when compared with the original image. This can be more evidently seen from the Airplane part of the image. Both don't look the same. Hence there can be better ways to resize the image instead of just linear approximation, like interpolating based on the characteristics of the pixel values.

PROBLEM 2
HISTOGRAM EQUALIZATION

a) HISTOGRAM EQUALIZATION

➢ ABSTRACT AND MOTIVATION:

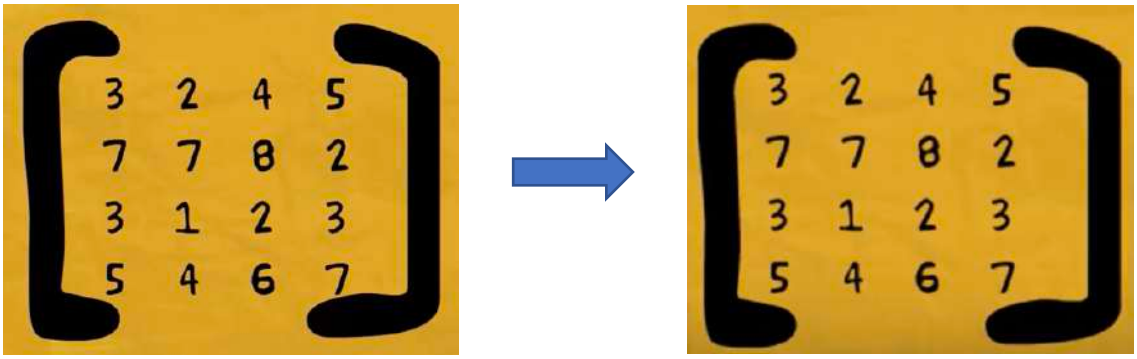
Contrast is one aspect of the image that gives an image desired highlights and shadows. A low contrast image appears flat/dull without them and doesn't exhibit good difference between its lights and dark. The contrast of an image can be adjusted by sophisticatedly modeling the histogram of the image. Histogram is nothing but the count of pixel values (0 to 255) in an image. Thus, equalizing the histogram means, modeling the histogram in such a way that the count of every pixel becomes equal. In this problem, we will compare two different methods of histogram equalization. We will be able to see how an image contrast increases by modifying the image based on modifying its histogram.

➢ APPROACH AND PROCEDURES:

METHOD A: THE TRANSFER-FUNCTION-BASED HISTOGRAM EQUALIZATION METHOD

(Source: https://www.youtube.com/watch?v=PD5d7EKYLcA)

Sample image pixel 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

First image is transformed to second image using the given mapping table:

- First row is the unique pixel intensities located in the given low contrast image
- Second row is the histogram – count of the number of times pixel has occurred
- Third row is the PDF – second row / total count of pixels located in the image.
- Fourth row is the CDF – cumulatively adding the pdf values
- Fifth row is scaling the values to the desired maximum value. Low contrast image histogram is concentrated at one end of the axis. This scaling by multiplication factor allows expanding the histogram
- Sixth row is floor rounding the scaled values. This gives the transfer function mapping of first row pixels
- Every pixel in the input image is now replaced by the corresponding sixth row intensity values. Thus, the image enhancement is done based on transfer function

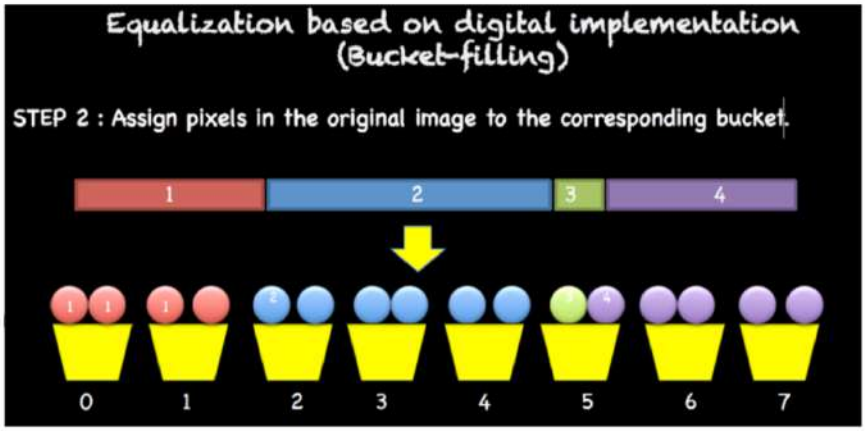
ALGORITHM IMPLEMENTATION (C++):

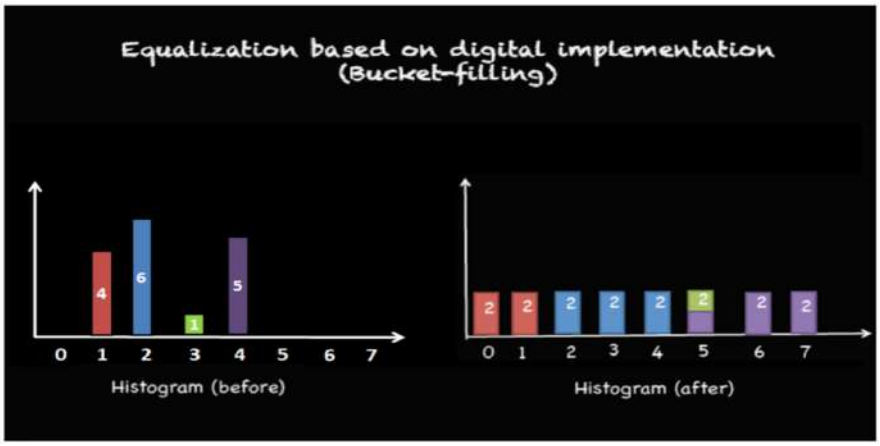
main() function:

- Read *Desk.raw* image using *fileRead()* function
- Convert 1D image to 3D using *image1Dto3D()* function
- Separate R, G and B channels using *seperateChannels()*
- Allocate memory for for histogram equalized channels using allocMemory2D()
- Allocate memory for histograms of original image, equalized image and transfer functions
- For each R, G and B channel:
  - Find the count of 0 to 255 pixels in the channel
  - Normalize the count to get pdf of the pixel values
  - Cumulatively add the pdf to get CDF of pixel values
  - Multiply by 255
  - Get the floor rounded values for every 0 to 255 pixel
  - Replace original image pixel values with the corresponding rounded pixel values
- Combine histogram equalized R, G and B channels to get final contrast enhanced image
- Write all the histograms to text file
- Deallocate all the memories using *delete*, *freeMemory2D()* and *freeMemory3D()*

METHOD B: THE TRANSFER-FUNCTION-BASED HISTOGRAM EQUALIZATION METHOD

The method B assigns equal count values for every pixel 0 to 255:(Source: EE569 – Discussion 2)





- The figure shows that we have to arrange the pixels from 0 to 255 and then assign equal number of pixels to every bucket.
- Bucket size = Total number of pixels in an image / 256
- Once equal number of pixels are assigned in a bucket, the original image is replaced with pixels that are assigned for each bucket.
- By this method we ensure that equal number of pixels are present in every 0 to 255 buckets.

ALGORITHM IMPLEMENTATION (C++):

main() function:

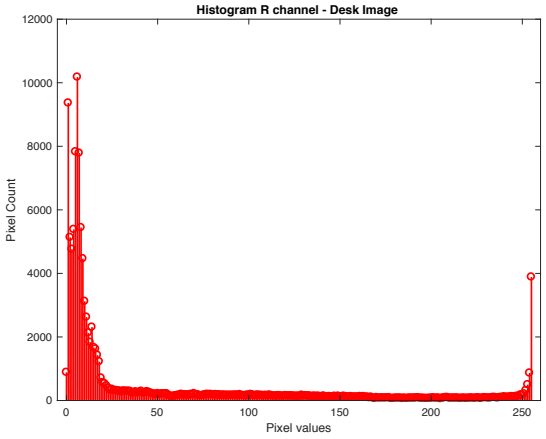
- Read *Desk.raw* image using *fileRead()* function
- Convert 1D image to 3D using *image1Dto3D()* function
- Separate R, G and B channels using *seperateChannels()*
- Allocate memory for for histogram equalized channels using allocMemory2D()
- Allocate memory for histograms of original image, equalized image and transfer functions
- For each R, G and B channel:
    - Call histEqualizationCDFBased() fuction and get back equalized image
- Combine histogram equalized R, G and B channels to get final contrast enhanced image
- Write all the histograms to text file
- Deallocate all the memories using *delete*, *freeMemory2D()* and *freeMemory3D()*

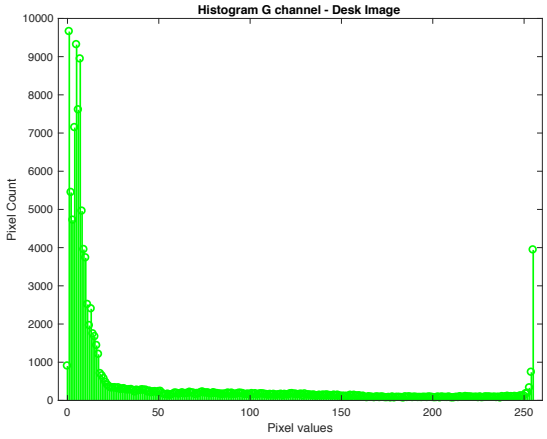
histEqualizationCDFBased() function:

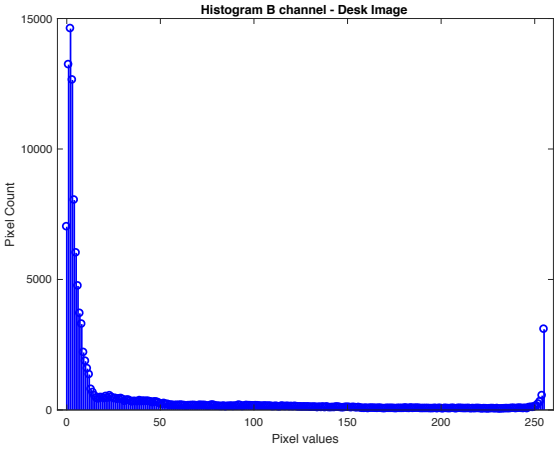
- Initialize three 1D pointer arrays to keep track of row index, column index and pixel value
- Arrange the pixels in the order of 0 to 255. Store the corresponding row index and column index in the 1D arrays
- According to the bucket size calculated Change the pixel values such that each bucket has equal number of pixels
- Using the row and column index tracked in 1D arrays, replace the pixel value in the original image to the new bucketed pixel values
- Return CDF based histogram equalized image

➢ EXPERIMENTAL RESULTS:

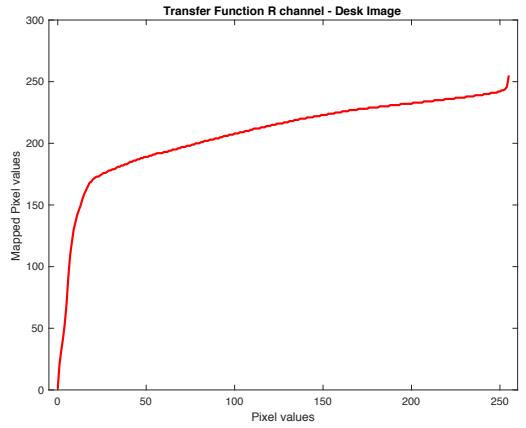
Desk image - Input Low Contrast

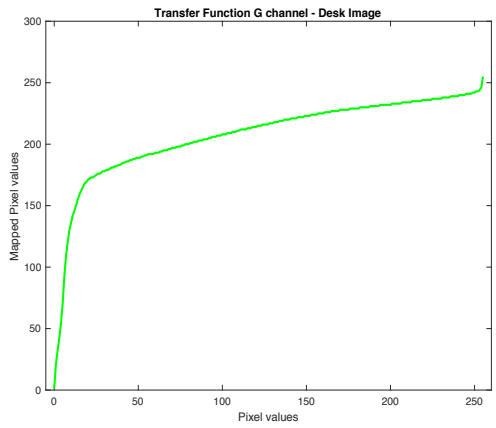


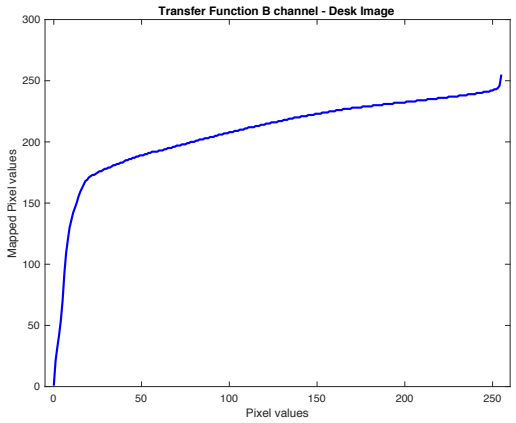


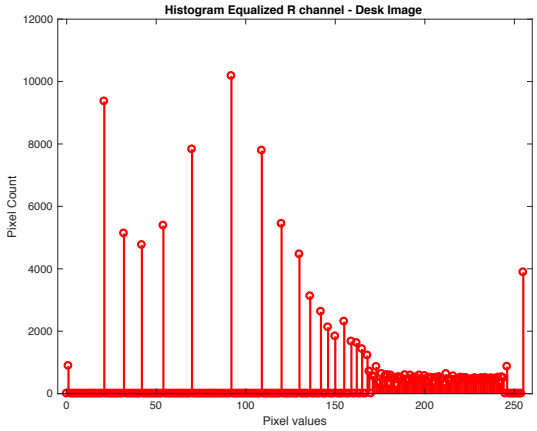


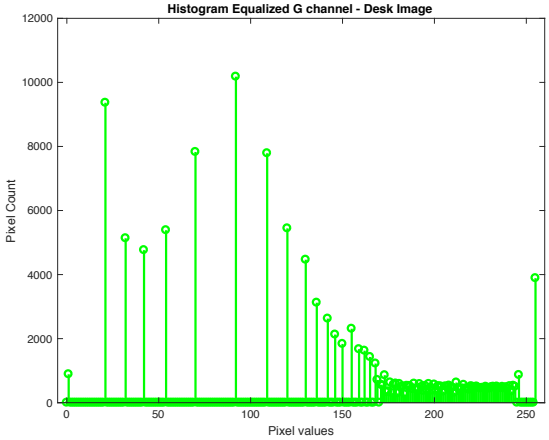
Input Image given with Histograms of Red, Green and Blue Channels

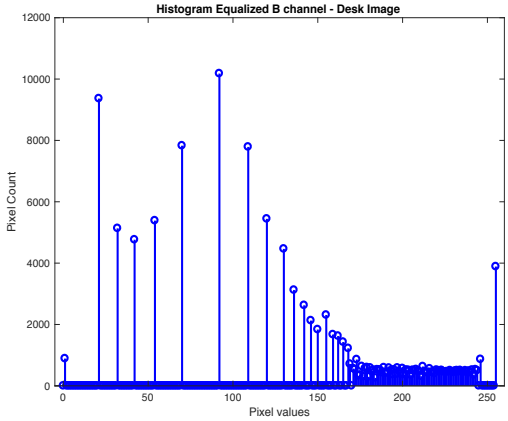




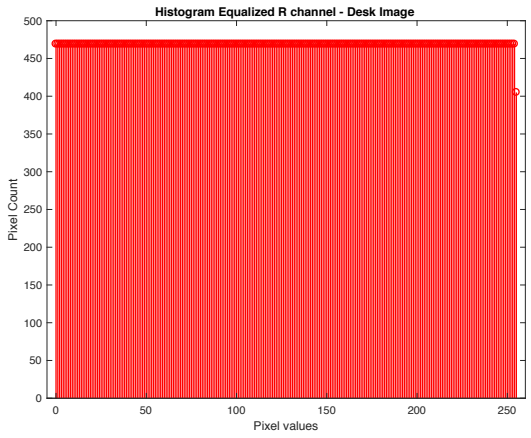


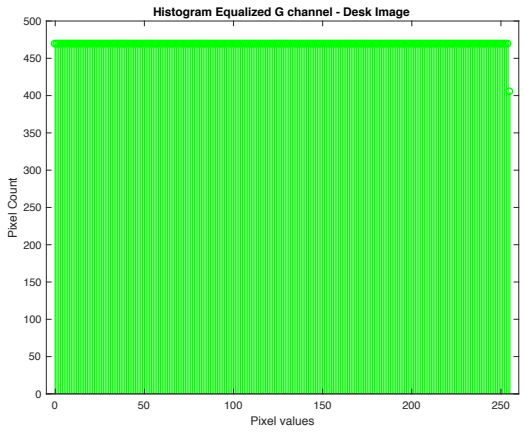


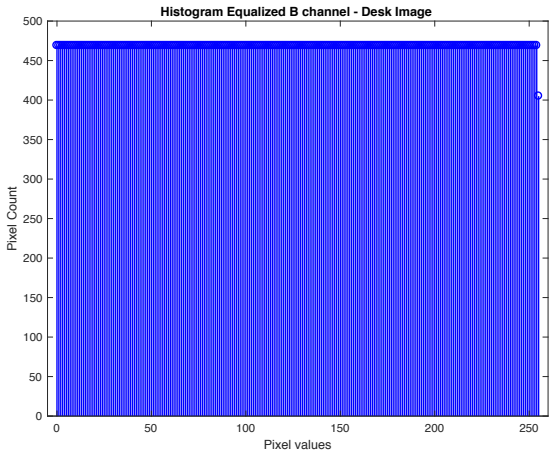


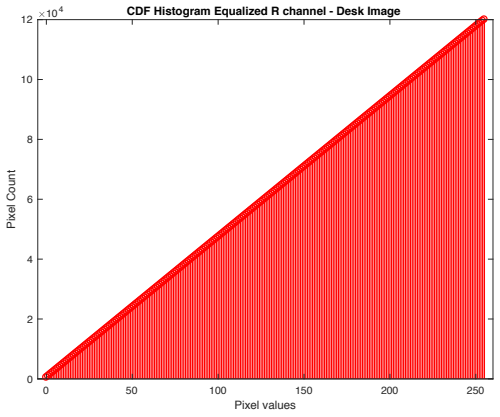


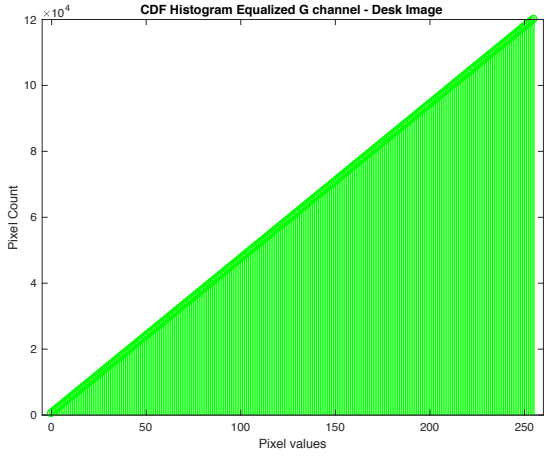
Method A: Transfer Functions and Histograms of Contrast Enhanced image

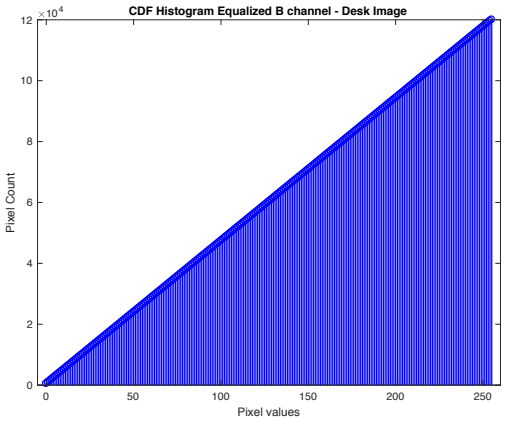












Method B: Histograms and CDF of Contrast Enhanced Image

Desk image - Output Contrast Enhanced - Method A

Desk image - Output Contrast Enhanced - Method B

➢ DISCUSSION:

- The histograms of original low contrast R, G and B channels are given above
- Method A was applied to the original image and the transfer functions for all three R, G and B channels are given above
- Method B was applied to the original image and the cumulative histograms for all three R, G and B channels are given above
- From the histograms of the enhanced image – The following can be observed:
    - Method A is like expanding the histogram. It's simple and enhances the contrast of the image. This method worked in our given image because the histogram was concentrated towards the left end. But if the pixel values were far apart from each other, then contrast enhancement using this method will fail. Also the brightness of the image looks altered.
    - Method B perfectly equalizes the histogram. The number of pixels in every bin is equal. It has good performance than method A in contrast enhancement. But since every pixel has to be checked and binned, the process can become computationally expensive for larger images.
- Since both the methods have their own cons in enhancing the image, we can go with some other histogram equalization method, where it is computationally simple (like method A) and also equalizes the histogram for every bin (like method B).
- There are many other algorithms like Bi-Histogram Equalization, Par Sectioning, Odd Sectioning etc. We can choose the best method over Method A and Method B that meets the requirement.

b) IMAGE FILTERING – CREATING OIL PAINTING EFFECT

➢ ABSTRACT AND MOTIVATION:

The image when showed in its raw format are not very appealing. People, nowadays prefer various types of filter and effects in their image. There are various applications booming internet boasting of various effects to change the raw image. This problem will help us understand how to create the famous oil painting effect on an image. At the end of problem, given an input image, we will be able to apply an oil-painting effect on that image. This is done in two steps – Quantization and Masking. The variations in choosing various parameters are also discussed.

➢ APPROACH AND PROCEDURES:

- Quantization: Given an input image, all the colors are quantized to 64 colors
  - If R, G, B should contain 64 colors together, each channel should consist of 4 colors – cube root (64)
  - Should bin pixels into 4 bins and replace all the bin values using the mean of that bin
- Masking: For each pixel, the most frequent color in its N*N neighborhood is replaced
- Quantization and Masking can be tried with various levels and mask size

ALGORITHM IMPLEMENTATION (C++):

main() function:

- Read *Trojans.raw* image using *fileRead()* function
- Convert 1D image to 3D using *image1Dto3D()* function
- Separate R, G and B channels using *seperateChannels()*
- Allocate different 2D image arrays for quantization and the oil effect for each R, G and B channel
- Pass every channel to *quantize2DImage()* function given the quantization level
- Pass the quantized images to oilEffectImage2D() function given the mask size
- Combine oil painting effect R, G and B channels to get final image using *combineChannels()*
- Write both the quantized image and final oil painting effect image the output path using *fileWrite()*
- Deallocate all the memories using *delete*, *freeMemory2D()* and *freeMemory3D()*
- Repeat the same procedure for *Star_Wars.raw*

quantize2DImage() function:

- Initialize three 1D pointer arrays to keep track of row index, column index and pixel value
- Arrange the pixels in the order of 0 to 255. Store the corresponding row index and column index in the 1D arrays
- Find the bin size based on the quantization level for each channel
- Obtain the mean value for every bin
- Replace all the pixel values in that bin to the mean value
- Track back the pixels from 1D array to 2D array based on row index and column index

oilEffectImage2D() function:

- Traverse every pixel using 2 nested for loops
  - Get the N*N surrounding pixel row and column index values
  - If the indexes are < 0, then replace with 0. Similarly, if they exceed image size, then replace with the corresponding max value
  - Get the surrounding pixel values and find the most frequently occurring pixel
  - Replace the pixel value by the previously obtained pixel value

➢ EXPERIMENTAL RESULTS:

Quantization Level - 64

Quantization Level - 64, Mask size - 3

Quantization Level - 64, Mask size - 5

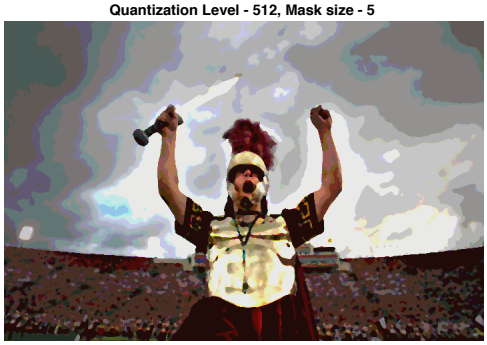
Quantization Level - 64, Mask size - 7

Trojans image – Quantization levels 64 with 3 different masks (above)

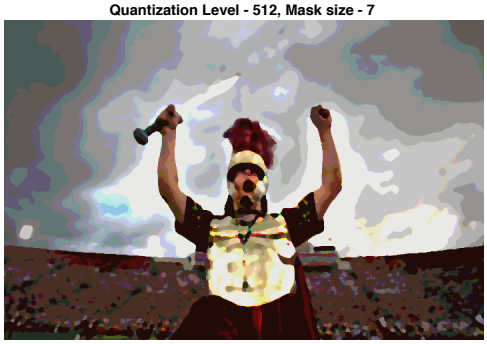
Quantization Level - 512

Quantization Level - 512, Mask size - 3

Quantization Level - 512, Mask size - 5

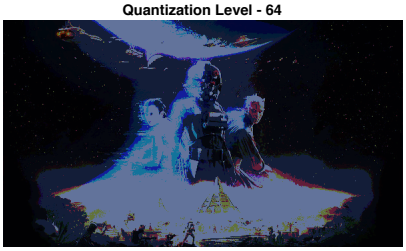


Quantization Level - 512, Mask size - 7

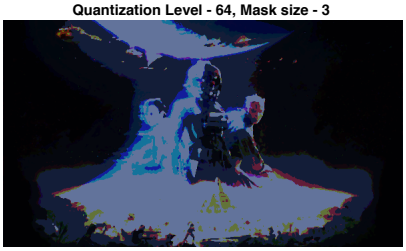


Trojans image – Quantization levels 512 with 3 different masks (above)

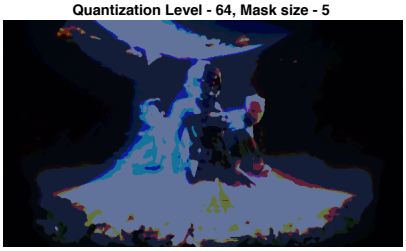
Quantization Level - 64



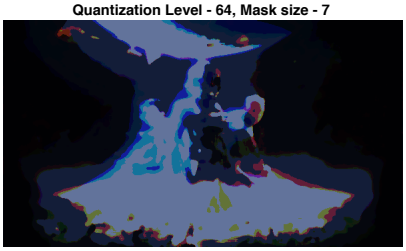
Quantization Level - 64, Mask size - 3



Quantization Level - 64, Mask size - 5



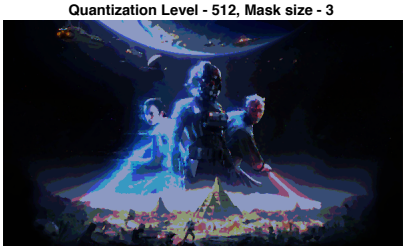
Quantization Level - 64, Mask size - 7



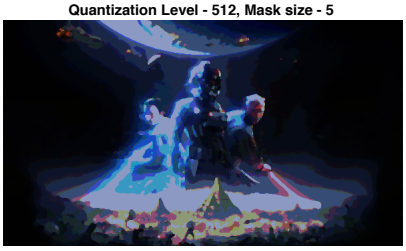
Star Wars image – Quantization levels 64 (above) and 512 (below) with 3 different masks

Quantization Level - 512

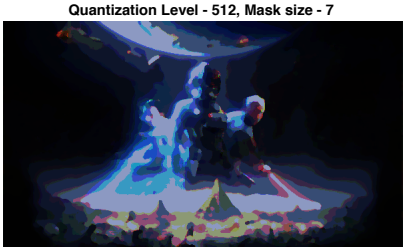
Quantization Level - 512, Mask size - 3



Quantization Level - 512, Mask size - 5



Quantization Level - 512, Mask size - 7



➢ DISCUSSION:

- We tried creating oil-painting effect on two different images. We should note that Star_Wars.raw image is very low in brightness and the Trojans.raw image is better
- We tried two different color quantization levels – 64 and 512
  - 64 quantization level has obviously very less number of colors than 512 quantization level. Hence 512 is better and have different variations of color in the quantized image
  - Comparing 64 and 512 quantized images of Trojans – The audience is visible well in 512 level than 64 level
  - Comparing 64 and 512 quantized images of Starwars – The mountains are well visible in 512 level than 64 level.
  - Thus, we can conclude, higher the quantization level, good differentiation of colors will happen in oil-painting effect
- We also tried various N*N neighborhood – 3*3, 5*5 and 7*7
  - The 7*7 and 5*5 applied images have good oil-effect when compared to 3*3 images
  - Oil effect is better done with higher mask size since the smoothening effect aligns well with the desired effect
- Overall 512 quantization level with 7*7 neighborhood makes a good pair for oil effect

c) IMAGE FILTERING – CREATING FILM SPECIAL EFFECT

➢ ABSTRACT AND MOTIVATION:

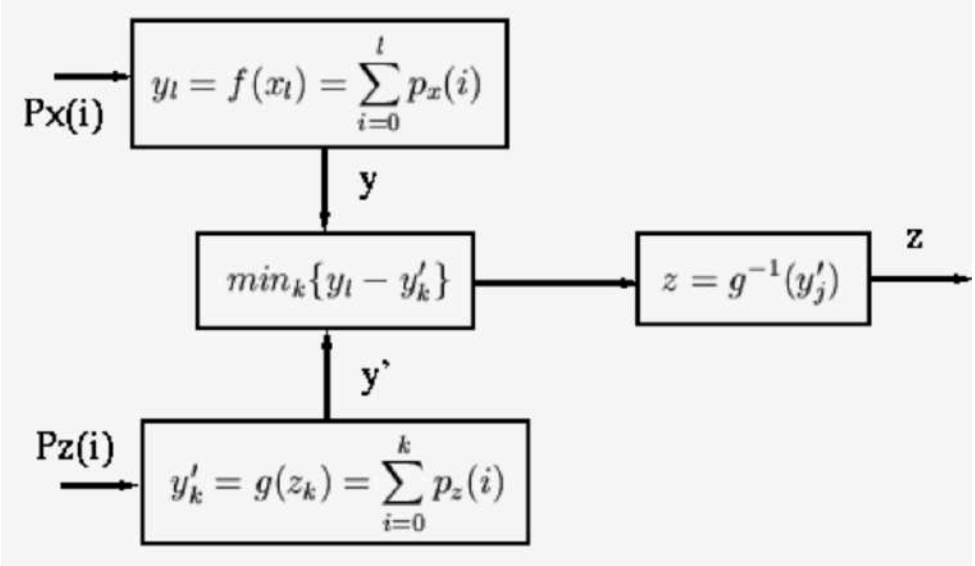
In this problem, we will be developing an algorithm for Film Special Effect. There is an input image and output Film effect image given. We would investigate what is the relationship between the input and output image. This would help us come up with an algorithm to create a special film effect for the Girl image given

➢ APPROACH AND PROCEDURES:

- On comparison with the given Original.raw image and output Film.raw image, the following observations can be made:
  - The image is inverted along columns. Which means, the image is mirrored
  - The image is also color inverted. Every pixel is approximately (255 – pixel) value
  - Post mirroring and inverting, we can compare it's R, G and B channel histogram with the given Film.raw image R, G and B channel's histogram
  - On comparison with each channel histogram, we can find a transfer function using Histogram Specification

  (Source: http://fourier.eng.hmc.edu/e161/lectures/contrast_transform/node3.html)
  - The details of the algorithm is given below:
    - Find the histogram of original image (mirrored and inverted) h_x and its cumulative H_x
    - Find the histogram of Film.raw image h_z and its cumulative H_z

- We can relate the cumulative histogram : for every pixel 's cdf value in the original image, get the closest cdf value from Film image
  $|H_x(i) - H_z(j)| = \min_k |H_x(i) - H_z(k)|$
- Get the corresponding pixel index for the chosen closest cdf value
- Use this and obtain the transfer function to modify the original image to get the Film Special Effect



ALGORITHM IMPLEMENTATION (C++):

main() function:

- Read *Original.raw, Film.raw and Girl.raw* image using *fileRead()* function
- Convert all 1D image to 3D using *image1Dto3D()* function
- Mirror Original.raw image using *mirror3DImage()*
- Invert colors of the mirrored image using *invert3DImage()*
- Separate R, G and B channels of mirrored and inverted original image and Film.raw image using *seperateChannels()*
- Get the histograms of R, G and B channels of original and Film image using *histogram2DImage()*
- Calculate the pdf and cdf of both the image channel's histograms
- Obtain the transfer functions for every channel using *histogramSpecification()*
- Using the transfer function change the pixel intensity values of the original image.
- Mirror and invert colors of the girl.raw image using *mirror3DImage()* and *invert3DImage()*
- Using the transfer function obtained get the Film effect in Girl.raw image
- Combine special effect R,G and B channels to get final image using *combineChannels()*
- Write image to the output path using *fileWrite()*
- Deallocate all the memories using *delete*, *freeMemory2D()* and *freeMemory3D()*

mirror3DImage() function:

- Traverse through the image pixels using three nested for loops
- Invert the columns of the image
- Return mirrored image

invert3DImage() function:

- Traverse through every pixel value using three nested for loops
- Get the inverted values by subtracting from 255
- Return the color inverted image

histogramSpecification() function:

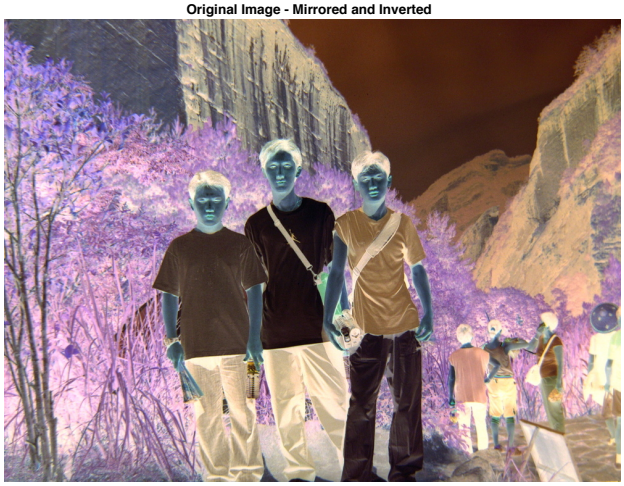
- Traverse through the cdf value of the given image using a single for loop
    - Subtract the cdf from all the cdf values of the target image
    - Obtain the minimum difference
    - Find the pixel index corresponding to the minimum difference
    - Use that value to form the transfer function
- Return the transfer function mapping array

➢ EXPERIMENTAL RESULTS:

Original Image - Input

Original Image - Mirrored

Original Image - Mirrored and Inverted

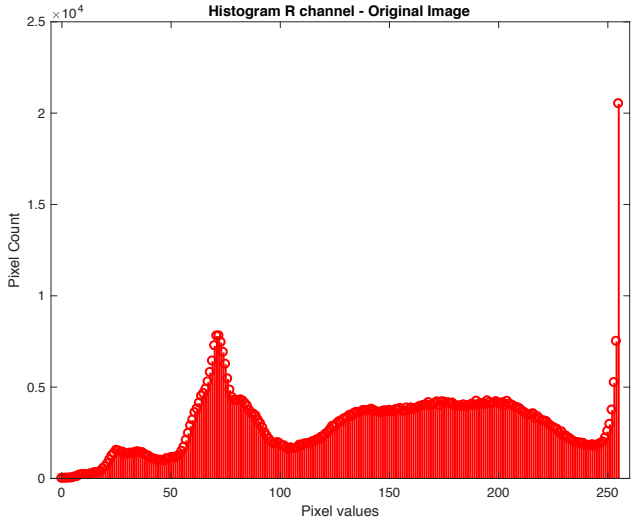


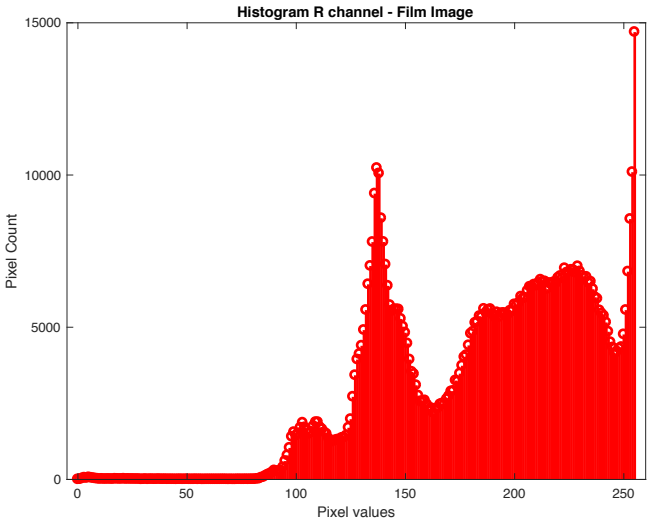
Girl Image - Input

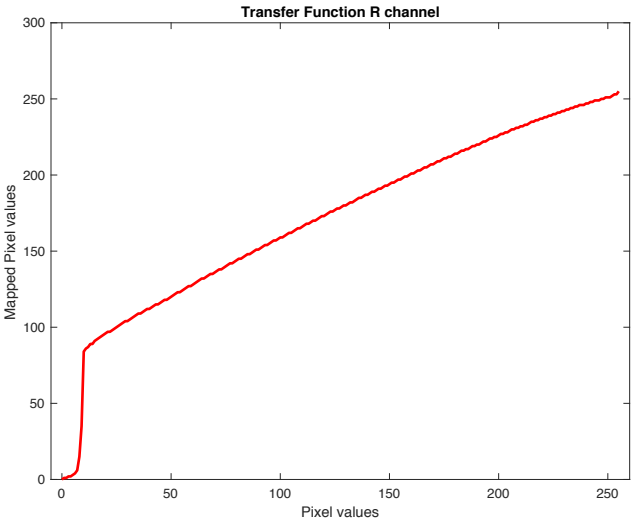
Girl Image - Mirrored

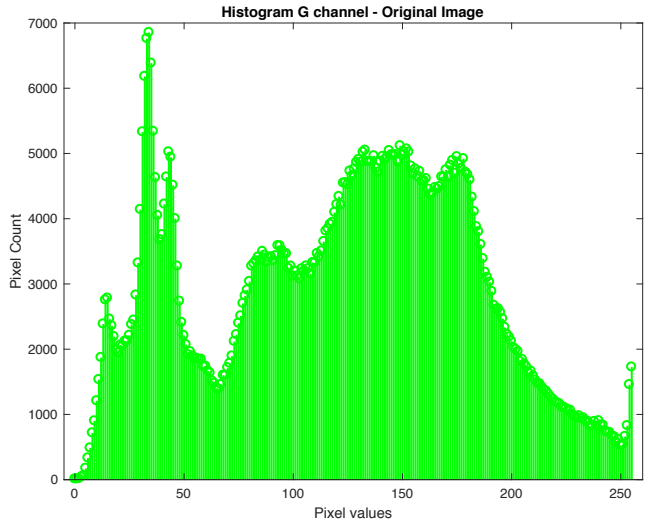
Girl Image - Mirrored and Inverted

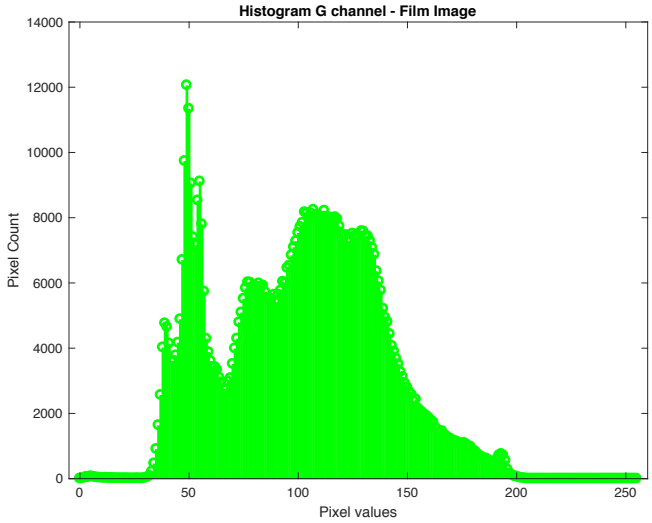
Film Image - Given Input

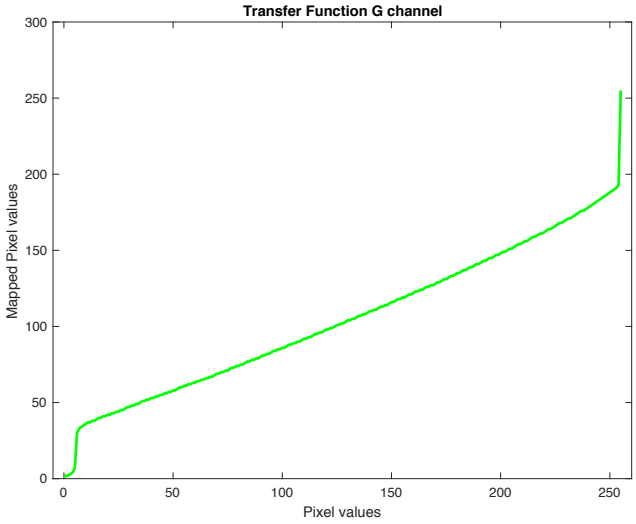

Histogram R channel - Original Image
×10^4
2.5
2
1.5
1
0.5
0
Pixel Count
0
50
100
150
200
250
Pixel values

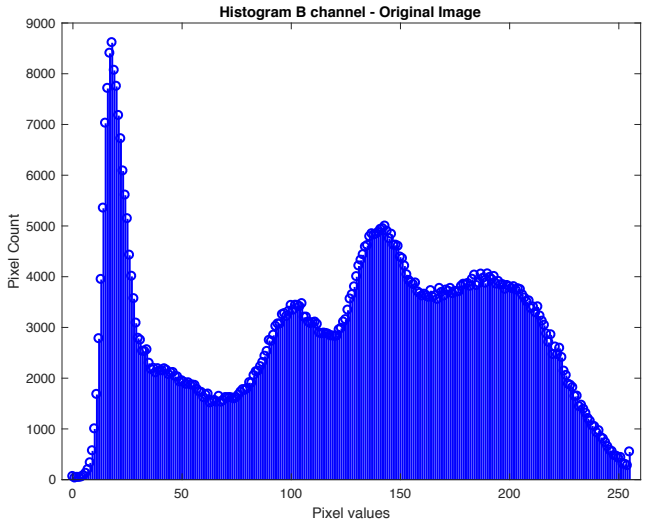

Histogram R channel - Film Image
15000
10000
5000
0
Pixel Count
0
50
100
150
200
250
Pixel values

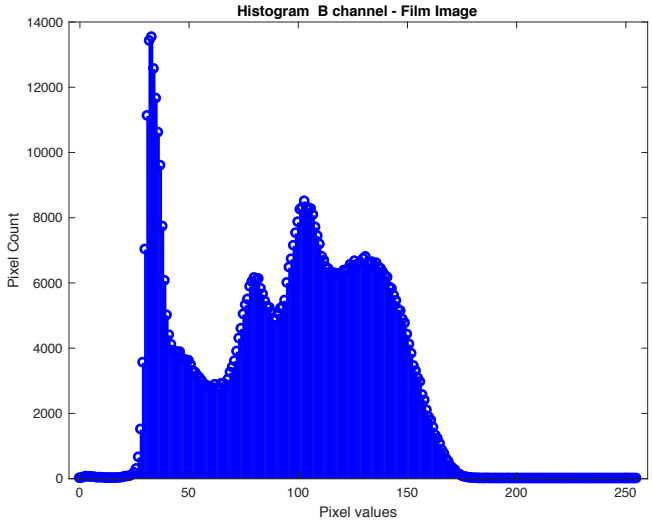

Transfer Function R channel
300
250
200
150
100
50
0
Mapped Pixel values
0
50
100
150
200
250
Pixel values

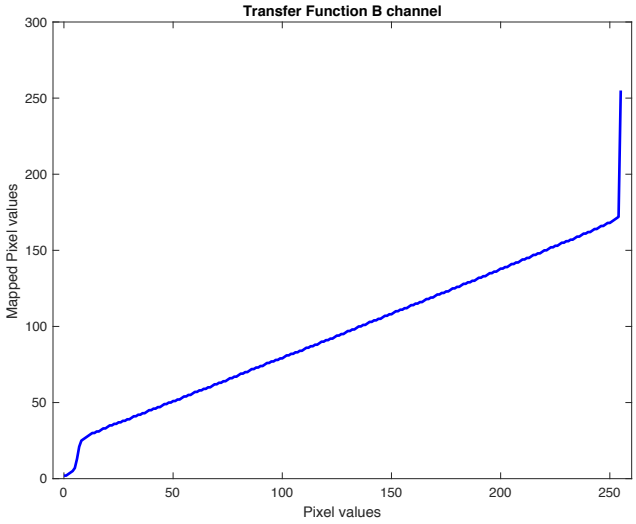

Histogram G channel - Original Image
Pixel Count
Pixel values
0
1000
2000
3000
4000
5000
6000
7000
0
50
100
150
200
250


Histogram G channel - Film Image
Pixel Count
Pixel values
0
2000
4000
6000
8000
10000
12000
14000
0
50
100
150
200
250


Transfer Function G channel
Mapped Pixel values
Pixel values
0
50
100
150
200
250
300
0
50
100
150
200
250


Histogram B channel - Original Image
Pixel Count
Pixel values
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
0
50
100
150
200
250


Histogram B channel - Film Image
Pixel Count
Pixel values
0
2000
4000
6000
8000
10000
12000
14000
0
50
100
150
200
250


Transfer Function B channel
Mapped Pixel values
Pixel values
0
50
100
150
200
250
300
0
50
100
150
200
250

Girl Image - Special Firm Effect - Output

➢ **DISCUSSION:**

- The special film effect was thus achieved using the following 3 steps:
    - Mirroring the image
    - Color Inverting the image
    - Finding the transfer function to match the present histogram with the desired histogram – Histogram Specification
    - Changing the pixel values according to the obtained transfer function

PROBLEM 3
NOISE REMOVAL

a) MIX NOISE IN COLOR IMAGE

➢ ABSTRACT AND MOTIVATION:

In real-world any image we capture is affected by noise. There are many reasons why noises are added to the image. More often, the reason and the type of noise is unidentifiable. And images are exposed to different types of noise and denoising an image has always been a perfect research topic. This problem will help us understand mixed noise in an image and different types of filter to denoise the given image.

The mixed noise types that can be found in the image are:

- Salt Noise (Extra 255's in the image - white)
- Pepper Noise (Extra 0's in the image - black)
- Salt and Pepper Noise (Extra 0's and 255's in the image)
- Gaussian Noise (Noise added to the image follows Gaussian Distribution)

The type of filters that can be applied to the image to denoise are:

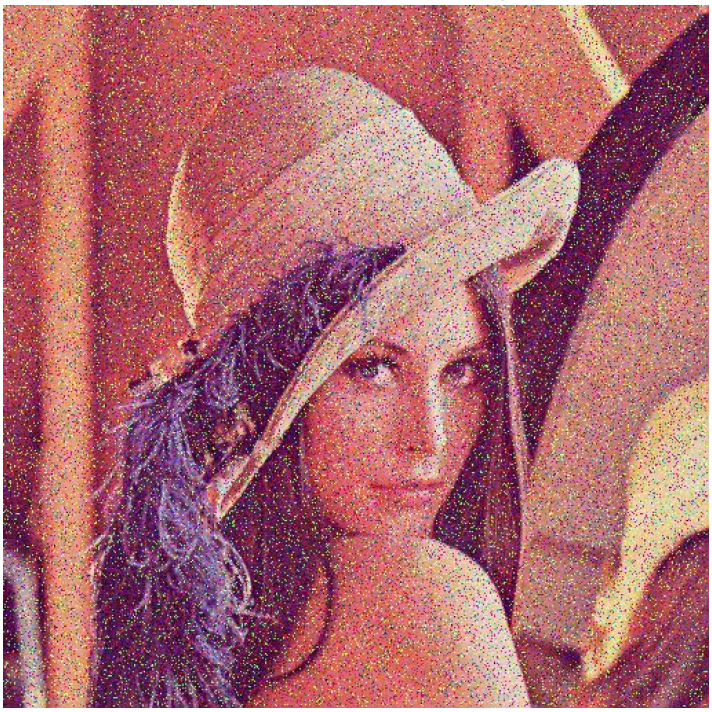
- Median Filter (3*3, 5*5 and 7*7) – Denoise Salt and Pepper Noise
- Low Pass Filter (3*3, 5*5 and 7*7) – Denoise Gaussian Noise
- Gaussian Filter ((3*3, 5*5 and 7*7) – Denoise Gaussian Noise

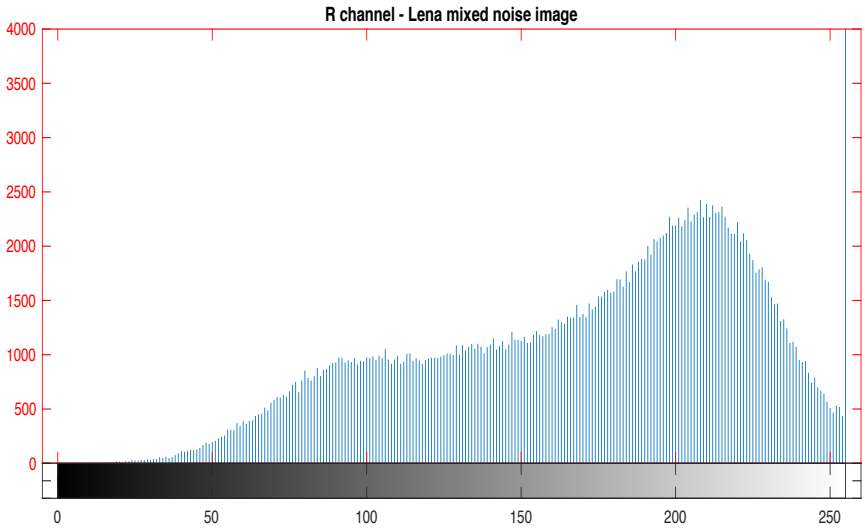
➢ APPROACH AND PROCEDURES:

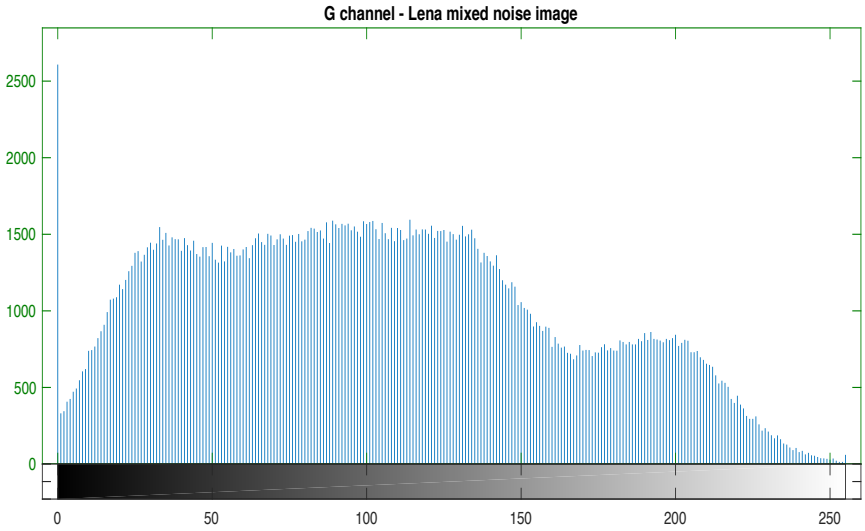
Theoretical Approach:

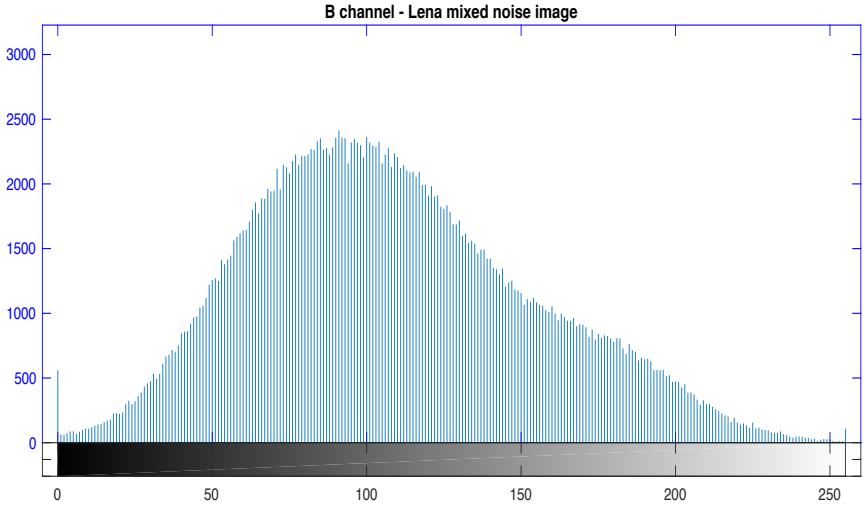
The lena mixed noise image given and corresponding histograms of R, G and B channels are given below to analyze the type of Noise.

Lena mixed noise image




R channel - Lena mixed noise image
4000
3500
3000
2500
2000
1500
1000
500
0
0
50
100
150
200
250

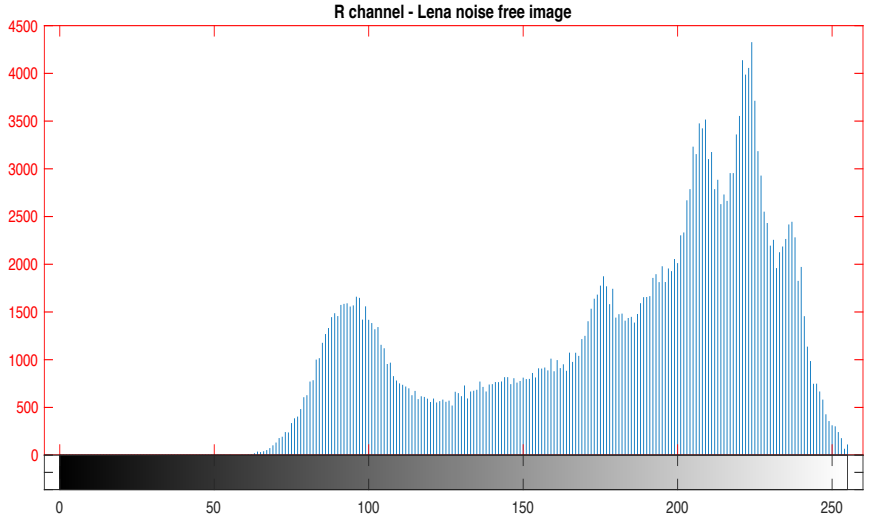

G channel - Lena mixed noise image
2500
2000
1500
1000
500
0
0
50
100
150
200
250

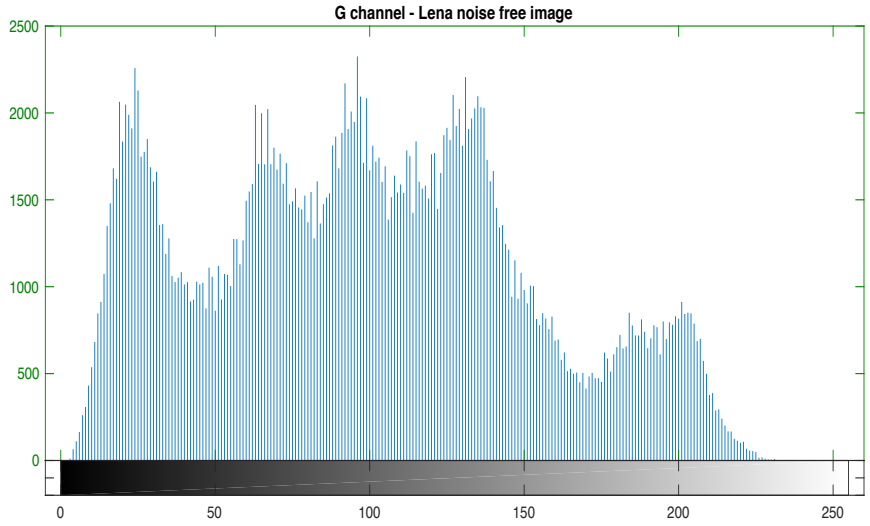

B channel - Lena mixed noise image
3000
2500
2000
1500
1000
500
0
0
50
100
150
200
250

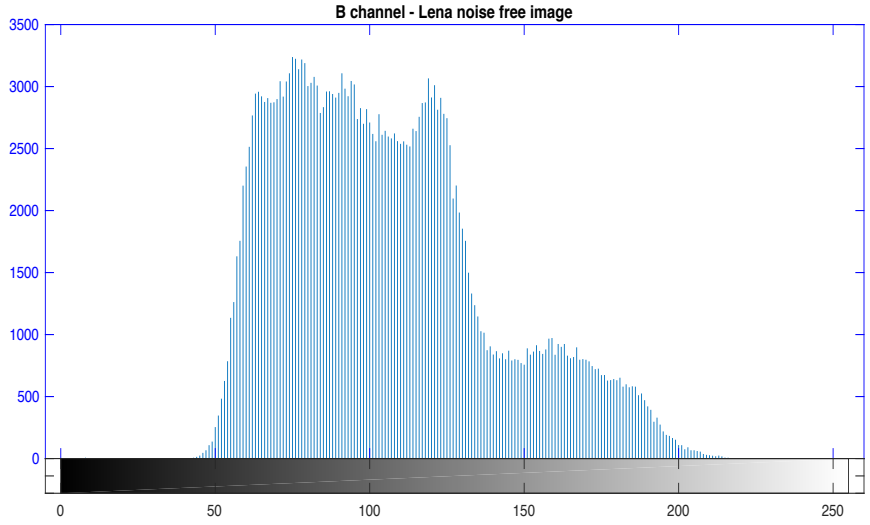
The Lena Noise free image and histograms of the R, G and B channels are given below:

The histograms can be compared with the noisy image histograms to identify the type of noise present in R, G and B channels.

Lena noise free image







On comparison with the Noise free image histograms, key observations are:

- R channel has Salt Noise – (There is a big peak at 255)
- G channel has Pepper Noise – (There is a big peak at 0)
- B channel has Salt and Pepper noise – (There are small peaks at 0 and 255)
- All the R, G and B channel histograms are smoothed out when compared to noise free channel histograms. This shows that Gaussian Noise is added to all the R, G and B channels. The mean of the noise added to the image would have been calculated from the mean of the pixel intensity values itself. That's why the image histograms are not shifted and rather the noise is smoothing out the histograms.

The theory to denoise the image is given below. Various types of filters are used and compared the results in the experiment session. The best filter combination is identified to denoise the given mixed Lena image. To apply the filters to boundary pixels, replication of the boundary pixels is done.

Quality Metric (PSNR):

The Peak-Signal-to-Noise-Ratio is one quality metric to access the performance of denoising algorithm. The formula for PSNR value calculation is given below:

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

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

X : Original Noise-free Image of size $N \times M$

Y : Filterd Image of size $N \times M$

Max: Maximum possible pixel intensity = 255

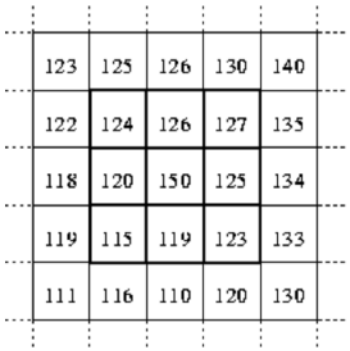
The higher the PSNR value, the quality of image is good. The purpose of this problem is to improve the PSNR value of the given mixed noise image

List of Filters used to denoise the image:

1. **Median Filter:**

   Source: https://homepages.inf.ed.ac.uk/rbf/HIPR2/median.htm

   It takes the N*N pixel values around each pixel and replaces it with the median value. Sample Median filter procedure is given below:



2. **Low Pass Filter:**

   Source: https://diffractionlimited.com/help/maximdl/Low-Pass_Filtering.htm

   It takes the N*N pixel values around each pixel value and replaces it the average of those values. Sample 3*3 Low pass filter mask is given below:

+1/9	+1/9	+1/9
+1/9	+1/9	+1/9
+1/9	+1/9	+1/9

3. **Gaussian Filter:**

   Source: https://goo.gl/qUdmdM

   Since we are working with functions we need 2 Dimensional Gaussian functions. The Kernal coefficients are sampled from the below Gaussian function given:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

   where, distribution is of $\mu = 0$ and $\sigma = 1$. A sample 5*5 Gaussian mask is given below:

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

Algorithm Implemented (C++):

main() Function:

- Read *Leena_mixed.raw* image using *fileRead()* function
- Convert 1D image to 3D using *image1Dto3D()* function
- Separate R, G and B channels using *seperateChannels()* function and store them in three different 2D arrays
- Allocate memory for Denoised R, G and B channels using *allocMemory2D()*
- Initialize the Denoised R, G and B with values from Noisy image intensity values
- Depending upon the argument given pass the Noisy image to *medianFilter()* or *lowPassFilter()* or *GaussianFilter()*. The size of the mask is also given as the argument
- PSNR values of R, G and B channels are calculated using *calculatePSNR2D()*
- PSNR value for the entire image is calculated using *calculatePSNR3D()*
- Write them to output raw files using *fileWrite()*
- Deallocate all the memories using *delete*, *freeMemory2D()* and *freeMemory3D()*

medianFilter() Function:

- Traverse through each pixel in the image using 2 nested for loops
- Calculate the indexes of the N*N neighborhood images to be considered for applying filter
  - If the row or column index becomes < 0, reassign to 0
  - If the row or column index becomes > row or > column, reassign to row and column respectively
  - Store all the N*N neighborhood pixels in a 1D array
  - Sort the array using bubble sort
  - Get the mid value of the sorted array
  - Assign the found mid value to the denoised image pixel value
- Return the denoised image

lowPassFilter() function

- Traverse through each pixel in the image using 2 nested for loops
- Calculate the indexes of the N*N neighborhood images to be considered for applying filter
    - If the row or column index becomes < 0, reassign to 0
    - If the row or column index becomes > row or > column, reassign to row and column respectively
    - Store all the N*N neighborhood pixels in a 1D array
    - Find the average of the 1D array
    - Assigned the found average value to the denoised image pixel value
- Return the denoised image

gaussianFilter() function

- Design the Gaussian N*N mask using the theoretical formula given above.
- Normalize the Gaussian mask
- Traverse through each pixel in the image using 2 nested for loops
- Calculate the indexes of the N*N neighborhood images to be considered for applying filter
    - If the row or column index becomes < 0, reassign to 0
    - If the row or column index becomes > row or > column, reassign to row and column respectively
    - Store all the N*N neighborhood pixels in a 1D array
    - Multiply the corresponding values from 1D array with the Gaussian mask sum up
    - Assigned the found value to the denoised image pixel value
- Return the denoised image

calculatePSNR2D() function:

- Traverse through each pixel value using 2 nested for loops
    - Find the difference between the noise free pixel value and denoised image pixel value
    - Find the square of the difference and sum them
- Using the PSNR formula given above, calculate the final psnr value
- Return PSNR value for the denoised image

calculatePSNR3D() function:

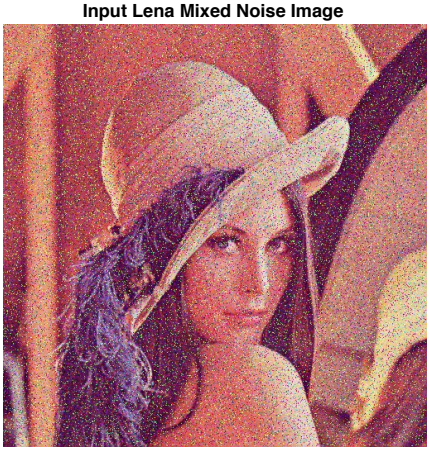
- Traverse through each pixel value using 3 nested for loops
    - Find the difference between the noise free pixel value and denoised image pixel value
    - Find the square of the difference and sum them
- Using the PSNR formula given above, calculate the final psnr value
- Return PSNR value for the denoised image

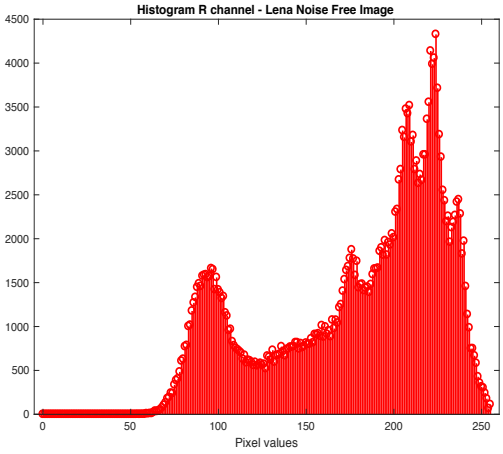
➢ EXPERIMENTAL RESULTS:

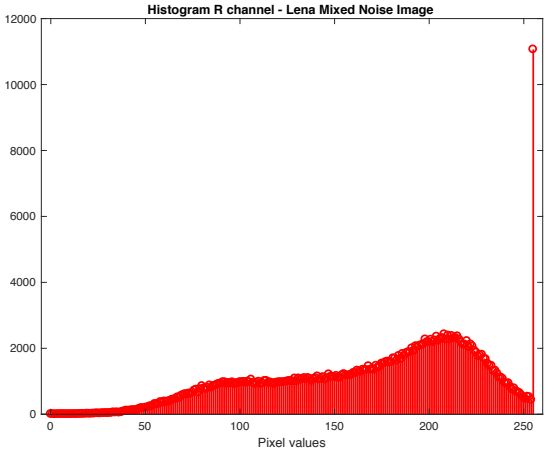
Mask	1st Filter	2nd filter	3rd filter	R – PSNR	G – PSNR	B - PSNR	PSNR
-	-	-	-	17.0867	16.5475	17.0143	16.8762
3	Median	-	-	25.2052	27.328	25.5484	25.9325
5	Median	-	-	24.5547	25.1342	25.0988	24.921
7	Median	-	-	22.5112	22.92	22.9999	22.805
3	Low Pass	-	-	23.6828	23.7352	25.4681	24.22
5	Low Pass	-	-	23.0777	23.0757	24.3112	23.4505
7	Low Pass	-	-	23.3677	23.2622	25.3473	23.8925
3	Gaussian	-	-	22.7376	22.6953	23.8453	23.0611
5	Gaussian	-	-	23.3125	23.3896	24.7115	23.7588
7	Gaussian	-	-	23.3531	23.4385	24.7834	23.8108
3	**Median**	**Low Pass**	**-**	**25.3722**	**27.0463**	**26.335**	**26.1966**
5	Median	Low Pass	-	23.8121	24.2006	24.8105	24.2552
3	**Median**	**Gaussian**	**-**	**25.3673**	**27.2355**	**26.2919**	**26.2315**
5	Median	Gaussian	-	24.1298	24.7523	25.0161	24.6167
3	Median	Low Pass	Gaussian	24.9907	26.5584	26.4177	25.9293
3	**Median**	**Gaussian (B alone)**	**-**	**25.2052**	**27.328**	**26.2919**	**26.1888**

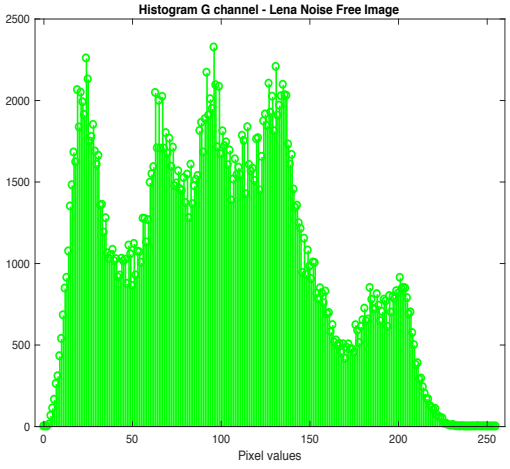
Input Lena Noise Free Image

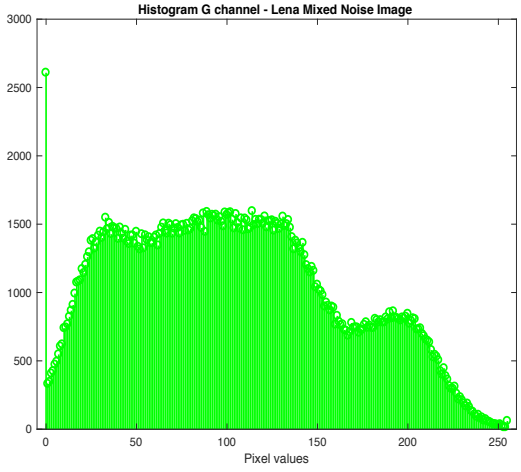
Input Lena Mixed Noise Image

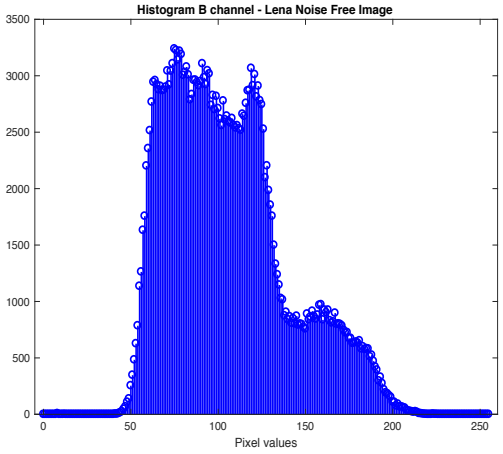


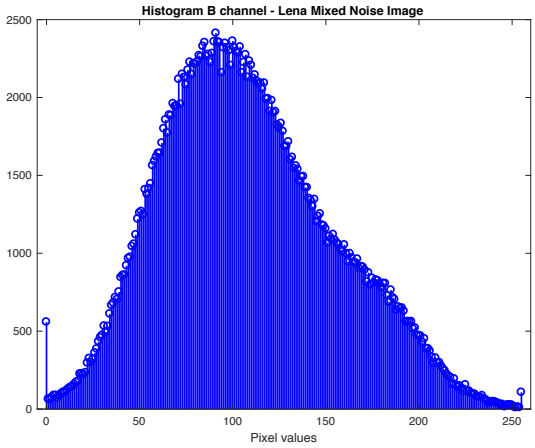






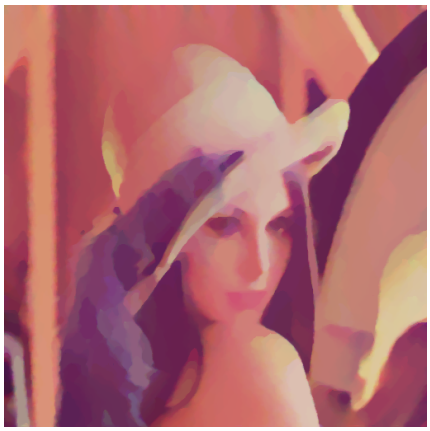






Sample Results for the given table:

Lena mixed denoised 7 median.raw



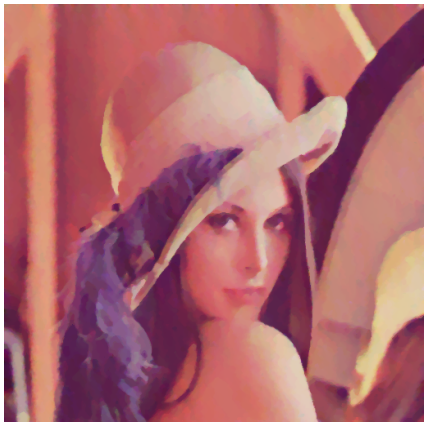
Lena mixed denoised 7 lowpass.raw

Lena mixed denoised 7 gaussian.raw

Lena mixed denoised 5 median lowpass.raw

Lena mixed denoised 5 median gaussian.raw

Lena mixed denoised 5 median.raw



Lena mixed denoised 5 lowpass.raw

Lena mixed denoised 5 gaussian.raw

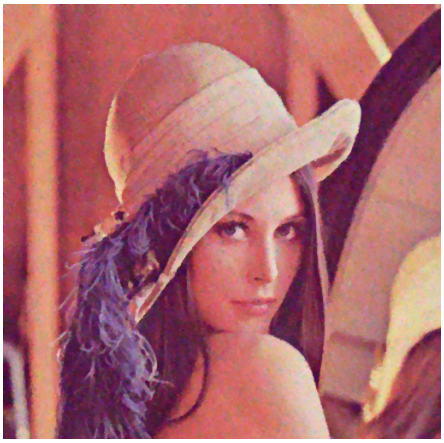
Lena mixed denoised 3 median lowpass.raw

Lena mixed denoised 3 median lowpass gaussian.raw

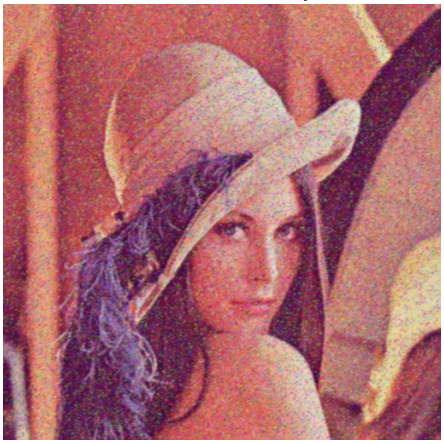
Lena mixed denoised 3 median gaussian.raw

Lena mixed denoised 3 median gaussian lowpass.raw

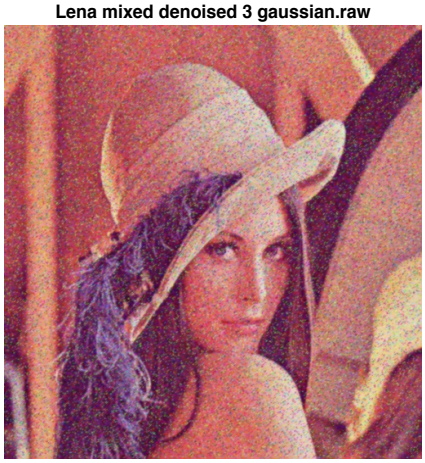
Lena mixed denoised 3 median.raw



Lena mixed denoised 3 lowpass.raw



Lena mixed denoised 3 gaussian.raw

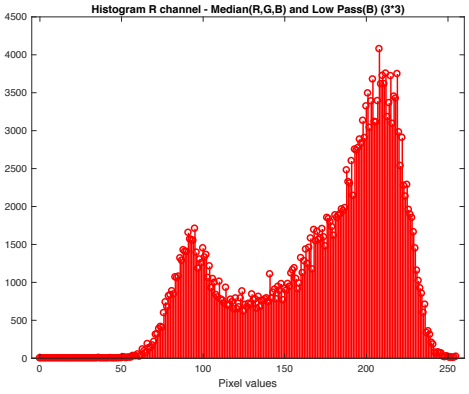


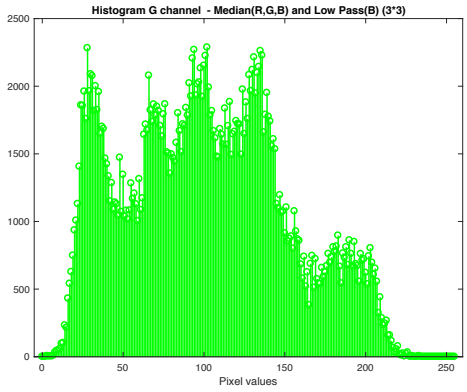
Lena mixed denoised3 median lowpass.raw

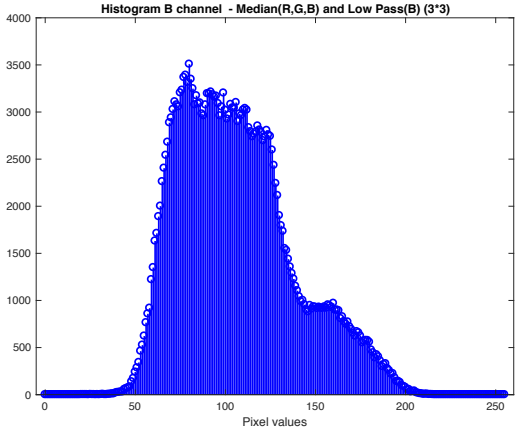
Best outputs (Highlighted in the table):

- **Median (for all R, G and B) and Gaussian (B alone) with mask size 3*3 – 26.1888 dB**

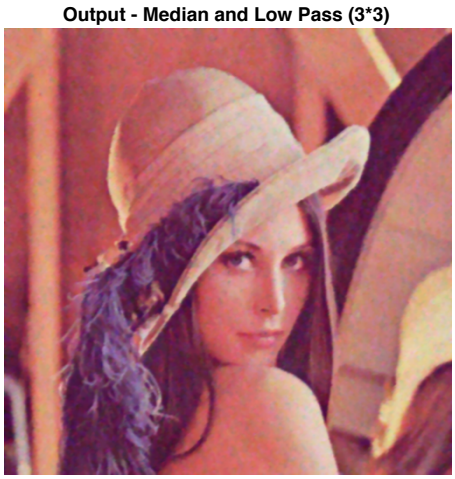
Output - Median(R,G,B) and Low Pass(B) (3*3)

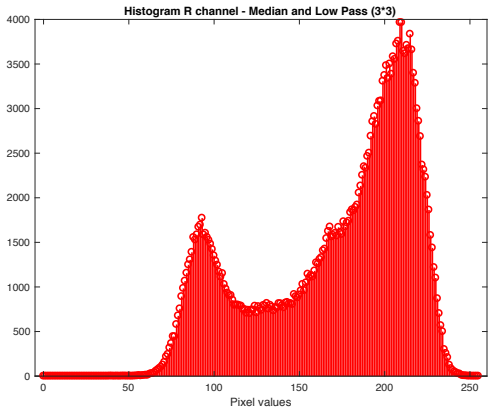


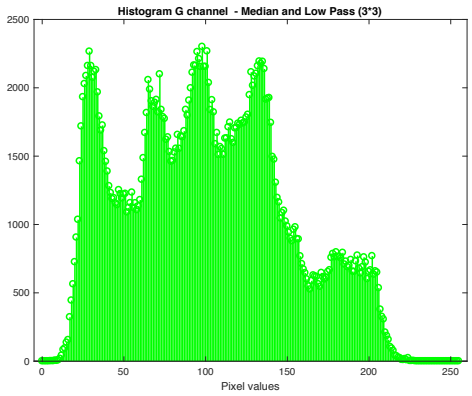


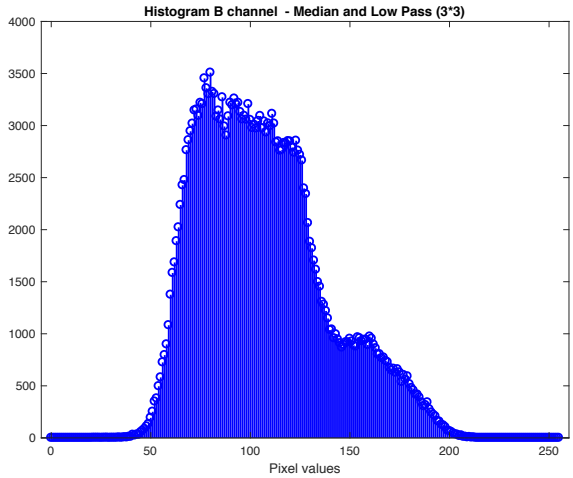


- **Median and Low Pass Filter with mask size 3*3 – 26.1966 dB**

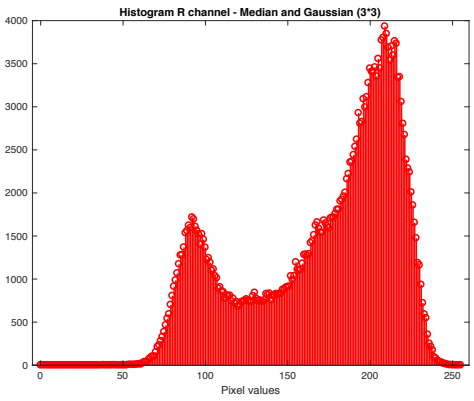


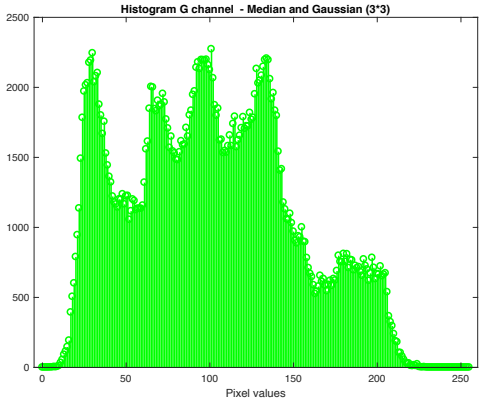


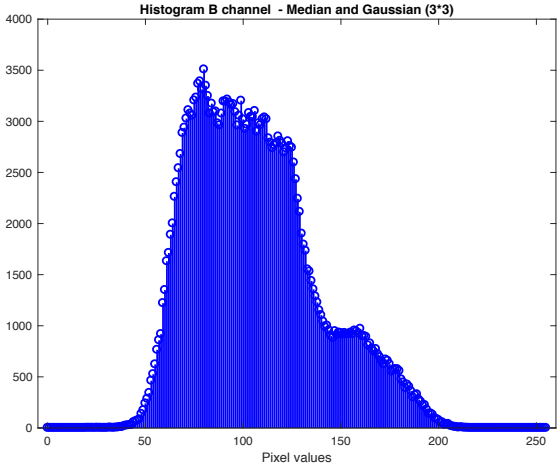




- **Median and Gaussian Filter with mask size 3*3 – 26. 2315 dB**







➢ DISCUSSION:

- Various study on Noisy Image histograms and denoising procedures were performed. Some observations and conclusions from the above experiments:
    - Not all the Channels have the same type of noise.
    - R – Channel has Salt noise and Gaussian Noise
    - G – Channel has Pepper noise and Gaussian Noise
    - B – Channel has Salt and Pepper Noise, and Gaussian Noise.
    - Salt and Pepper noise is identified by the peaks in in 255 and 0 respectively. Salt and Pepper Noise in B channel is very low when compared to the R and G channel
    - Gaussian Noise is identified by the histogram smoothed version in the Noisy image. Gaussian Noise is added with mean as the mean of each channel's pixel data.

- We have to perform filtering on the individual channels separately to see if different filters denoised better when applied on different channels

- Median Filter is used to remove the Salt and Pepper Noise whereas, Low Pass and Gaussian Filter is used to remove the Gaussian Noise. Mixed Noise can be removed by cascading median filter and Low Pass or Gaussian Filter.

- From the above best results shown I would prefer to use Median Filter first and then Gaussian Filter. Since it is always best to remove Salt and Pepper Noise first and then the Gaussian Noise. If we apply low pass filter or Gaussian filter before median filter, then the salt and pepper noise will get averaged out and will be hard to remove.

- From the above results I would prefer to use 3*3 filter instead of mask size 5 or 7 since as the size of the mask increases, the image becomes over smoothed. Hence the PSNR value decreases.

- Shortcomings of these filters-based noise removal is that the image gets over smoothened and the high frequency components like sharpness are not seen properly. Also, as we increase the mask size, the computational time becomes huge and will become difficult to remove noise

- Research is still going on this noise removal area to increase the PSNR value. We can try other filters like max-min filter, mid-point filter, alpha-trimmed mean filter. There are also many other techniques developed to remove noise like Non-Local Means, Wavelet transforms, block-matching algorithms and other statistical methods involving machine learning.

b) PRINCIPAL COMPONENT ANANLYSIS

➢ ABSTRACT AND MOTIVATION:

This problem will allow us to understand the difference between traditional filter-based noise-removal and PCA based approach. Principal Component Analysis is a dimensionality reduction algorithm. We will see how PCA will help in noise removal.

(Source: http://imagine.enpc.fr/publications/papers/BMVC11.pdf)

➢ APPROACH AND PROCEDURES:

WHY PCA FOR FILTERING NOISE DATA:

Orthogonal linear transformation is used to projection of the given data in n dimensions. These n dimensions might be of high variance. The eigen vectors and eigen values come into picture where we can rank the eigen vectors based on the eigen values. The eigen value has its significance from high variance. Also, eigen values of noisy data are affected because of the noise in the image data. Hence using the top few eigen values will influence the noise reduction.

DESCRIPTION OF PATCH-BASED LOCAL PCA:

The main outline of the Patch based PCA is that, we can first learn the orthogonal basis of noisy image. This can be obtained by performing Principal Component Analysis. The next step is obtaining the denoised patch. If we zero all the small coefficients in the noisy patch we can do this.

There are various ways to set the patches for PCA.

- Patch based Global PCA
- Patch based Local PCA
- Patch based Hierarchical PCA

We are concerned over the Patch based Local PCA. In this the axes are built dynamically rather than using traditional static based methods. A sliding window defined as Ws * Ws (square window) using which the patches are selected.

To overcome various drawbacks of PGPCA, PLPCA was proposed. Here we select patches and then subsets of patches are selected. This resulting basis is adaptive to the image and the image path region of interest. The searching window can overlap and drastically reduces the computational time. Also, redundancy is expected since the sliding window overlaps for estimating each patch. There is one more redundancy other than the sliding window, that is to be considered. Each pixel can belong to multiple patches and hence there would be various estimators for every pixel. To solve this issue, we can uniformly average the estimators for every pixel.

This method is more robust since we have the luxury to choose various parameters like patches, threshold level and searching zone width. Varying these might end up giving better results to denoise the image.

➢ EXPERIMENTAL RESULTS:

APPLYING PLPCA TO HOUSE.RAW IMAGE (Given Gaussian Noise sigma = 25):

EXP NO	hP (Half size of the patch)	Factor_thr (Factor for hard thresholding)	hW (Half Size of the Searching)	PSNR (dB)
1	7	3.00	20	35.03
2	7	2.00	20	32.72
3	**9**	**2.75**	**20**	**35.09**
4	10	2.75	15	35.01
5	10	2.75	13	34.95

PLPCA:
PSNR 35.03

PLPCA:
PSNR 32.72

PLPCA:
PSNR 35.09

PLPCA:
PSNR 35.01

PLPCA:
PSNR 34.95

Best PSNR and Optimized parameters from the above experiments:

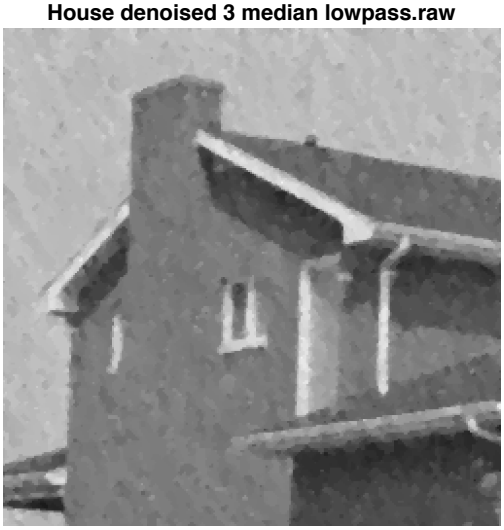
- hP (Half size of the patch) = 9
- factor_thr (Factor in front of the variance term for hard thresholding the coefficients) = 2.75
- hW (Half size of the searching zone) = 20
- Obtained PSNR = 35.09

Experiments using Best Filter Combination from Problem a – approach:

Mask	1st Filter	2nd filter	3rd filter	PSNR
-	-	-	-	20.281
3	Median	Gaussian	-	27.932
3	**Median**	**Low Pass**	**-**	**28.3243**
3	Median	Low Pass	Gaussian	26.9291

House denoised 3 median gaussian.raw

House denoised 3 median lowpass.raw



House denoised 3 median lowpass gaussian.raw

DISCUSSION:

From the above experimental results, we can observe various things:

- PLPCA approach works much better than the traditional Filter based approaches.
- This is evidently seen from the images itself. The Filter based approach, images are not sharp and is blurred. Even the noise is not removed properly.
- Also, more evidently the PSNR values of the PLPCA method is very high when compared to the Filter based approach.
- Advantages of PLPCA approach is that, the sliding window is adaptive to the region of patch interest. Also, there are many parameters associated with the PLPCA approach, when tuned properly would give better results. But the traditional Filter based approach can be varied only by mask size. As we increase the mask size the image becomes over smoothened and the PSNR value only decreases.
- Also, the redundancy issue in a way helps denoising to be done accurately, since we uniformly average each pixel's estimate.
- If the Wiener filter is used on top of this algorithm, a significant improvement is seen

c) BLOCK MATCHING AND 3-D (BM3D) TRANSFORM FILTER

➢ ABSTRACT AND MOTIVATION:

This part of the project will help us get familiar Block matching and 3-D (BM3D) algorithm. This is one famous algorithm developed by Tampere University of Technology – Department of Signal Processing. They have made the source code public and it is available for use to general public out there. We will experiment denoising House_noisy.raw image using BM3D source code downloaded from internet.
(Source: http://www.cs.tut.fi/~foi/GCF-BM3D/index.html#ref_people)

➢ APPROACH AND PROCEDURES:

BM3D ALGORITHM:

- This image enhancement algorithm relies on transforming similar image groups of 2D to a 3D array
- The obtained 3D array is transformed by 3D transformation
- The transformed 3D array spectrum is shrunk
- Inverse transform is performed to get back the 3D array in its original form
- After this process of filtering, the pixels from these blocks are returned to their original position.
- Similar to PLPCA algorithm, since the blocks can overlap, for each pixel we might get many estimates.
- Aggregating, meaning averaging is done to get maximum information for all the estimates for each pixel

➢ EXPERIMENTAL RESULTS:

The BM3D code from online was run on *house_noisy.raw* image by varying the parameters.

λ - 3D	β	N Step	N2	β-Weiner	N Step Weiner	PSNR without step 2	PSNR with Step 2
2.7	8	3	16	8	3	22.032	22.017
2.5	8	3	16	8	3	22.330	22.445
2.5	**4**	**2**	**64**	**4**	**2**	**22.566**	**22.549**

The First image is the Noisy image
Second image is the denoised image using BM3D (PSNR: 22.549dB)

➢ DISCUSSION:

WHY BLOCK MATCHING:

When image is fragmented and grouped together, the repetitive patterns in the image are identified. Also, the image groups don't have to be disjoint. Hence redundancy is allowed. This in turn increased the performance of the filtering since we are able to exploit the entire information available to us.

BM3D – SPATIAL AND FREQUENCY DOMAIN FILTER:

BM3D can be considered as a spatial domain filter since the blocks of images are grouped to form a 3D array. This exploits the information of the spatial information of every pixel from

the surrounding pixels. Also since Weiner filter is used to increase the performace of the algorithm, this can also be considered as frequency domain filter.

PERFORMANCE COMPARISON:

The above experimental results show that the house_noisy.raw image is not denoised well when compared to the other PLPCA and traditional filter-based methods. Though the BM3D algorithm can be considered as one of the best methods to denoise an image, it doesn't work well with the given house image. One reason for this result might be that the house image doesn't have repetitive patterns to be grouped and to be considered as the region of interest. I hope the sophisticated method of denoising using BM3D will work well in an image with repeating patterns.

The entire problem – c, makes us come to a conclusion that each algorithm is suited for particular type of image. Hence denoising an image is an art and it is performed by suitable judging which method will work better for what type of an image.

APPENDIX

(DESCRIPTION OF FUNCTIONS USED FROM DIP MyHeaderFile.h)

fileRead() function:

- Reads the given filename to a 1D array

allocMemory2D() function

- Allocate memory for a 2D array with the given row and column size and initialize to zero using 2 nested for loops
- Returns image2D

allocMemory3D() function

- Allocate memory for a 3D array with the given row and column size and initialize to zero using 3 nested for loops
- Returns image3D

Image1Dto2D() function

- Converts given 1D image to the output 2D image (grayscale – single channel)
- Returns image2D

Image1Dto3D() function

- Converts given 1D image to the output 3D image (reads RGBRGBRGBRGB..)
- Returns image3D

Image2Dto1D() function

- Converts given 2D image to the output 1D array

Image3Dto1D() function

- Converts given 3D image to the output 1D array

seperateChannels() function

- Separates the given 3D image to 2D array based on the index 0 – Red, 1 – Green , 2 – Blue
- Return the 2D image

combineChannels() function

- Combines given three 2D image arrays to 3D array
- Returns image3D

fileWrite() function

- Write the given 1D array of unsigned char to the given filename

fileWriteHist() function

- Write the given 1D array of unsigned integers to the given filename

freeMemory2D () function

- Free memory of every allocated array in the 2D array using delete []

freeMemory3D () function

- Free memory of every allocated array in the 3D array using delete []

Histogram2DImage () function

- Traverse through the given 2D array using 2 nested for loops
    - Get the pixel intensity value at the location
    - Increment the histogram array based on the pixel obtained