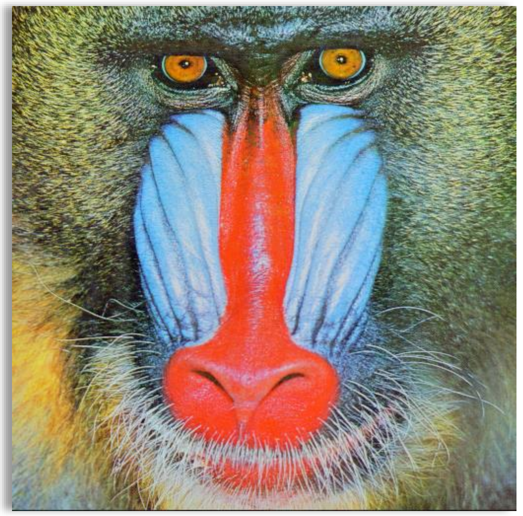
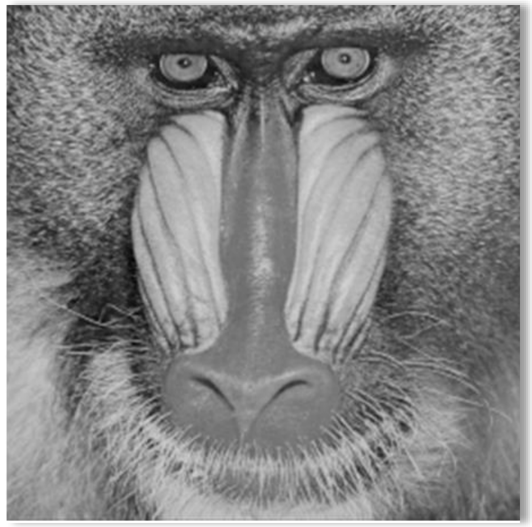
Problem 1 – Simple Image Manipulation

a) Color-to-Grayscale Conversion


Mandril.raw


Mandril_bw.raw

The objective of the problem is to convert a color image to a grayscale image. The most popular format to store a color image is to use three channels viz. Red, Green and Blue (RGB). A grayscale image is defined using one channel that represents quantized levels of intensity values, usually 256 levels.

The problem of color to grayscale conversion is an interesting topic in the field of image processing as there are many benefits of doing that.

Some of the reasons for conversion are:

1. Image processing is best done on single channel images. It reduces complexity without losing essential information.
2. The grayscale image captures the luminance Y in an image; studies show that human vision is more perceptible to luminance information than chrominance information.
3. It reduces the size of the images which makes it suitable for fast processing.
4. The reduction in size also helps in network based operations where a grayscale can be transferred more quickly to upload it on a server or sending it to a peer.
5. Feature extraction is relatively easier on grayscale images.
6. They at times represent a nice effect on film or images.

Modern descriptor-based image recognition systems often operate on grayscale images. Most researchers assume that the color-to-grayscale method is of little consequence when using robust descriptors. However, since many methods for converting to grayscale have been employed in computer vision, it is prudent to assess whether this assumption is warranted. There are quite a lot of methods in Image Processing literature for conversion of a color image to a grayscale image. The most common techniques are based on weighted means of the red, green, and blue image channels (e.g., *Intensity* and *Luminance*), but some methods adopt alternative strategies to generate a more perceptually accurate representation (e.g., *Luma* and *Lightness*[1]) or to preserve subjectively appealing color contrast information in grayscale images (e.g. *Decolorize* [2]). The main reason why grayscale representations are often used for extracting descriptors instead of operating on color images directly is that grayscale simplifies the algorithm and reduces computational requirements. Indeed, color may be of limited benefit in many applications and introducing unnecessary information could increase the amount of training data required to achieve good performance [3].

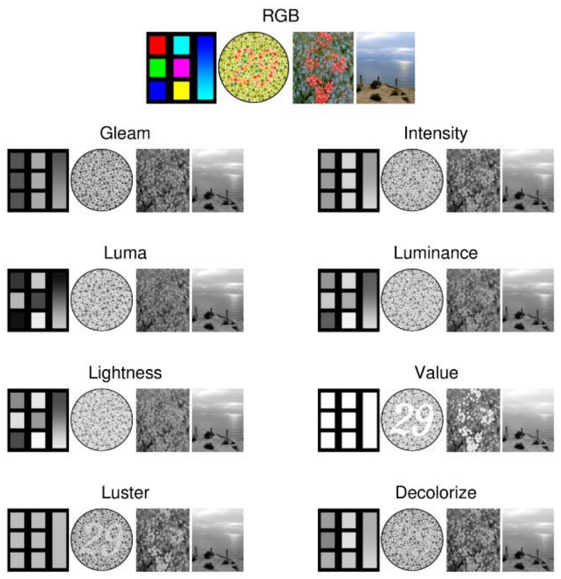
Here in this problem we use a method based on weighted Luminosity. The method is called Luma and inspired from the Luminance. Luminance [4] is designed to match human brightness perception by using a weighted combination of the RGB channels:

$$\mathcal{G}_{Luminance} \leftarrow 0.3R + 0.59G + 0.11B.$$

Luminance does not try to match the logarithmic nature of human brightness perception, but this is achieved to an extent with subsequent gamma correction. Luminance is the standard algorithm used by image processing software (e.g., GIMP). It is implemented by MATLAB's “rgb2gray” function, and it is frequently used in computer vision (e.g. [5]). Luma is a similar gamma corrected form used in high-definition televisions (HDTVs) [1]:

$$\mathcal{G}_{Luma} \leftarrow 0.2126R' + 0.7152G' + 0.0722B'.$$

Emulating the way humans perceive certain colors as brighter than others appears to be of limited benefit for grayscale image recognition. However, methods that incorporate a form of gamma correction (e.g., Lightness, Gleam, Luma, Luster, etc.) usually perform better than purely linear methods such as Intensity and Luminance.



The code written has class “Image” in header file Image.h that defines Image Input and Output. I’ve designed data structures to read an image into a dynamically allocated single dimensional array. The Image class has procedures to read and write images to file and get and set value of an image pixel.

I use the Image class in another header file ImageProc.h that handles all the image processing methods. I have written a method ImageProc::color_to_bw() to handle the conversion technique.

The color_to_bw() function reads in a color image and outputs a grayscale image. The image to be read has to be encoded as RGB values with 3 channels and the output file generated has corresponding gray value in 1 channel.

Code snippet:

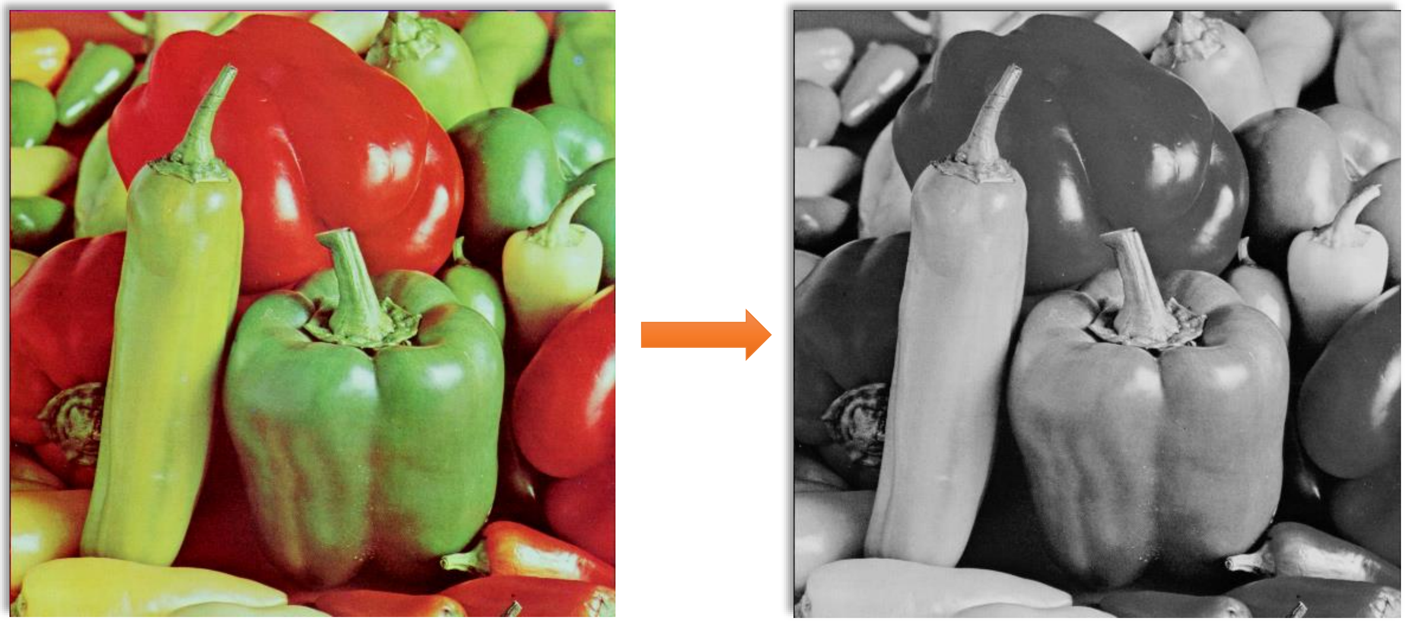
```
Image img_color("color", width, height); //object for input color image
Image img_bw("bw", width, height);       //object for output b-&-w image

img_color.read_image(infile, img_color.cols, img_color.rows); //reading input image from
file

for(int i = 0; i < img_bw.rows; i++)
       for(int j = 0; j < img_bw.cols; j++)
              for(int k = 0; k < img_bw.channels; k++)
                     img_bw.setvalue(i,j,k,((img_color.getvalue(i,j,0)*0.21)\
                                           +(img_color.getvalue(i,j,1)*0.72)\
                                           +(img_color.getvalue(i,j,2)*0.07))); //using
Luminosity formula: grayscale_value = 0.21*R + 0.72*G + 0.07*B

img_bw.write_image(outfile, img_bw.cols, img_bw.rows); //writing image to output file
```

I use the following form of conversion as stated in the homework document to convert the “rgb” channel values to get the grayscale value.

$$\mathcal{G}_{Luma} \leftarrow 0.2126R' + 0.7152G' + 0.0722B'.$$

Following are the results of the above implementation on the image Pepper.raw

```
./Main.exe -prob color_to_bw -i ../images/Pepper.raw -o ../images/Pepper_bw.raw -w 512 -h 512
```



Pepper.raw

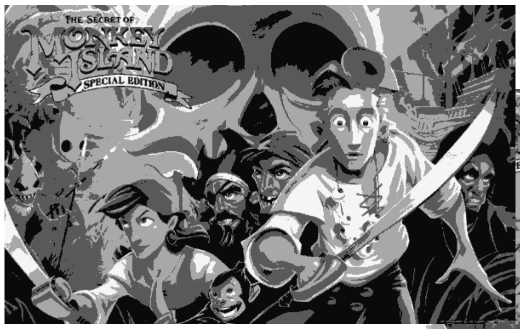
Pepper_bw.raw

In conclusion, it can be stated that conversion techniques for color images to grayscale images play an important role in image processing as well as in computer vision. It can be observed from the above result that the perception of the image doesn't change much even after the conversion. The inference and semantics we can gather from a grayscale image is as good as that from a color image.

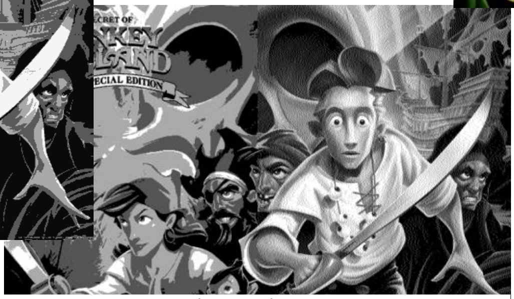
Another interesting thing about grayscale images is the levels that we define for these images. The above images have been converted to grayscale images with 256 levels. As the gray levels decrease, we start observing averaging and gradually the image tends to maintain only high level structure and no texture detail.

THE SECRET OF
MONKEY ISLAND
SPECIAL EDITION

Original

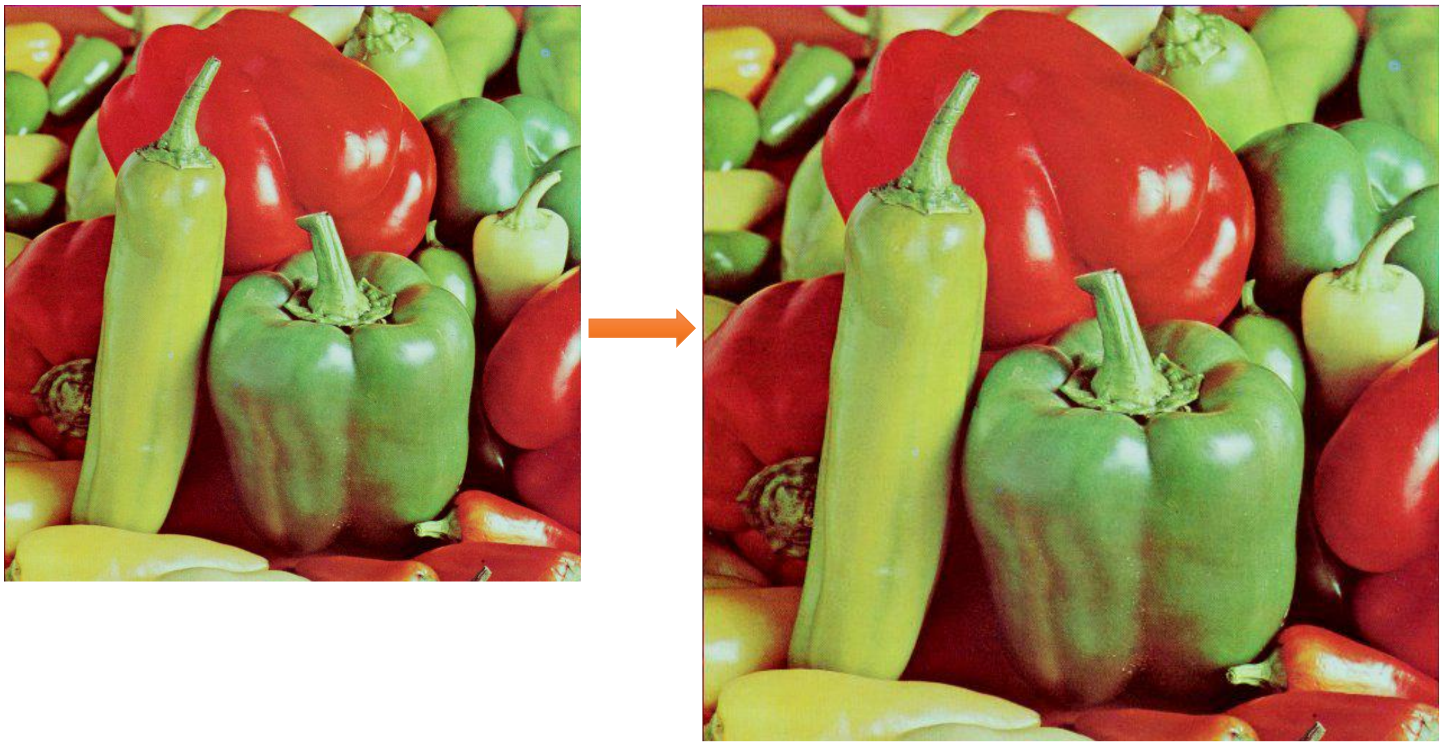


Gray Levels < 256



With Dither

b) Image Resizing via Bilinear Interpolation



The objective in this problem is to resize the image. An image is defined by the number of pixels each row and the number of rows. When resizing an image, we increase the number of pixels without changing or extending the image. In other words, we change the resolution of the images.

Image resizing is the most used application in image processing and we can see it happen everywhere. The pinch and zoom function on a touchscreen is a process of image resizing and is a method used so often. High resolution cameras sample the image data for individual pixels and carry more information and detail. Most of the times, such a high resolution is unnecessary and we might want to resize them down to a lower resolution.

Benefits of resizing an image

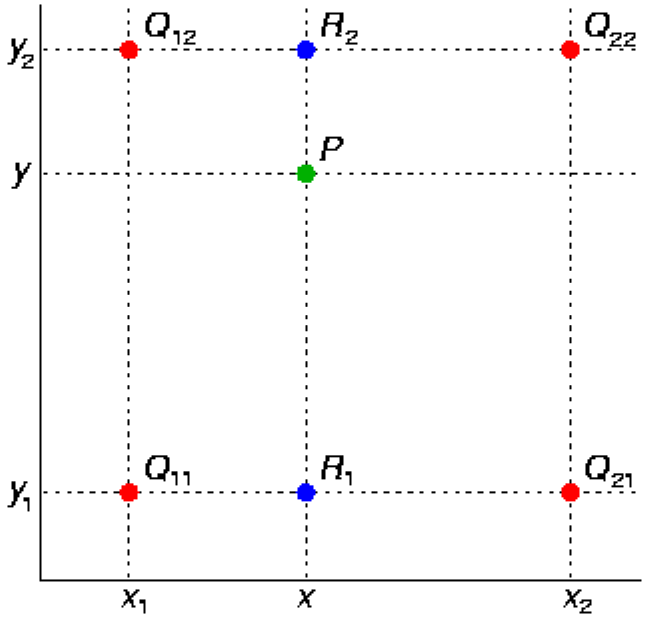
1) Reduce upload time over a network by making them smaller, so they carry less information and take less space on storage.
2) Over the years since the first digital image was created, there has been a wide change in aspect ratio for cinema and standard photo sizes. Resizing of images is important to keep up with the standard. One of the popular techniques used for that is Seam-Carving.

3) At times sensor pixel sizes are not true to the world image and corrective measures of resizing the image needs to be taken for restoration.
4) In Computer Vision, to do context aware feature matching, we might want to start out with a lower resolution image and progressively increase the size to analyze further or cut the procedure to save time.

There are a few context aware image resizing algorithms like Seam Carving that are good to change the aspect ratio of an image. It acts as a non-linear transformation kernel. The additional points in the resulting image that don't map directly to any point in the actual image are interpolated using the intensity values of its neighbors.

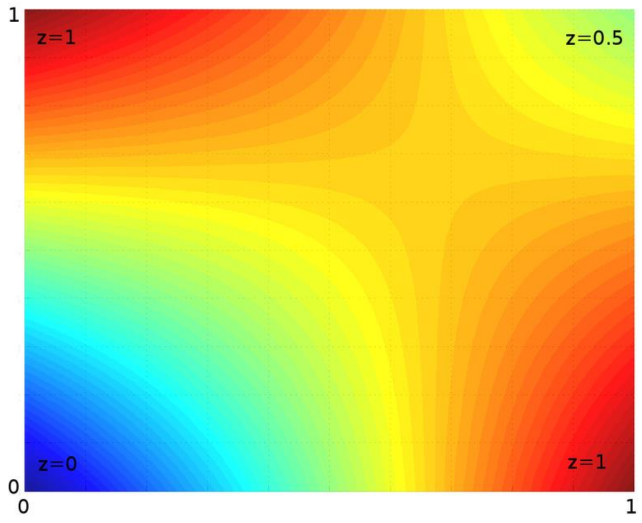
Weighted interpolation is an important technique in resizing an image. It resolves the issue of not having one-by-one mapping between pixels of source image and resulting image. The lack of mapping is cause of the discrete nature of the medium for image capture. Interpolation helps fills these gaps in a discrete system.

In this problem, we use one of the popular techniques of interpolation in 2D space – Bilinear Interpolation. It is a basically a weighted sum of the pixel intensities of 4 pixels around a given point in the source image that maps to discrete location in the resulting resized image.



Bilinear Interpolation Model

Red dots represent actual data points and the green dot is the interpolated point(source: Wikipedia)



The formulation of Bilinear Interpolation is straight-forward and can be stated as follows:

If the value of a location (x, y) has to be found using interpolation, we find its surrounding 4 points. We do that by evaluating X= floor(x) and Y= floor(y). (X, Y) are the pixel co-ordinates of one of the pixels surrounding location (x, y). The other points that surround (x, y) would be (X, Y+1), (X+1, Y) and (X+1, Y+1). We compute the distance of (x, y) from (X, Y) on x-axis and store it as weight 'a'; like-wise for y-axis as weight 'b'. The interpolated intensity value is given by:

$$\begin{aligned} I(x,y) = {} & I(X, Y)*(1.0-a)*(1.0-b) \\ & + I(X, Y+1)*(a)*(1.0-b) \\ & + I(X+1, Y)*(1.0-a)*(b) \\ & + I(X+1, Y+1)*(a)*(b) \end{aligned}$$

Bilinear interpolation is good for changing the size of an image, but causes undesirable softening of details and can be somewhat jagged.

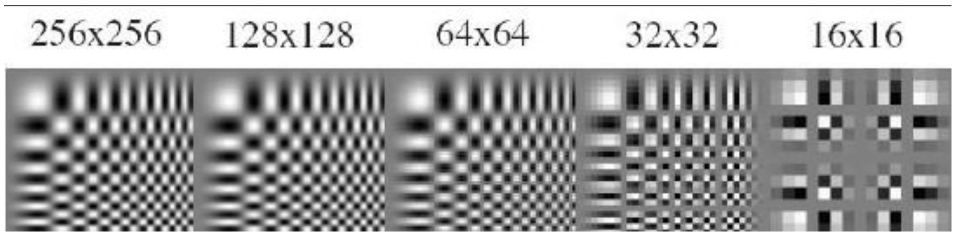
Some of the other methods used for Image Scaling are:

1) Nearest Neighbor Interpolation which assigns the intensity value of the nearest pixel
2) Bicubic Interpolation
3) Lanczos Resampling
4) Hqx
5) Super Sampling
6) Vectorization, which is one of the efficient ways to do scaling as it is resolution independent
7) Mipmap

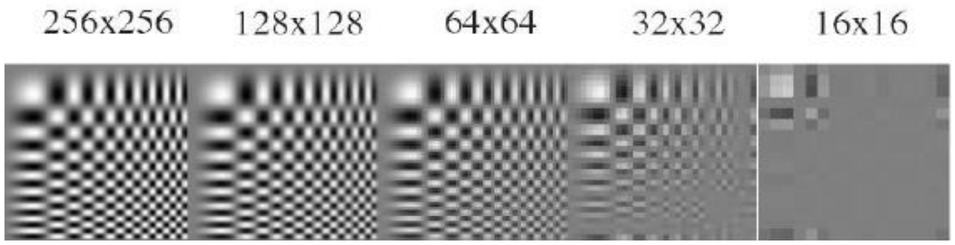
One of the effects of image downscaling that is often discussed is **aliasing**. The effect is caused because of having higher frequency bandwidth than the sampling rate, which in this case is the resolution. Samples of the effect are shown on the next page. This an important problem to tackle as it spoils the viewing experience.

The problem of aliasing can be solved by folding the frequency bandwidth and getting rid of the high frequencies in the data. The reduction in sampling rate is now compensated for, by lowering the max frequency value in the image. It is imperative that we adhere to Nyquist-Shannon theorem that defines the required sampling rate based on the highest frequency value to faithfully reproduce the analog signal.

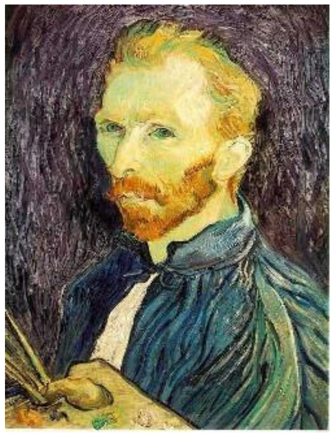
Aliasing

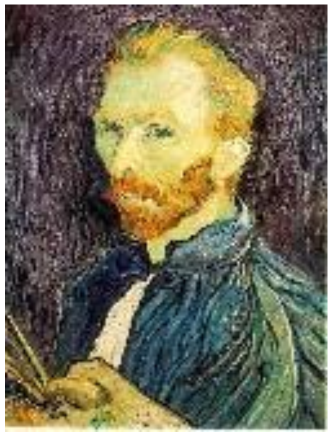


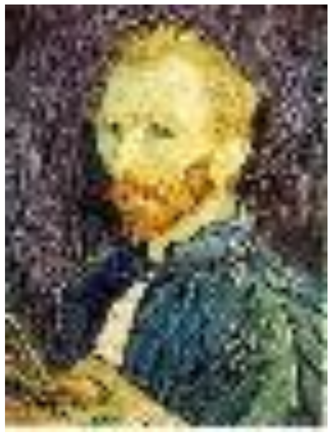
Anti-aliasing



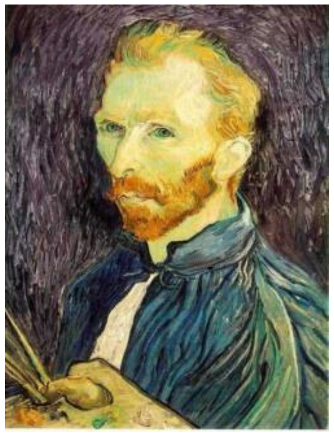
Subsampling without pre-filtering[6]

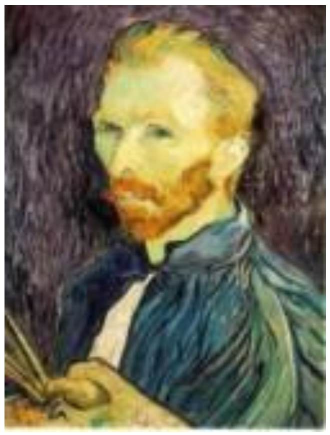

1 / 2

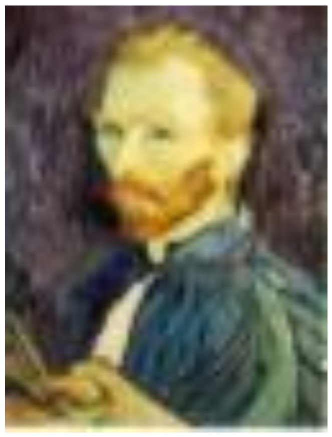

1 / 4 (2x zoom)


1 / 8 (4x zoom)

Subsampling with pre-filtering using Gaussian filter


G + 1 / 2


G + 1 / 4 (2x zoom)


G + 1 / 8 (4x zoom)

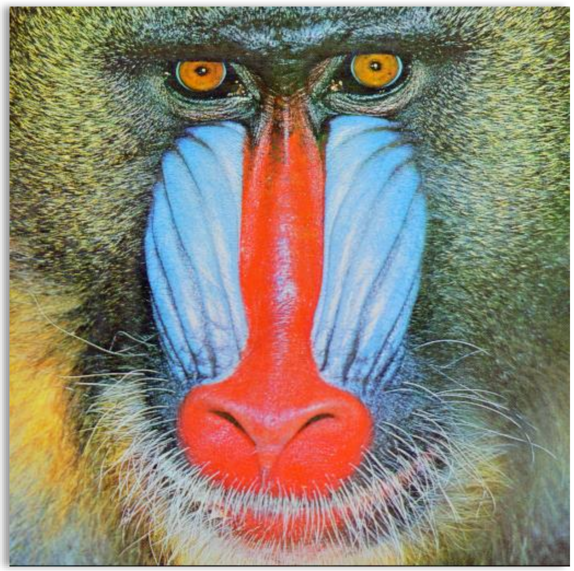
The code logic for image resizing can be encapsulated in the following pseudo-code:

Code snippet:

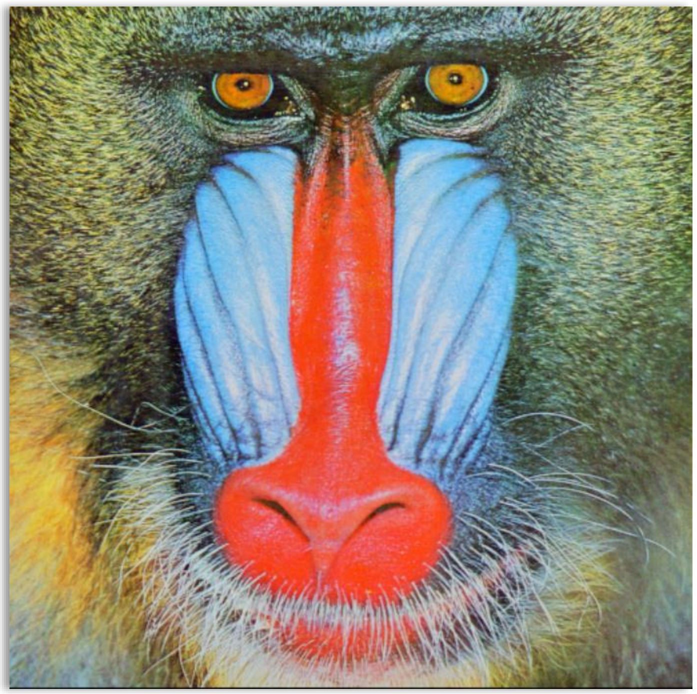
```
for(int i = 0; i < img_resized.rows; i++)
       for(int j = 0; j < img_resized.cols; j++)
              for(int k = 0; k < img_resized.channels; k++)
              img_resized.setvalue(i,j,k,
                                        ((img_orig.getvalue(X  ,Y  ,k)*(1.0-a)*(1.0-b))
                                        +(img_orig.getvalue(X  ,Y+1,k)*(    a)*(1.0-b))
                                        +(img_orig.getvalue(X+1,Y  ,k)*(1.0-a)*(    b))
                                        +(img_orig.getvalue(X+1,Y+1,k)*(    a)*(    b))
                                                          //using 4 surrounding co-ordinates
                                                          //BILINEAR INTERPOLATION
```

Following are the results of the implementation above on the image “mandril.raw”

```
./Main.exe -prob color_to_bw -i ../images/mandril.raw -o ../images/mandril_resized.raw -w 512 -h
512 -tw 700 -th 700
```



mandril.raw (512x512)



mandril_resized.raw (700x700)

Bilinear Interpolation is a fast technique for resizing and executes in linear time. The reproduction of image that is resized is very accurate and faithful.

Problem 2 – Histogram Equalization and Image Filtering

a) Histogram Equalization

The most important factor in capturing images is the light on the subject. Good lighting conditions are necessary for a well exposed image. Human vision too is more perceptible to luminance than chrominance in an image. But most of the times, it is not possible to have the ideal lighting conditions and the image need to processed post-capture.

Image Enhancement is a step in post-processing of an image and contrast adjustment is a key part of this step. Contrast is the difference in luminance and/or color that makes an object (or its representation in an image or display) distinguishable. In visual perception of the real world, contrast is determined by the difference in the color and brightness of the object and other objects within the same field of view. Because the human visual system is more sensitive to contrast than absolute luminance, we can perceive the world similarly regardless of the huge changes in illumination over the day or from place to place. The maximum contrast of an image is the contrast ratio or dynamic range.

A way to correct contrast levels in an image and control the luminance in it is to equalize Histogram levels of the image. Histograms are a quantification of the pixels and their pixel intensities. For a single channel in an image with 256 gray levels for each channel, the histogram is the distribution of number of pixels in each gray level. The requirement for contrast rectification is to have a uniform distribution for these gray levels in each channel.

Histogram Equalization

The basic idea of Histogram Equalization is to go from a random variable distribution of intensity values in an image to a uniform distribution. The process flattens this distribution and spreads it across the spectrum. There are a lot of variants of Histogram Equalization method out in the Image Processing literature and each one of those has its own benefit. None of the equalization technique is a one stop solution for all images, but are ad-hoc strategies to handle various shortcomings of the basic histogram equalization.

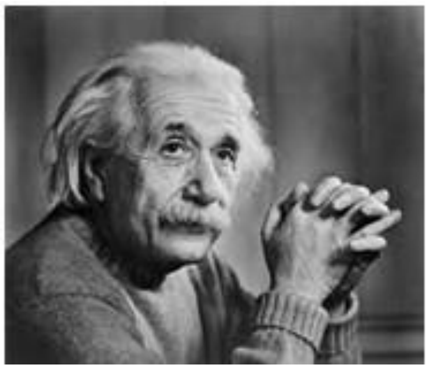
To increase the contrast ratio, we can also just stretch the histogram of the image, in which case we'd be up-scaling the histogram. In histogram stretching, we'd essentially be mapping original histogram distribution to a wider distribution on the spectrum. But the results generated by this method is unsatisfactory.

An effective way of Histogram Equalization is histogram flattening. We can do this leveling of distribution using the two methods that we implement for this problem.

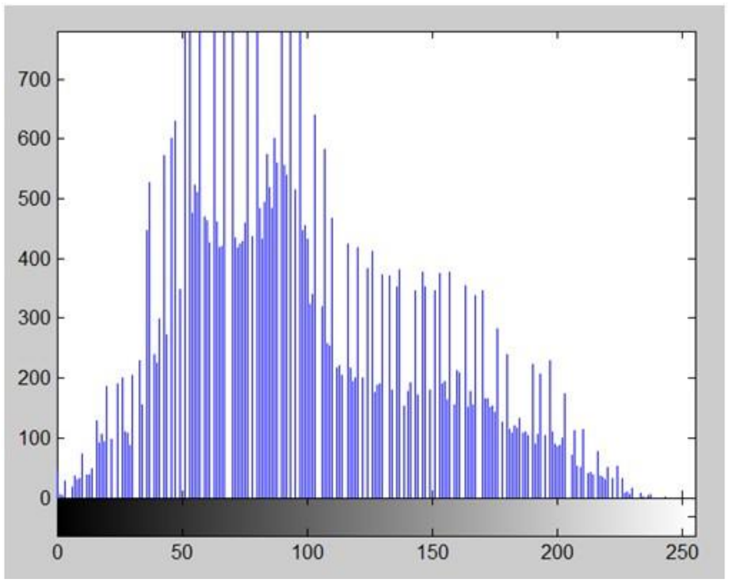
Method one: Using cumulative distributive function (cdf)

We first create histogram bins equal to the number of gray levels in the image (ex. 256). We assign these histogram bins the total number of pixels in the image that have the corresponding gray value.

```
for(int i = 0; i < img_color.rows; i++)
        for(int j = 0; j < img_color.cols; j++)
                for(int k = 0; k < img_color.channels; k++)
                        if((img_color.getvalue(i,j,k) >= 0) && (img_color.getvalue(i,j,k) < 256))
                                histogram_bins[img_color.getvalue(i,j,k)][k]++;
```



Image

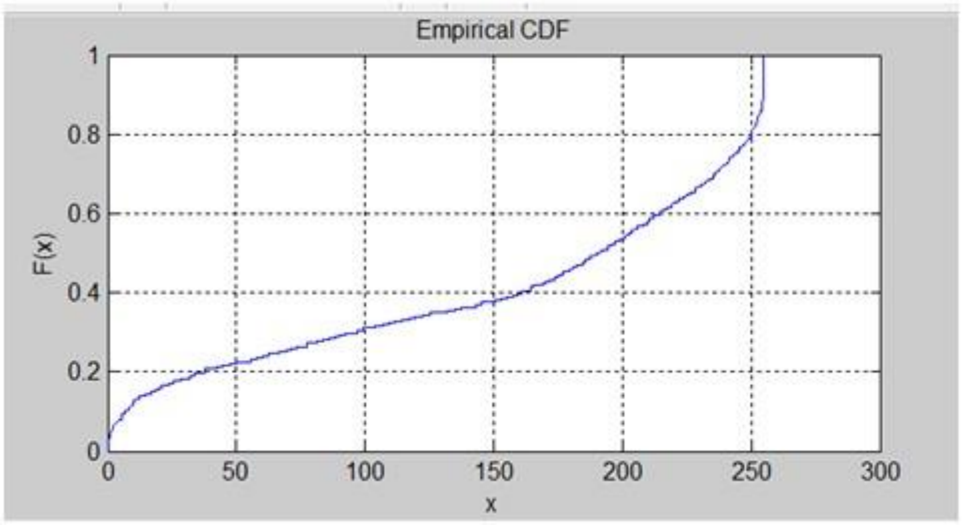


Histogram Distribution

Each gray value is a bin index and the number of pixels of that gray value is the value assigned to that bin.

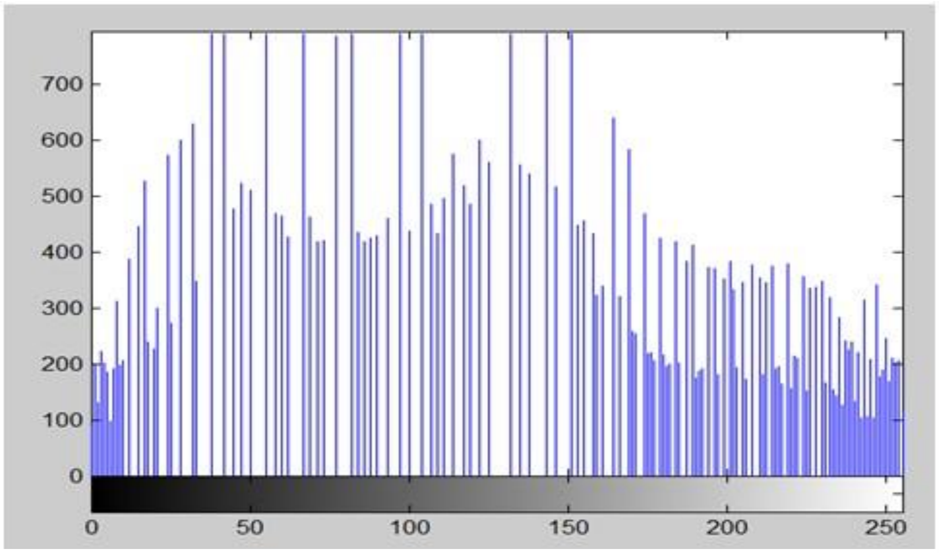
Next, we compute the Cumulative Distributive Function of the Histogram Distribution. It is computed by integrating the histogram distribution. The cumulative distributive function is also an array of total number of gray levels. Each element in this array stores the summation of number of pixels in all histogram bins until the corresponding histogram bin.

```
for(int i = 1; i < 256; i++)
        for(int k = 0; k < 3; k++)
                hist_cumulative[i][k] = hist_cumulative[i-1][k] + histogram_bins[i][k];
```



Cumulative Distributive Function

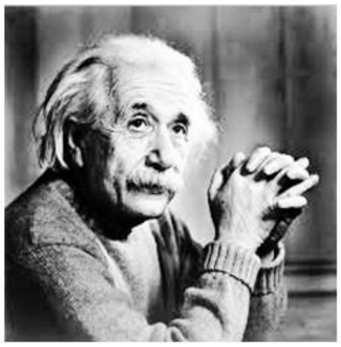
To find the corresponding gray value in histogram equalized image for pixels of a certain gray value, we divide its value in cumulative distributive function by total number of pixels and scale it with total number of gray levels. The floor of this resulting number is the mapped gray value from original image to equalized image.



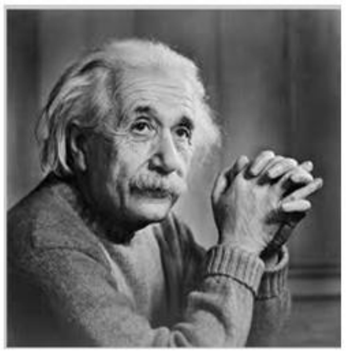
Equalized Histogram

The pixels are getting mapped on the basis their gray value in original image to a different gray value that equalizes the histogram and enhances contrast. A sample result is shown below of an Equalized Histogram.

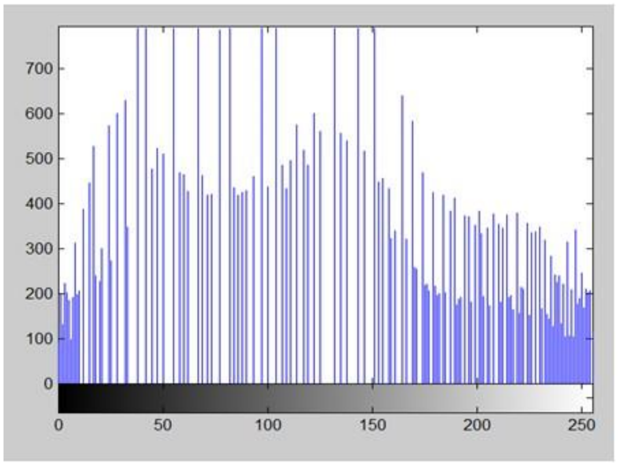
New Image



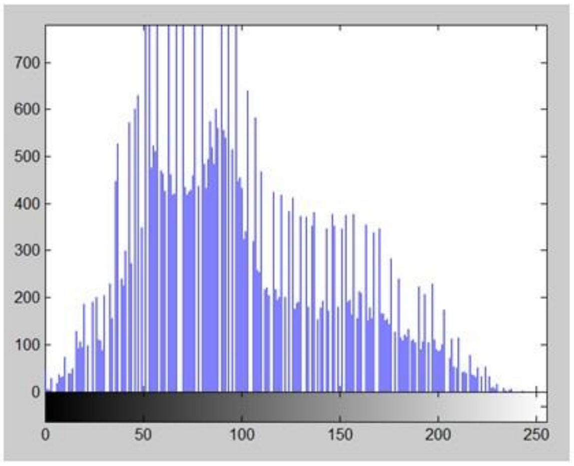
Old image



New Histogram



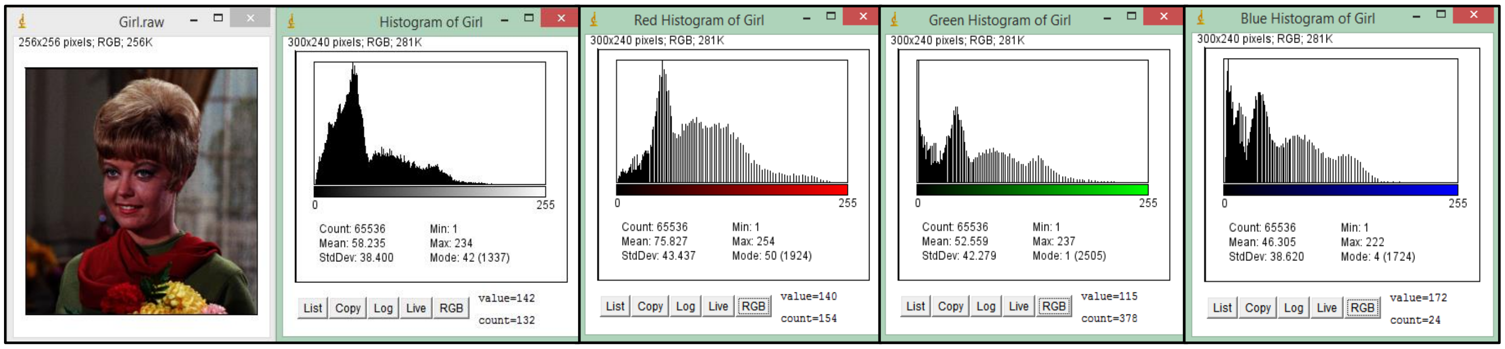
Old Histogram



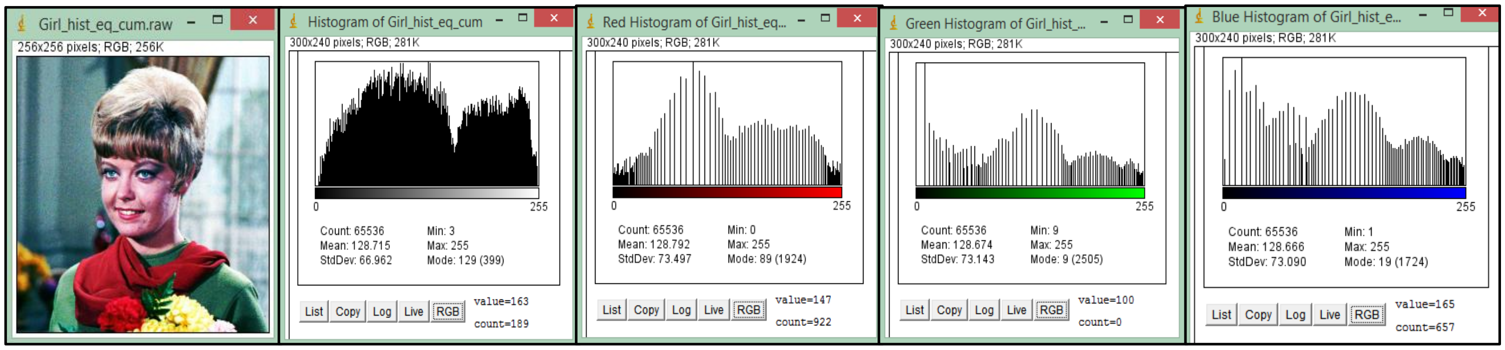
Pseudo – code for equalization

```
for(int i = 0; i < height; i++)
       for(int j = 0; j < width; j++)
              for(int k = 0; k < 3; k++)
              if(img_color.getvalue(i,j,k) < 0  ) img_color.setvalue(i,j,k,0)  ;
              else if (img_color.getvalue(i,j,k) > 255) img_color.setvalue(i,j,k,255);
              else
img_color.setvalue(i,j,k,(255*hist_cumulative[img_color.getvalue(i,j,k)][k]/hist_cumulative[25
5][k]));
```

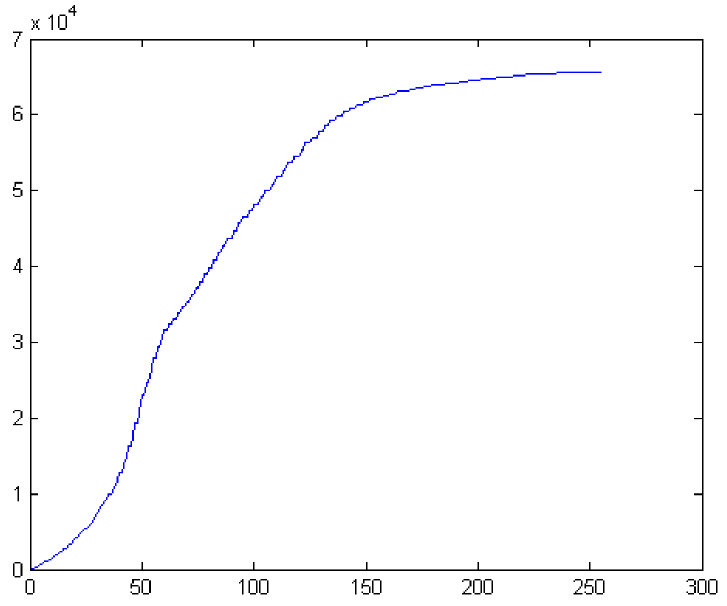
The following results of converting Girl.raw using the above method:



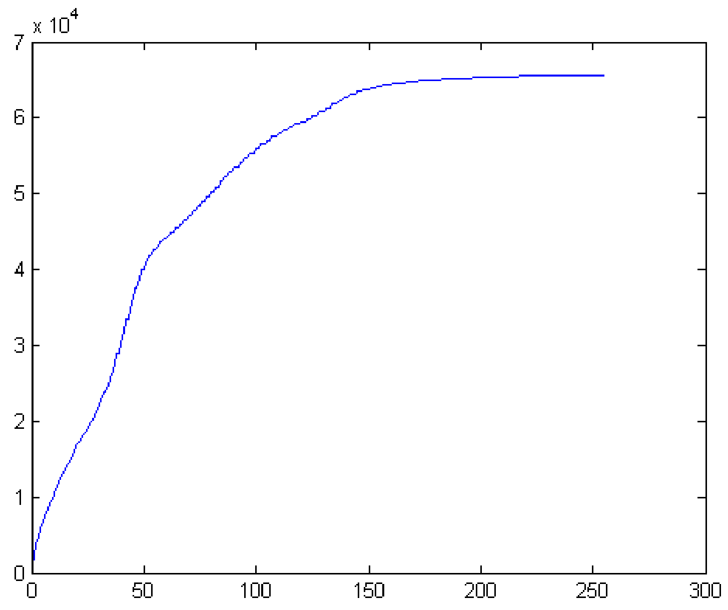
```
Girl.raw
256x256 pixels; RGB; 256K

Histogram of Girl
300x240 pixels; RGB; 281K
Count: 65536    Min: 1
Mean: 58.235    Max: 234
StdDev: 38.400  Mode: 42 (1337)
value=142
count=132

Red Histogram of Girl
300x240 pixels; RGB; 281K
Count: 65536    Min: 1
Mean: 75.827    Max: 254
StdDev: 43.437  Mode: 50 (1924)
value=140
count=154

Green Histogram of Girl
300x240 pixels; RGB; 281K
Count: 65536    Min: 1
Mean: 52.559    Max: 237
StdDev: 42.279  Mode: 1 (2505)
value=115
count=378

Blue Histogram of Girl
300x240 pixels; RGB; 281K
Count: 65536    Min: 1
Mean: 46.305    Max: 222
StdDev: 38.620  Mode: 4 (1724)
value=172
count=24
```



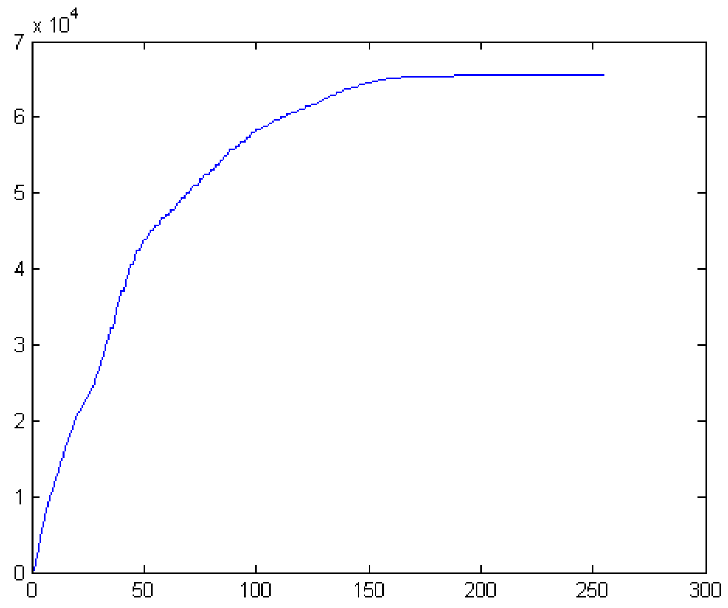
```
Girl_hist_eq_cum.raw
256x256 pixels; RGB; 256K

Histogram of Girl_hist_eq_cum
300x240 pixels; RGB; 281K
Count: 65536    Min: 3
Mean: 128.715   Max: 255
StdDev: 66.962  Mode: 129 (399)
value=163
count=189

Red Histogram of Girl_hist_eq...
300x240 pixels; RGB; 281K
Count: 65536    Min: 0
Mean: 128.792   Max: 255
StdDev: 73.497  Mode: 89 (1924)
value=147
count=922

Green Histogram of Girl_hist_...
300x240 pixels; RGB; 281K
Count: 65536    Min: 9
Mean: 128.674   Max: 255
StdDev: 73.143  Mode: 9 (2505)
value=100
count=0

Blue Histogram of Girl_hist_e...
300x240 pixels; RGB; 281K
Count: 65536    Min: 1
Mean: 128.666   Max: 255
StdDev: 73.090  Mode: 19 (1724)
value=165
count=657
```



Transfer function for red channel



Transfer function for green channel



Transfer function for blue channel

Method 2: Direct Mapping and equal division

In this method, we cascade all histogram bins to form a single long array and then divide that array into number of gray levels. Here we map pixels to gray values and it is not done on the basis of their original gray values. Pixels belonging to one histogram bin may map to multiple histogram bin in the equalized image. This method ensures that we have equal number of pixels in each gray level and hence the resulting histogram bins are equalized. The cumulative distributive function of the resulting histogram of the image is a linear function.

Following is the algorithm I used to accomplish this task. The following procedure reduces the memory and space and time to keep record of pixels and instead uses only their index in the cumulative histogram to assign them their correct value.

Similar to the earlier method, we create histogram bins from the image.

```
for(int i = 0; i < img_color.rows; i++)
	for(int j = 0; j < img_color.cols; j++)
		for(int k = 0; k < img_color.channels; k++)
			if((img_color(i, j, k) >= 0) && (img_color(i, j, k) < 256))
					histogram_bins[img_color(i, j, k)][k]++;
```

We again calculate the cumulative distributive function and

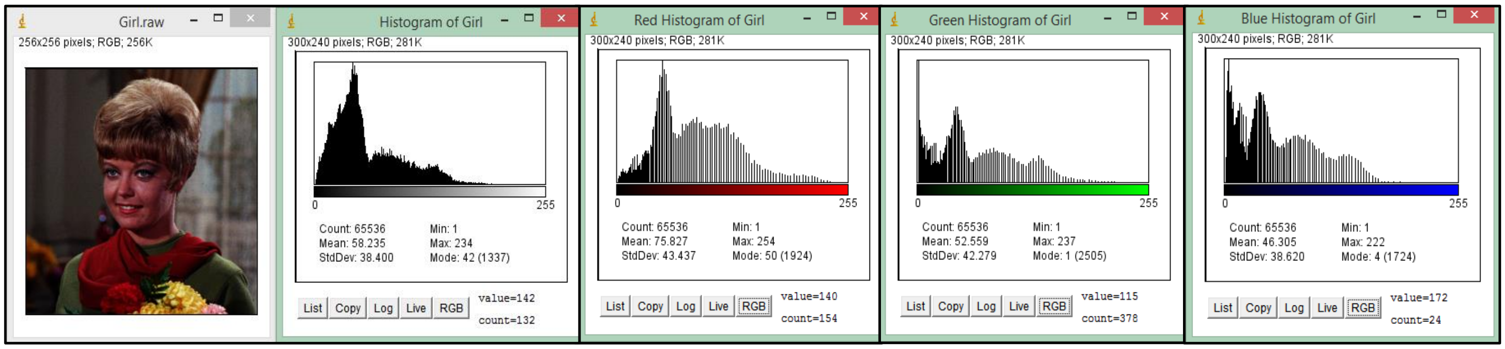
```
for(int k = 0; k < 3; k++)
	hist_cumulative[0][k] = histogram_bins[0][k];
	for(int i = 1; i < 256; i++)
		for(int k = 0; k < 3; k++)
		hist_cumulative[i][k] = hist_cumulative[i-1][k] + histogram_bins[i][k];
```

Now we make an empty copy of hist_cumulative[][], so we could find the index and position of any pixel in the original cumulative histogram and map it accordingly in the new the equalized histogram.

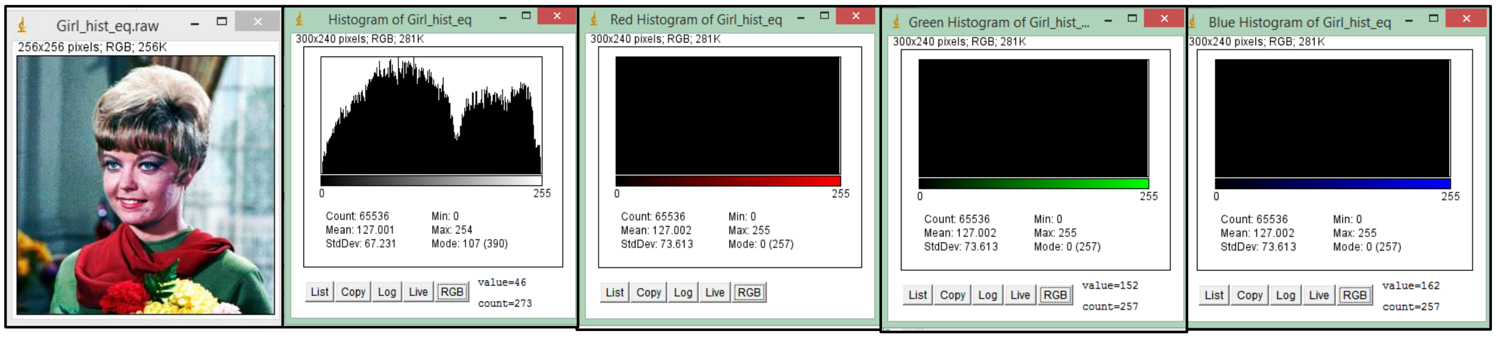
Following procedure would perform the aforementioned task

```
for(int i = 0; i < img_color.rows; i++)
for(int j = 0; j < img_color.cols; j++)
for(int k = 0; k < 3; k++)
	if      (img_color.getvalue(i,j,k) < 0  ) img_color.setvalue(i,j,k,0);
	else if (img_color.getvalue(i,j,k) > 255) img_color.setvalue(i,j,k,255);
	else
	{
	++hist_cumulative_copy[img_color(i, j, k)][k];
	curr_bin_count = hist_cumulative_copy[img_color(i, j, k)][k];
	prev_bin_count = hist_cumulative[img_color(i, j, k) - 1][k];
	value = floor(255.0 * (prev_bin_count + curr_bin_count)/ hist_cumulative[255][k]);
	img_color.setvalue(i,j,k,value);
	value = 0; curr_bin_count = 0; prev_bin_count = 0;
	}
```

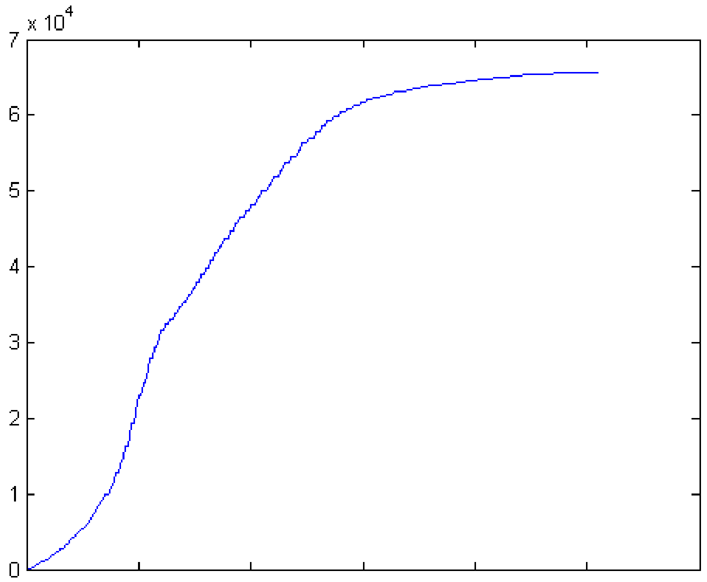
Following are the results of performing the above procedure on “Girl.raw”



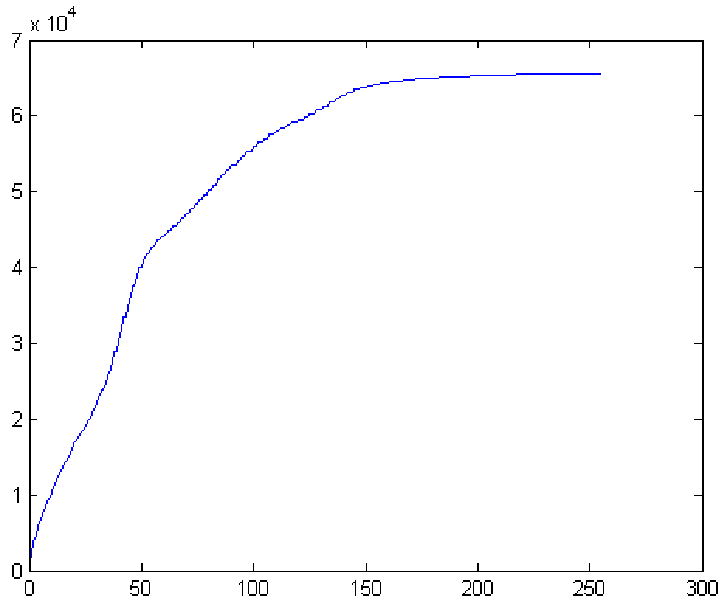
```
Girl.raw
256x256 pixels; RGB; 256K

Histogram of Girl
300x240 pixels; RGB; 281K
Count: 65536      Min: 1
Mean: 58.235      Max: 234
StdDev: 38.400    Mode: 42 (1337)
value=142
count=132

Red Histogram of Girl
300x240 pixels; RGB; 281K
Count: 65536      Min: 1
Mean: 75.827      Max: 254
StdDev: 43.437    Mode: 50 (1924)
value=140
count=154

Green Histogram of Girl
300x240 pixels; RGB; 281K
Count: 65536      Min: 1
Mean: 52.559      Max: 237
StdDev: 42.279    Mode: 1 (2505)
value=115
count=378

Blue Histogram of Girl
300x240 pixels; RGB; 281K
Count: 65536      Min: 1
Mean: 46.305      Max: 222
StdDev: 38.620    Mode: 4 (1724)
value=172
count=24
```


Girl_hist_eq.raw
Histogram of Girl_hist_eq
Red Histogram of Girl_hist_eq
Green Histogram of Girl_hist_...
Blue Histogram of Girl_hist_eq
0
255
List
Copy
Log
Live
RGB

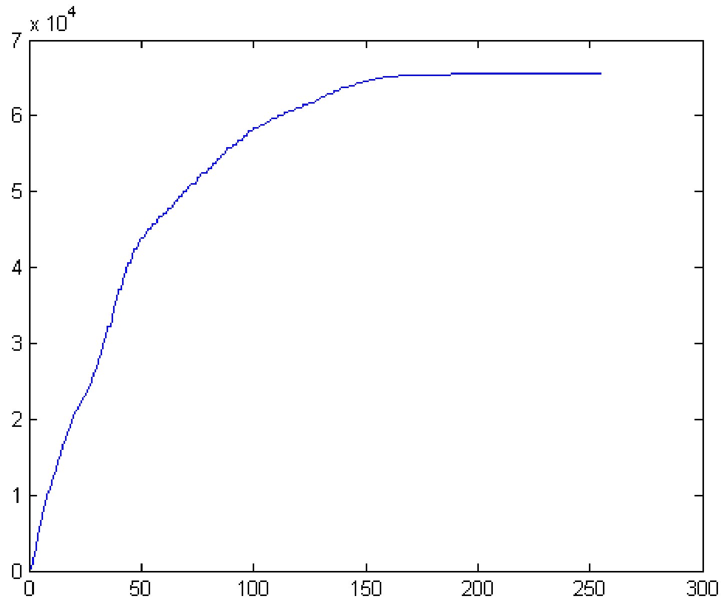
```
Girl_hist_eq.raw
256x256 pixels; RGB; 256K

Histogram of Girl_hist_eq
300x240 pixels; RGB; 281K
Count: 65536      Min: 0
Mean: 127.001     Max: 254
StdDev: 67.231    Mode: 107 (390)
value=46
count=273

Red Histogram of Girl_hist_eq
300x240 pixels; RGB; 281K
Count: 65536      Min: 0
Mean: 127.002     Max: 255
StdDev: 73.613    Mode: 0 (257)

Green Histogram of Girl_hist_...
300x240 pixels; RGB; 281K
Count: 65536      Min: 0
Mean: 127.002     Max: 255
StdDev: 73.613    Mode: 0 (257)
value=152
count=257

Blue Histogram of Girl_hist_eq
300x240 pixels; RGB; 281K
Count: 65536      Min: 0
Mean: 127.002     Max: 255
StdDev: 73.613    Mode: 0 (257)
value=162
count=257
```



Transfer function for red channel



Transfer function for green channel



Transfer function for blue channel

b) Oil Painting Effect

Oil Paint Effect is one of the most popular effect used in image processing, mostly for artistic purposes. The uniqueness of Oil Paint effect is that it uses a reduced palette of colors. It portrays the reality of using colors that painters use. The palette we've chosen for this problem has 256 colors as against 2 to the power 24 colors that a true color display can show.

The reduction of color palette can be done by k-means clustering that reduces the range of colors that can be displayed in the image. In this problem, we reduce the memory consumption by two-thirds, which is a potential application in image transfer.

Images that don't have wider dynamic range are most suited for such a reduction as the original and modified images don't look too far apart.

Apart from reduction in color range, we also implement clustering of neighboring pixels and assign them the same 8-bit value. The reason for doing this is create an effect of canvas drawing. As canvas used for such painting are not as fine grained as the resolution of our screen, we must show an effect of reduced resolution.

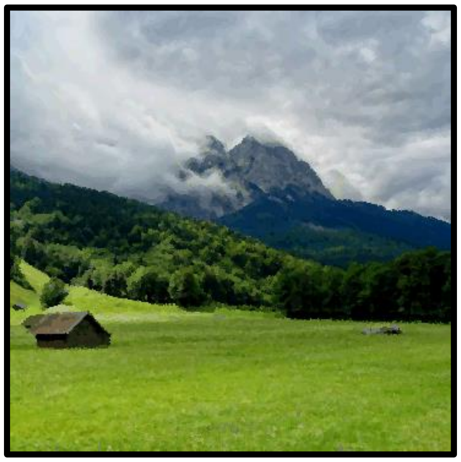
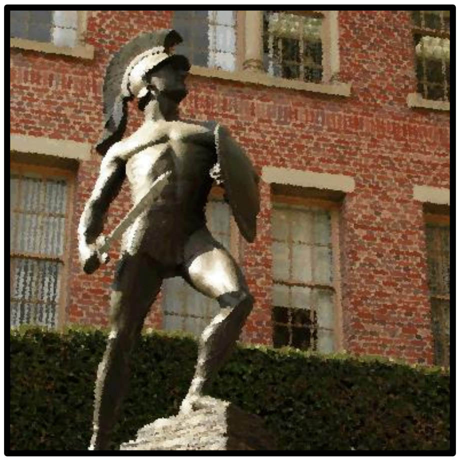
Oil-Paint Filter only depends on the original pixel intensities and can hence be parallelized which would make it extremely efficient for large images as well as for large kernel sizes. Running oil-paint filter on single thread can be performance-hungry with large kernel sizes.

For this problem, we've already been provided with two 256 color images viz. "scene2_256.raw" and "Trojan_256.raw".

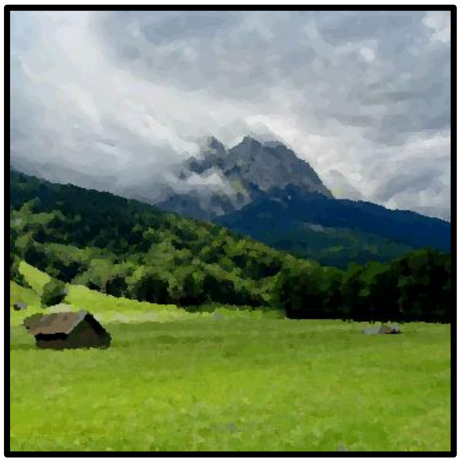
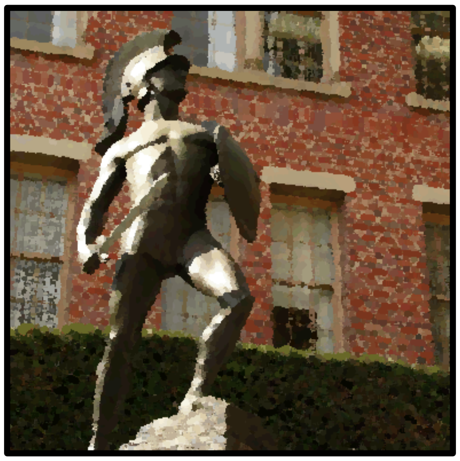
The filter is applied on original filters with different kernel sizes and following are the results:

Kernel Size

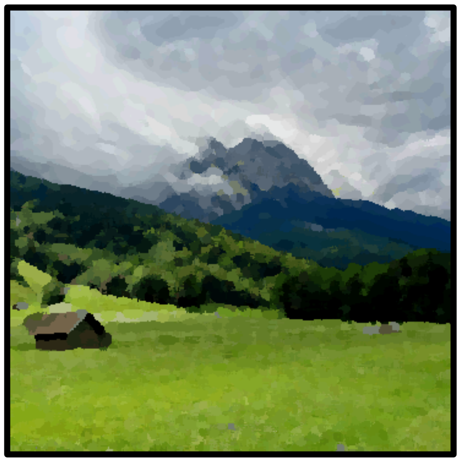
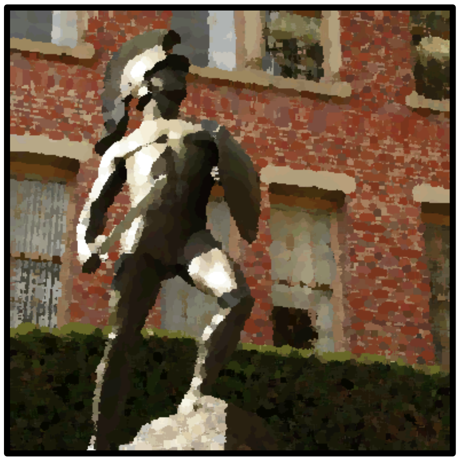
3:

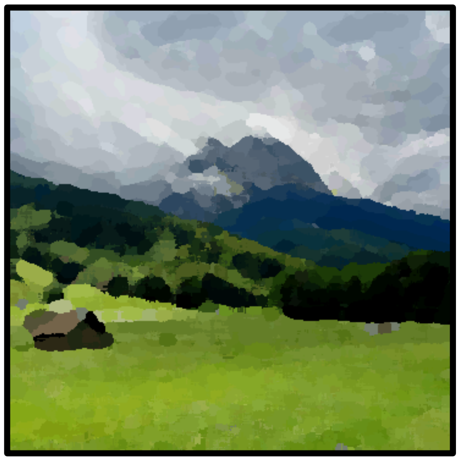
5:

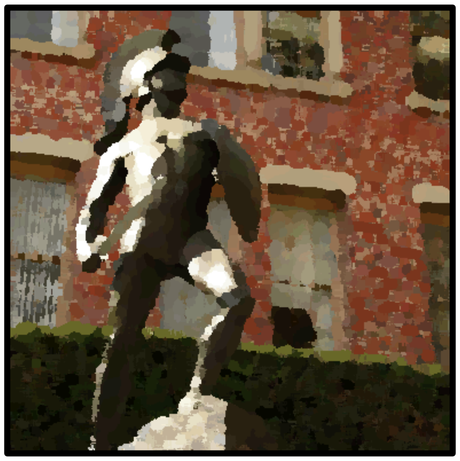



7:

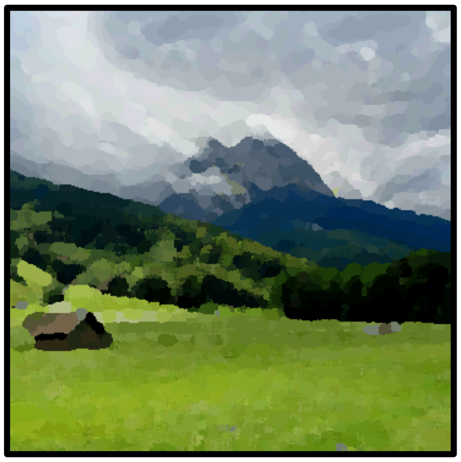



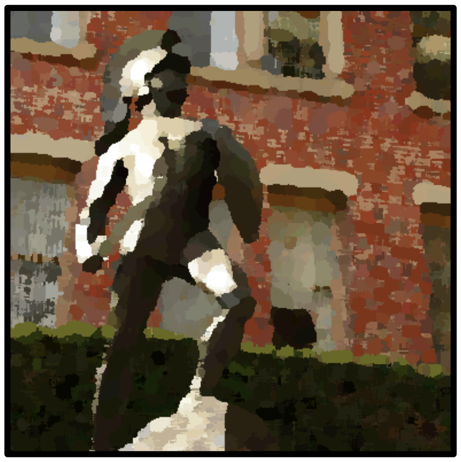
9:





11:





The approach taken for this problem is to use kernel over the image that replaces the center pixel in the window with the most frequently appearing intensity in the window.

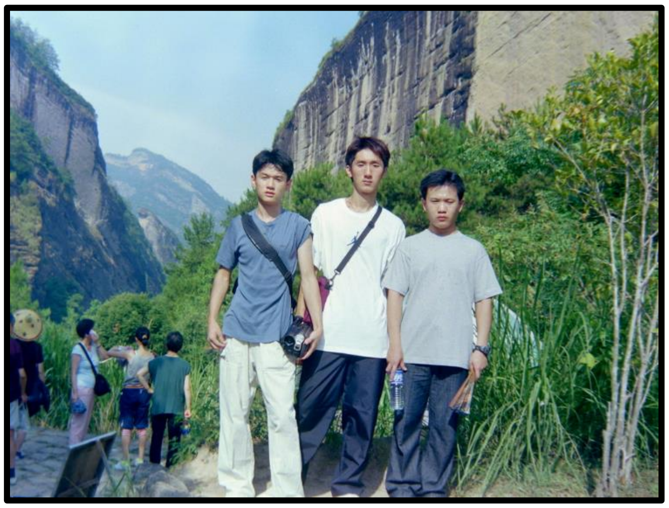
Firstly a nested loop is setup which goes over all the pixels in the image and over all its channels one-by-one. The center element is where the kernel is placed over. The surrounding elements become the other elements in the kernel. All these elements that come in the kernel are collected in an array.

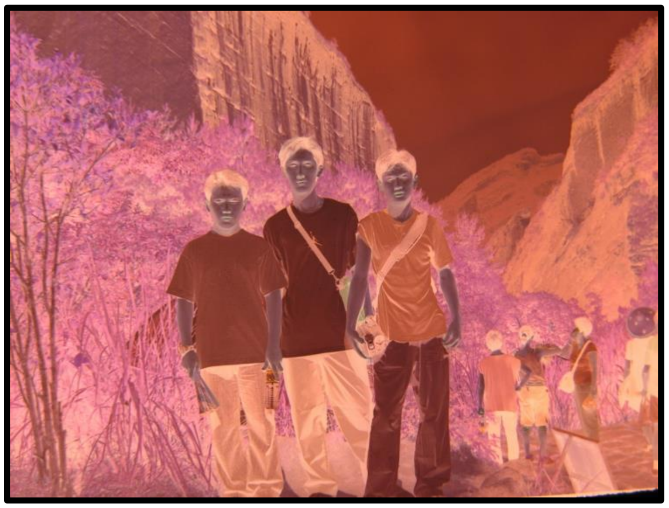
The array of window size squared is parsed to find the most frequently occurring element in the window. This intensity is then assigned to the center pixel in consideration. All the results are written to a new file to avoid overwriting and conflicts.

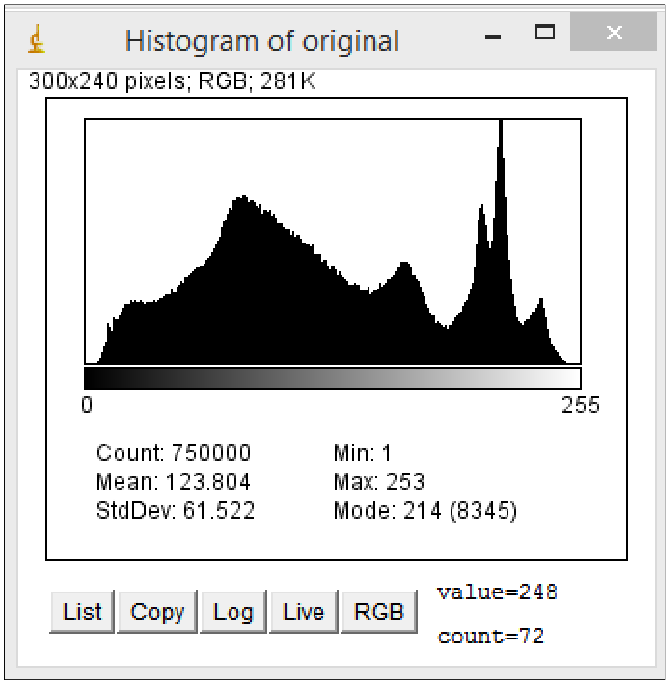
The first image scene2_256.raw is better candidate for such an effect cause the dynamic range of colors in the original image is quite narrow, which is suitable for oil paint filter. It gives a more convincing result than Trojan_256.raw. Kernel size/Window size of 7*7 is the best to use as it preserves the geometrical inferences and still causes enough distortion to properly portray it as an oil painting.

3) Creating Film Special Effect

The objective of this problem is to duplicate a special effect seen in the following images.





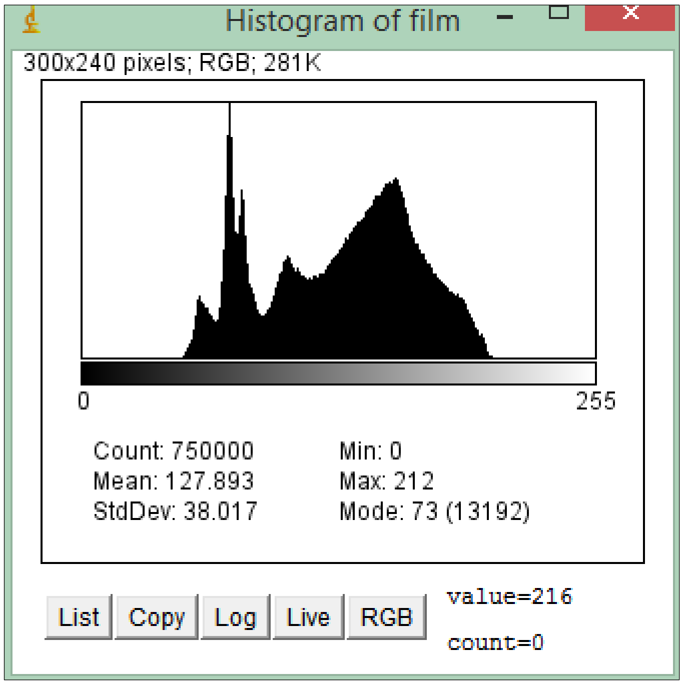


Histogram of original

300x240 pixels; RGB; 281K

```
Count: 750000      Min: 1
Mean: 123.804      Max: 253
StdDev: 61.522     Mode: 214 (8345)
```

```
value=248
count=72
```

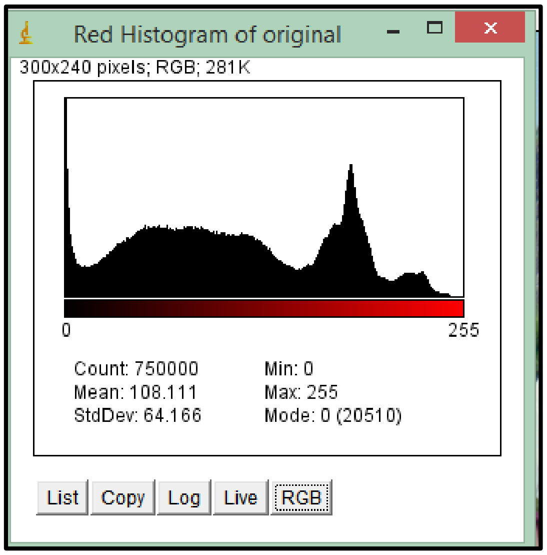


Histogram of film

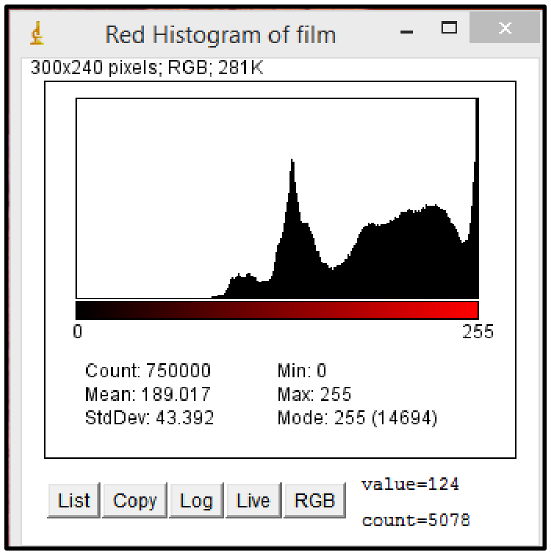
300x240 pixels; RGB; 281K

```
Count: 750000      Min: 0
Mean: 127.893      Max: 212
StdDev: 38.017     Mode: 73 (13192)
```

```
value=216
count=0
```

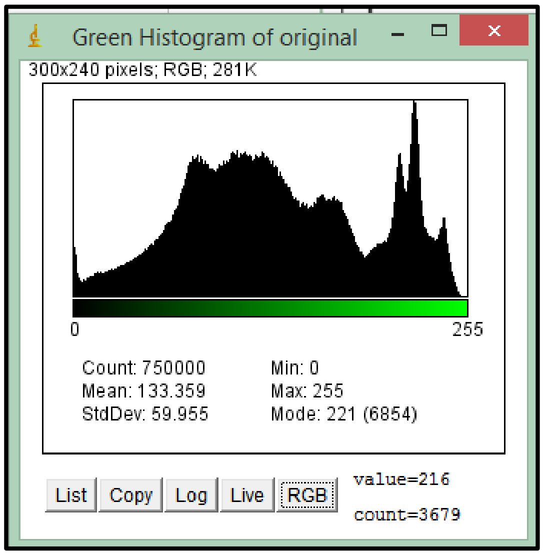


Count: 750000
Mean: 108.111
StdDev: 64.166
Min: 0
Max: 255
Mode: 0 (20510)



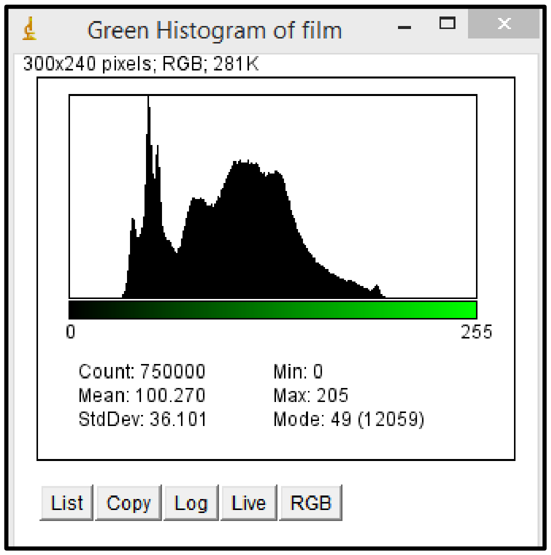
Count: 750000
Mean: 189.017
StdDev: 43.392
Min: 0
Max: 255
Mode: 255 (14694)

value=124
count=5078

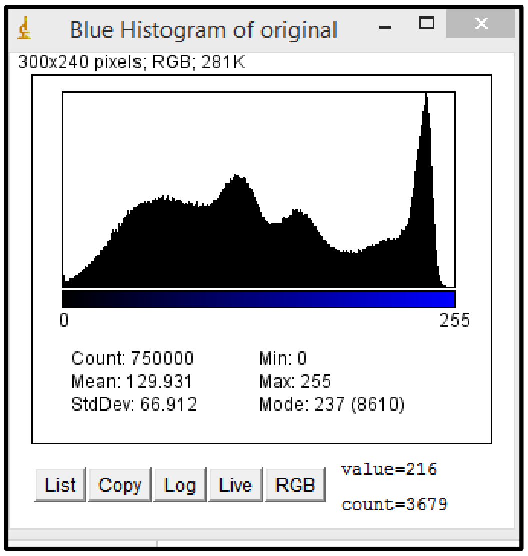


Count: 750000
Mean: 133.359
StdDev: 59.955
Min: 0
Max: 255
Mode: 221 (6854)

value=216
count=3679

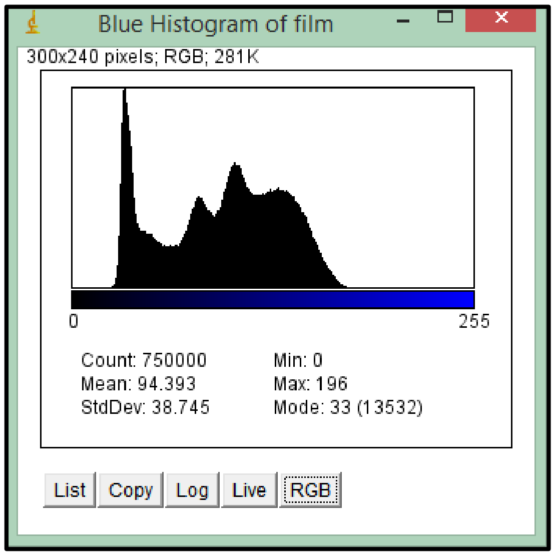


Count: 750000
Mean: 100.270
StdDev: 36.101
Min: 0
Max: 205
Mode: 49 (12059)



Count: 750000
Mean: 129.931
StdDev: 66.912
Min: 0
Max: 255
Mode: 237 (8610)

value=216
count=3679



Count: 750000
Mean: 94.393
StdDev: 38.745
Min: 0
Max: 196
Mode: 33 (13532)

From all the above information in the images and its histograms, we can faithfully say that,

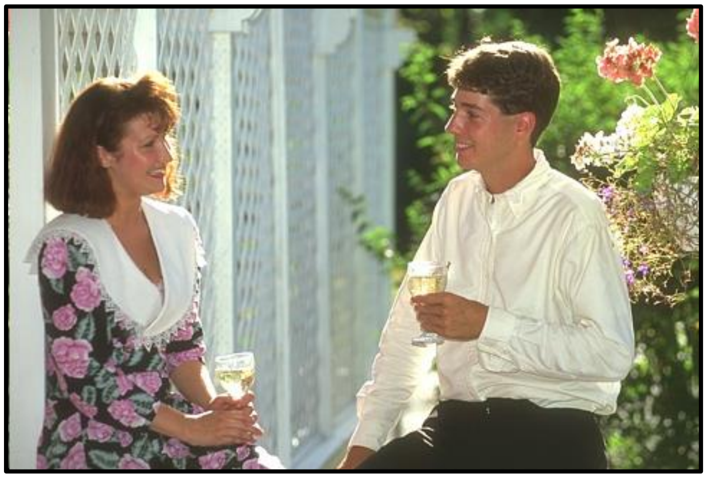
1) The image in inverted horizontally.
2) The histograms of all channels in the image are also inverted.
3) The histograms in the resulting image are scaled down.

These three effects need to be encapsulated in one film special effect filter and applied to a sample image.

We first handle inverting of the image on horizontal axis,

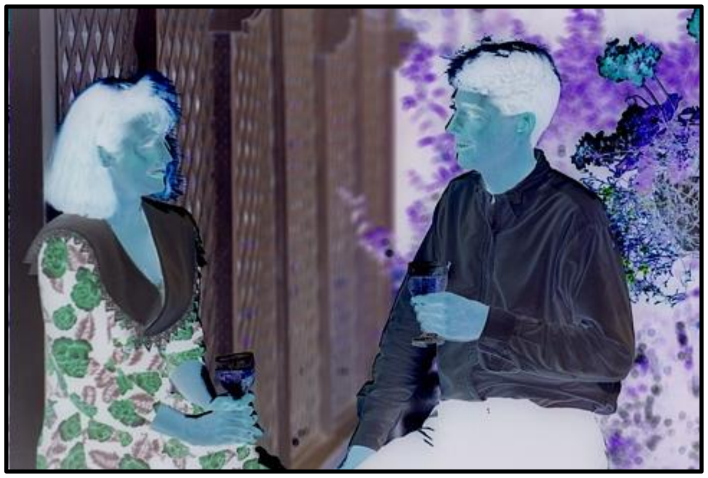
```
for(int i = 0; i < img_modified.rows; i++)
for(int j = 0; j < img_modified.cols; j++)
       for(int k = 0; k < img_modified.channels; k++)
       {
       img_modified.setvalue(i,j,k,(img_orig.getvalue(i,(img_orig.cols-j),k)));
       }
```

We invert the column number on all pixels.



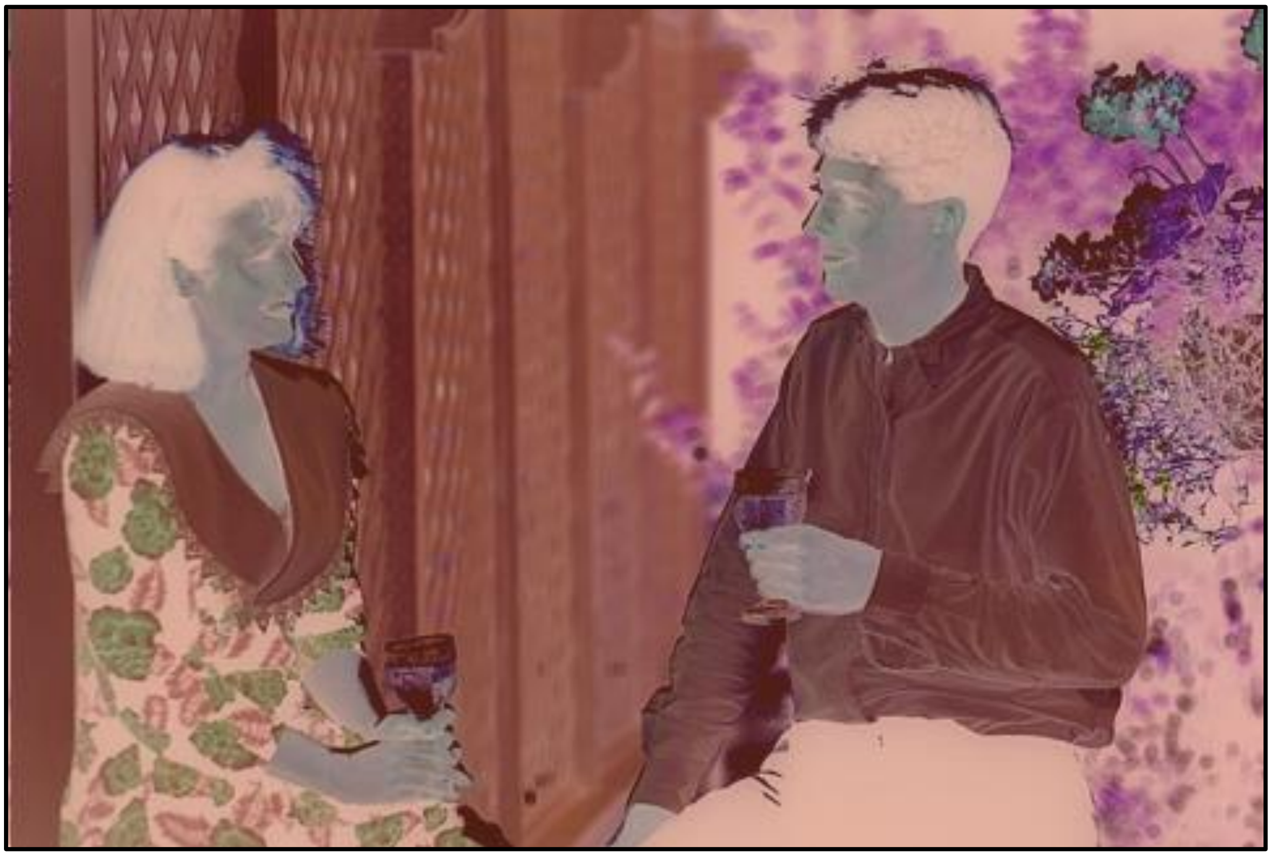
Then we handle inverting all the channels by finding their complement in 255

```
for(int i = 0; i < img_modified.rows; i++)
for(int j = 0; j < img_modified.cols; j++)
       for(int k = 0; k < img_modified.channels; k++)
       {
       img_modified.setvalue(i,j,k,(255-img_orig.getvalue(i,j,k)));
       }
```



Then we scale the gray values on all the channels such that their resulting histograms are scaled down. By finding the min and max gray value from the sample images we can determine the scaling factor for all channels.

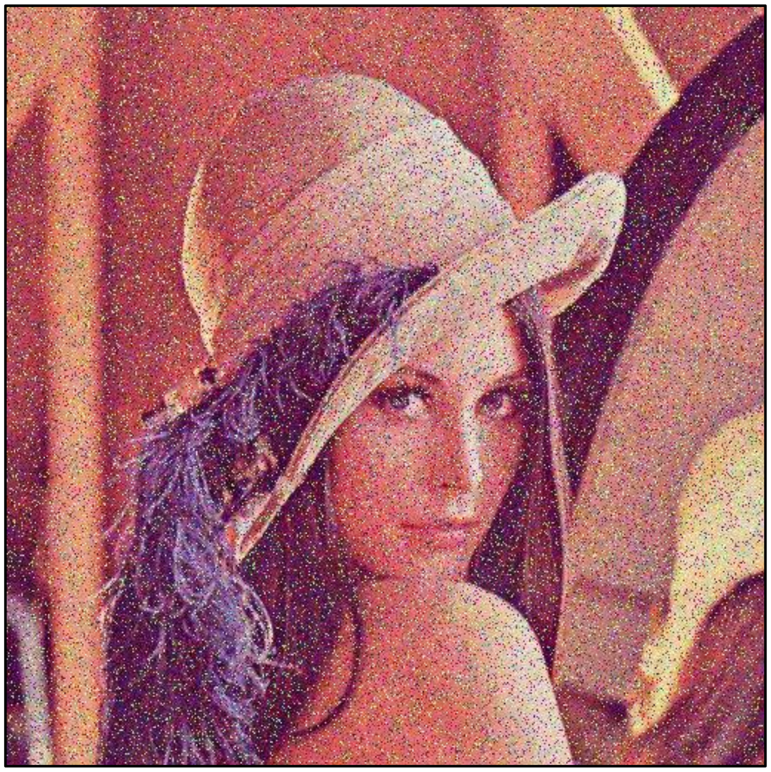
```
int gmax[3] = {255,205,180};
int gmin[3] = { 80, 30, 20};
float scaling_factor[3];
for(int i = 0; i < 3; i++)
       scaling_factor[i] = (gmax[i]-gmin[i]) / 255.0;
for(int i = 0; i < img_modified.rows; i++)
for(int j = 0; j < img_modified.cols; j++)
for(int k = 0; k < img_modified.channels; k++)
{
       img_modified.setvalue(i,j,k,((img_modified.getvalue(i,j,k)
         * scaling_factor[k])
         + gmin[k]));
}
```

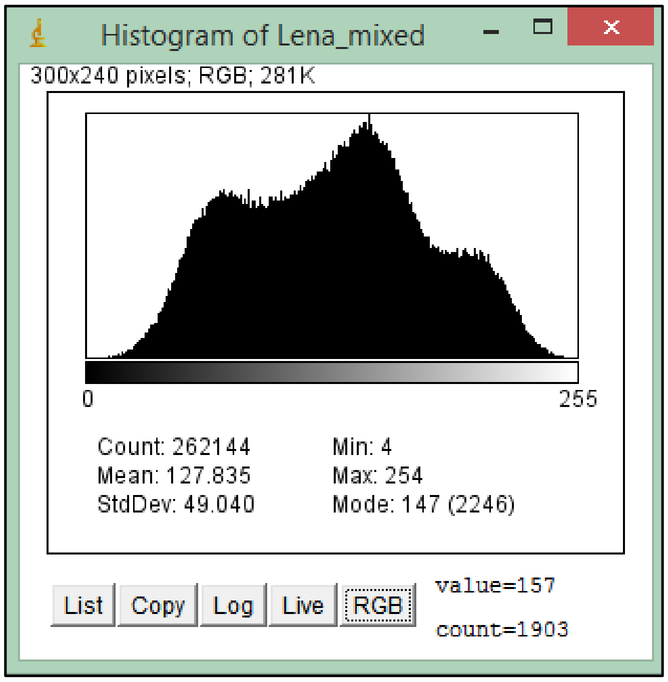


Problem 3: Noise Removal

a) Mix Noise in Color image

The following image has to be de-noised.





Histogram of Lena_mixed

300x240 pixels; RGB; 281K

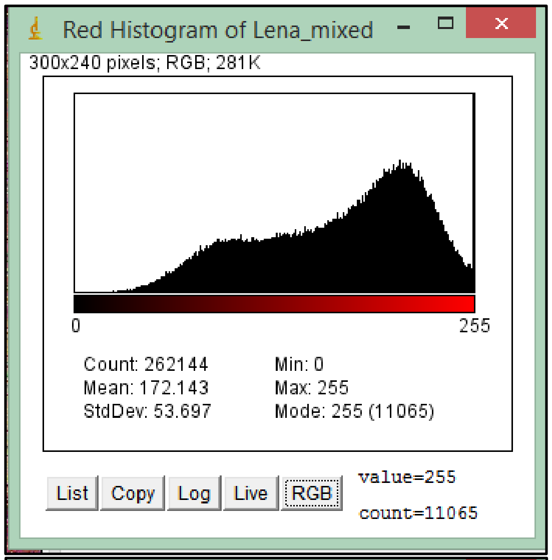
Count: 262144
Mean: 127.835
StdDev: 49.040

Min: 4
Max: 254
Mode: 147 (2246)

value=157
count=1903

Ans. 1 a) No, all channels don’t have the same noise type

It is observed in red channel that there is salt noise as the intensity value of 255 has a spike in histogram graph. It may also have Gaussian noise as the histogram is fairly smooth at peaks.

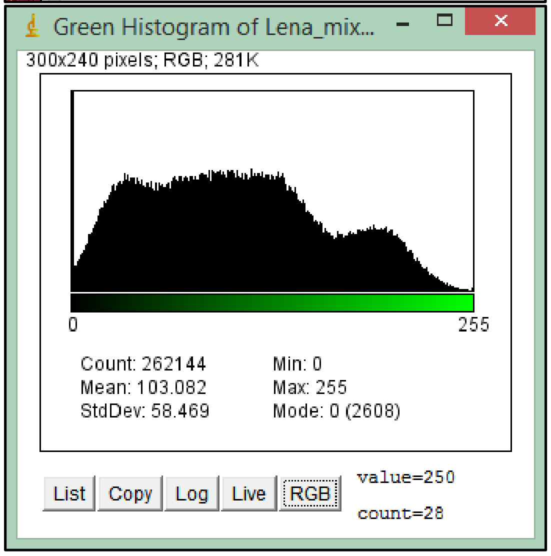


300x240 pixels; RGB; 281K

Count: 262144
Mean: 172.143
StdDev: 53.697
Min: 0
Max: 255
Mode: 255 (11065)

value=255
count=11065

It is observed in green channel that there is pepper noise as the intensity value of 0 has a spike in histogram graph. It may also have Gaussian noise as the histogram is fairly smooth at peaks.

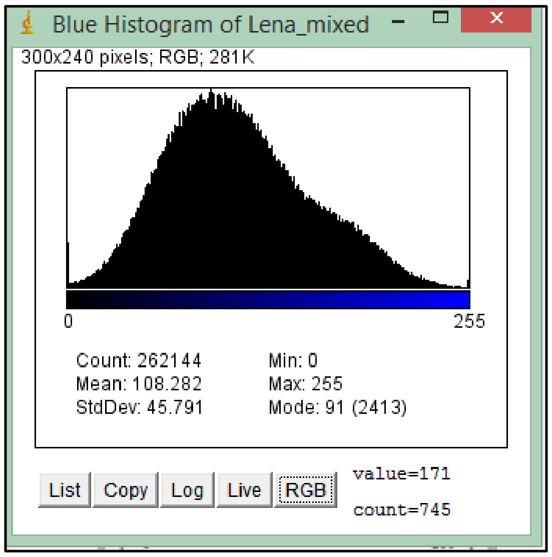


300x240 pixels; RGB; 281K

Count: 262144
Mean: 103.082
StdDev: 58.469
Min: 0
Max: 255
Mode: 0 (2608)

value=250
count=28

It is observed in blue channel that there is Gaussian noise as the histogram is fairly smooth at peaks.



300x240 pixels; RGB; 281K

Count: 262144
Mean: 108.282
StdDev: 45.791
Min: 0
Max: 255
Mode: 91 (2413)

value=171
count=745

It may also have salt and pepper noise as there are moderate levels of spikes on intensity values 0 and 255.

Ans. 1 b)

Yes, it is imperative that we apply filters to the channels individually because the noise is noticeably different in all channels. The histogram of the combination of R, G and B doesn't show any peculiar patterns which may lead to a conclusion that there isn't any benefit in applying filters to the color image itself.

Another reason to apply filter individually to color channels is that no to channels have the exact same noise and filter to their combination won't remove noise in those channels separately. For example, red channel has a lot of salt noise and blue channel has a Gaussian noise and there is no way to resolve this noise collectively in one step.

Ans. 1 c)

As it's evident that salt and pepper noise and Gaussian noise are present in all channels collectively, it makes sense to have a non-linear and linear filter to remove that noise. To remove the non-linear noise in particular, we must use a **median filter** which is most effective in removing outlier and not disturbing the mean value of the window it parses. To remove the linear noise in particular, we must use a Gaussian filter which can tackle the **Gaussian noise** present in the color channels.

Ans. 1 d)

No, it is not possible to use these filters in any order. A median filter is used to remove the non-linear noise such as salt and pepper noise. These are outliers with intensities of 0 or 255. A Gaussian filter averages the intensities and blurs the image. If we apply the Gaussian filter before median filter, we'll average the salt and pepper noise too and then removing them with Median filter won't be possible. Therefore, it's better to apply median filter before Gaussian to have max removal of noise.

Ans. 1 e)

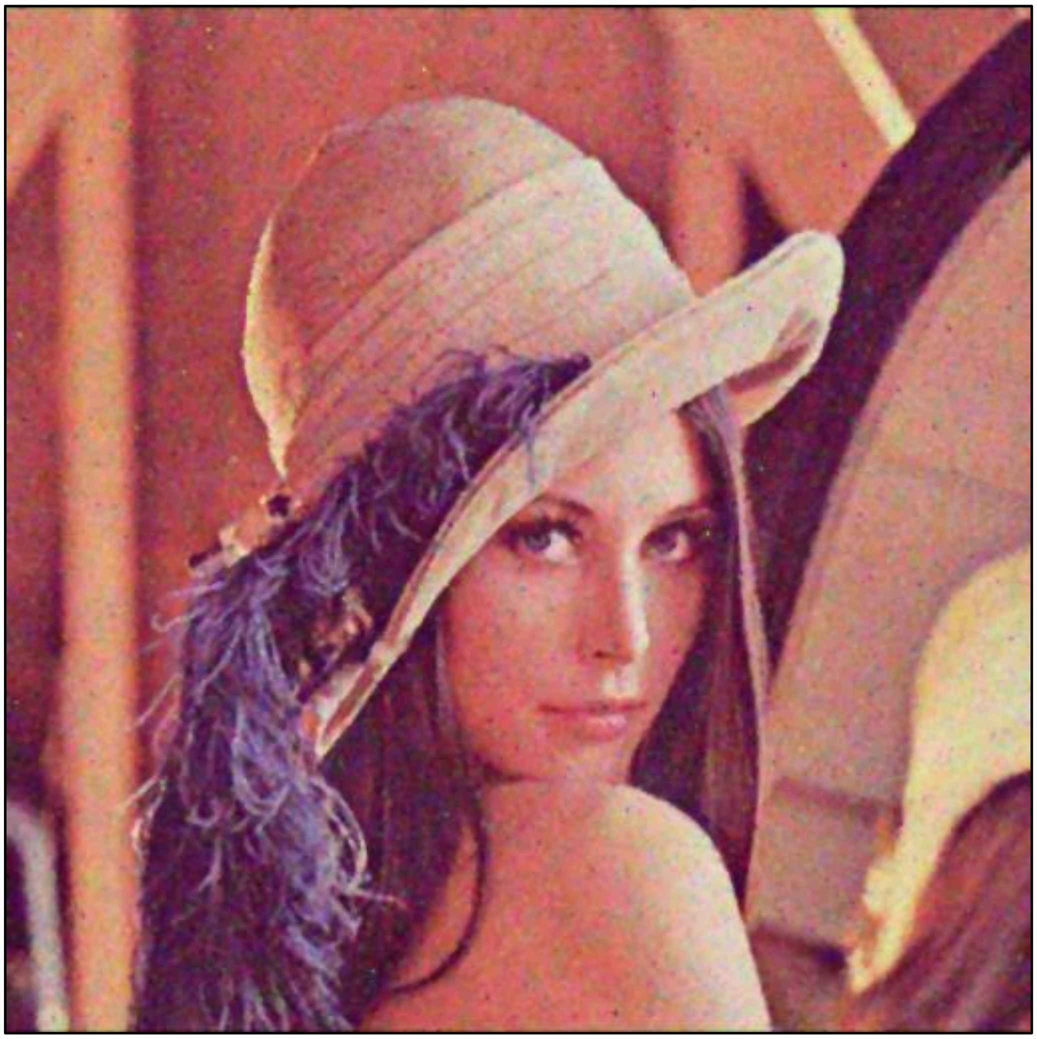
Window sizes in case of both filters make a lot of difference. With a relatively smaller window for median filter, it is possible to remove sparse non-linear noise. With a relatively bigger window for median filter, even if the non-linear noise is in sizeable clusters, they can be removed as we have more data available to parse out outliers.

With a bigger window for Gaussian Filter, it does a good job of averaging values across the image but gives an undesirable blurring effect all over the image that leads to loss in sharpness.A smaller window size is better Gaussian Filter as it limits the averaging to a smaller region and hence the effect of dynamic change is limited as well.

2 a)

The following is the result with window size of 3 for the filters and sigma equals 9 for Gaussian filter

```
$ ./Main -prob denoising -i ../images/Lena_mixed.raw -o ../images/Lena_clean.raw -w 512
-h 512 -sigma 9 -ws 3
```



Firstly, I implemented a median filter to be used for removal of non-linear noise or salt and pepper noise in specific. The median filter is kernel that takes in all the values in the window in the image it's parsing and assigns its median to the center pixel.

The pseudo code for median filter is as follows:

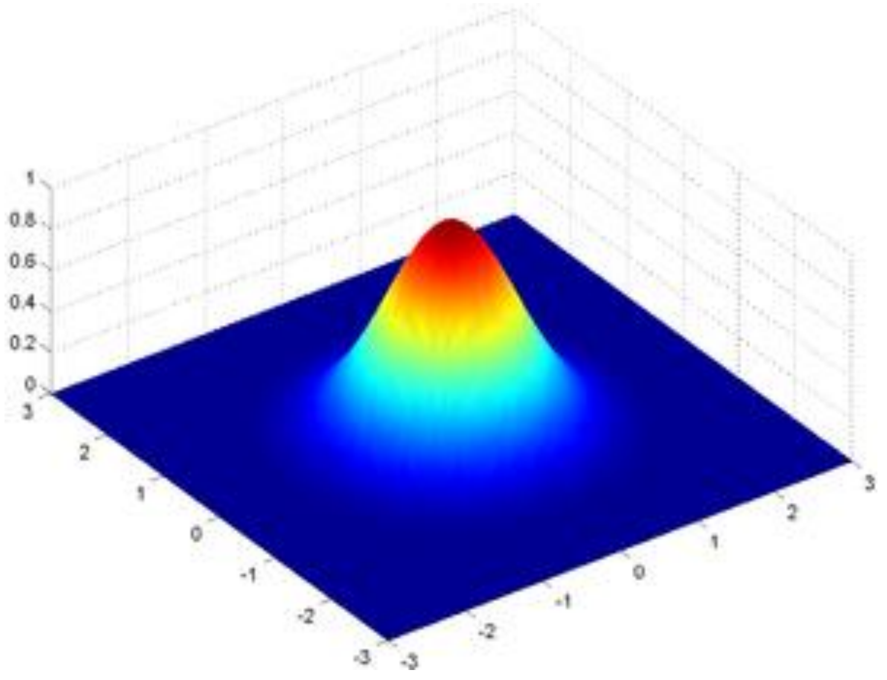
```
Pass the window for median filter over every pixel
{
    Let center pixel be the point of focus; store all the intensity values in the window in an array
    Sort the array
    Get the median of the array
    Assign center pixel the value of the obtained median
}
```

```
for(int i = 0; i < img_orig.rows; i++)
for(int j = 0; j < img_orig.cols; j++)
for(int k = 0; k < img_orig.channels; k++)
{
for(index = 0; index < (ws*ws); index++)
       values[index] = 0;
index = 0;

for(int m = -kernel_radius; m <= kernel_radius; m++)
for(int n = -kernel_radius; n <= kernel_radius; n++)
{
       if((i+m >= 0) && (i+m < img_orig.rows) && (j+n >= 0) && (j+n < img_orig.cols))
              values[index] = img_orig.getvalue((i+m),(j+n),k);
       else
              values[index] = img_orig.getvalue((i+m),(j-n),k);
       index++;
}

mergesort(values, 0, ((ws*ws)-1));
median_value = get_median(values, (ws*ws));
img_modified.setvalue(i,j,k,median_value);
}
```

Next, we implement Gaussian Filter. Gaussian Filter is discrete convolution operator based on the Gaussian function defined by

$$f(x,y) = A\exp\left(-\left(\frac{(x-x_o)^2}{2\sigma_x^2} + \frac{(y-y_o)^2}{2\sigma_y^2}\right)\right).$$

Where x0, y0 is the center of the Gaussian bell and sigma is the span of the bell



For computational simplicity we create a Gaussian kernel operator with center (0,0) and compute normalized weights at each discrete location around it for certain window size.

```
for (int row = -kernel_radius; row <= kernel_radius; row++)
        for (int col = -kernel_radius; col <= kernel_radius; col++)
        {
            frac = exp(-1.0 * ((row*row) + (col*col)) / (2.0 * (sigma*sigma)));
            kernel[row + kernel_radius][col + kernel_radius] = frac;
            kernel_sum += kernel[row + kernel_radius][col + kernel_radius];
        }
for (int i = 0; i < ws; i++)
       for (int j = 0; j < ws; j++)
            kernel[i][j] = kernel[i][j] / kernel_sum; //Normalized
```

Where kernel[ ][ ] is the Gaussian operator.

Next, I made a Denoising() function to read the images and apply the median and Gaussian on individual color channels in the image.

Ans. 2 b) Such an implementation denoises the image to quite some extent but not fully. The Gaussian filter implementation won’t be able to remove Gaussian noise if it’s not uniformly distributed over the whole image. In that case the filter may cause blurring in some region or may not be fully able to remove noise in other regions. The median filter too could be tuned to remove salt or pepper noise individually more effectively if we have prior knowledge of such a discrepancy.

Ans.2 c) One way to handle such non-linearity in noise would be to use approaches that can approximate absolute solutions to such problems. One of the way to tackle non-uniform distribution of Gaussian noise analyze the data in the window and approximate it to a surface. If the surface is close a Gaussian bell, then we can apply the filter, else not.

Another way to tackle non-linearity would be to use Least Squares approach to find an approximate linear solution by taking samples of data uniformly from all over the image.

b) Bilateral Filtering

Bilateral Filter is one of the efficient ways to remove noise without affecting edges or high-gradient changes. This is possible because Bilateral Filter takes similarity into factor when it run a Gaussian filter over the image for de-noising.

Gaussian filter produces an undesirable blurring of high-gradient changes which causes reduction in contrast levels as well. Bilateral filter factors in similarity constraint based on a Gaussian distribution with the value in the center of the window to be the center of that Gaussian distribution.

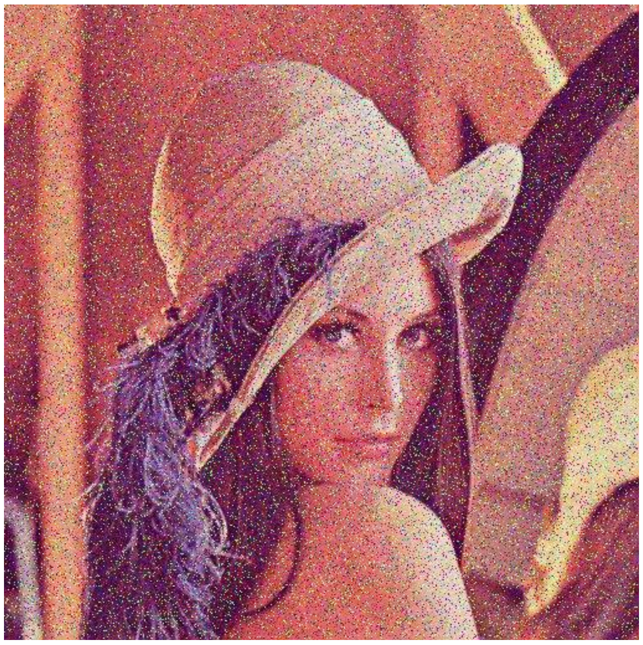
Bilateral filter is implemented according to the following function:

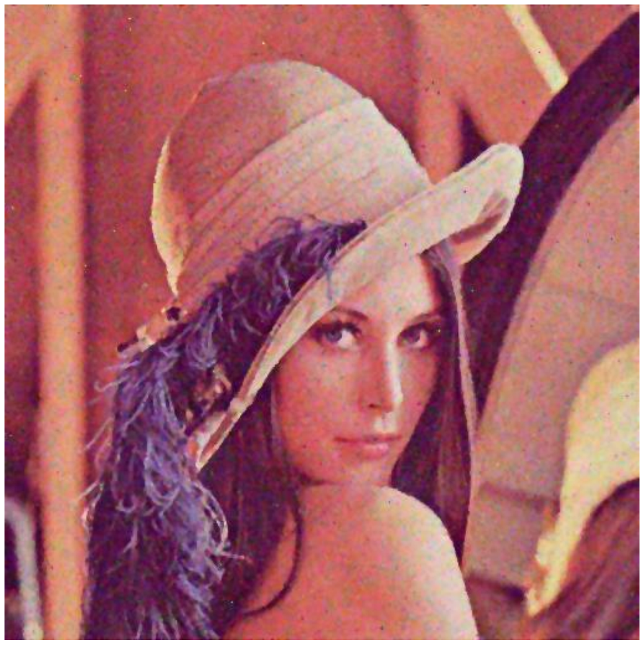
$$\mathbf{h}(\mathbf{x}) = k^{-1}(\mathbf{x}) \sum_{\xi \in \Omega_{\mathbf{x}}} \mathbf{f}(\xi)\ \exp\left(-\frac{\|\xi - \mathbf{x}\|_2^2}{2\sigma_c^2}\right)\ \exp\left(-\frac{\|\mathbf{f}(\xi) - \mathbf{f}(\mathbf{x})\|_2^2}{2\sigma_s^2}\right)$$

Where 1/k is the normalization factor and f(a) is the intensity at a(location in the image)

Sigma(c) is the factor in Gaussian function based on distance and sigma(s) is the factor in Gaussian function based on similarity.

The results of running the Bilateral filter on Lena_mixed after applying the median filter is as follows:





The observation that can be made here is that application of bilateral filter has averaged regions in the image but have maintained the high-gradient contours.

Ans. 2) ‘c()’ in the formulation of bilateral filter describes spatial closeness as Gaussian distribution. It evaluates the weight of surrounding elements in a window based on Gaussian function. This way elements farther away have less influence and element close by have more influence on the final value for intensity.

‘s()’ in the formulation of bilateral filter describes similarity constraint as a Gaussian distribution. It evaluates the weight of similar elements in the window based on 1D Gaussian function defined over the scale of intensity values. So the farther away an intensity value on the scale, lesser its influence and vice-versa.

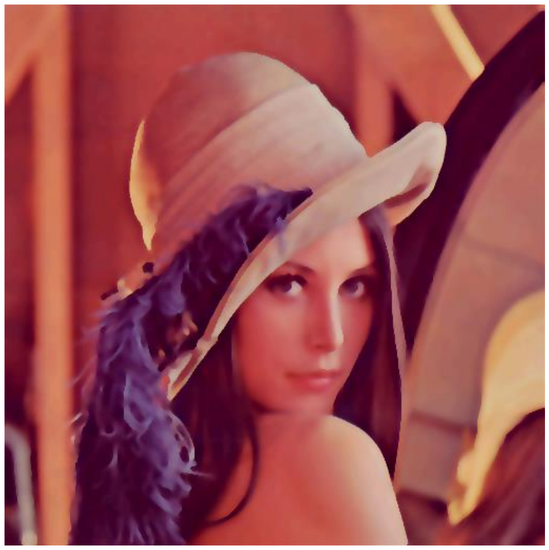
Sigma(c) defines the span of the Gaussian bell on the surface of the window. If sigma is relatively big then its decay in value is slower and vice-versa.

Sigma(s) defines the span of the Gaussian bell on the scale of intensities. If sigma is relatively bigger, then its decay in value is slower and hence has more influence even when the values are distant.

Window-size in case of bilateral filter could be bigger than usual as we are guaranteed to maintain our strong contours even if there are too many element surrounding to influence the center pixel in the window. But bigger window size would necessarily mean longer execution time. The execution time increases quadratic proportions on increasing the window size.

Ans. 3) Yes, this filter performs better than most linear filters out there as it maintain strong gradient changes and hence the contrast. It takes similarity into account which ensures better results for too less an overhead. This filter is based on a more mature heuristic than the simple mathematical operator that the other linear filters are.

Ans. 4) sigma(c) = 3 , sigma(s) = 20, iterations = 6, window size = 3



References

[1] Jack K (2007) Video demystified, 5th edition. Newnes.

[2] Grundland M, Dodgson N (2007) Decolorize: Fast, contrast enhancing, color to grayscale conversion. Pattern Recognition 40: 2891–2896.

[3] Christopher Kanan, Garrison W. Cottrell. Color-to-Grayscale: Does the Method Matter in Image Recognition?

[4] Pratt W (2007) Digital image processing. Wiley-Interscience

[5] Bosch A, Zisserman A, Munoz X (2007) Image classification using random forests and ferns. International Conf on Computer Vision (ICCV-2007).

[6] Forsyth and Ponce. Computer Vision 2002