EE569 HW#2

Student Name: Shi-Lin Chen
Student ID: 2991911997
Student Email: shilinch@usc.edu
Submission Date: Feb 16, 2020

Problem 1: Edge Detection

1.1 Motivation and Abstract

In this problem, we will implement Edge Detection with different method, including (a.) Sobel Edge Detector (b.) Canny Edge Detector (c.) Structure Edge. In the end we will evaluate our performance of edge maps in part(d.). Through this problem, we will learn several ways to detect and output the edge of images.

1.2 Approach and Result

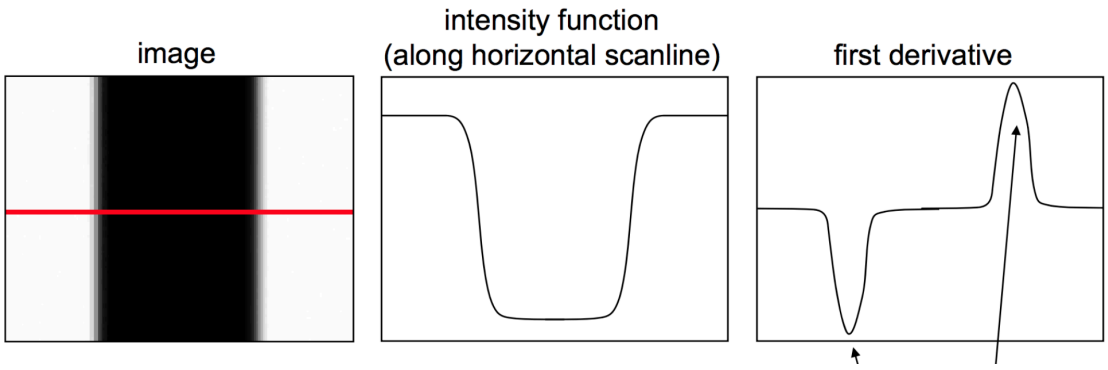
1.2(a.) Sobel Edge Detector

This method requires grayscale image, so we need to convert both RGB value Image to grayscale image first, with this formula:

$$Y = 0.2989 * R + 0.5870 * G + 0.1140 * B$$

Then we could start out implementation on image.

First of all, we need to understand what form edges in the image.



We could observe the edges in the image, when the pixel values variates intensely. In order to capture these edges, we need to check the gradient of pixels!

Gradient

$$\nabla f = \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$

Magnitude

$$|\nabla f| = \sqrt{g_y^2 + g_x^2}$$

a	b	c
d	e	f
g	h	i

In Sobel edge detection, we need to compute four **directional derivative** adjacent the center pixel "e". First, we need to sum them up and get the average of four directional derivative which are (a-i),(b-h),(c-g)(f-d). The computing method of directional derivative is to subtract the value of two pixel (ex: value of a – value of i) and divided by their distances (nearby pixels are 2, diagonal pixels are 4). In order to get the gradient, we still need to multiply **derivation's direction** to directional derivative.

>> G = (c-g)/4 * [1, 1] + (a-i)/4 * [-1, 1] + (b-h)/2 * [0, 1] + (f-d)/2 * [1, 0].

When we multiply 4 to this G we will get:

>> G' = 4*G = [c-g-a+i + 2*(f-d), c-g+a-i + 2*(b-h)], that how the coefficients of Sobel filter come from!

-1	0	+1
-2	0	+2
-1	0	+1

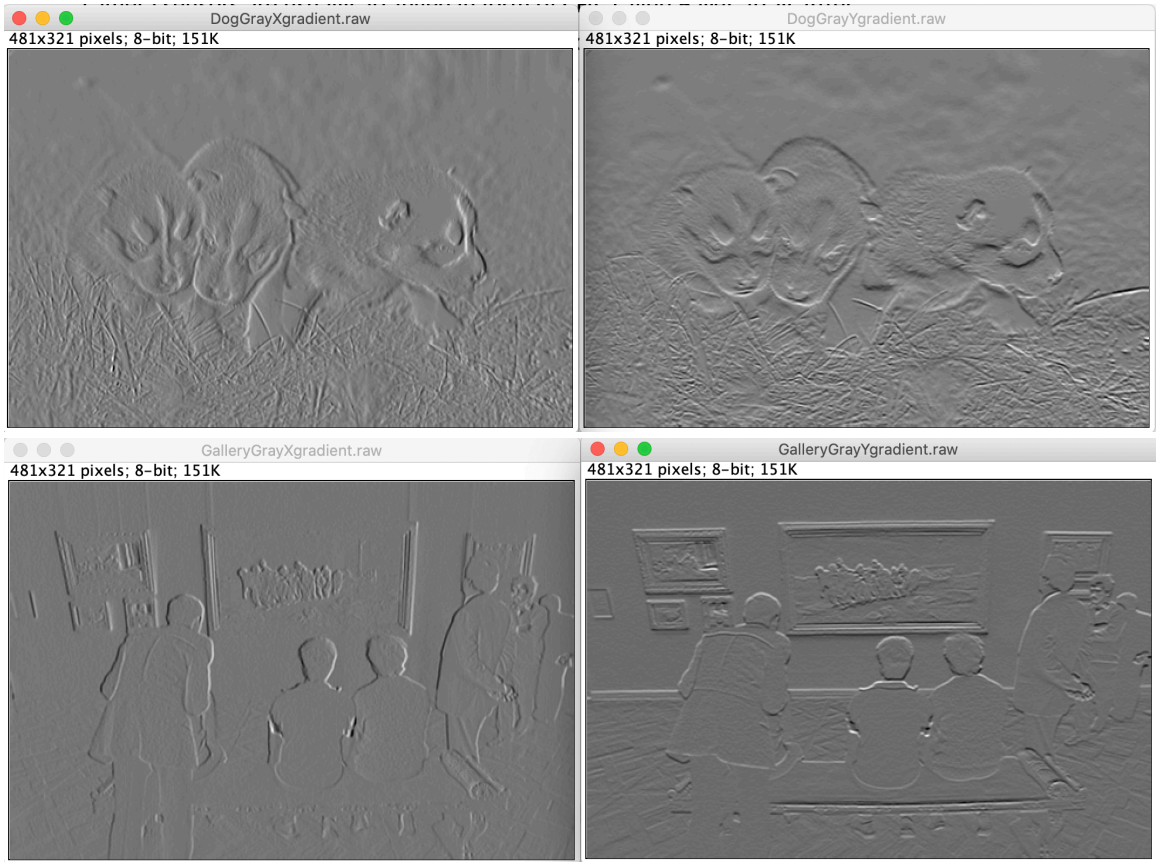
Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

Step 1: Computer the finite difference at the center pixel by convolution, we can get X-gradient and Y-gradient of the Image in this part. In order to visualize this step, we need to implement minmax normalization to both gradient. **Output = (input -min)/(max-min)**

These are converted grayscale images:

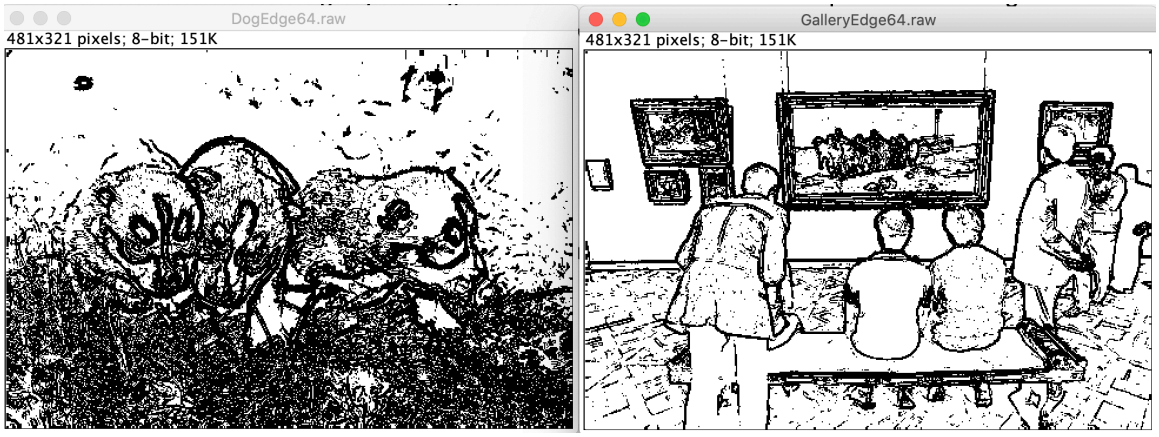


These are X,Y-gradient result and are also the result of Q(1)!

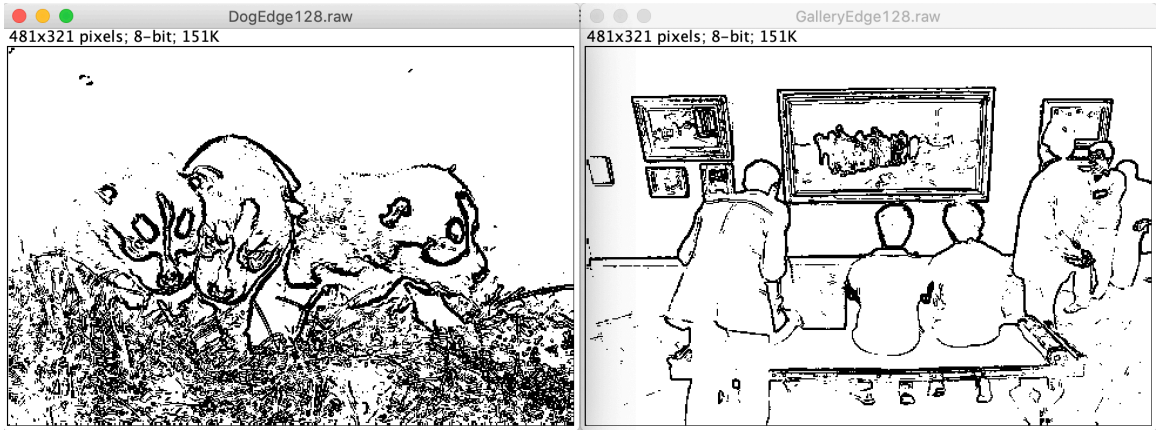
Step 2: we need to calculate the magnitude of both directional gradient by sum x-gradient square and y-gradient square and then square root the sum. Then we will get the magnitude map of the image, which shows below:

Step 3: Last, we need to set the threshold and display the edge of the images. In this step, I try different thresholds with (0.25, 0.5, 0.75 and 0.85), if the pixel value is greater

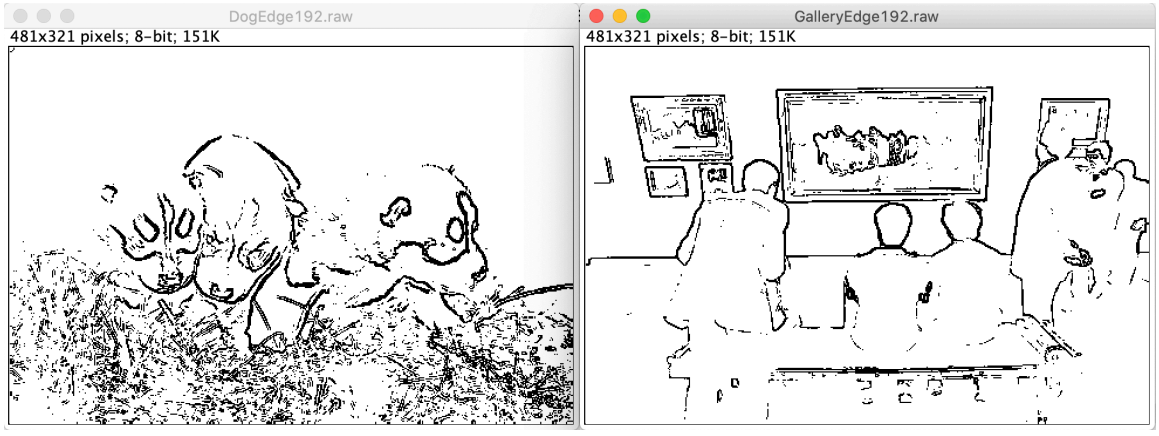
than threshold, set it to 0(black, edge), otherwise, set it to 255(white, background). Then we can get:



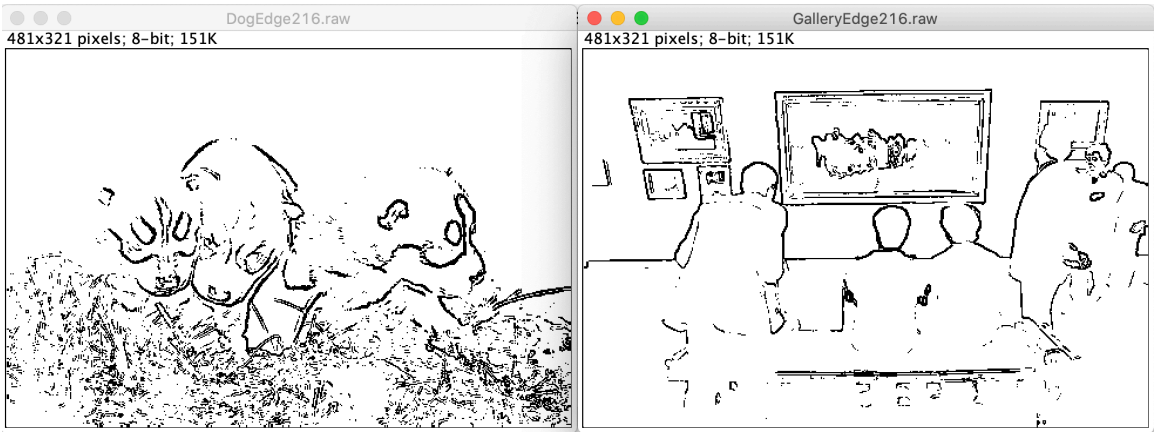
Threshold value = 64 (25%)



Threshold value = 128(50%)



Threshold value = 192(75%)



Threshold value = 216(85%)

These are the results of Q3(in different threshold).

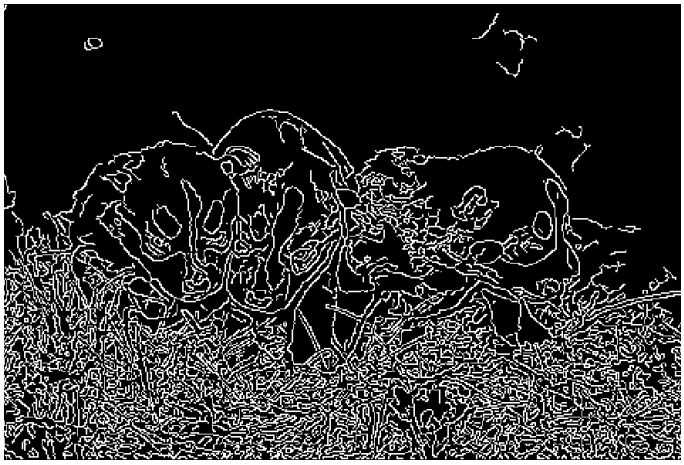
1.2(b.) Canny Edge Detector

There are five steps in Canny edge implementation, which are Gaussian filter, finding the gradient, non-maximum suppression, double threshold and edge tracking by hysteresis. The first step, gaussian filter will filter out the noise existing in the image. Second step need to find the gradient of the image, which are similar to the process used in Sobel. Non-maximum suppression is an edge thinning technique, it is applied to find the "largest edge". Then double threshold could filter out edge pixels with a weak gradient value and preserve edge pixels with high gradient values. The final step, edge tracking by hysteresis, could track the edge connection. It will track weak edge pixels and see if they are connected to string edge pixel or not, if they do, then they might be the true edge pixel, otherwise, they might be noises!

In this implementation, I used matlab built tool to implement canny edge detector.

The results are showed below: (Dogs w/ different sigma, Gallery w/ different threshold)

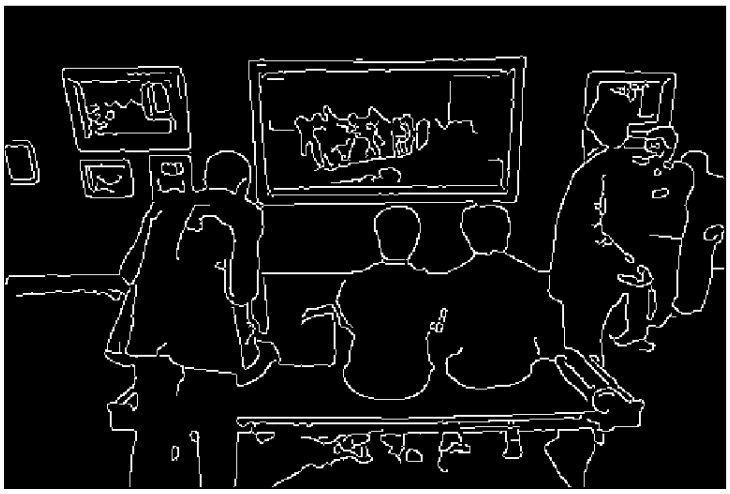
With double threshold [0.0563, 0.1406] and sigma = 1.414 (default)



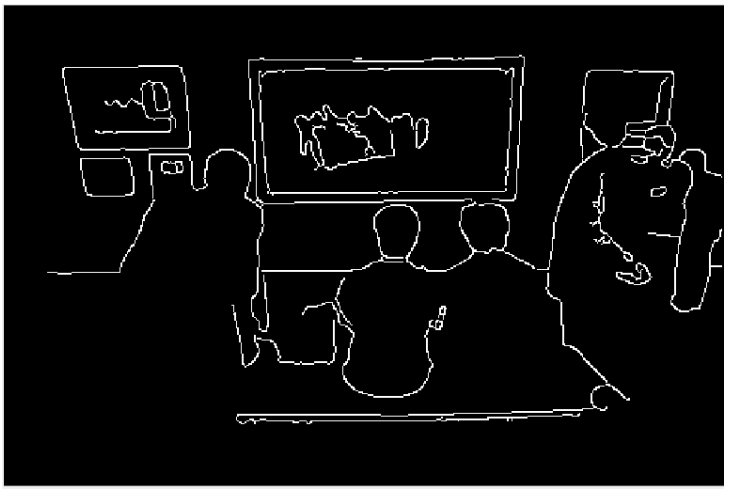
With same threshold but sigma = 0.8

With same threshold but sigma = 1.8

With threshold [0.0313, 0.0781], sigma = 1.4



With threshold [0.0313, **0.2**], same sigma (=1.4)



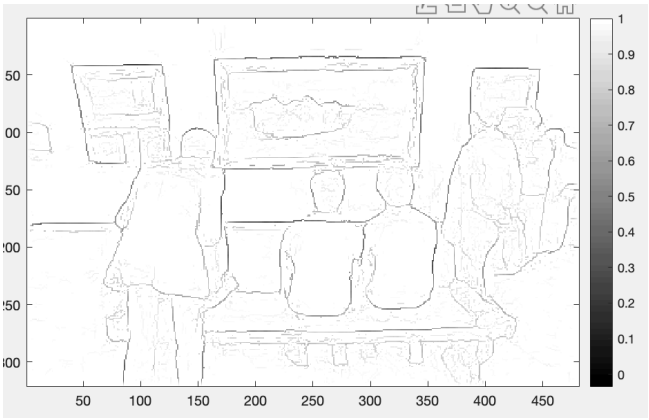
With threshold [0.0313, **0.4**], sigma = 1.4

With threshold [**0.05**, 0.0781], sigma = 1.4

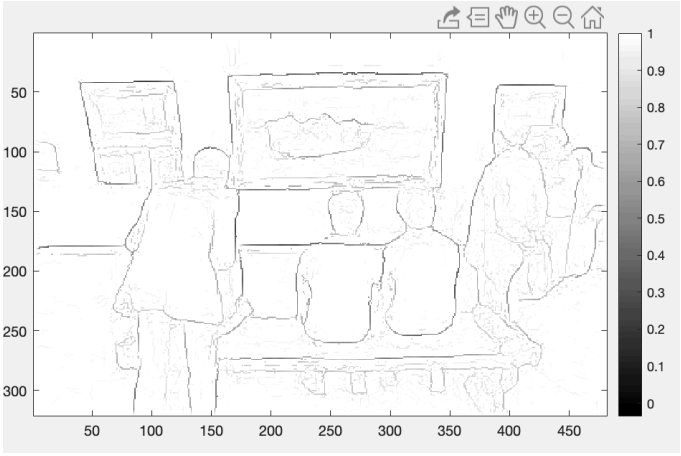
The comparison and answers to the question will be discussed in discussion part!

1.2(c.) – Structure Edge

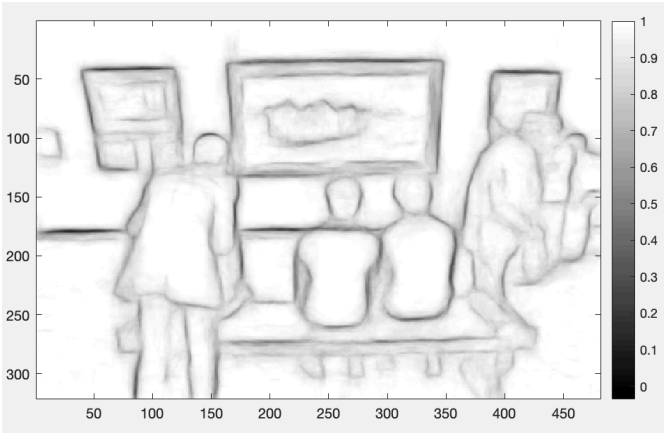
I use open source code from MATLAB: http://github.com/pdollar/edges for this problem! (n Thread below are four



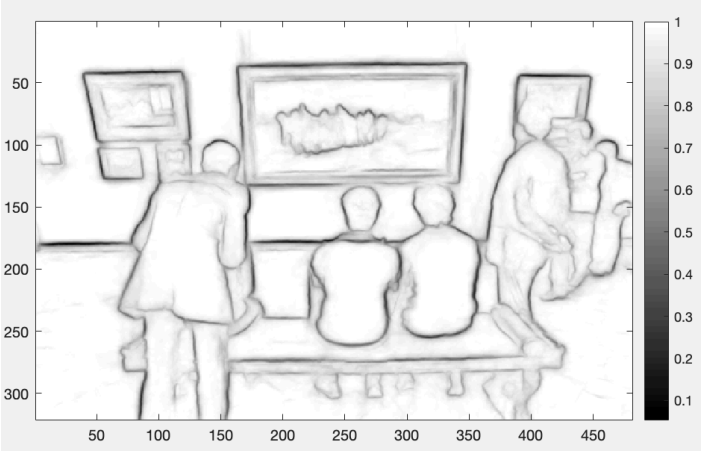
Multiscale = 1, sharpen = 0, n, TreesEval = 1, nms = 1



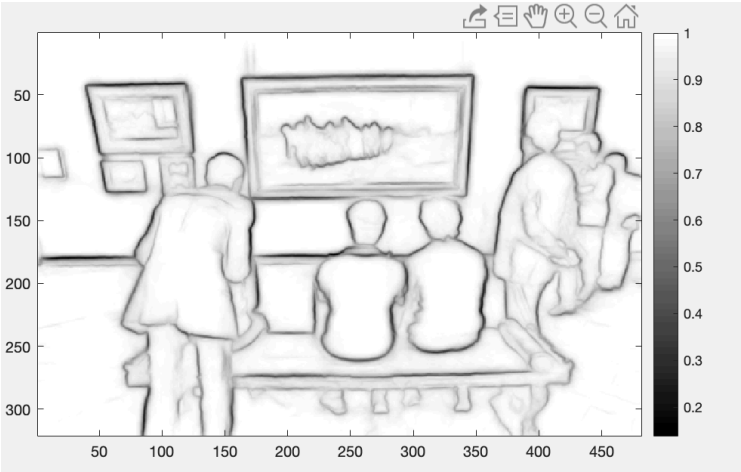
Mutliscale = 1, sharpen = 1, n TreesEval = 1, nms = 1



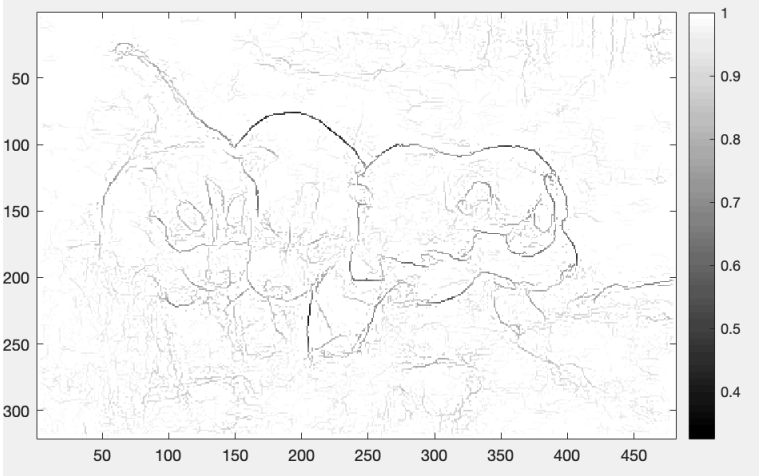
Multiscale = 1, sharpen = 0, n, TreesEval = 1, nms = 0



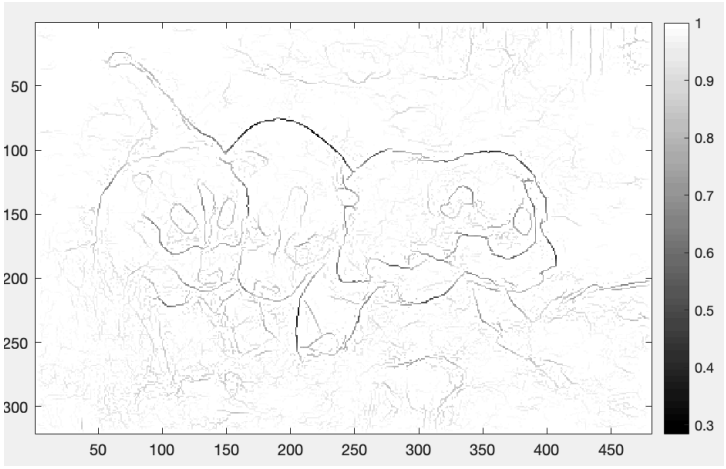
Multiscale = 1, sharpen = 4, n, TreesEval = 1, nms = 0



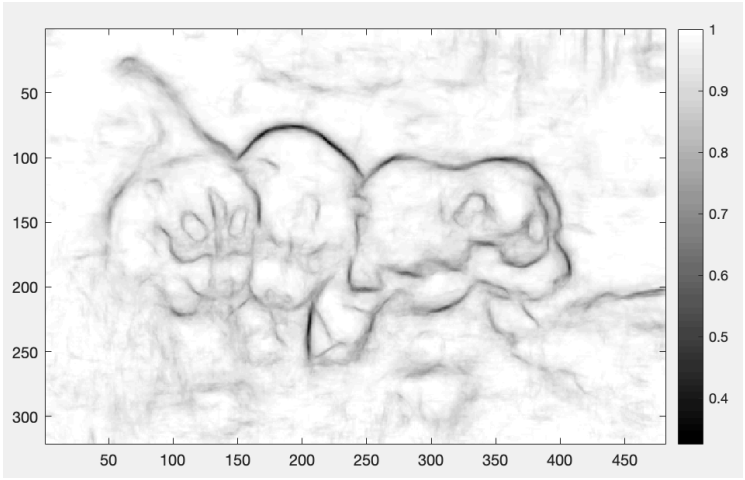
Multiscale = 1, sharpen = 4, n, TreesEval = 4, nms = 0



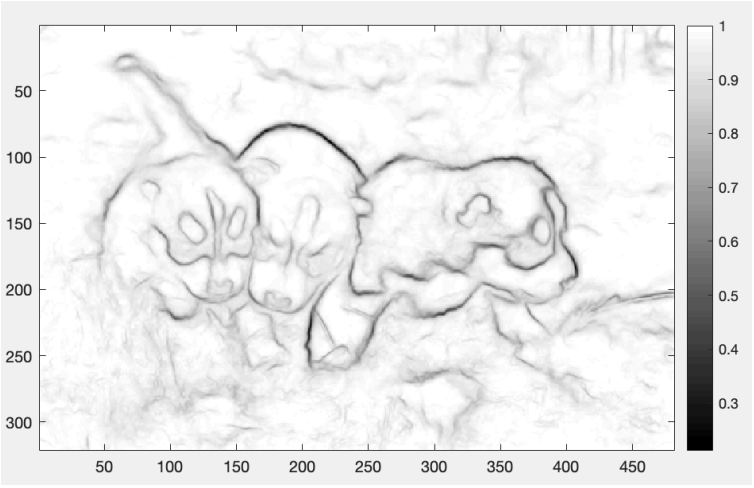
Multiscale = 1, sharpen = 0, n, TreesEval = 1, nms = 1



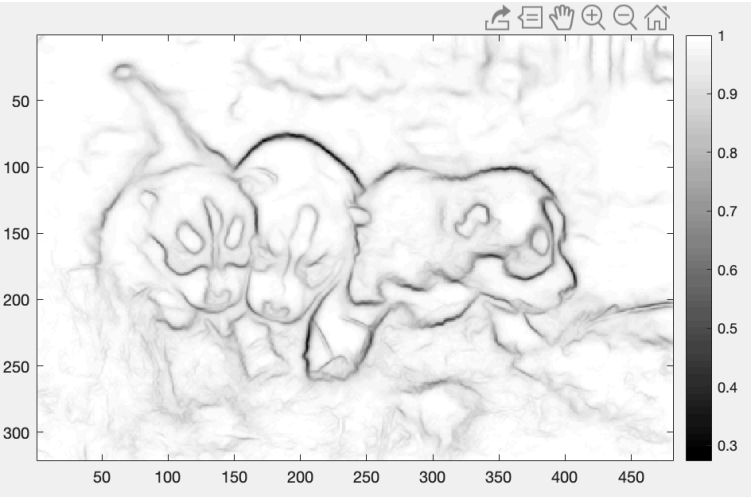
Mutliscale = 1, sharpen = 1, n TreesEval = 1, nms = 1



Multiscale = 1, sharpen = 0, n, TreesEval = 1, nms = 0



Multiscale = 1, sharpen = 4, n, TreesEval = 1, nms = 0



Multiscale = 1, sharpen = 4, n, TreesEval = 4, nms = 1

1.4(d.) – F measure

Sobel Edge Detector								
	Dogs				**Gallery**			
GT	**Precision**	**Recall**	**F**	**Mean**	**Precision**	**Reca**	**F**	**Mean**
1	**0.046**	**0.864**	**0.087**	**0.124**	**0.175**	**0.929**	**0.295**	**0.288**
2	**0.074**	**0.663**	**0.173**		**0.158**	**0.939**	**0.268**	
3	**0.070**	**0.901**	**0.125**		**0.170**	**0.946**	**0.282**	
4	**0.038**	**0.843**	**0.074**		**0.159**	**0.953**	**0.269**	
5	**0.094**	**0.922**	**0.1593**		**0.234**	**0.945**	**0.369**	

Canny Edge Detector								
	Dogs				**Gallery**			
GT	**Precision**	**Recall**	**F**	**Mean**	**Precision**	**Reca**	**F**	**Mean**
1	**0.207**	**0.721**	**0.334**	**0.420**	**0.394**	**0.932**	**0.553**	**0.549**
2	**0.267**	**0.443**	**0.543**		**0.351**	**0.956**	**0.501**	
3	**0.233**	**0.565**	**0.432**		**0.372**	**0.932**	**0.531**	
4	**0.167**	**0.713**	**0.282**		**0.341**	**0.929**	**0.499**	
5	**0.436**	**0.792**	**0.507**		**0.507**	**0.938**	**0.664**	

SE Detector								
	Dogs				**Gallery**			
GT	**Precision**	**Recall**	**F**	**Mean**	**Precision**	**Reca**	**F**	**Mean**

1	**0.274**	**0.224**	**0.253**	**0.283**	**0.671**	**0.319**	**0.441**	**0.439**
2	**0.359**	**0.134**	**0.251**		**0.634**	**0.343**	**0.439**	
3	**0.314**	**0.168**	**0.317**		**0.627**	**0.314**	**0.427**	
4	**0.238**	**0.221**	**0.279**		**0.618**	**0.332**	**0.419**	
5	**0.472**	**0.187**	**0.314**		**0.819**	**0.313**	**0.472**	

1.3 Discussion

1.3(a.) – Sobel edge

From the result, we could observe that when the threshold is tuning up, the edge will be more clearly and more obvious, however the details of some regions in the image will be filtered out. I personally think the best visualization is depend on what kind of result we want to obtain and the image. For example, there are many trivial edges in *Dog.raw,* while there are many obvious edges in *Gallery.raw* (like the square, people shape.) Therefore, in *Dog.raw*, I think higher threshold is better to filter out unimportant edges. However, in *Gallery.raw*, lower threshold can keep some interesting details in image (like the style of floor and cloth shape).

1.3(b.) – Canny edge

(1.)

Non-maximum suppression helps this algorithm to "find the largest edges", it will suppress all the gradient value expect the local maximum, by setting the them to zero. Through this procedure, we will consider the direction of gradient and compare the edge strength, then we could preserve the right value and suppress others. By doing this, we could thin the edge!

(2.)

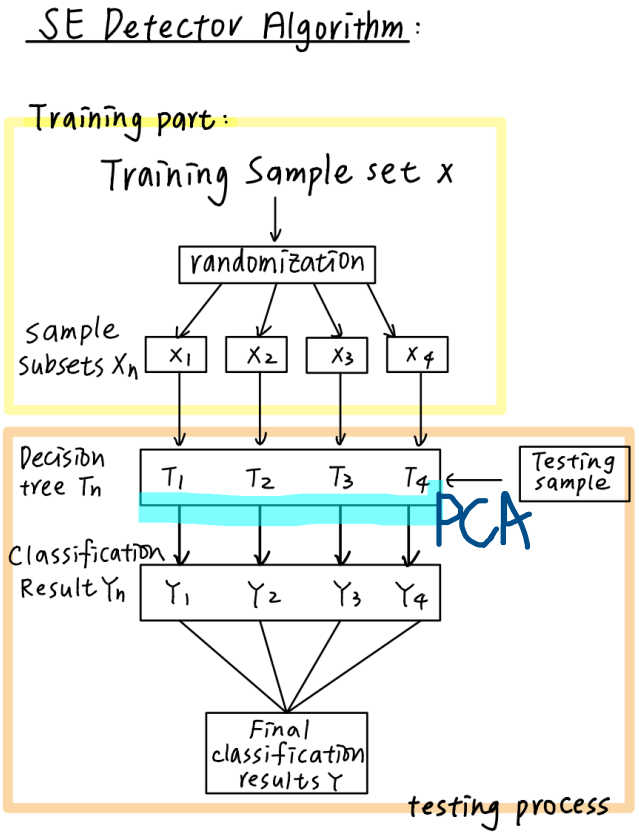
After NMS, remaining edge pixels are almost accurate of real edges in an image. However, some edge pixels remain that are caused by noise and color variation. Double threshold could filter out edge pixels with a weak gradient value and preserve edge pixels with a high gradient value. For example, there are two threshold ***A & B***, A is the lower threshold and B is the higher threshold. Pixel values greater than B will be marked as strong pixel value, if pixel values are greater than A but smaller than B, they will be marked as weak pixel value, otherwise, they will be suppressed!

(3.) **Different Threshold and Different Sigma** of canny edge

As the result I provided above, we can see how different threshold and sigma could affect the edge performance. Lower sigma will cause the edge more roughly, there are more not-to-important edges will be captured through the low sigma. Higher sigma could get more important edges in this case. As for the threshold part, we know there are how threshold and low threshold in this algorithm. If we adjust the higher threshold, we can see some of the edges will be filtered out, since there are less pixels to be marked as “strong pixel value”. (can easily observed the difference if we adjust the high threshold from 0.2 to 0.4). On the other hand, if we lower the low threshold, there will be less unimportant pixel to be suppressed. It might cause the algorithm preserve more unimportant edges or detail when implementing.

1.3(c.) Structured Edge

(1.) – SE detection algorithm explanation



Since SE detector is a data-driven edge detector, there are training part and testing part. First, we need a training sample set, then we choose some randomly into subsets. Then we have to build the structured decision trees, then we have to prepare training data and labels for training purpose. Later, we need to extract features (like gradients) and use classifier (structured random forest). Through this processing, we could train our

data and classify them to the result we desire.

(2.) – Decision Tree and RF classifier

Decision tree will split the source set into subsets based on an attribute value test. This process is repeated on each derived subset in a recursive manner and the recursion is completed when the subset at a node has the same value of the target variable, or when splitting no longer adds values to the prediction.

RF classifier is ensemble algorithm, which means it will combine more than one algorithm of same or different kind for classifying objects. RF classifier creates a set of decision tress from randomly selected subset of training set. Then it aggregates the votes from different decision trees to decide the final class of the best object.

(3.)

Compare to Canny detector, edges using SE detector seems more smoothly. Moreover, the edges obtained from SE detector have different intensity in the image! Take *Gallery.raw* for example, the important edges (like the shape of people and their shirts) are more intense than the edges which are not that important (like the detail of shirts or the detail of painting on the wall). Though SE detector, we could obtain more information of the image, and it seems more natural for visual experience.

Parameters:

From the result, we can see if we adjust nms from 0 to 1, the edges will be thinned, and the sharpen parameter also will affect a bit on edges. I adjusted sharpen from 0 to 4, the edges in gallery are more intense(darker)! Other parameters seems won't influence the image a lot.

1.3(d.)

1. F measure Result show on approach part!

Sobel

Pros:

easy to implement and fast.

Cons:

result are the worst (compare to others), thick and broken edges, threshold is hard to choose.

Canny

Pros:

Thin and good quality of edges.

Cons:

Not easy to implement, two thresholds should be chosen in order to have good

result.

Structured Edge

Pros:

Best results. Clean

Cons: Slow, take long time to train, hard to implement.

2. F measure

Gallery image has higher F measures than Dogs image.

Since the boundaries in Gallery are more obvious, while the boundaries in Dogs image are trivial (there are much grass near three dogs, which will be captured by the detector.) We can see these trivial edges among dog in the edges by Sobel edge detector!

3. Rationale

F measure is a combination of Recall(R) and Precision(P). The coefficient 2 means we give the same weights to Recall and Precision. There are four kinds of result, (true/false/positive/negative). R and P only take two of them into consideration, which might cause a biased result. That the reason why we need F measure, in order to have a overall consideration.

Problem 2: Digital Half-toning

2.1 Motivation and Abstract

In this problem, we will use different method to implement Image Halftoning. There are **Dithering** (Fixed Threshold, Random Threshold and Dithering Matrix), **Error Diffusion** (Floyd-Steingberg, JJN and Stucki) in grayscale Image. In part(c.) we will use **Separable Error Diffusion** and **MBVQ-based Error Diffusion** to deal with color halftoning. Image Half-toning could convert continuous tune image to discrete dot. (For example, halftoning could convert grayscale image from 0~255 to only 0(black) and 255(white)). It could preserve most details of the image! The motivation of Image Halftoning is that it has been applied on printer for a long time, this method could use less ink to print the image with close appearance.

2.2 Approach and Result

2.2.(a.) Dithering

In first part of Dithering, we will use **fixed threshold**. I set threshold as 128 in this problem. From the formula below, we can see if input greater than threshold, then change it to 255, otherwise, change it to 0. In the end, we will get a half tone image with only 0 and 255(black and white).

$$G(i,j) = \begin{cases} 0 & \text{if } 0 \le F(i,j) < T \\ 255 & \text{if } T \le F(i,j) < 256 \end{cases}$$

In second part, we will use **random thresholding** to generate half tone image, I use a c++ library syntax to get the random thresholds. It simply generate a random threshold every time when deciding a pixel set to 255 or 0.

$$G(i,j) = \begin{cases} 0 & if\ 0 \le F(i,j) < rand(i,j) \\ 255 & if\ rand(i,j) \le F(i,j) < 256 \end{cases}$$

```
std::default_random_engine generator;
std::uniform_int_distribution<int> distribution(0,255);
```

In third part, we will use **Dithering Matrix**, the value in an index matrix indicate how likely a dot will be turn on. Take I2 for example, where 0 indicate the pixel that is most likely to be turned on.

$$I_2(i,j) = \begin{bmatrix} 1 & 2 \\ 3 & 0 \end{bmatrix}$$ --(1)

$$I_{2n}(i,j) = \begin{bmatrix} 4 \times I_n(i,j) + 1 & 4 \times I_n(i,j) + 2 \\ 4 \times I_n(i,j) + 3 & 4 \times I_n(i,j) \end{bmatrix}$$ -- (2)

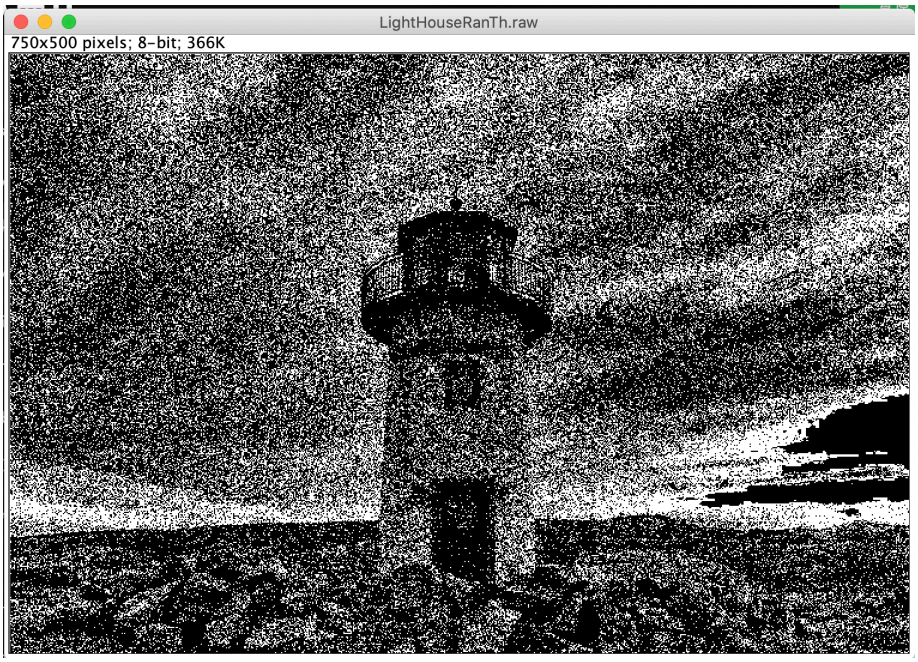
The larger size of Dither Matrix can be generated by this formula.

$$T(x,y) = \frac{I_N(x,y) + 0.5}{N^2} \times 255$$ --(3) threshold value.

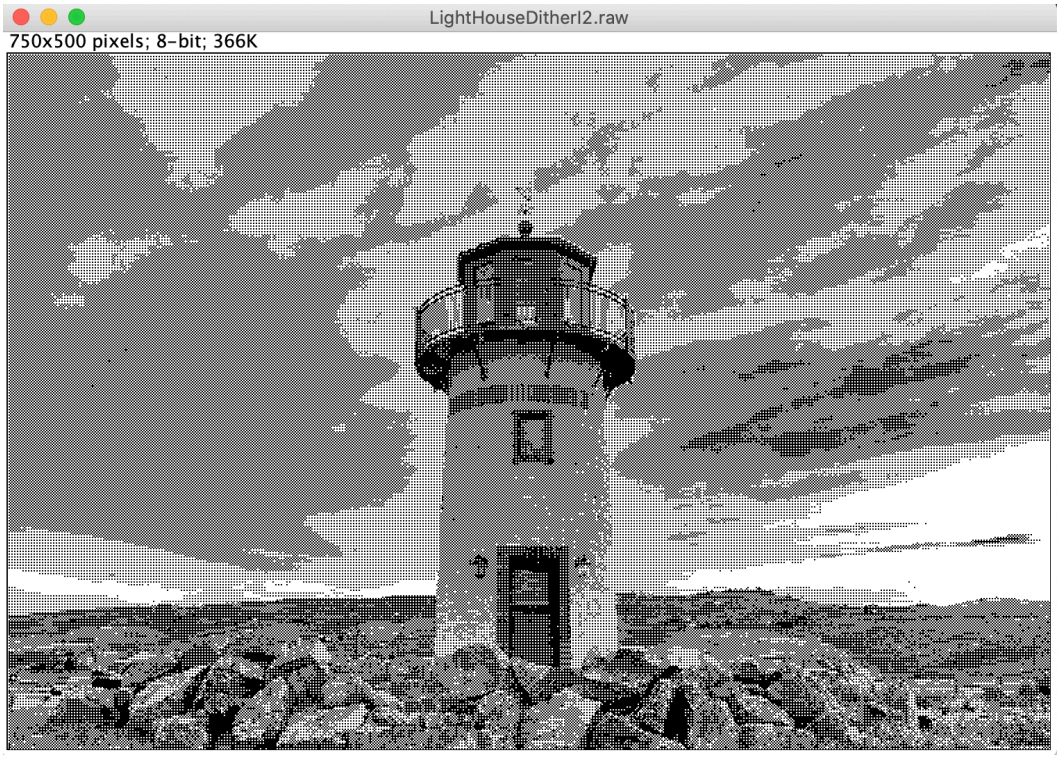
$$G(i,j) = \begin{cases} 0 & if\ F(i,j) \le T(i\ mod\ N, j\ mod\ N) \\ 255 & otherwise \end{cases}$$ --(4)

We use (3) to determine the threshold value we will use in (4), simply by taking the modulo pixel location i and j to decide which location in Dither matrix we will use. The Dither result will be showed below:

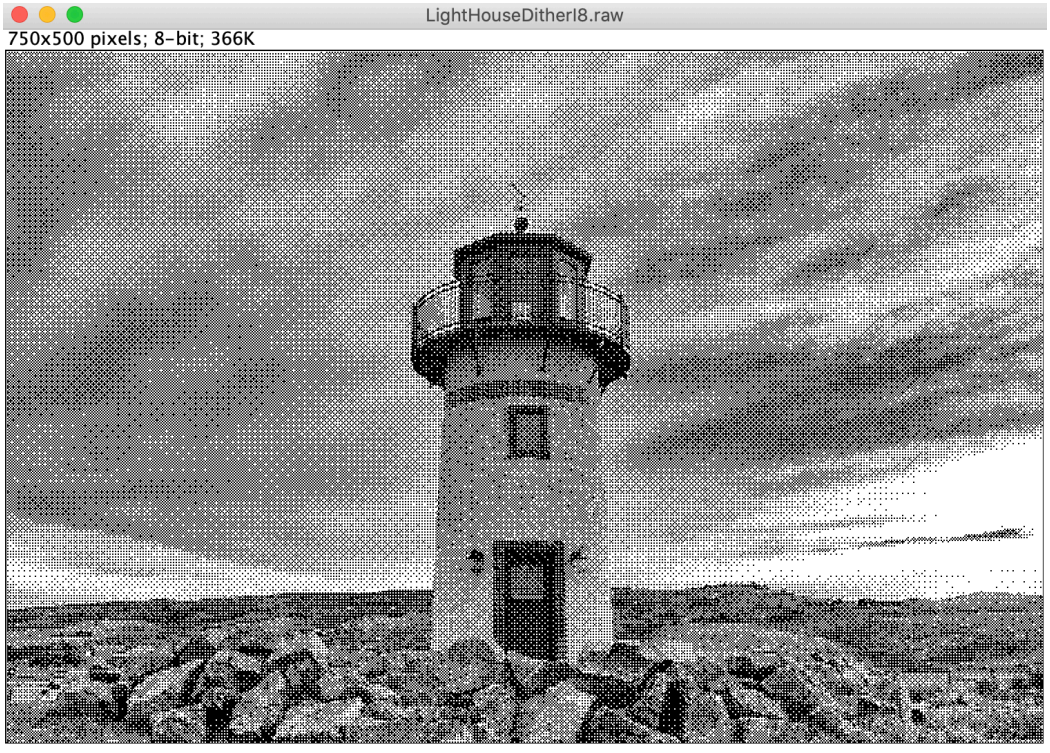
Fixed Threshold (threshold = 128)



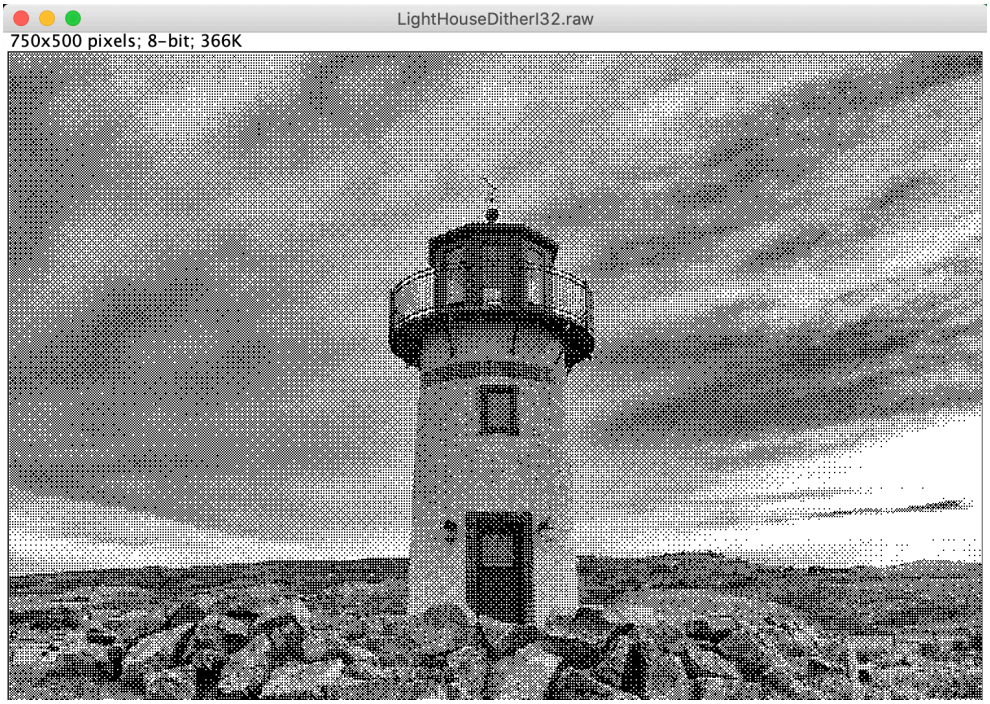
Random Threshold (threshold is uniformly distributed at 0~255)



Dithering Matrix with I2 (2x2 matrix)



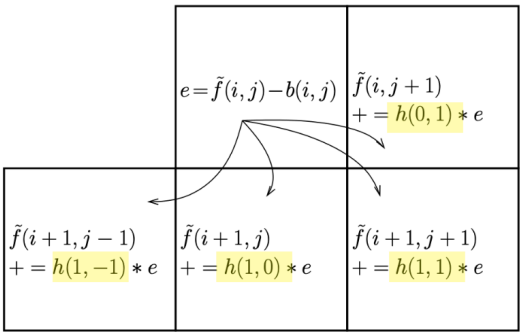
Dithering Matrix with I8 (8x8 matrix)

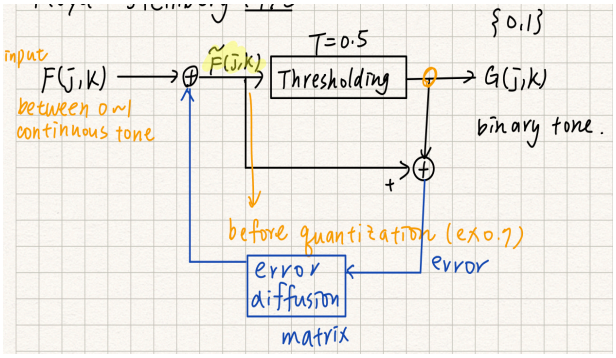


Dither Matrix with I32 (32x32 matrix)

2.2(b.) Error Diffusion

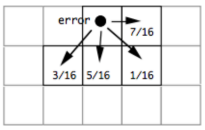
This part uses Error Diffusion method to convert 8-bit grayscale image to half-toned one, there are three error diffusion matrices to use. The idea is to diffuse the error from current pixel to future pixel.



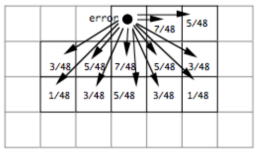


The procedure is showed on the right. The error comes from the original pixel value and quantized pixel value (0,255), and it will be accumulated at each pixel, the output pixel will be binary (either 0 or 255). An important part of this method is that we will use **"Serpentine scanning"** method to process the error diffusion! Use the value of row of input image, if the row is even, the processing direction will be from left to right, if the row is odd, then it will process from right to left. As the picture, we could get the output pixel when we sum ~f + error and set the threshold!

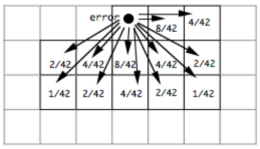
- **Floyd-Steinberg** $\frac{1}{16}\begin{bmatrix}0 & 0 & 0\\ 0 & 0 & 7\\ 3 & 5 & 1\end{bmatrix}$



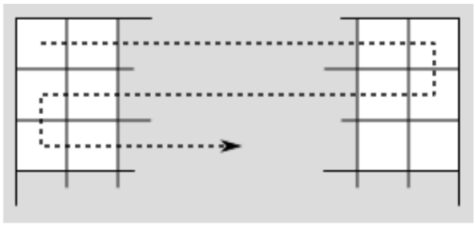
- **JJN** $\frac{1}{48}\begin{bmatrix}0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 7 & 5\\ 3 & 5 & 7 & 5 & 3\\ 1 & 3 & 5 & 3 & 1\end{bmatrix}$



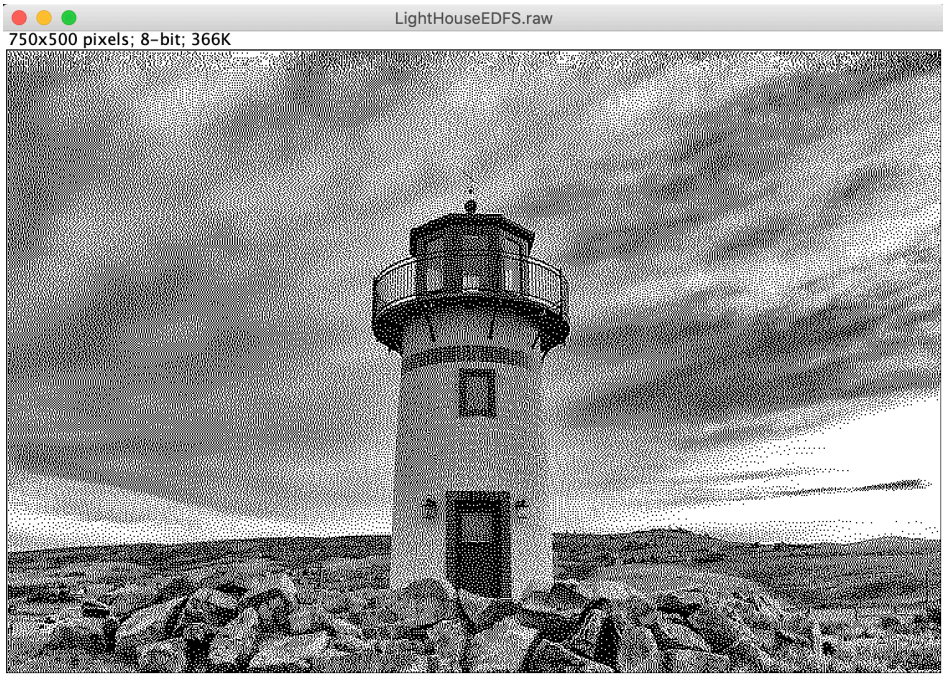
- **Stucki** $\frac{1}{42}\begin{bmatrix}0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 8 & 4\\ 2 & 4 & 8 & 4 & 2\\ 1 & 2 & 4 & 2 & 1\end{bmatrix}$



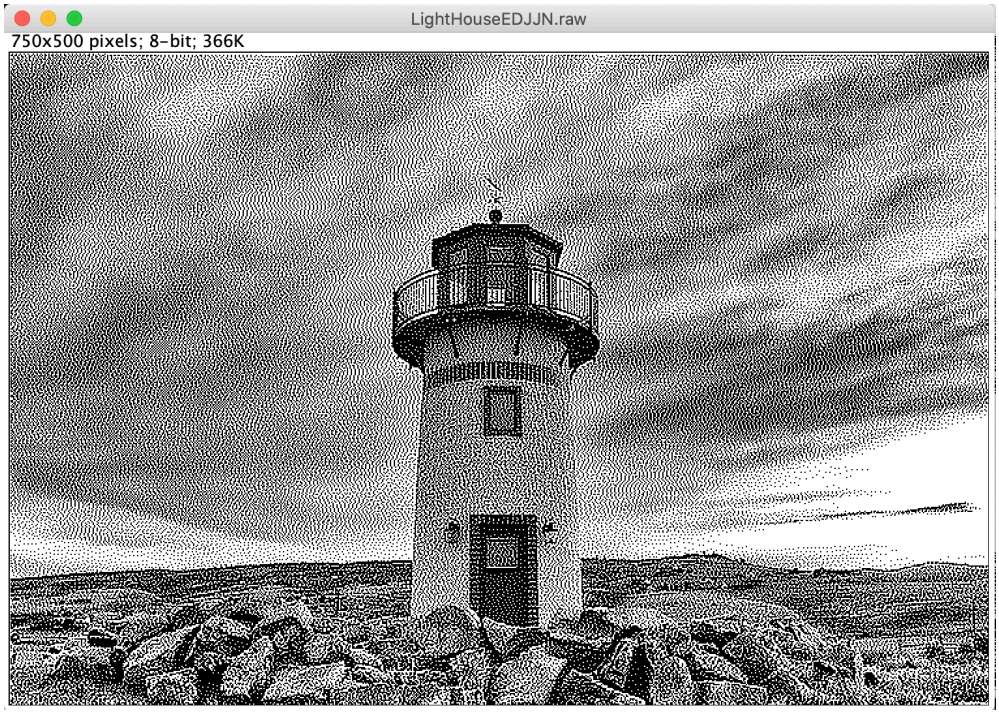
- **Serpentine parsing**



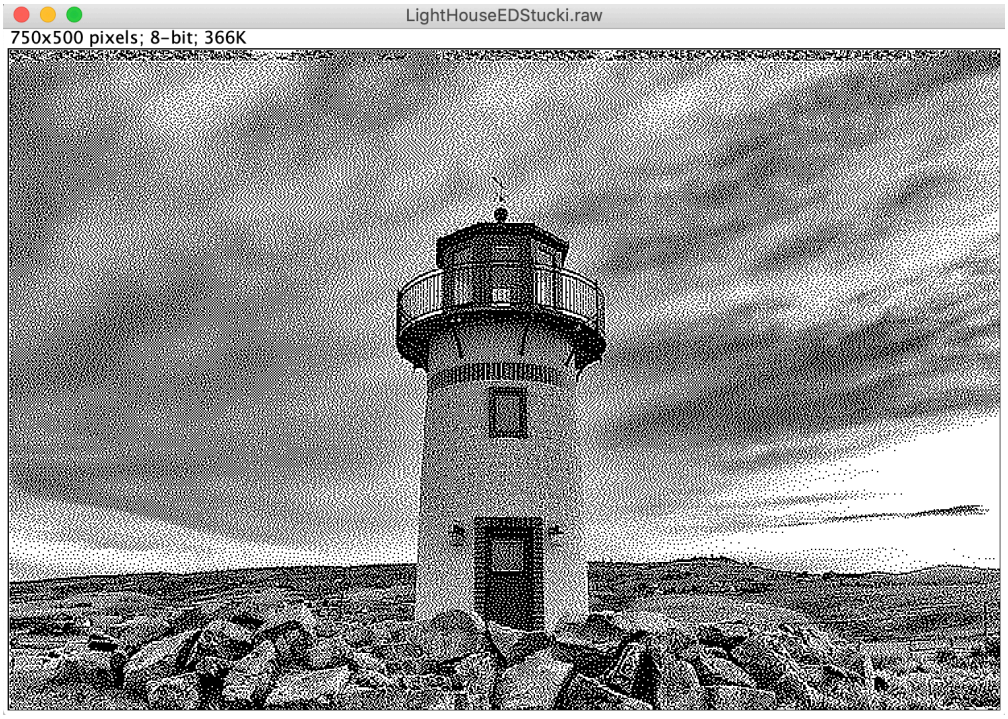
The result three different matrices results will be showed below:



Floyd-Steinberg



Jarvis, Judice and Ninke (JJN)



Stucki

2.2(c.) Color Halftoning with Error Diffusion

Part 1: Separable Error Diffusion

To achieve color halftoning, we need to separate the image into **CMY** three channels and

apply error diffusion algorithm we used on last question (like Floyd-Steinberg's) to quantize each channel separately.

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix} \qquad \begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix}$$

Step 1: Convert RGB to CMY by the formula above.

Step 2: Apply error diffusion on CMY channel separately and convert them back to RGB. Then we will have one of the following 8 colors, corresponding to the 8 vertices of the CMY cube at each pixel:

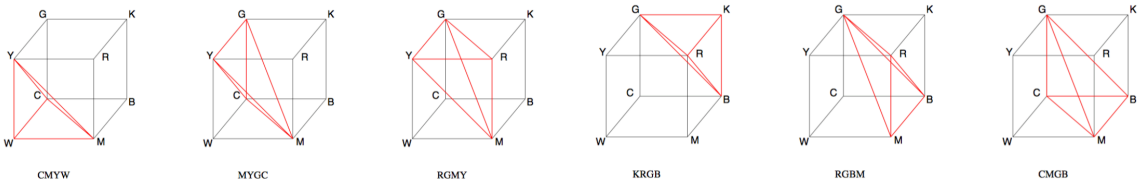
W = (0,0,0), Y = (0,0,1), C = (0,1,0), M = (1,0,0),

G = (0,1,1), R = (1,0,1), B = (1,1,0), K = (1,1,1)

Part 2: MBVQ-based Error Diffusion

This method proceeds by separating the RGB color space into minimum brightness variation quadrants (MBVQs). >> we will first rent every input pixel using one of six complementary quadruples, which shows below:

MBVQ (Minimal Brightness Variation Quadruples)

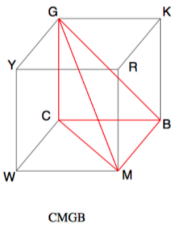


Step 1: Determine MBVQ first, I write these if-else condition in C++ by referencing given matlab code.

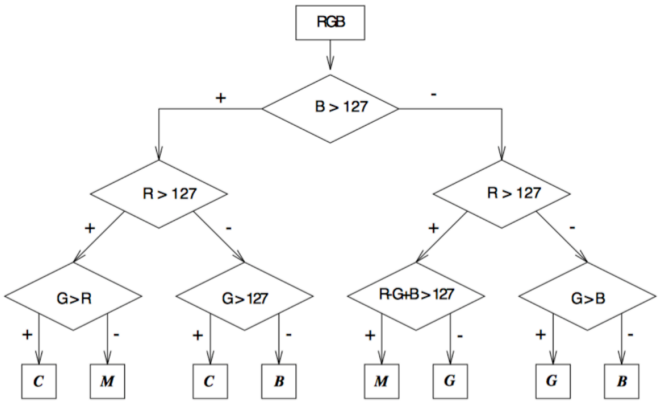
```
pyramid MBVQ(BYTE R, BYTE G, BYTE B)
{
  if((R+G) > 255)
    if((G+B) > 255)
      if((R+G+B) > 510)     return CMYW;
      else                  return MYGC;
    else                    return RGMY;
  else
    if(!((G+B) > 255))
      if(!((R+G+B) > 255)) return KRGB;
      else                  return RGBM;
    else                    return CMGB;
}
```

Step 2: Once we know each cube we will go, we have to determine the closet vertex, there are six MBVQ cube, just take one in discussion for example:

- (e.g. for CMGB Quadruple)



CMGB



Once we determine the vertex, for example, if it is Yellow, the value will be R = 1, G = 1 , B = 0, then we could set the input value to (1,1,0). Every pixel will do this procedure in the beginning before implement error diffusion. The error in this part is:
Input.pixel - MBVQ(input.pixel).

Note: The error will accumulate into the input pixel.

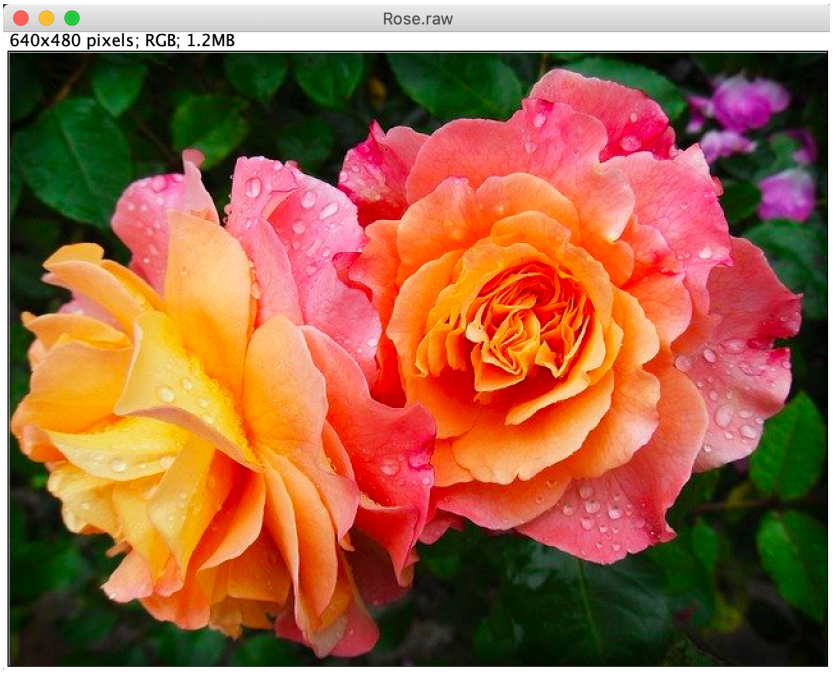
The procedure will be:

Error = input**(will be accumulated error)** – MBVQ(input) > use this error implement Error Diffusion

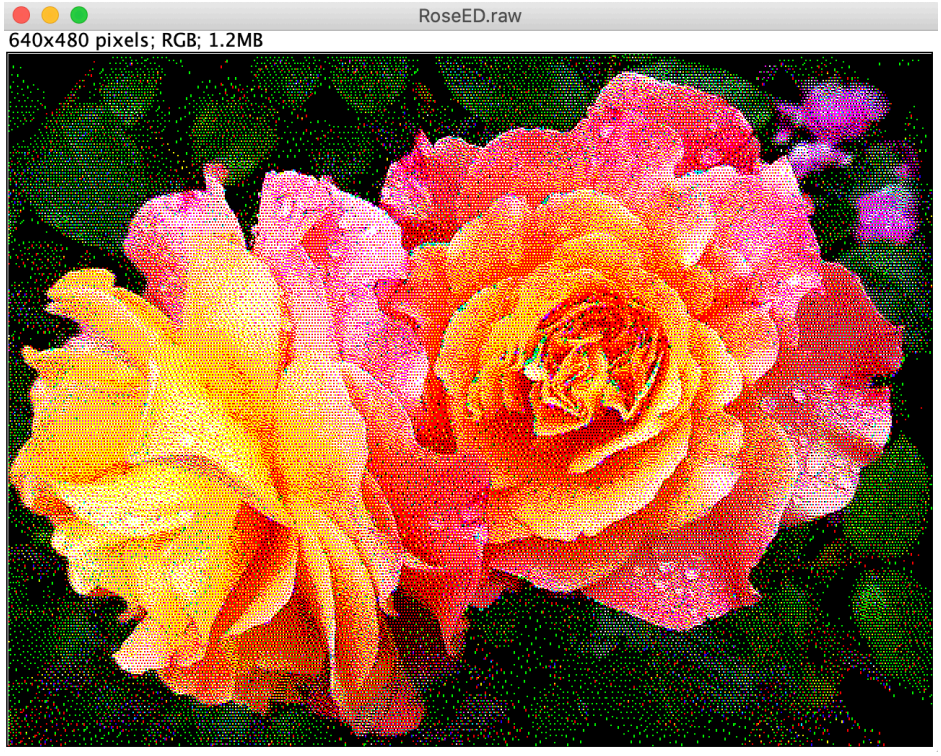
Step 3: We need to compute the quantization error and distribute the error to future pixels. I used Floyd Steinberg's error diffusion in this problem.

Step 4: The output pixel could got by using **Step1** and **Step2.** (MBVQ->VERTEX->assign one of eight color values).

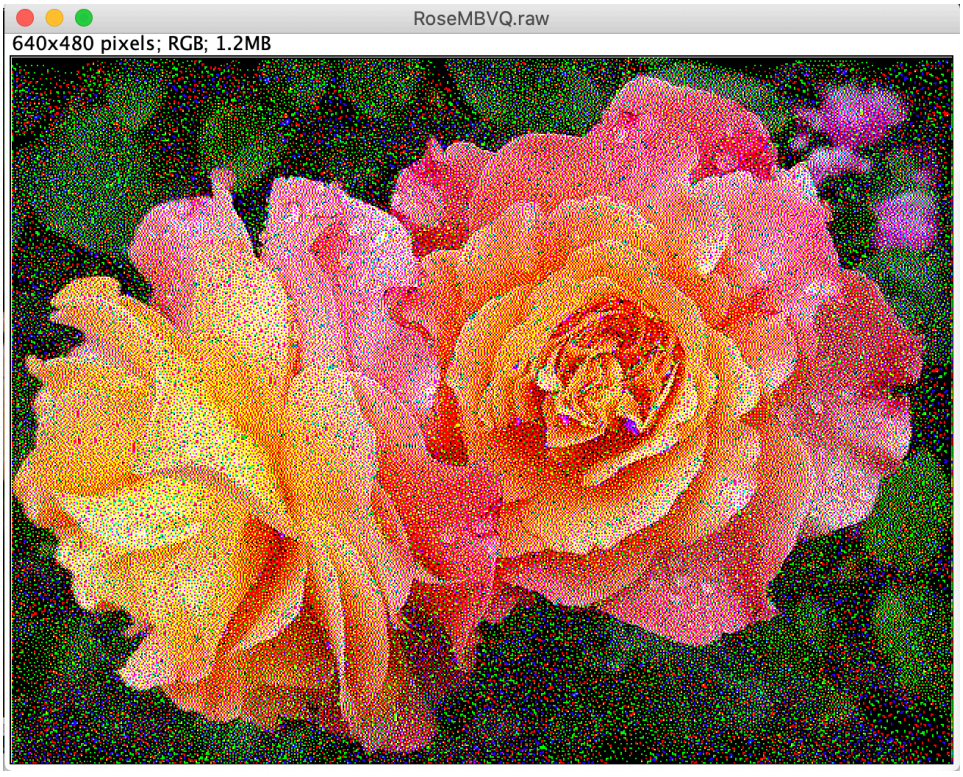
The result will showed below:



Original



Separable Error Diffusion



MBVQ-based Error Diffusion

2.3 Discussion

2.3.(a.3) -- Dithering

From the result, we could observe that the halftoning image using Dither Matrix I2 are quite different from the image using I8 and I32. I2 image lost a lot of details, like the variation of black on clouds and the texture of rocks! However, I8 image and I32 image look similar. There are also many crossing-patterns in these Dithered images.

2.3(b.) – Error Diffusion

(1.) Result Discussion:

The main different between Dithering and Error Diffusion is that, the image using Dithering has a lot of “crossing-pattern”, while the pattern in Error Diffusion is more natural. (Note: there are still some artifacts like “worm” in error Diffusion.

I prefer Floyd-Steinberg method. The image using JJN and Stucki method might be better for recovering detail of original image, but there are more worms-like artifact than the image using Floyd-Steinberg. Therefore, I rather loss a little bit quality to have more natural image.

(2.) improvement:

If we observe closely on the images we produced by FS, JJN and Stuck, we could see there are still some undesirable artifacts such as “worm” and undesired pattern. One way I could think of to break up the undesirable output pattern is to alternative scanning pattern, such as Hibert Curve. However, I don’t have time to implement this suggestion, but I believe when Serpentine scanning is better than Raster scanning. Therefore, it must exist a better scanning method to apply with implementing Error Diffusion.

2.3(c.) – Separable Error Diffusion

I think the shortcoming of this method is that we consider RGB 3 channels separately, unlike MBVQ-based Error Diffusion. Moreover, since the threshold Is 0.5, if the two adjacent have the pixel of 0.49 and 0.51 value on the same channel, they might be set to other value in error diffusion, while they are almost the same color in original image.

If we zoom in the image of halftoning image, we can see that the image is composed by only 8 colors (RGBCMYKW)! However, the image looks very similar when observing at

the original size.

2.3(c.2) – MBVQ-based Error Diffusion

(1.) The key idea which makes MBVQ different from Separable Error Diffusion is that MBVQ needs to look for the closet vertex in the MBVQ (Minimal Brightness Variation Quadruple). This step needs to determine which quadrant and then find the vertex. During this step, RGB 3 channels value are considered together. There are many conditions to decide quadrant and vertex. Through this procedure, we could better assign the value to input pixel in order to implement error diffusion. That is, the inputs converted to vertices could be more relative!

(2.) Compared to the image using Separable Error Diffusion, I think the image using MBVQ is a little bit lighter, but in my implementation, MBVQ image seems to have more noise (like the color points are denser than the one I got in Separable Error Diffusion).