EE569 HW2

Problem 1 Edge Detection

Motivation

Edge is a technique that recognize the incontinuous pixels in the image. It has great significance in object detection. Even in deep learning task, edge detection is a good start before exploring other complex feature of images and objects. In this part, we implement two classic edge detector, Sobel and Canny, which is based on gradient, and a modern detector, sturctured edge detector, which is based on structured forest.

Procedure and results

a.Sobel Edge Detector

$$\frac{1}{4}\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \qquad \frac{1}{4}\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Command:

g++ -o SobelEdgeDetector SobelEdgeDetector.cpp

SobelEdgeDetector HW2_images/Tiger.raw

Output: x Gradient.raw y Gradient.raw

To switch between Tiger and Pig, just edit the source file to and change output name properly. To switch threshold percentage, edit ``
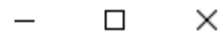
```
Threshold(GF,ImageOutput,15);
OutputImage(ImageOutput,"Output/PigEdge15.raw",1);
```

and input a number between 0 and 100 as the third parameter of Threshold() funtion.
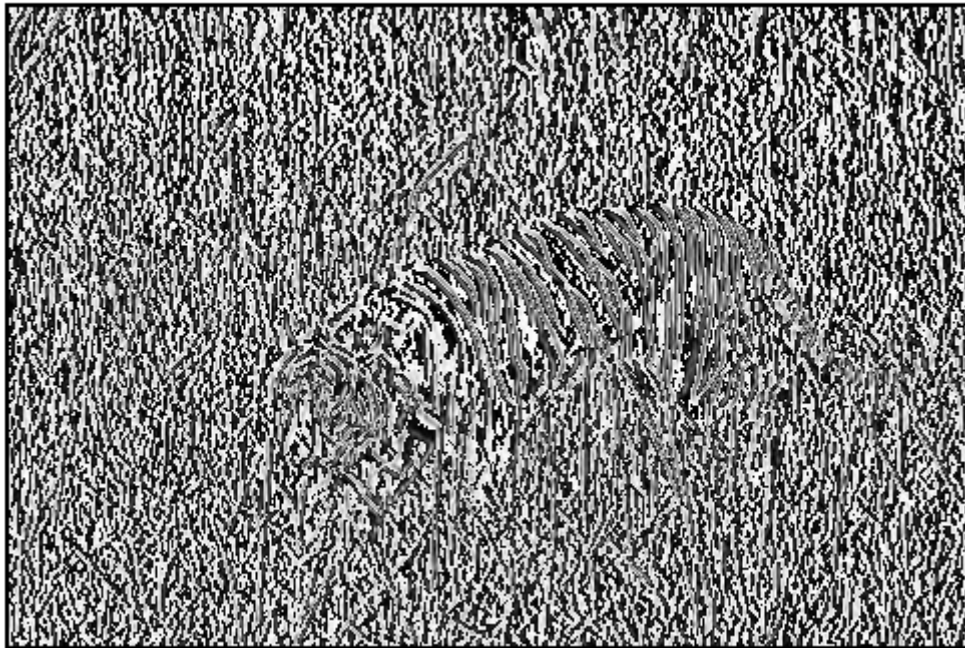
a.1 X gradient and Y gradient

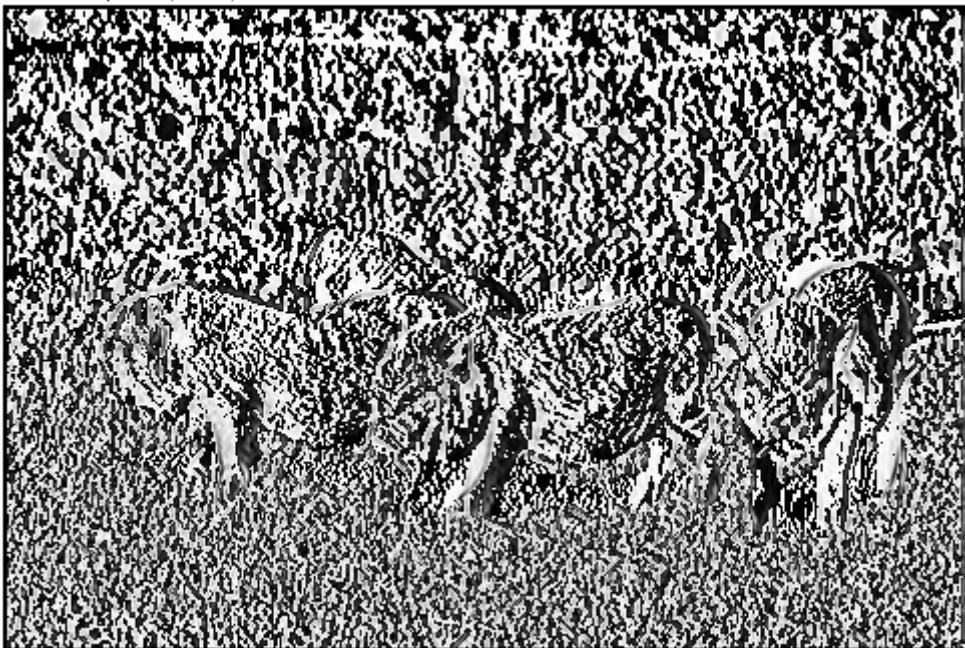Gradient in x direction looks like this

x Gradient.raw
481x321 pixels; 8-bit; 151K

Pig x Gradient.raw
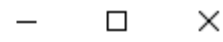481x321 pixels; 8-bit; 151K

Gradient in y direction looks like this

y Gradient.raw
481x321 pixels; 8-bit; 151K

Pig y Gradient.raw
481x321 pixels; 8-bit; 151K

a.2 Gradient map

a.3 Edge display

481x321 pixels; 8-bit; 151K

edge = top 5 percent

481x321 pixels; 8-bit; 151K
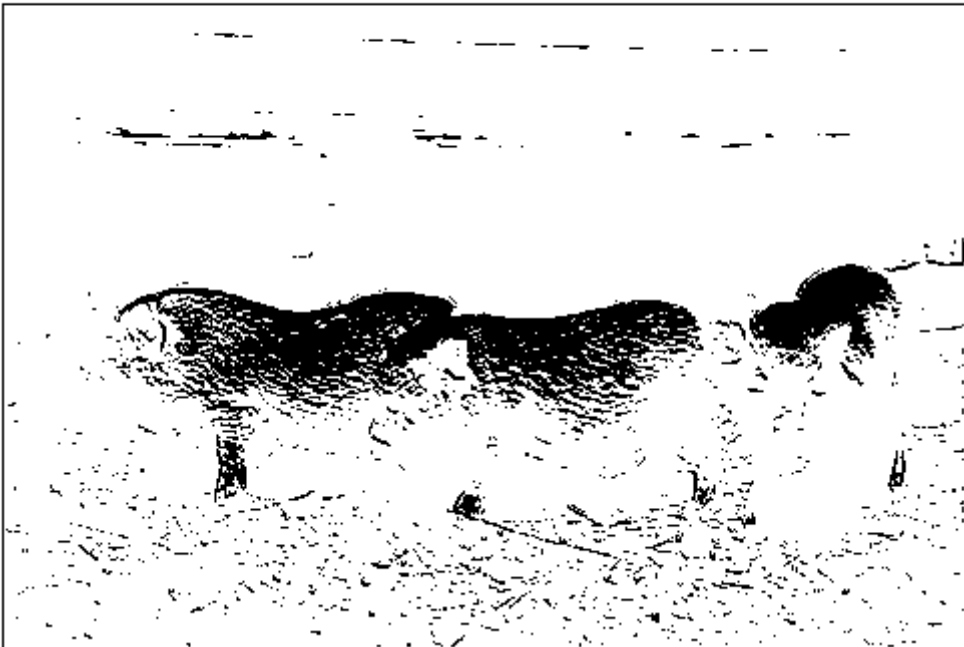
edge=top 10 percent

PigEdge15.raw

481x321 pixels; 8-bit; 151K

edge = top 15 percent

The high threshold make less pixels becoming edges. Personally I'll prefer high threshold(5%) to low threshold because high threshold will include fewer useless background information as edges, and the structure is no worse than low threshold product.

b. Canny Edge detector

b.1 Non maximum suppression

Canny edge detector calculate gradient in four different direction. In every direction, canny will try to find the local maximum gradient, and only the local maximum will finally become edge, all the other pixels that has high gradient but not the local maximum will be suppressed. Thus, the edge will be expected to be thinner and cleaner.

b.2 High & low threshold

There're two threshold work together in canny edge detector, called Hysteresis Thresholding. Magnitudes above the higher threshold are definitely edge points. Those below the lower threshold are definitely not. Those stay in the middle are going to be further processed. Sometimes the gradient that are not high enough may be caused by either true edge, or the color changes or noises. We don't want the color changes or noise to generated wrong edges, but we still want to reserve those weak true edges. Because weak true edges are usually generated by strong true edges, so if it is connected to a strong true edge, then we consider it a s weak true edge and reserve it. Otherwise it is considered as non edge and be suppressed.

b.3
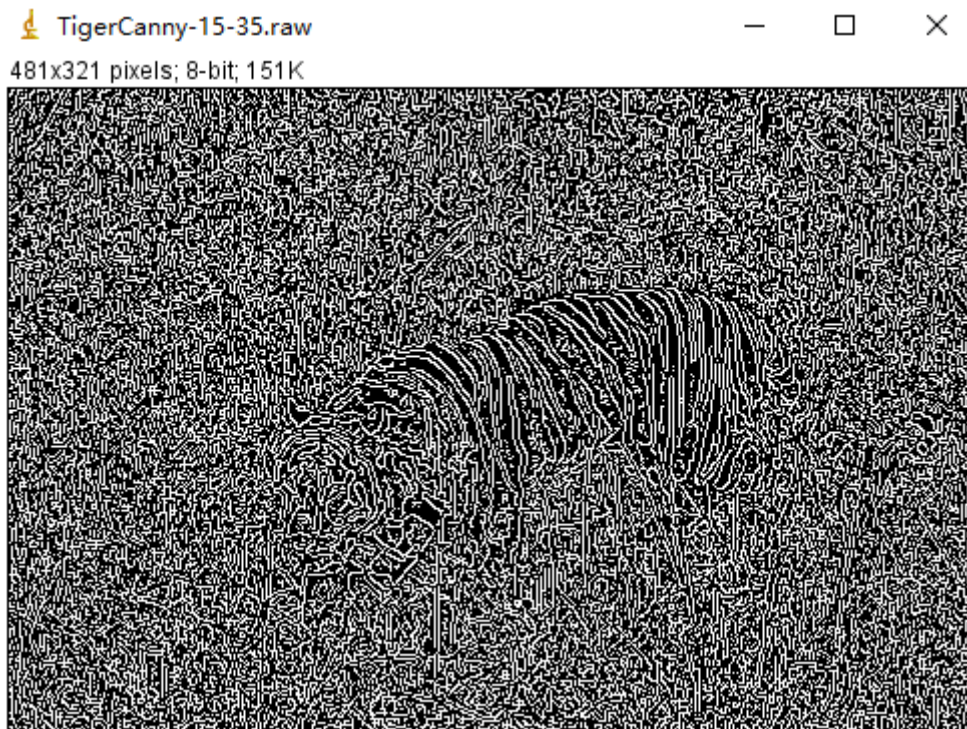
low=15 High=35
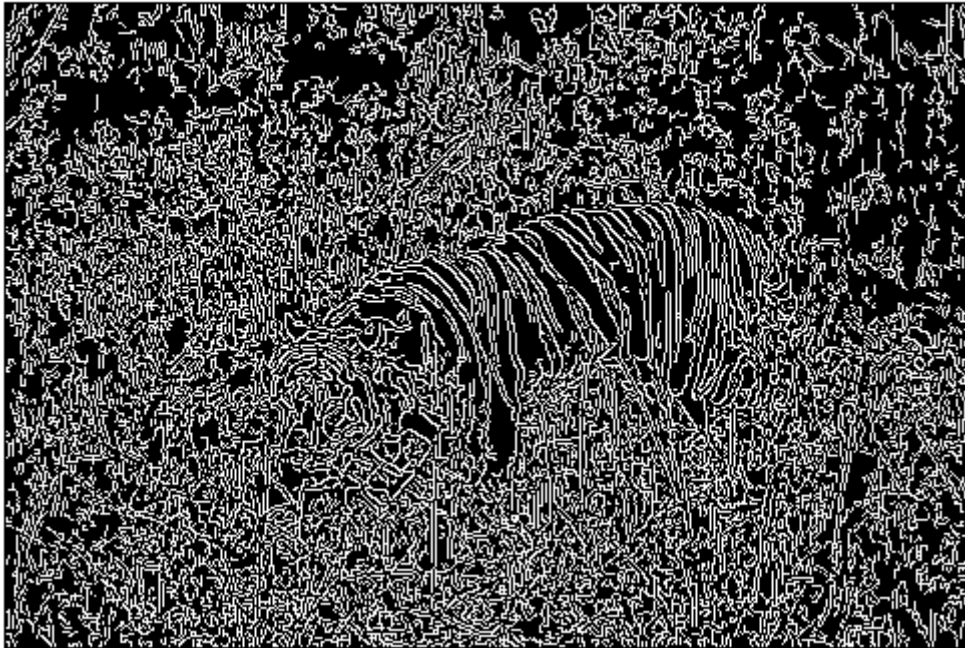
Low=100 High=200
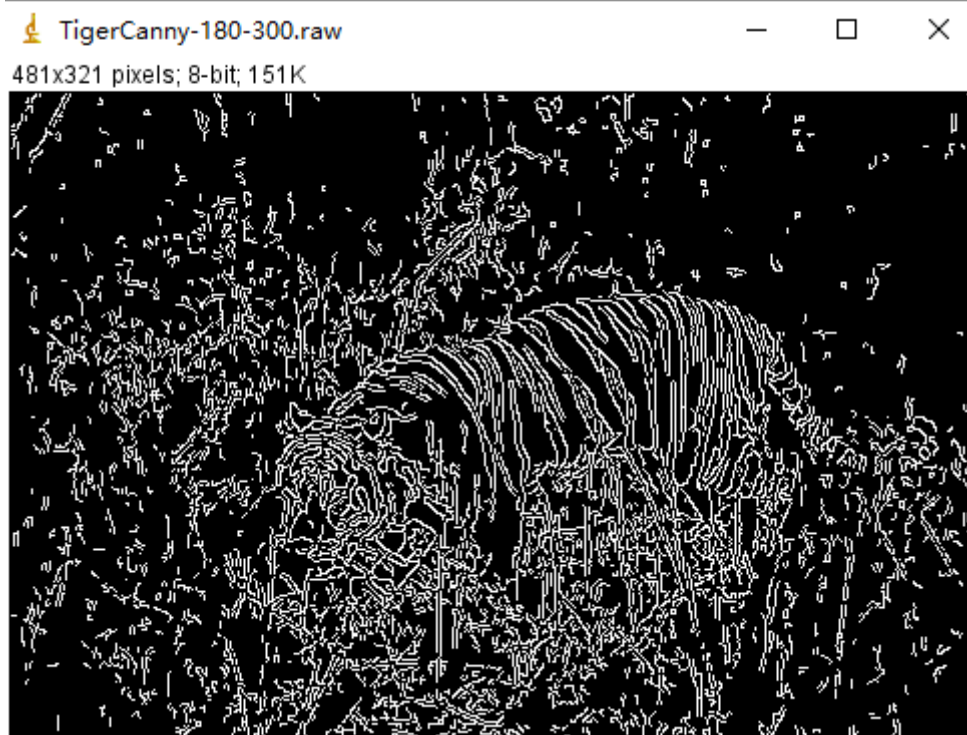
Low=180, High=200

Low=180, High=300
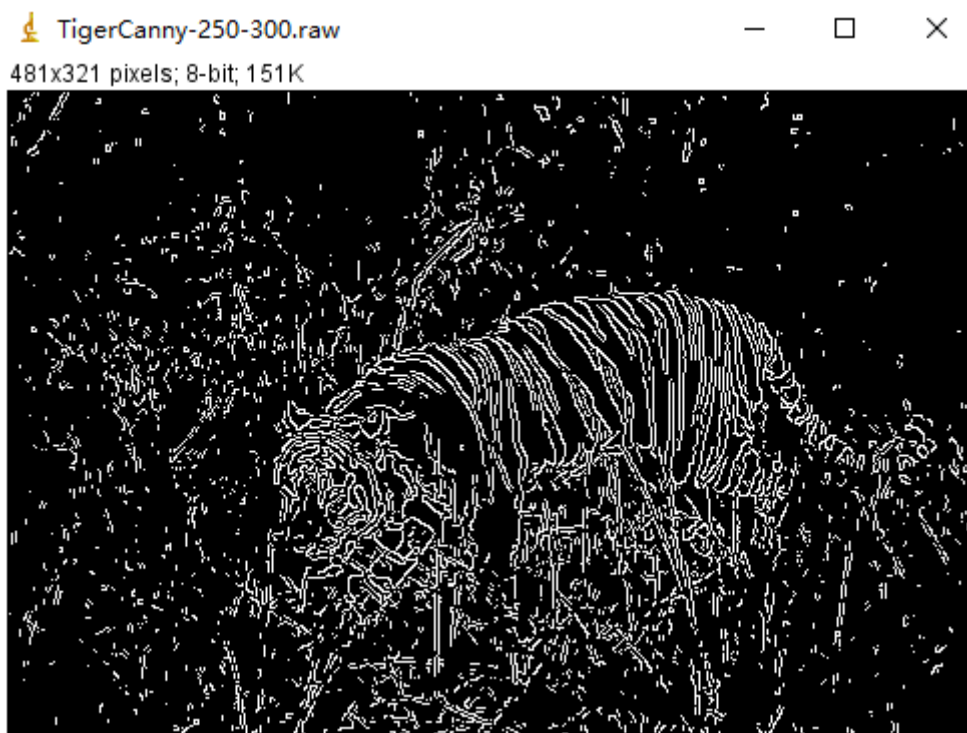
Low=250, High=300

Low=400, High=500

For Pig.jpg, I just include the best image I think in the report
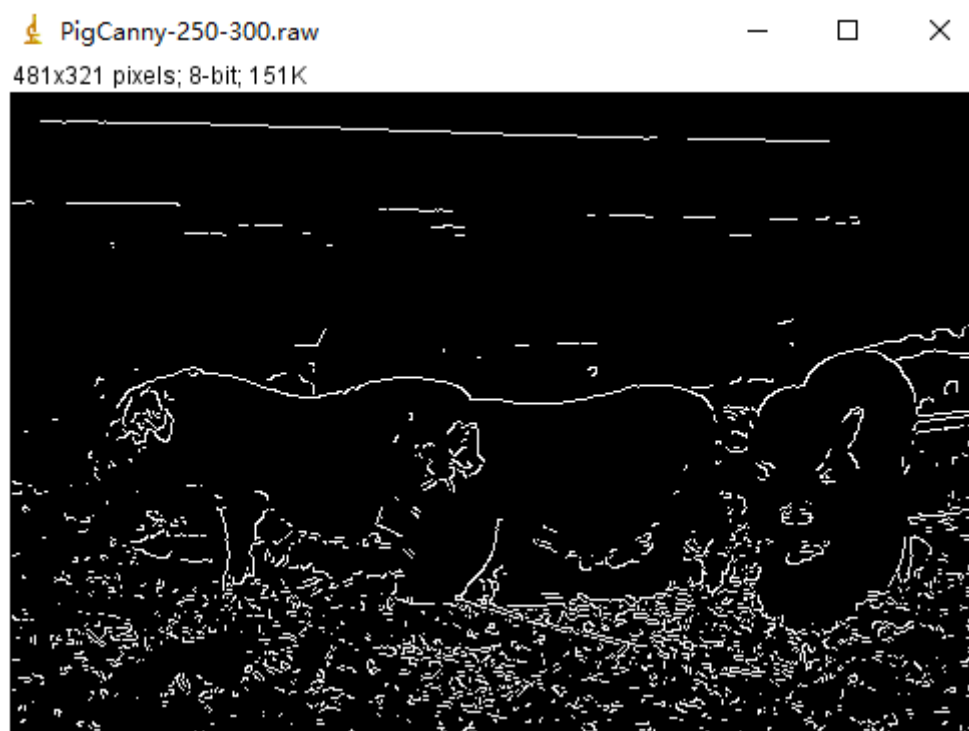
Low=250, High=300

In Canny edge detector, we can see there're no longer any color blocks. The edges are just with width of a single pixels. We can see when the high threshold goes higher, or when the lower threshold goes closer to the high threshold, less and less possible edges are included.

Command:

activate

```
add_executable(EE569_hw2_8831981929_HanyunZhao fileIO.h CannyEdgeDetector.cpp)
```

in CmakeLists.txt and run in CLion

Threshold can be tuned in the source code

c. Structured Edge

1.SE algorithm

Structured edge is an upgrade of sketch token. Sketch token. ST compares the data with the predefined label sets, while SE consider all possible cases of 16*16 grid, which has $2^{256}$ dimensional output. To lower the dimension, we map it to a space Z, which represent whether two pairs are similar, that reduce dimension to 32640. To further reduce, we can sample m dimension, e.g. 256, and then use PCA to extract the most significant dimensions. In Z space, labels are separate to k classes through Kmeans or PCA at each node of a decision tree. Average of the decision tree will get a soft edge response. To further improve the result, we can run the detection on half, original, and double resolution, and average result, which is called multiscale detection.
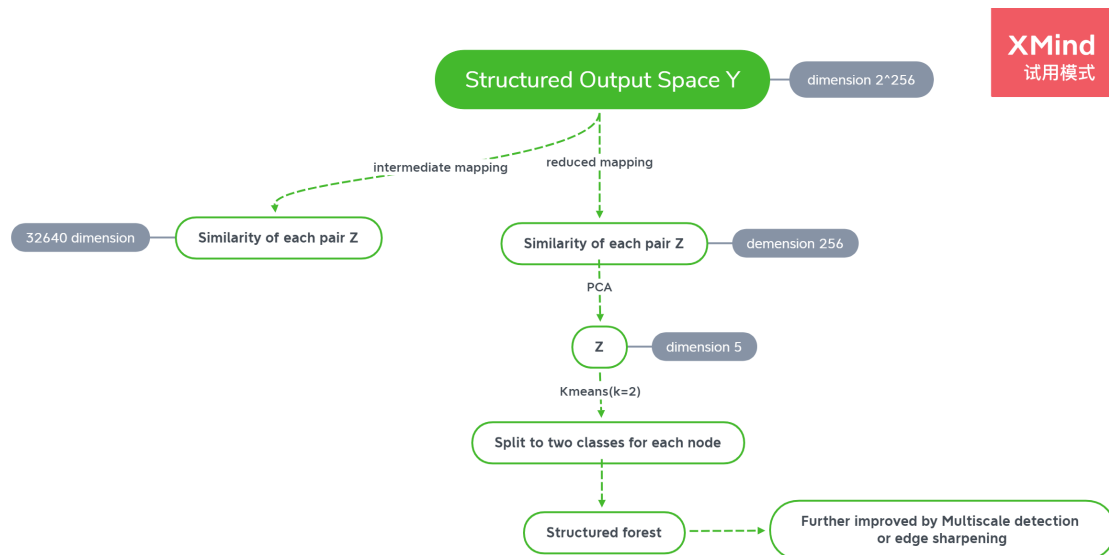
2.RF classifier

To construct a decision tree, first step is to randomly pick some samples from the training set. These samples is the root node of a decision tree. Then, at each node, we randomly pick a subset of feature, and try to find a feature in the subset and a threshold to split the node that maximize the information gain (or minimize the entropy).

Random forest is consist of lots of independent decision trees. It takes the most common classifying result of all the decision trees as its own classifying result.

3.Matlab SE Detector
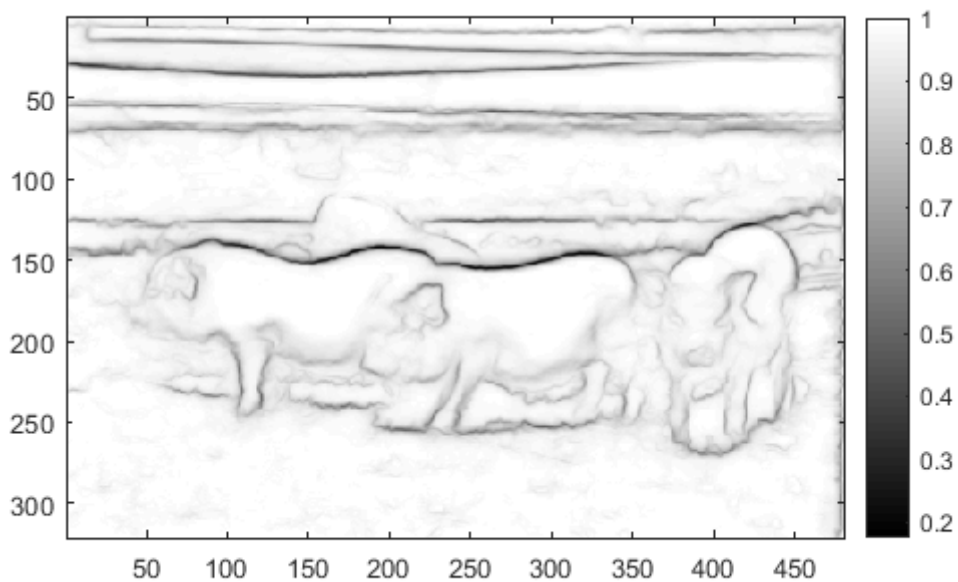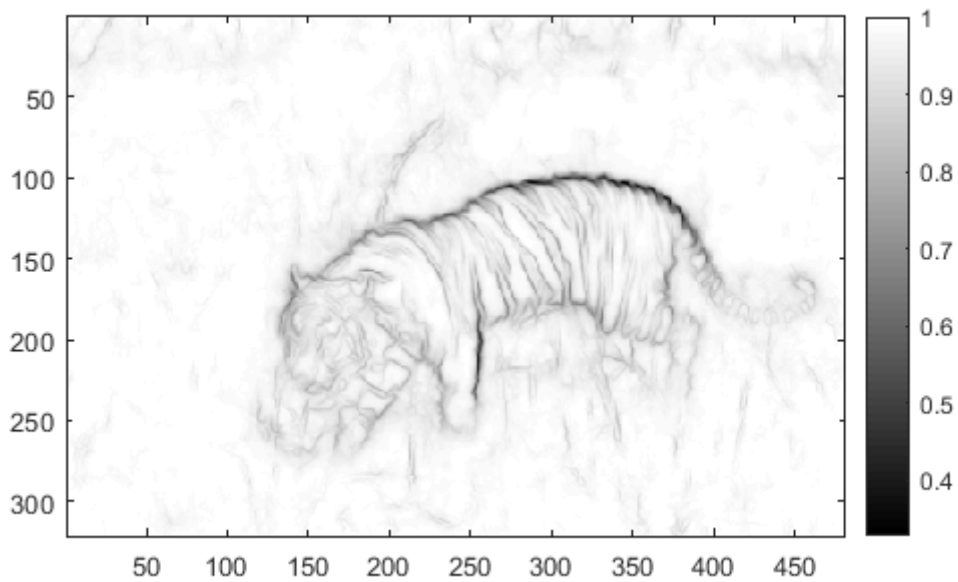
There're five parameter that can be tuned.

```
model.opts.multiscale=1;          % for top accuracy set multiscale=1
model.opts.sharpen=2;             % for top speed set sharpen=0
model.opts.nTreesEval=8;          % for top speed set nTreesEval=1
model.opts.nThreads=4;            % max number threads for evaluation
model.opts.nms=0;                 % set to true to enable nms
```

The reason for option is to increase the accuracy. For multiscale, it is the method mentioned above, that run the algorithm in half, original, and double resolution an take average to improve performance. So I enabled it. For sharpen, it'll make the edges clearer. I set it to be the maximum supported value, which is 2. For nTreesEval, the default value is 4, then I doubled it. For higher value, I did not see any obvious improvement, so 8 is my final choice. For nThreads, it may speed up the process. I just leave it default. For nms (non maximum suppress), I don't think it makes the result more beautiful, so I disabled it. The figure below is a typical result with NMS on.
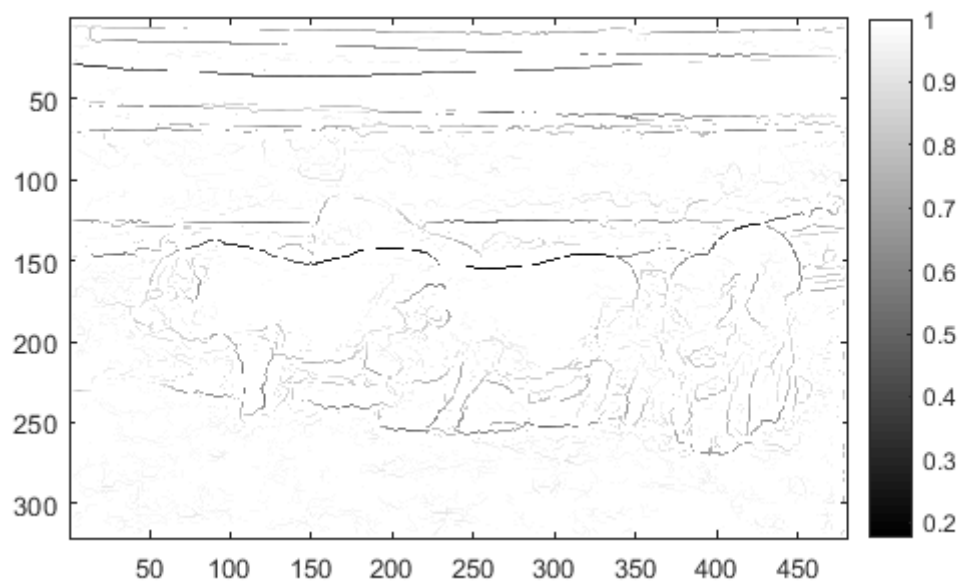
SE with NMS

d. Performance evaluation

   1.

In this part, threshold is 0.5

Sobel：we use the boundary plot with top 5% threshold

Tiger	GT1	GT2	GT3	GT4	GT5
precision	0.0354	0.0376	0.0420	0.1598	0.0487
recall	0.9652	0.9680	0.9623	0.9283	0.9622

Overall, **meanP=0.0647, meanR=0.9572, F=0.1212**

Pig	GT1	GT2	GT3	GT4	GT5
precision	0.0831	0.0882	0.1334	0.1596	0.1297
recall	0.8072	0.7921	0.7482	0.7126	0.7455

Overall, **meanP=0.1188, meanR=0.7611, F=0.2055**

Canny Edge Detector: I use lower bound=250, higher bound=300

Tiger	GT1	GT2	GT3	GT4	GT5
precision	0.0666	0.0704	0.0793	0.2378	0.0901
recall	0.9047	0.9006	0.9038	0.6876	0.8852

Overall, **meanP=0.1088, meanR=0.8564, F=0.1931**

Pig	GT1	GT2	GT3	GT4	GT5
precision	0.1322	0.1433	0.2188	0.2565	0.2037
recall	0.4577	0.4591	0.4377	0.4083	0.4175

Overall, **meanP=0.1909, meanR=0.4361, F=0.2655**

Structured Edge:

Tiger	GT1	GT2	GT3	GT4	GT5
precision	0.1227	0.1232	0.1412	0.5226	0.1203
recall	0.2741	0.2593	0.2648	0.2385	0.1944

Overall, **meanP=0.2060, meanR=0.2482, F=0.2251**

| Pig	GT1	GT2	GT3	GT4	GT5
precision	0.1189	0.1286	0.2441	0.3083	0.2752
recall	0.2257	0.2257	0.2677	0.2690	0.3091

Overall, **meanP=0.2150, meanR=0.2595, F=0.2352**

The Sobel has lowest precision, and highest recall (as expected, since it regards too many pixels as edges). It means it can recognize almost all the true edges, but meanwhile picks too much wrong information. And the SE has highest precision and lowest recall. What it picks is more likely to be correct, but it may omit more true edges. Canny has the highest F for Pig and SE has highest F for Tiger. These two perform better than Sobel. And their PR value indicates Canny provides higher recall and SE provides higher precision.

2.

In this part, I test 99 threshold between 0 and 1.

Sobel:

Tiger

Pig
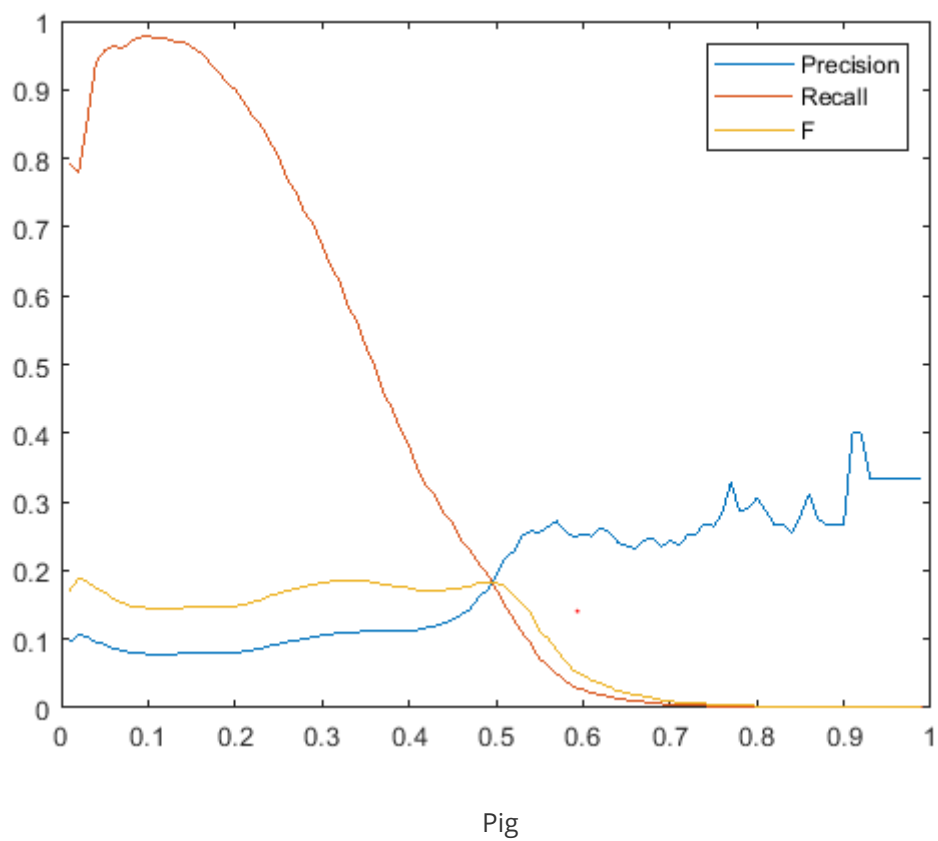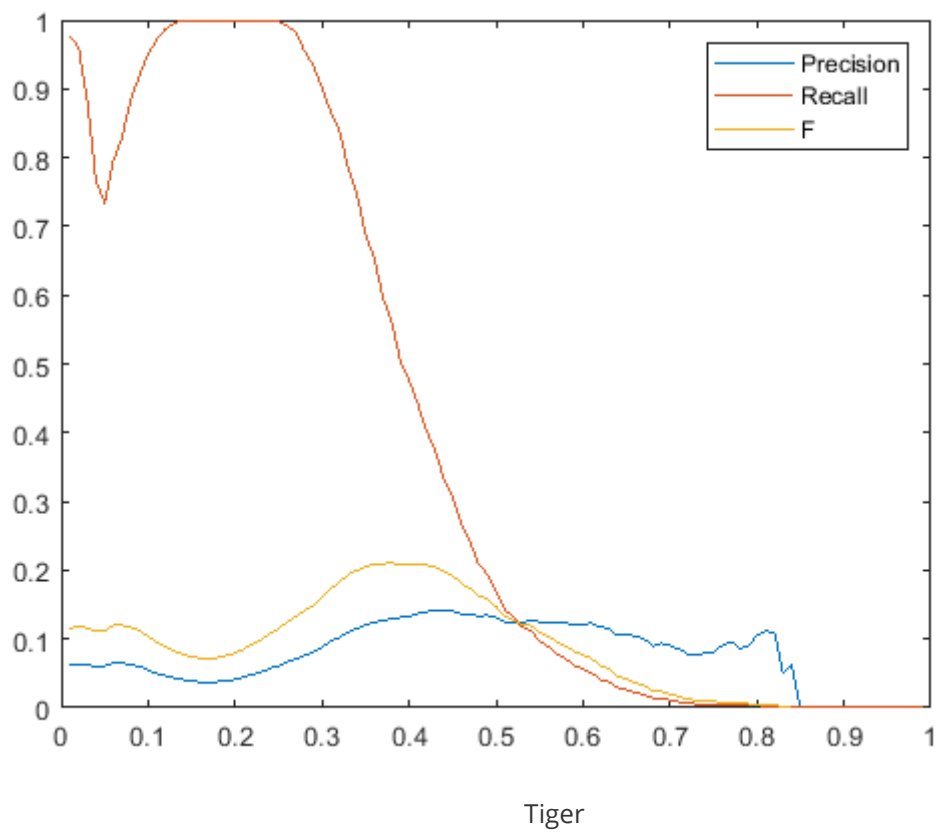
Canny:

Tiger	100/200	180/300	250/300	400/500
Precision	0.0523	0.0893	0.1088	0.1414
Recall	1	0.9347	0.8564	0.4698
F	0.0995	0.1630	0.1931	0.2174

pig	200/300	250/300	300/400
Precision	0.1739	0.1909	0.2115
Recall	0.5205	0.4361	0.2304
F	0.2606	0.2655	0.2206

Structured Edge:

Tiger

Pig
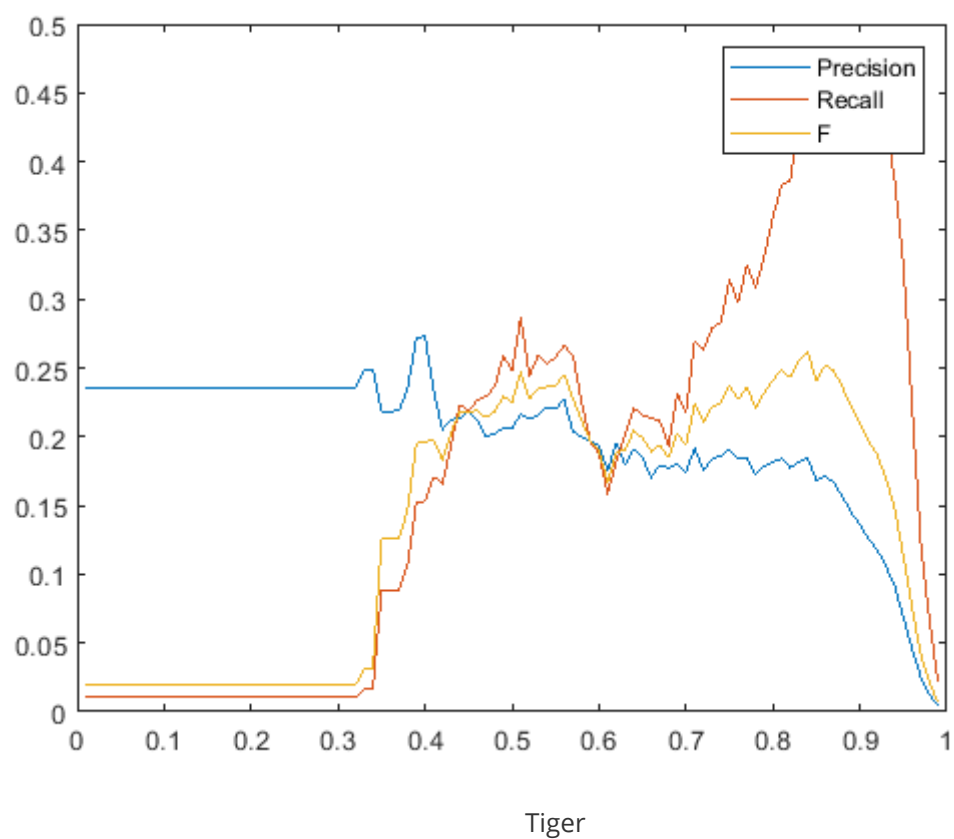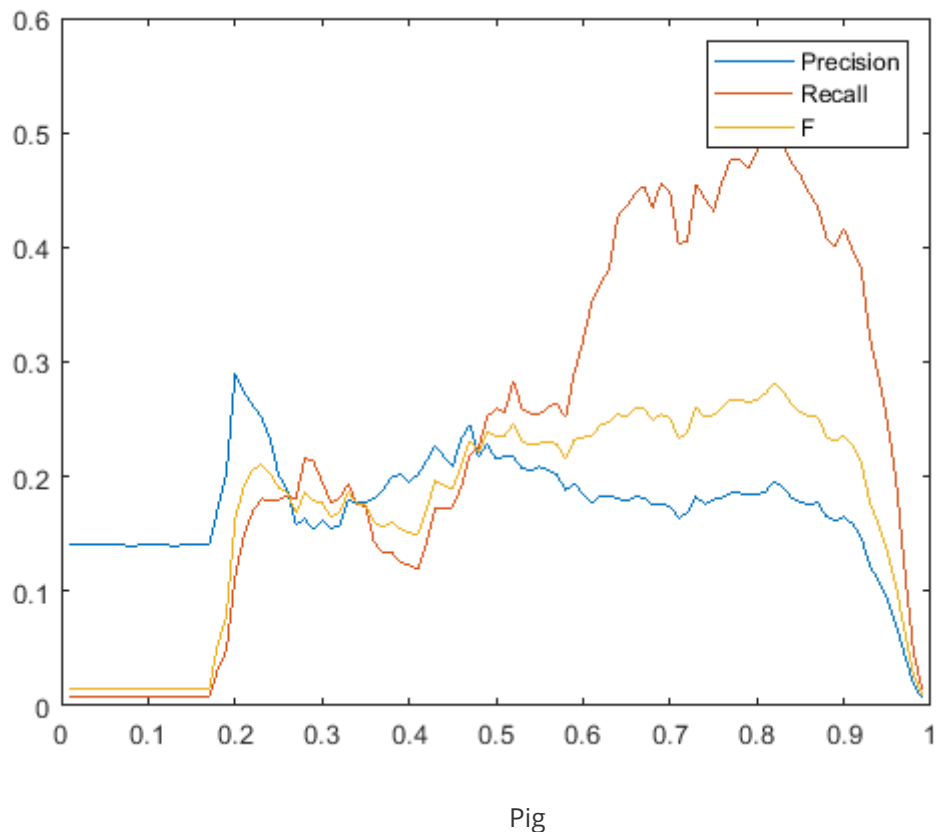
For Sobel, the threshold represent the normalized gradient. Best F for tiger is about 0.4, and any value for pig smaller than 0.5. What is interesting is that for threshold greater than 0.7, recall goes to 0, that make F also goes to 0.

For Canny, higher threshold lead to higher precision and lower recall. The best value has been discussed in part b.

For SE, thres=0.8 seems to be the optimal threshold for both picture, small threshold doesn't work.

3.It seems Pig is easier to get higher F score. Maybe the background matters. In tiger.jpg, the background grass are messy, and is more likely to provide mis-information to the machine. In pig.jpg, the background is more plain, and the upper part is with clear gradation.

4.The rationale behind F definition is, we have to balance between precision and recall. For example, picking all the pixels as edges, or none of them as edges, will gives big recall, or precision. But obviously this is nonsense. When one of them is much higher than another, the whole term will goes to zero, like the case in Sobel when threshold is large. If P+R=C, a constant, then $PR \leq C^2/4$ according to fundamental inequality.

Discussion

In Sobel edge detector, the pigs' back has big area that are considered as edges, which is not expected. For human eyes, I don't think there're large gradient, but the program think so. It may due to the color fluctuation. When we apply Canny edge detector, they are all suppressed. Meaning they're not connected to the strong true edges, thus not weak true edges.

The Opencv Canny edge detector return results that has black background and white edges. But typically, we would expect the opposite because printing black background will cost a lot of ink.

When evaluating the precision, I'm not sure whether the *edgesEvalImg* function would accept white as background or black as background. I suppose background should be white and edges to be black because the result seems more pursuading. For the edges detected by SE, I just send the outcome Matrix to the edgesEvalImg function. The PigSE's F value is lower than Canny's, and both has low recall value. That makes me worried whether I use the functions correctly.

Problem 2 Digital Half-toning

Motivation

In many practical cases, we have only binary output for images, like printer, but we want to reserve the grey scale. Half-toning is used for creating continous tone under such limit. By adjusting the density of black and white pixels, it can approximate the grey scale. Efficient half-toning algorithms can achieve good image quality with relatively low cost.

In this part, we implement two algorithms. Dithering, which add limited randomness to the threshold, and error diffusion, which pass the error to subsequent pixels to compensate for excessive loss or gain.

Procedure and results

a.Dithering

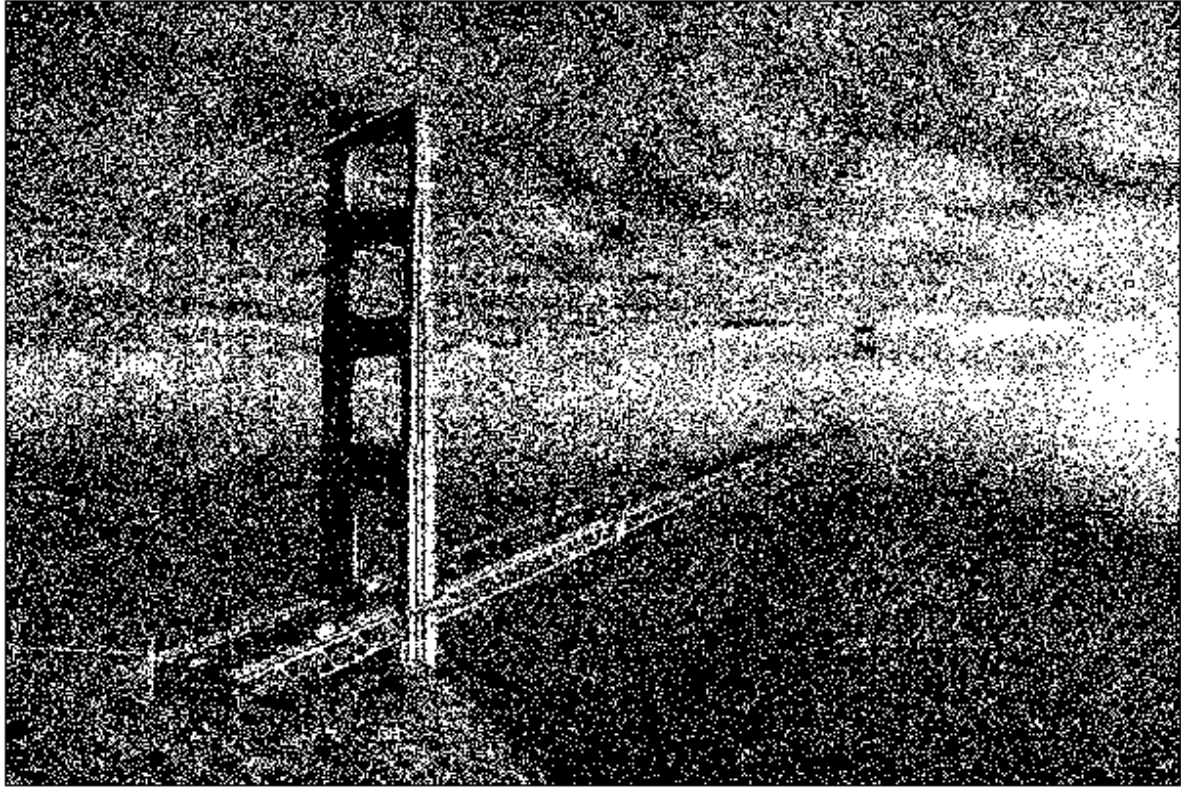1. Fixed thresholding (T=128)
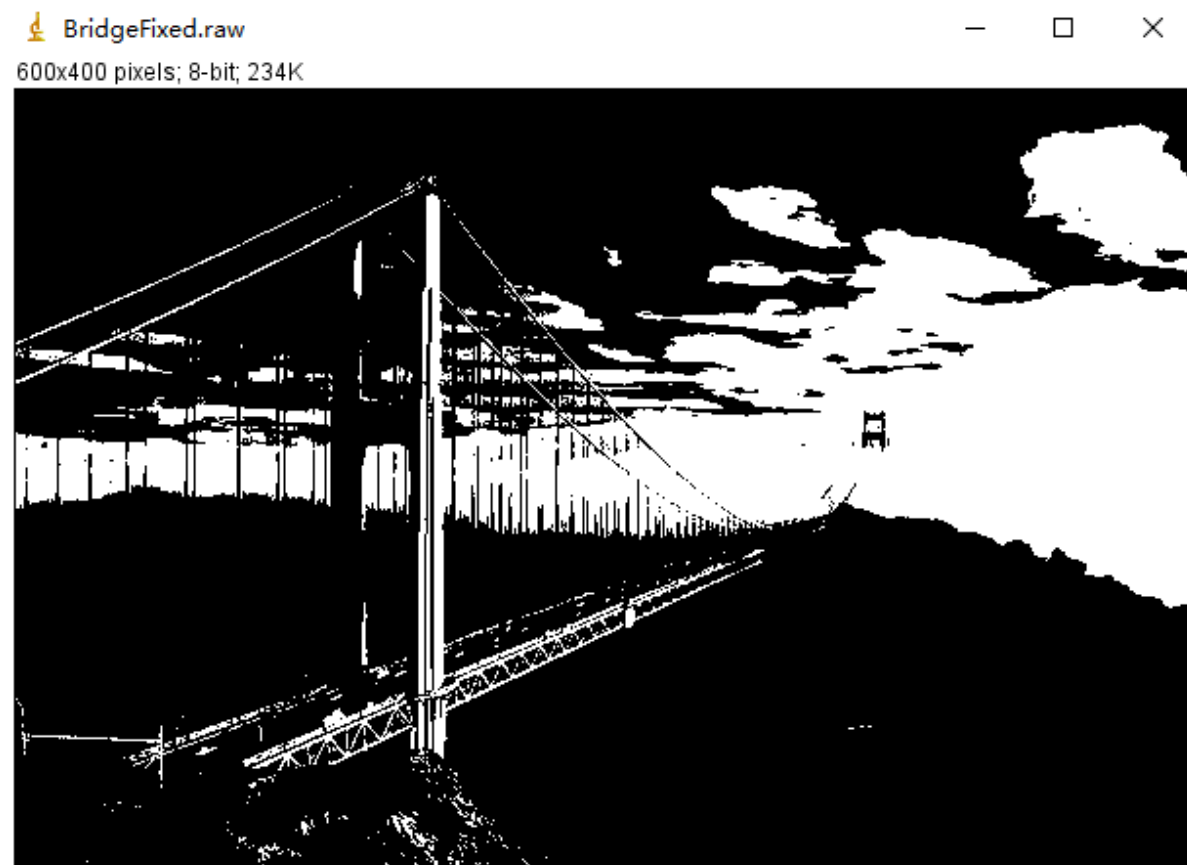
2. Random Thresholding

Command for the two thresholding part

g++ -o Thresholding Thresholding.cpp

Thresholding

Output: BridgeFixed.raw, BridgeRandom.raw

3. Dithering Matrix

31 223

159 95

threshold matrix

I_2

7 199 55 247

135 71 183 119

39 231 23 215

167 103 151 87

threshold matrix

I_4

1 193 49 241 13 205 61 253

129 65 177 113 141 77 189 125

33 225 17 209 45 237 29 221

161 97 145 81 173 109 157 93

9 201 57 249 5 197 53 245

137 73 185 121 133 69 181 117

41 233 25 217 37 229 21 213

169 105 153 89 165 101 149 85

$$I_8$$

When the dithering matrix goes larger, it seems the image has more detail and more grey scale, making it more 'colorful'. It's easy to understand because, for example, all intensity from 0 to 64 will be classified to the same scale by I_2, but it will be classified to 16 different scales by I_8, since I_8 has more refined thresholds.

Command

g++ -o DitheringMatrix DitheringMatrix.cpp

DitheringMatrix {n} Output/Dithering{n}.raw

Output: Dithering{n}.raw

b.Error Diffusion

Floyd-Steinberg's error diffusion

JJN error diffusion

Stucki's error diffusion

I think JJN and Stucki are almost identical, and both of them perform better than FS, because they seem clearer,for example, the cables on the bridge. All the three diffusion image generate some kind of texture and dots.

Compared with dithering, error diffusion seems more uniform. In the dithering, you can literally see those small squares stack on each other. But personally I would still prefer dithering because those salt dots in error diffusion images makes me uncomfortable.

How to improve

Maybe we can combine Dithering and error diffusion. Currently in error diffusion, we use fixed threshold of 128. If they're combined together, I suppose it may introduce more randomness and therefore reduce the unwanted texture. Also, I'd like to try bigger diffusion matrix. Since JJN and Stcuki are 5 * 5 and performs better than FS 3 * 3, it's natural to think about bigger one like 7 * 7. And if the error diffuse further instead of concentrate on few of the pixels nearby, pixel value will not be changed too much by others' error, maybe it will suppress those salt dots.

Another idea was using linear filter. We've already learnt how to deal with salts and peppers. And in both dithering and error diffusion, we can see there're lots of impluse generated in the half-toning process. However, I'm worried that the impluses are too dense for the linear filter to properly work everywhere.

Command:

g++ -o ErrorDiffusion ErrorDiffusion.cpp

ErrorDiffusion

Discussion

In dithering matrix part, I found my matrix *I* is [[0,3],[2,1]] instead of [[1,2],[3,0]]. It's very subtle and I tried many times but fail to flip it across the center. But, I firmly believe they have exactly the same performance because they have the same randomness over rows and columns.

In both dithering and error diffusion, there're certain kind of patterns generated. Like the tiny squares in dithering and the mazes in error diffusion. One possible reason is there' not enough randomness. For example, the dithering matrix is completely uniform, and repeat in a strict way. What will happen if we randomly patch the dithering matrixs on the image? These are all possible methods we can try.

# Problem 3: Color Half-toning with Error Diffusion

Motivation

This is an extension of previous part. The most obvious difference for color image and grey image is that color image has two more channels. Most simply, we can repeat it for three times. But there's more we can think about. Color channels are related to each other, so it must be a better choice to deal with them as a whole. In this part, we compare the color half toning methods of separately process each channel, and process them together.

Procedures and results

a.separable error diffusion

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$
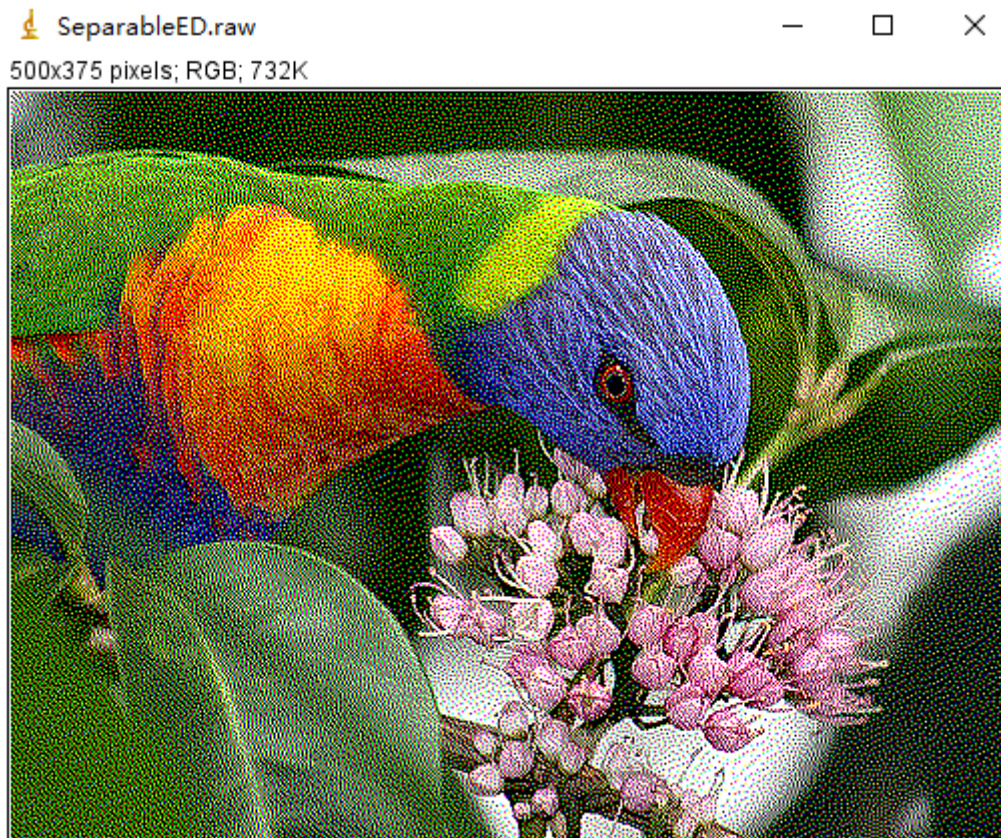
The result seems not bad. But the problem for separatable error diffusion is that is treat three channels independently. But actually they may have some connection. For example, on channel has high intensity probably indicate the other two also have high intensity. By separating them, we lose some information.
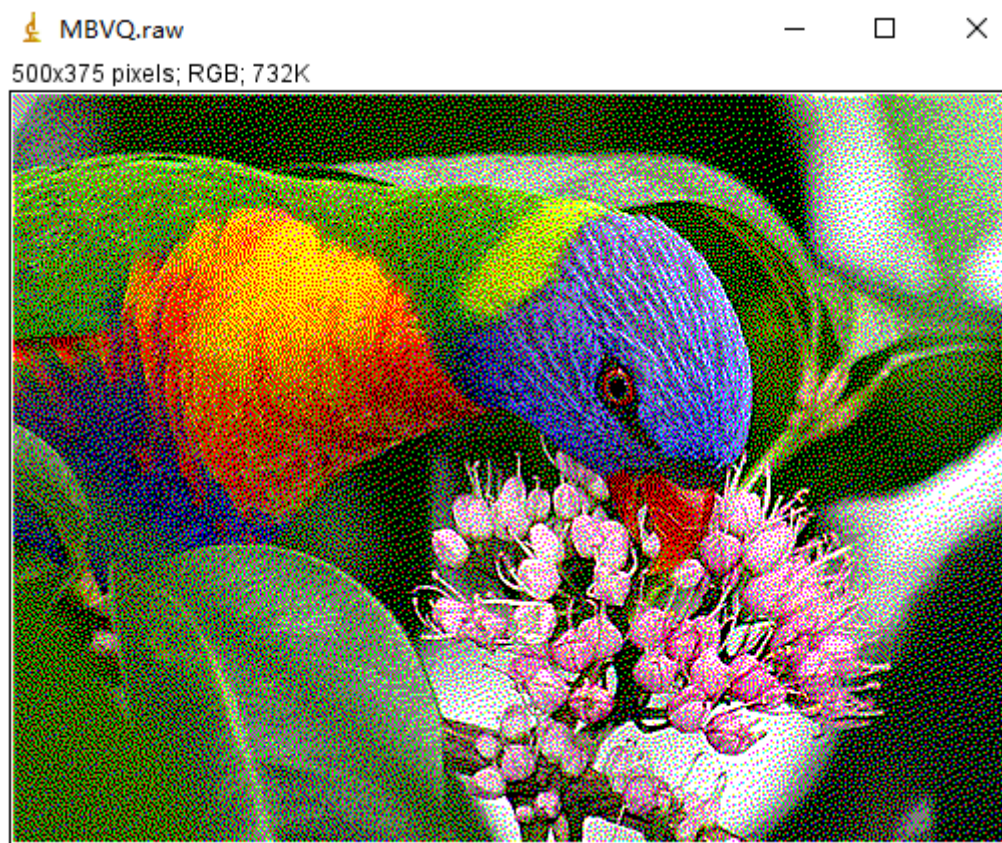
Cammand:

g++ -o SeparableED SeparableED.cpp

SeparableED

Output：SeparableED.raw

b.MVBQ-based error diffusion

The key idea for MBVQ error diffusion is that, it first classify the pixels to six Brightness Quadrants, using all three channel information. Then error diffusion is applied. This will probably gives out half-toing outcome, that is closer to the original intensity, because the three channel together limit the range of adjustment (within the Quadrants), and within one quadrants, the intensity of four possible vertex are similar.
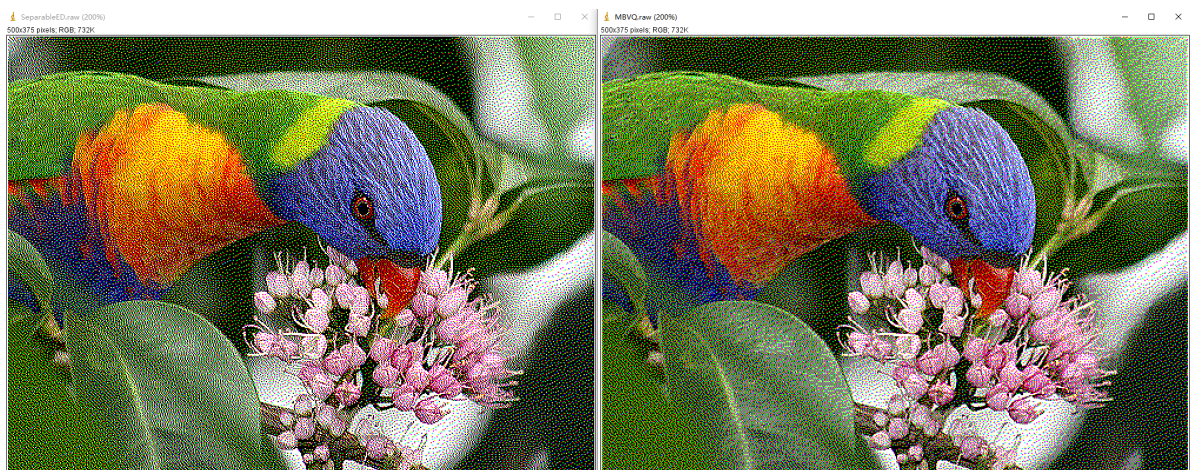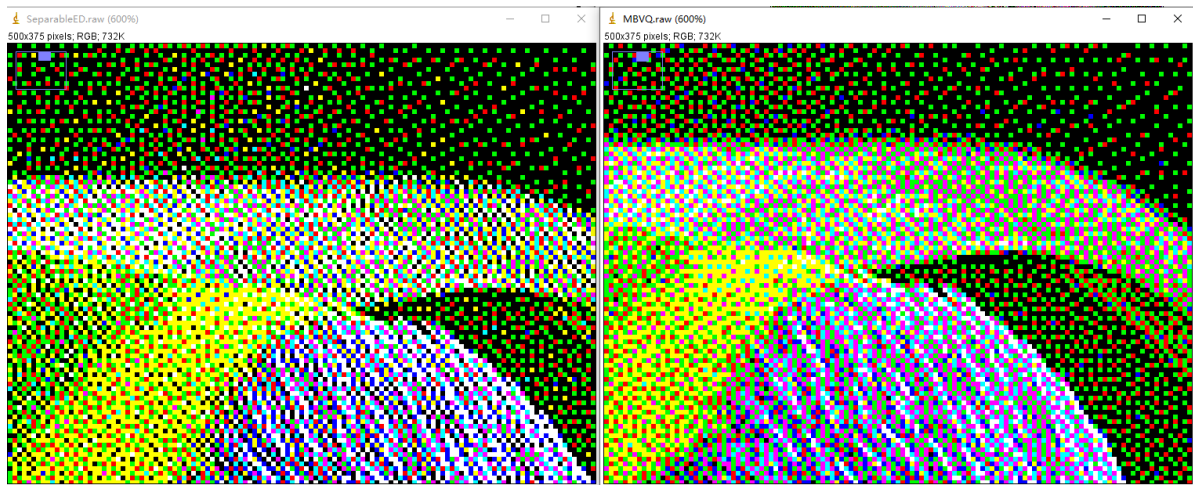
500x375 pixels; RGB; 732K

Cammand:

g++ -o MBVQ MBVQ.cpp

MBVQ

Output:MBVQ.raw

Left: separable error diffusion Right: MBVQ error diffusion

On the first look, I may say the two pictures are identical. But if we check it carefully, we can see from the zoom-in figure that there're less black and white dots. This means MBVQ method preserve more color information than separable ED.

Discussion

When doing error diffusion, I do not understand why we should operate in CMY domain. Actually I've tried to do these in RGB domain and get the same outcome. The only reason I can think about is, when we are required to generate binary document that is used to be printed, then we have to keep the representation in CMY. But now we are going to display them on computer, so we should transfer to CMY domain then transfer back to RGB.

About the difference of separable error diffusion, honestly I think they can be ignored in most cases without high standard. But it may be useful when pre-process the images for maybe deep learning purpose (will deep learning use half-toning? I'm not sure). According to Doron Shaked's article, MBVQ take 1.55 times of time than separable ED. It may provide an alternative of higher quality but slightly longer time for printer users.

Feedback:

Q1: (40')

1. (3') -1 Need to mention that Sobel is a type of 1st derivative filter. -4 Wrong normalized gx, gy map. -2 Wrong Normalized magnitude map.
2. (10') good
3. (13') -0.5 Wrong Flow chart. -0.5 Missing explain what the difference of Structured Random Forest from Random Forest is, and what does the purpose introduce such difference. -1 Missing explain the process of decision tree construction.
4. (14') -1 For each threshold: show the mean P and mean R over GT (99*1 vector, 99 is #of threshold);

Q2: (27')

1. (14') -1 Need to explain that halftoning with random threshold introduces random noise, which is an obvious disadvantage of this method.
2. (13') -1 Missing explain Boundary case: the part of error diffusion mask that is outside image borders is omitted. -1 Missing explain Serpentine parsing requires horizontally flip error diffusion matrix when scanning from right to left.

Q3: (19')

1. (4') good
2. (15') -1 Explain both have maze-like pattern due to the FS error diffusion algorithm