EE 569 Homework 1 Write Up

Armin Bazarjani

February 5th, 2021

1 Image Demosaicing and Histogram Manipulation

Note I would like to say that although some output images may seem to have border issues that is purely because of the cropping of me taking a screenshot of the results from ImageJ. I have made sure that all of my image borders come out perfect and that there are no unfilled areas.

1.1 Motivation for Image Demosaicing

When taking a digital photograph neither a CMOS nor a CCD image sensor can accurately distinguish between the different wavelengths of light. Thus, this lack of wavelength specificity does not allow for the separation of color information.

The current method to deal with this is to overlay a color filter array (CFA) on top of the sensors. A CFA is made up of different color filters that only allow a certain range of light wavelength to pass through. This gives us information about the intensity of light for a single color. In order to combine these disparate color filters in a meaningful way we can place them in a specific pattern. The most common CFA pattern is with the Bayer filter, a 2x2 filter which contains twice as much green as it does red and blue. The motivation behind using more green is simply because the human eye is more sensitive to green.

Now that we have color information in the form of intensity values for our specified CFA (most likely a bayer filter), we only solved half the problem. The next step is to use these different intensity values to estimate the missing color intensity values for every other pixel in our image. This process is called demosaicing, of which there are many methods we can use, here we focus on bilinear demosaicing. Afterwards, we are left with a digital color image that is hopefully representative of what we are trying to take a photograph of.

1.2 Motivation for Histogram Manipulation

Sometimes when we take photos the resulting lighting in the digital image is not the best, making the finer details of a the photo harder to see. This most commonly occurs when the photo is either too dark or too light. We refer to these types of photos as having low contrast, meaning that objects in the same field of view have similar levels of brightness. This can quite often lead to a visually displeasing photograph.

One method of contrast adjustment is through histogram manipulation. Where the histogram can be thought of as a distribution of intensity frequency values. In other words, on the x-axis we have our pixel values and on the y-axis we have the frequency of occurrence of said pixel value. The idea is relatively simple, we want to "stretch out" the frequency histogram of an image in order to make it more uniform. Intuitively this makes quite a bit of sense as to how this process will yield favorable results as we are making sure one intensity value isn't dominating any others.

1.3 Approach and Procedures

1.3.1 Convolution

Before we begin discussing the approach, I believe there should be a brief aside on convolution (discussed in this section) and padding (discussed in the next section) as they both show up extensively throughout this homework.

Because we are operating with Linear Shift Invariant (LSI) systems, we can characterize them completely through their responses to an impulse. In our situations these impulses will be our filters and we can slide these filters over our image and perform multiply accumulate operations to get the convolution output.

The Convolution equation is as follows:

$$g(x,y) = w * f(x,y) = \sum_{dx=-a}^{dx=a} \sum_{dy=-b}^{dy=b} w(dx,dy) f(x+dx, y+dy)$$

Where $g(x,y)$ is the filtered image, $f(x,y)$ is the original image, and w is the filter kernel which has a height of $2b$ and a width of $2a$.

1.3.2 Padding

The idea of padding is very straightforward. When we do a convolution operation on a matrix, we are downsizing the matrix that we are applying the convolution on. In order to maintain the same size afterwards, it is common practice to "pad" the image's border so that we are convolving on an enlarged matrix. Thus, once this enlarged matrix downsizes, it will be the same size as

our original matrix.

For Example, if we have a matrix with height H_1 and width W_1, a square filter of size F, a stride of S, and padding P.

Our new matrix values will have a height of

$$H_2 = \frac{H_1 - F + 2P}{S} + 1$$

and a width of

$$W_2 = \frac{W_1 - F + 2P}{S} + 1$$

However, throughout this assignment Stride will be a constant value at 1.

1.3.3 Bilinear Demosaicing

The main idea behind bilinear demosaicing is that for each color channel of a pixel, we will use the neighbors that obtained intensity information of that color to interpolate the missing color for the current pixel. The neighbors must be adjacent to the current pixel and thus must be in a 3x3 window centered around the pixel of interest.

Let's assume that $\hat{G}_{x,y}$ represents the estimated amount of green at pixel location (x, y), and $G_{x,y}$ represents the actual amount of green at the pixel location (meaning it is overlaid by the green color filter in our CFA).

At any red location in our bayer filter:

$$\hat{R}_{x,y} = R_{x,y}$$

$$\hat{B}_{x,y} = \frac{1}{4}(B_{x+1,y+1} + B_{x+1,y-1} + B_{x-1,y+1} + B_{x-1,y-1})$$

$$\hat{G}_{x,y} = \frac{1}{4}(G_{x+1,y} + G_{x-1,y} + G_{x,y+1} + G_{x,y-1})$$

At any blue location in our bayer filter:

$$\hat{B}_{x,y} = B_{x,y}$$

$$\hat{R}_{x,y} = \frac{1}{4}(R_{x+1,y+1} + R_{x+1,y-1} + R_{x-1,y+1} + R_{x-1,y-1})$$

$$\hat{G}_{x,y} = \frac{1}{4}(G_{x+1,y} + G_{x-1,y} + G_{x,y+1} + G_{x,y-1})$$

At a green location on an even row and even col (assuming zero indexing)

$$\hat{G}_{x,y} = G_{x,y}$$

$$\hat{R}_{x,y} = \frac{1}{2}(R_{x+1,y} + R_{x-1,y})$$

$$\hat{B}_{x,y} = \frac{1}{2}(B_{x,y+1} + B_{x,y-1})$$

At a green location on an odd row and odd col (assuming zero indexing)

$$\hat{G}_{x,y} = G_{x,y}$$

$$\hat{R}_{x,y} = \frac{1}{2}(R_{x,y+1} + R_{x,y-1})$$

$$\hat{B}_{x,y} = \frac{1}{2}(B_{x+1,y} + B_{x-1,y})$$

These values can be easily computed through convolution and the use of four different 3x3 filters, where your current location within the array influences which filters you will use to interpolate the missing color value. In order to preserve the original size of the image, I mirror padded the image with a padding size of $P = 1$. Mirror padding serves to both retain the original dimensions of the image as well as "continue" the bayer pattern outwards allowing for us to easily convolve with our 3x3 filters.

Following, is a table showing the different 3x3 filters that I used.

diagonal filter	adjacent filter	vertical filter	horizontal filter
0.25 0.00 0.25	0.00 0.25 0.00	0.0 0.5 0.0	0.0 0.0 0.0
0.00 0.00 0.00	0.25 0.00 0.25	0.0 0.0 0.0	0.5 0.0 0.5
0.25 0.00 0.25	0.00 0.25 0.00	0.0 0.5 0.0	0.0 0.0 0.0

1.3.4 Transfer Function Based Equalization

The transfer function approach comes from the probabilistic theory that says if we have a continuous random variable X and we define a new random variable as $Y = CDF(X)$, then our new random variable Y will have a uniform distribution.

The process is relatively simple. If we let $p_n = \frac{number\ of\ pixels\ with\ intensity\ n}{total\ number\ of\ pixels}$

Then our transfer function becomes:

$$y = T(x) = floor[(L-1)\sum_{n=0}^{x} p_n] = (L-1)CDF(x)$$

Where L is our total number of pixels (usually 256), and we multiply by $(L-1)$ as a means of obtaining a pixel intensity as our cdf only gives a value between 0 and 1.

To implement this I looped through the input image, calculated the histogram values for p_n where $0 \leq n \leq 255$. And then looped through the histogram and summed each value up to where I was in the loop to calculate the CDF. From there, I looped through the original image again and applied the transfer function to each pixel value.

1.3.5 Cumalative Probability Based Equalization

This method can also be referred to as the "bucket filling" method. Instead of the transfer function method, this one opts to first make the uniform distribution as a sequence of "buckets" and then "fill" each one accordingly.

For example, in our case, we would have $B = L = 255$ buckets and each one will have $\frac{num\_pixels}{num\_buckets}$ pixels per bucket.

I decided to implement this by first looping through the original image and filling a vector of size $L = 256$ with the location values (x,y coordinates) of every pixel. Making sure to use the pixel's value as its key in the vector. So in the end I am left with a size 256 vector where each value (1,..,256) in the main vector holds another vector of pair locations to store the exact location of corresponding pixel intensities.

Afterwards, I created an outer loop to loop through the entire vector and an inner loop to loop through all of the different location pairs within each pixel value key in the vector. I then map each pixel location in the vector to the corresponding value in the uniform bins array. I do this by keeping track of (1) what the current bin value is and (2) by keeping track of how many pixels I have already mapped to the current bin value. If (2) reaches the allowed number of pixels per bin, I increment (1) and set (2) back to zero.

1.4 Experimental Results

For demosaicing, we applied the bilinear method to the grayscale House image. I have also included the original House image for comparison.

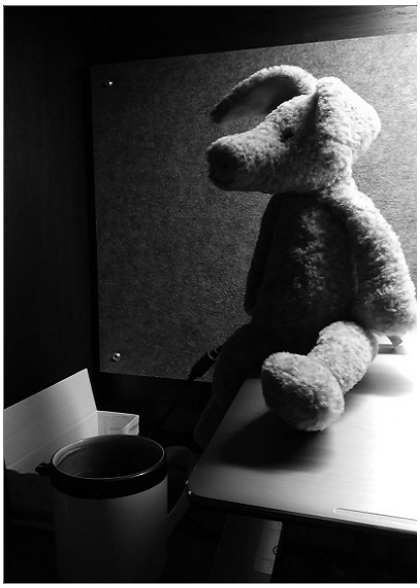
(a) Gray Scale House

(b) Original House

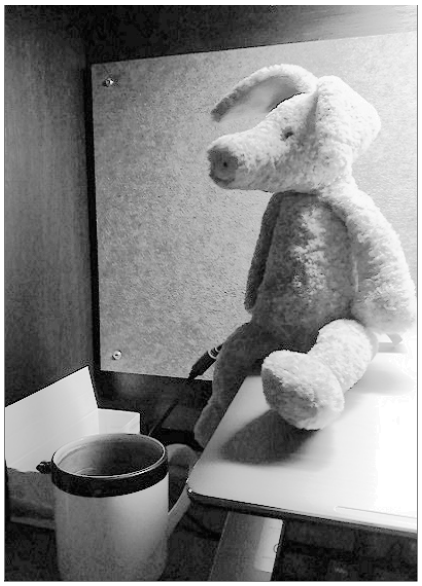
(c) Demosaiced House

Figure 1: Bilinear Demosaicing on House Image

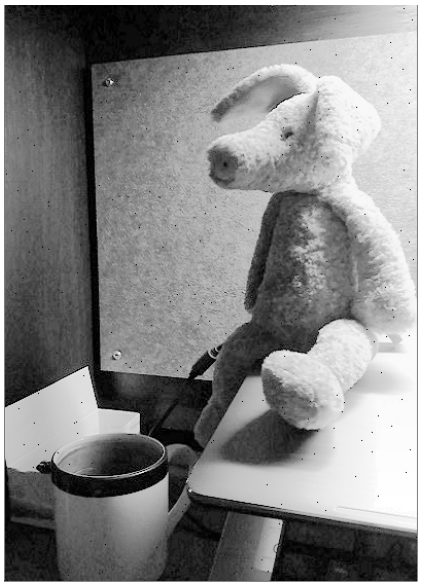
For Histogram Equalization, we were tasked with applying both the transfer function method and the bucket filling method to the Toy image. I have included images of both methods applied to the toy image as well as the original toy image for comparison.



(a) Original Toy



(b) Transfer Function Toy



(c) Bucket Filling Toy

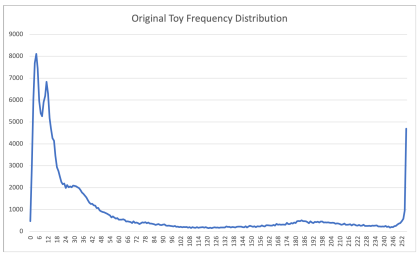
Figure 2: Histogram Equalization on Toy Image

1.5 Discussion

Q(a-1) - Please look above in section "1.4 Experimental Results" to see the results of the bilinear demosaicing algorithm and its comparison to House_ori.

Q(a-2) - With respect to the demosaicing problem. One can see that bilinear interpolation does a great job. However, if you were to look closely, you would notice that there are artifacts that appear around the edges of objects in the demosaiced image. The best explanation for this is because the bilinear interpolation algorithm does not take into account the correlation among the RGB values. We need to use the values of the adjacent pixels as well as the current value of our pixel. In order to improve these results, we must take into account spatial and spectral correlation of our pixel values.

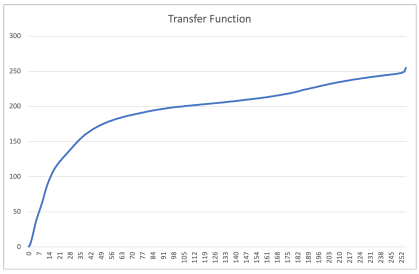
Q(b-1) - The histogram plot of the original image is as follows.



(a) Original Toy Histogram

Figure 3: Histogram of original Toy image

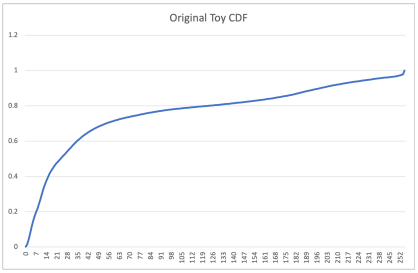
Q(b-2) - The transfer function plot is as follows.



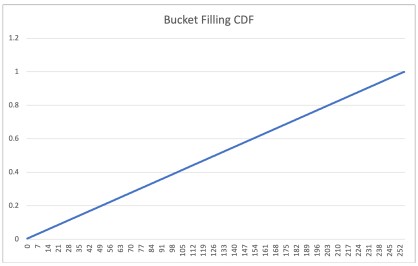
(a) Transfer Function

Figure 4: Transfer Function plot

Q(b-3) - The CDF's for the original toy image and after applying method B are as follows.



(a) Original CDF



(b) Method B CDF

Figure 5: CDF Comparison Plot

Q(b-4) - When observing the two histogram equalization methods it is clear, at least to me, that the transfer function based method outperforms cumulative probability based method. I believe this is because. The bucket filling method artificially imposes a uniform distribution while the transfer function method relies on the mathematics of probability theory to transform the current distribution into a uniform distribution. While both methods do their job well, I believe the transfer function based approach is slightly "smoother" with the transitions from light to dark (or dark to light depending on the images contrast).

2 Image Denoising

2.1 Motivation

Typically with image processing, and the process of acquiring digital photos in general, there is a certain amount of noise that naturally comes along with it. Noise contamination can come from either intrinsic (ex. sensor) or extrinsic (ex. environment) sources. This is simply an unavoidable reality in most practical applications. For this reason, the ability to suppress the underlying noise is a very important (and still open) problem.

That being said there are many great existing techniques for dealing with noise. For the purposes of our course, we focus on three. Those being a linear filter (both uniform and gaussian), a bilateral filter, and a non-local means (NLM) filter.

2.2 Approach and Procedures

2.2.1 Peak signal-to-noise ratio (PSNR)

Again before we dive into the details of the denoising applications, it is useful to develop the idea of the peak signal-to-noise ratio (psnr) as it is a very useful quantitative way to measure the efficacy of a denoising method. The psnr is

the ratio of the maximum possible power of a signal and the power of the noise that distorts it.

Mathematically this can be seen as:

$$PSNR = 10log_{10}(\frac{MAX^2}{MSE})$$

Where the MSE can be calculated as:

$$MSE = \frac{1}{MN}\sum_{i=1}^{N}\sum_{j=1}^{M}[Y(i,j) - X(i,j)]^2$$

Here, MAX is our maximum possible value (usually 255)
$Y(i,j)$ is our filtered image of size NxM,
and $X(i,j)$ is our original image of size NxM.

Mathematically we can see that a higher pnsr value corresponds to a lower mse value and thus a lower "error" between our filtered and original images.

2.2.2 Linear Denoising

The linear approach to denoising involves using a lowpass filter as most image content is low frequency and most noise content is high frequency. One side affect of this approach however is that because edges also contain high frequency, they are also blurred in the denoising process. We explore two different types of lowpass filters, those being the uniform filter and the gaussian filter. The filters are all NxN with $N > 1$ and odd.

To implement the linear filter I used convolution. Naturally with convolution I also had to pad the original image in order to retain its size. The amount of border padding can be calculated by using the $filter\_size$ variable.

$$pad\_size = \frac{filter\_size - 1}{2}$$

After constructing a padded image, I loop through the new padded image and convolve the padded image with the linear filter to get the output for the denoised image.

$$I_{out}(i,j) = I_{pad}(i + pad\_size, j + pad\_size) * w$$

The **uniform** filter works as such. If we assume that the pixel to be filtered is in the middle of our NxN filter, we simply take the average of all of the values within the filter.
Each value of our filter has the same weight and can be calculated as:

$$f(i,j) = \frac{1}{N^2}$$

The **gaussian** filter works in the same way. The only difference being in how the filter weights are calculated. As the name would suggest the weights are calculated according to a gaussian distribution.

$$f(i,j) = \frac{1}{2\pi\sigma^2} e^{-\frac{i^2+j^2}{2\sigma^2}}$$

Where σ is the variance of the gaussian distribution and a controllable parameter, and i and j are the location within the filter with the center of the filter corresponding to $(i = 0, j = 0)$.

2.2.3 Bilateral Filtering

The bilateral filter can be thought of as an extension of the gaussian filter in a way. The main motivation for its construction was to develop a non-linear filter that allows the preservation of edges. It works like the previous filters by replacing a pixel's intensity value with a weighted average of its neighbors. Crucially, the weighting includes radiometric differences (ie. color intensity, depth distance, etc.) which is why it preserves edges.

To develop some more intuition behind bilateral filtering, it's useful to look at the equation.

$$BF[I]_p = \frac{1}{w_p} \sum_{q \in S} G_{\sigma_c}(||p-q||) G_{\sigma_s}(I_p - I_q) I_q$$

Where S is the set of neighbors (depending on what the neighborhood_size is set to) and I_p and I_q are the intensity values of p and q respectively.
G_{σ_c} is the spatial gaussian that decreases the influence of distant pixels
G_{σ_r} is the range gaussian that decreases the influence of pixels, q, with an intensity value different than I_p.

The equation can also be written as:

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

where the weight:

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

Following the previous intuition we developed we can see that:
σ_c is the spatial weight controlling the spatial gaussian, and
σ_s is the range weight controlling the range gaussian.

Implementing this was relatively straightforward. I first pad the image based on what the users desired $neighborhood\_size$ value is to ensure that the output remains the same size as the original image. Where, $pad\_size = \frac{neighborhood\_size-1}{2}$. In the main loop I then call a function to calculate the bilateral filter's value for each pixel location in the padded image.

The bilateral filter function works by looping from our current pixel location (i,j) over the entire neighborhood region (k=i-neighborhood_size, l=j-neighborhood_size) to (k=i+neighborhood_size, l=j+neighborhood_size). For each neighbor location (k,l), I then call another function to calculate the $bilateral\_weight$ w(i,j,k,l).

The bilateral filter function then takes all of these bilateral weight outputs and uses them to construct the bilateral output. It does this by summing over all of the $w\_I = \sum_{k,l} w(i,j,k,l) * I(k,l)$ values and also constructing a sum of weight values, $w\_sum = \sum_{k,l} w(i,j,k,l)$, the output is then $\frac{w\_I}{w\_sum}$.

2.2.4 Non-Local Means Filtering

The Non-Local Means (NLM) filter is another non-linear filter that incorporates information from the entire image. As with the bilateral filter, the non-locality assures less loss of detail and thus greater preservation of edges. The NLM filter attempts to find patterns of pixels over the image and then average all of these self similarities. If we take similar regions and average over all of them, we should get a denoised version of that region in return. The more similar a region is to the region in question, the higher of a weight it contributes.

The weight of the convolution kernel for NLM is:

$$w(i,j,k,l) = exp[\frac{||I(W_{i,j}) - I(W_{k,l})||^2_{2\sigma}}{h^2}]$$

and

$$||I(W_{i,j}) - I(W_{k,l})||^2_{2\sigma} = \sum_{x,y \in W} G(x,y)||I(i-x, j-y) - I(k-x, l-y)||^2$$

and

$$G(x,y) = \frac{1}{\sqrt{2\pi}\sigma} exp[-\frac{x^2+y^2}{2\sigma^2}]$$

Where W is the nearby pixel region, I is the input image, and σ and h are parameters.

For my NLM implementation, I decided to use the open source code available in MATLAB's Image Processing Toolbox. MATLABS NLM allows us to change three parameters: (1) DegreeOfSmoothing, (2) SearchWindowSize, and (3) ComparisonWindowSize.

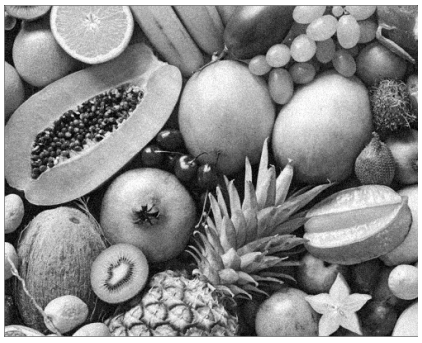
2.2.5 Mixed Noises in Color Image

When working with color images there are two different approaches to denoising. The first is to apply a denoising method, any one that we have just talked about would work, to each color channel separately, and combine the denoised channels into a final image. The second approach is to use the relationship information between the color channels as a means of potentially doing a better job.

You can calculate the psnr value of a color image representation by taking the psnr value of each channel individually. You can then choose to look at each channel's psnr value or take the average.

2.3 Experimental Results

Before showing the results, it will be useful to see what the noisy and clean images both look like:



(a) Fruits Noisy

(b) Fruits Clean

Figure 6: Noisy and Clean Grayscale Fruits Image to be Denoised

Now, let's observe the effects of the linear uniform filter. As I have stated, this method obtained very poor results. I have opted to include the 3x3 filter because it did the best, as well as the 5x5 filter to show how drastic the drop off in quality is when sizing up a single level.

(a) 3x3 Uniform Filter

(b) 5x5 Uniform Filter

Figure 7: Linear Uniform Denoising

The linear gaussian filter yielded much more favorable results. Here I show both the 3x3 and 5x5 filter each with a σ value of 0.5, this was the best σ value for each respective filter. I opted to show both because they both performed similarly well.

(a) 3x3 Gaussian Filter

(b) 5x5 Gaussian Filter

Figure 8: Linear Gaussian Denoising

Finally, I will show the results from the non linear methods, each with the best parameters from the respective method. Those methods are bilateral filtering and non-local mean filtering.

(a) Bilateral Filter

(b) NLM FIlter

Figure 9: Non-Linear Denoising

These methods performed very well with the NLM filter achieving the highest psnr score. However, looking at the images, even though the gaussian filter got a higher psnr score than the bilateral filter (albeit ever so slightly). The bilateral filter appears more visually appealing to me as it is less grainy.

Following are the noisy and clean color fruit images followed by the deniosed fruit image that my cascade of median and gaussian filters produced.

(a) Fruit Color Clean

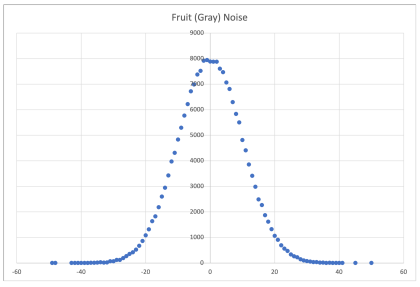
(b) Fruit Color Noisy

(c) Cascaded Denoised

Figure 10: Clean and Mixed Noise Color Fruit Followed by Denoised

2.4 Discussion

Q(a-1) - The type of embedded noise is gaussian, as you can see from the plot of the noise distribution below. This is also inline with our experimental findings.



(a) Noise Distribution

Figure 11: Noise of Gray Fruit Image

Q(a-2) - After experimentation, I found the best gaussian filter to be of size 3x3 with a σ value of 0.5, this combination boasted a psnr value of 30.335. I believe it is worthwhile to note that the 5x5 gaussian filter with the same σ value got a 30.333 psnr score, so it only did ever so slightly worse. After testing out different uniform filter sizes, I found the psnr value to fluctuate quite drastically from one size to the next. The best scoring size was a 3x3 uniform filter that got a psnr score of 26.246. From here it was only downhill has the psnr value got significantly worse with any increase in size.

Q(b-1) - Please refer to section "2.3 Experimental Results" to see the result of applying the bilateral filter on the noisy fruit image as well as a comparison with the original image.

Q(b-2) - As I stated earlier in the report when talking about the bilateral filter (see section 2.2.3) G_{σ_c} is the spatial gaussian that decreases the influence of distant pixels G_{σ_r} is the range gaussian that decreases the influence of pixels, q, with an intensity value different than I_p. Thus, σ_c is the spatial weight that controls the spatial gaussian and σ_s is the range weight that controls the range gaussian. Through adjusting the values, I saw that increasing σ_c will smooth large features and increasing σ_s will make the filter closer to gaussian blur.

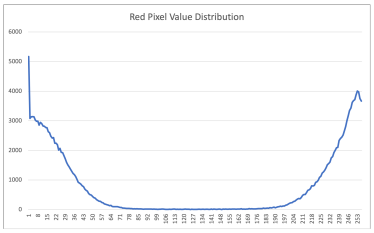
Through experimentation, I found the best σ_s value to be 20 and the best σ_c value to be 10 with a neighbor radius of 3 (ie. a 7x7 area). This is slightly inline with the filter creator's observations that generally the σ_s value needs to be, on average, slightly greater than twice the value of σ_c. The psnr value achieved with these parameters was 30.0912.

Q(b-3) - Somewhat surprisingly the filter did not outperform the simple linear gaussian filter. The optimal gaussian filter achieved a slightly higher psnr value, although the results were very close. This closeness in psnr however would tempt me to use the gaussian filter for denoising this image as it is simpler implement and slightly quicker to run. The bilateral filter also heavily outperformed all version of the linear uniform filter.

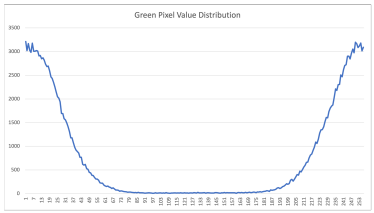
Q(c-1) - With the MATLAB implementation of the NLM filter we are allowed to adjust three parameters. The DegreeOfSmoothing parameter is self explanatory, as we increase that the resulting denoised image becomes smoother and smoother. This decreases noise, but also heavily distorts edges. The second parameter, SearchWindowSize is useful, however past a certain point it loses its benefits of image denoising and increases the runtime of the NLM algorithm. The third and final parameter, ComparisonWindowSize, operated much like SearchWindowSize. After a certain point it too lost its efficacy and only started hindering the runtime. From my experiments I found the optimal values to be: (DegreeOfSmoothing = 11.85, SearchWindowSize = 17, and ComparisonWindowSize = 3). This combination of parameters resulted in a psnr value of 32.1126.

Q(c-2) - The NLM filter by and large performed the best out of all of the denoising filters. With optimal parameters, it acheived a psnr score of 32.1126.

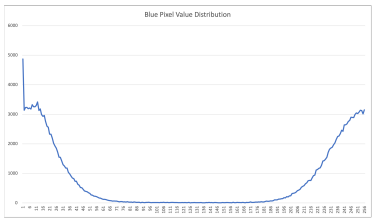
Q(d-1) - From observing the noise distributions from each color channel we can see that they all follow a roughly similar pattern. This being an inverted gaussian with spikes on either end of the intensity spectrum. For this reason, I believe the type of noise that we are dealing with in this image is a mixture of salt and pepper noise with some gaussian noise.



(a) Red Channel



(b) Green Channel



(c) Blue Channel

Figure 12: Noise Distribution Across RGB Channels

Q(d-2) - The types of filters that I would like to use are (1) a median filter to deal with the salt and pepper noise, and (2) a gaussian filter. You can casacade these filters in any order, however I believe you should first apply the median filter before the gaussian filter. This is because you have to remove the impulse noise first, if you don't and say apply the gaussian filter first, you are just making it more difficult for the median filter to localize the impulse noise.

Q(d-3) - I decided to filter the image on each color channel separately, and then bring them all together in the end. My method was to first perform median filtering using a 3x3 neighborhood of adjacent values. After using the median filter, I decided to use a gaussian filter on top of that. The results of which can be seen above in section "2.3 Experimental Results".

Through experimentation I found the best gaussian filter to be one with a size of 3x3 with a σ value of 0.6. The psnr associated with this filter was 25.226. The psnr was calculated across each color channel and then averaged.

The results were not entirely satisfactory for me. I believe I made a correct initial step my using a median filter. However, the use of a gaussian filter, in hindsight, could have been the shortcoming here. If I were to improve upon this I think I would instead opt to cascade the median filter with a NLM filter so as to not blur the edges too much. I think I would also try experimenting with using the other color channels to assist in the denoising process rather than localizing it to a single channel at a time.

3 Special Effect Image Filters: Creating Oil Painting Effect

3.1 Motivation

One useful and fun task of image processing is the ability to create a filter that we can apply to an image to produce some sort of special effect. This type of thing can have many use cases as the limits of its efficacy can be constrained by our imagination. One obvious use case would be the construction of fun and interesting filters to overlay on a photo that we have taken. Another interesting application could be to use these special effect filters as a way of transferring the image into another "domain" to have a more robust training set for a machine learning problem.

In the end though, it is nice to be able to apply a special effect on an image that gives it a different appearance. While the task of creating these sorts of filters is not an entirely difficult one, it is nuanced and does require a bit of thought to make an effective filter that gives a visually pleasing output.

3.2 Approach and Procedures

The process of constructing a filter that gives off the effect of an oil painting requires two steps. In the first step we quantize the current image intensity spectrum for each color into a smaller one. For example if we wanted our output image to only contain 64 colors, then each color channel (RGB) will have 8 values as $8^3 = 63$. A sub-process of the first step is to also select a method for giving a new value for each quantization region. The second step is to take our new quantized image and apply an NxN filter on it. The most frequent value in this NxN neighborhood will be the new value of our output image.

I implement the outlined steps below on each color channel. Thus, I am seperately quantizing and most frequent neighbor mapping for each color channel to construct the output "oil painted" image.

To implement **step one**, I recycled my code that I used for the bucket filling method of histogram equalization (to see details of method please refer to

section 1.3.5). Afterwards I am left with a vector of length 256 where each index of the vector corresponds to an intensity value. Each index of the vector also contains a variable size vector containing location pairs (x,y) of where exactly in the input image these intensity values occur. Let's call this vector *color_hist*.

The next step is to pass this vector into another function that I use to quantize the bins. I do this by constructing another vector of pairs, let's call this one *quant_bins*. Except now the vector (*quant_bins*) is of size *num_bins*, where $num\_bins = 4$ *or* 8 (depending on *num_colors*). I loop through *color_hist* and add each (x,y) pair location into my new vector of *quant_bins*. I do this by keeping track of which bin I'm on using a integer that I initialize to 0. Once I fill up the bin with the desired range of intensity values, I increment the integer to start filling up the next bin. The desired range of intensity values is either 64 if we are using $num\_colors = 64$, or 32 if we are using $num\_colors = 512$.

After filling *quant_bins* I calculate the respective value for each bin by taking a weighted mean of the pixel intensity values. For example, if we have 8 pixel values with 3 pixels having an intensity of 2 and 5 having an intensity of 6. The weighted mean would be $\frac{(3*2)+(5*6)}{8}$.

To implement **step two**, I first mirror pad the image (where the padding is decided by the value of our desired NxN neighborhood). Then, I loop through the mirror padded image and construct a vector to hold all the intensity values within the NxN neighborhood. I sort this vector, then I use the sorted vector to determine the most frequent value. Then, $I_{out}(i,j) = most\_frequent[I_{pad}(i + pad\_size, j + pad\_size)]$

3.3 Experimental Results

The following images are after implementing step 1 and quantizing the images. I have included both the 64 and 512 color versions for both Fruits.raw and Fruits_noisy.raw.

(a) Clean 64-color

(b) Clean 512-color

(c) Noisy 64-color

(d) Noisy 512-color

Figure 13: Quantized Images of Clean and Noisy Fruit

Now, are the images after applying step 2, most frequent neighbor in NxN window. For the clean and noisy images for various sizes of N (3 and 7) and for 64 and 512 colors. I have decided to only show two sizes as I think the difference between 3 and 7 is adequate enough to see any dissimilarities that arise from increasing the filter size.

(a) Clean 64-color 3-N

(b) Clean 64-color 7-N

(c) Clean 512-color 3-N

(d) Clean 512-color 7-N

Figure 14: Oil-Painting effect on clean images

(a) Noisy 64-color 3-N

(b) Noisy 64-color 7-N

(c) Noisy 512-color 3-N

(d) Noisy 512-color 7-N

Figure 15: Oil-Painting effect on noisy images

3.4 Discussion

Q(a-1) - The 64-color version can be seen in section "3.3 Experimental Results", and the quantization method was described in detail in section "3.2 Approach and Procedures".

Q(a-2) - The results from N=3 and N=7 can be seen in section "3.3 Experimental Results". I believe that asking which one "gives better results" is very subjective and depends on what you are expecting out of the filter. If you want some preservation of detail then you might prefer a smaller neighborhood size. However, if you really want bleeding of borders and to get a very "wet" looking oil painting then you would instead opt for a larger neighborhood size.

Q(a-3) - All the corresponding images can be seen in section "3.3 Experimental Results". As one can see, if we increase the number of colors, we get a more detailed image as we allow for more preservation across the color channels. Thus, with an increased number of colors we would need a higher neighborhood size to achieve similar results to a lower number of colors with a smaller neighborhood

size. Again, this is very subjective, if you want more preservation of detail with a lingering oil-painting effect overlaid on the image, then you might prefer to increase the number of colors.

Q(a-4) - All the corresponding images can be seen in section "3.3 Experimental Results". From what I saw the quantization step did not seem to have much effect on the noisy image. This is very interesting as it shows there is a direct relationship between distribution of noise in an image and how smoothly you can quantize the image into bins. I believe because the noisy image has salt and pepper noise, the quantization does not appear to be doing anything as the frequency distribution of the quantized image will still have salt and pepper noise itself.

Interestingly after applying the NxN most frequent filter, the noisy fruits photo starts to take on the appearance of an oil-painted photo. With a smaller neighborhood size there are still many noise artifacts. Hoewver, once you increase the neighborhood size, the salt and pepper noise becomes less and less dominant as the frequency of the normal un-corrupted pixels starts to take over and you converge onto what looks like a norma oil-painting effect. With the noisy photo increasing the number of colors similarly makes us need to increase the neighborhood size to see any results that we would attribute to "oil-painting".