Homework #2

B05902120 / Yu-Ting, TSENG

Mar 23, 2019

Basic Execution

- Platform: Unix (MacBook)
- Language: C++

Problem1: Edge Detection

Given three images as shown in Fig.1. For each given image, you are required to generate several edge maps using the following methods. [Please mark the edge points with intensity value 1 and background points with intensity value 0.]
For each method, please apply different parameters and provide some discussions on how they would affect the resultant edge maps. From the observations of your results, please list the pros and cons of each method.

(a) Perform 1^{st} order edge detection and output the edge map as E_1.
1^{st} order is actually the first gradient. In a discrete case, we can approximately obtain the gradient by taking eight grids into consider. We might be capable of calculating the row gradient and the column gradient respectively; and furthermore combine them into G.

$$G_r = \frac{1}{K+2}[(A_2 + KA_3 + A_4) - (A_0 + KA_7 + A_6)]$$
$$G_c = \frac{1}{K+2}[(A_0 + KA_1 + A_2) - (A_6 + KA_5 + A_4)]$$
$$G(i,j) = \sqrt{G_r(i,j)^2 + G_c(i,j)}$$

In addition, we need to find a proper threshold that can identify edge and non-edge. The gradient value that bigger than threshold would be view as edge, the smaller one would be see as non-edge on the contrary. The following is the corresponding code.

```
for (int i = 1; i < size - 1; i ++)
```

```
for (int j = 1; j < size - 1; j ++){
    double tmpi = 0;
    tmpi += (img[i][j + 1] - img[i][j - 1]) * c;
    tmpi += img[i - 1][j + 1] - img[i - 1][j - 1];
    tmpi += img[i + 1][j + 1] - img[i + 1][j - 1];
    tmpi /= c + 2;

    double tmpj = 0;
    tmpj += (img[i + 1][j] - img[i - 1][j]) * c;
    tmpj += img[i + 1][j - 1] - img[i - 1][j - 1];
    tmpj += img[i + 1][j + 1] - img[i - 1][j + 1];
    tmpj /= c + 2;

    int grad = sqrt(tmpi * tmpi + tmpj * tmpj);
    if (grad > T) tmp[i][j] = 0;
    if (grad <= T) tmp[i][j] = 255;
}
```

After computing the 1_{st} order gradient of each pixels, we draw a histogram of the distribution function to find the best threshold.

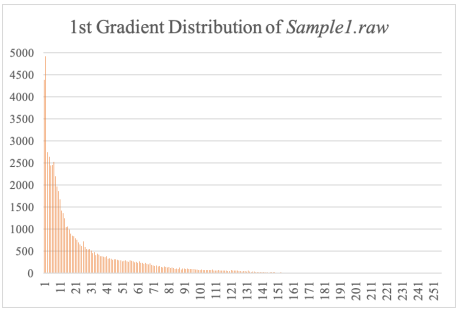

Figure 1: G's Histogram of *sample*1

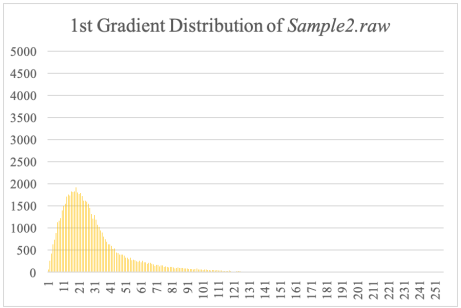

Figure 2: G's Histogram of *sample*2

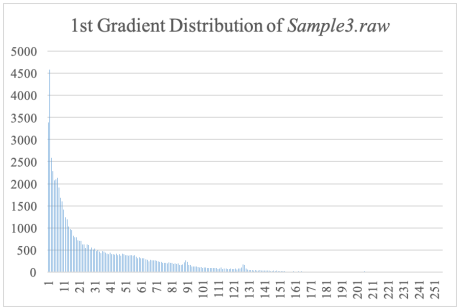

Figure 3: G's Histogram of *sample*3

In the basis on the histograms, we can still give several tries. We choose four different parameters for each figure, two distinct K and two distinct T. (Prewitt is where $K = 1$ and Sobel is which $K = 2$.) The following images is the results of *sample*1.

Figure 4: Prewitt when $T = 30$

Figure 5: Prewitt when $T = 40$

Figure 6: E_1.png Sobel when $T = 30$

Figure 7: Sobel when $T = 40$

It seems this method do well. Let's take a look at the results of *sample*2!

Figure 8: Prewitt when $T = 35$

Figure 9: Prewitt when $T = 40$

Figure 10: E_1.png Sobel when $T = 35$

Figure 11: Sobel when $T = 40$

We can obviously observe that the noise of the image is amplified and become more clear in the picture! The images below comes from *sample*3.

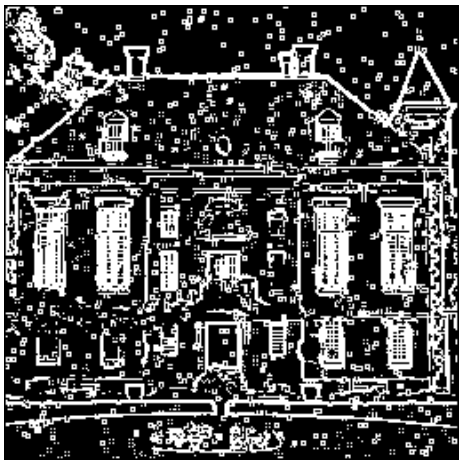

Figure 12: E_1.png Prewitt when $T = 40$

Figure 13: Sobel when $T = 40$

(b) Perform 2^{nd} order edge detection and output the edge map as E_2. The most common seen method of Second Order Gradient is probably "LoG (Laplacian of Gaussian). We reduce noise by Gaussian Low Pass Filter at the beginning, which was the requirement of last assignment. The next step, is to compute the gradient and find the edge by zero-crossing. In this step, we might meet some problems. Zero-crossing is not where the second order gradient equals to 0! We choose the threshold, the value within this interval would be think as potential zero-crossing point.

```
for (int i = 1; i < size - 1; i ++)
    for (int j = 1; j < size - 1; j ++){
        grad[i][j] += smo[i][j] * 4;
        grad[i][j] += smo[i][j - 1] + smo[i - 1][j];
        grad[i][j] += smo[i][j + 1] + smo[i + 1][j];
```

```
            grad[i][j] += smo[i - 1][j - 1] * -2;
            grad[i][j] += smo[i - 1][j + 1] * -2;
            grad[i][j] += smo[i + 1][j - 1] * -2;
            grad[i][j] += smo[i + 1][j + 1] * -2;
            grad[i][j] /= 8;

            if (abs(grad[i][j]) < T) grad[i][j] = 0;
            if (abs(grad[i][j]) >= T) grad[i][j] = grad[i][j
               ];
        }
```

After labeling all the pixels and finding out the possible edge points, we would need to calculate more precisely. We check if the gradient values of two sides is different in order to determine the edge, since the edge means the two pixels of two side is significantly different. At last, we might be able to get the eventual images.

Figure 14: E_2.png'

Figure 15: E_2.png'

Figure 16: E_2.png'

(c) Perform Canny order edge detection and output the edge map as E_3.

It's a method similar to the First Order Gradient. We reduce the noise before calculating. Besides computing the gradient, we double check whether it is a real edge or not. We select two thresholds T_H and T_L, we consider the one which gradient value greater than T_H is definitely an edge, so-called "strong-edge", and the one that smaller than T_L is a non-edge on the other hand. The values that within two threshold would be double check whether there are a strong-edge in their neighbor which make the possibility of they are edge become greater, which is called "weak-edge". The implementation is somehow resemble with the attached code shown above.

Figure 17: E_3.png

Figure 18: E_3.png

Figure 19: E_3.png

We might observe the noise has been reduce a lot, it looks clear compared to the resultant images of Problem1.(a). Without doubt, the noise reduction at the first step as well as the double check mechanism improve the results dramatically.

(d) Other Findings:

Observing the images above, we could discover some interesting phenomenon. The table analyze the pros and cons of each methods!

-	1^{st} Order	2^{nd} Order	Canny Order
Pros	1. Easily implement; 2. Relative fast.	1. Gaussian leads to better Edge Detection; 2. Sensitive toward Noise.	1. Clearly Obtain the Edge without impacting by Noise.
Cons	1. Extremely Sensitive to Noise.	1. Lots of Noise makes Images Unclear; 2. Two Steps Increase the Implement Time.	1. Threshold Selection isn't that Simple; 2. Many steps Slow Down the Speed more than the Former.

Problem2: Geometrical Modification

Given a gray-level image I_2 as shown in Fig.2(a), please follow the instructions below to create several new images.

(a) Perform edge crispening on I_2 and denote the result as C. Show the parameters and provide some discussions on the result as well.

The motivation for us to do edge crispening is to enhance the difference between two sides of the edge, which mean we can identify the edge more accurately. There are two method could be adopted: "High-Pass Filter" and "Unsharp Masking". I choose the latter since the first method amplify the noise simultaneously.

The first thing we need to do is LowPass Filtering. We blur the image at the begin by the filter we decide. As we have mentioned in the report of last assignment, the most common filter we be like this, where we choose the parameter b to be 8:

$$\begin{bmatrix} 1 & b & 1 \\ b & b^2 & b \\ 1 & b & 1 \end{bmatrix} = \begin{bmatrix} 1 & 8 & 1 \\ 8 & 64 & 8 \\ 1 & 8 & 1 \end{bmatrix} \tag{1}$$

The next step is combined the original image and the one after LowPass Filter. Too some degree, what is being done is similar to enhance the two sides of the edge by computing as well as enhancing through the proportion of original image and the LowPass Filtering image.

$$G(i,j) = \frac{c}{2c-1}F_{ori}(i,j) - \frac{1-c}{2c-1}F_{low}(i,j), \text{ where } 0.6 \leq c \leq 0.83$$

Look deeply into the equation above, it seems like the formula of outer division point, it somewhat prove my assumption. After several testing, we find that when $c = 0.65$ perform the best. Furthermore, to implement we write the code:

```
for (int i = 0; i < size; i ++)
```

```
for (int j = 0; j < size; j ++){
    tmp[i][j] = img[i][j] * c - low[i][j] * (1 - c);
    tmp[i][j] = tmp[i][j] / (2 * c - 1);
}
```

Now, we show the two images which are the resultant of LowPass Filtering and Combination respectively.

Figure 20: After LowPass Filtering

Figure 21: C.png

(b) Design a warping method to produce D from C. As shown in Fig.2(b), D is a swirled disk with diameter of 256 pixels.

First, I design an algorithm to turn the square-like image into circle-like image. The concept is "projection". Shrinking and expanding the length to the center based on the ratio of edge of the original image and 128 (the radius of the circle). There are two distinct types of adjusting the length: the expanded line connect the center and the point could be to the edge at top or at bottom; on the other hand, the expanded line could be on the left or right edge. There are some slight difference between them over computing:

1. $length_{new} = length * i * \sqrt{i^2 + j^2}$
2. $length_{new} = length * j * \sqrt{i^2 + j^2}$

According to the above mentioning, we can turn the image into the circle one by the code and get the result as shown below. During my first trial, I compute the value from the original image and fill the pixel of the resultant image. However, I find that there will be some pixels not being computed since we might ignore some location when changing `double` into `int`. To solve the problem, we must give

a second try, to do backward! We compute the original location we need based on the current location. In this case, we can make sure all the pixel values have been computed without skipping. `i` and `j` are the pixel location of the new image currently. `tmpi` and `tmpj` represent the original coordinating location which is computing from the above mentioning ration function. The center point is view as $(255/2, 255/2) = (127.5, 127.5)$.

```
for (int i = 0; i < size; i ++)
    for (int j = 0; j < size; j ++){
        // length between center and the point
        double tmpi = i - 127.5;
        double tmpj = j - 127.5;

        double rat = 0;
        if (abs(tmpi) > abs(tmpj))
            rat = tmpi / sqrt(tmpi*tmpi + tmpj*tmpj);
        if (abs(tmpi) <= abs(tmpj))
            rat = tmpj / sqrt(tmpi*tmpi + tmpj*tmpj);

        tmpi = tmpi / abs(rat) + 127.5;
        tmpj = tmpj / abs(rat) + 127.5;

        tmp[i][j] = img[int(tmpi)][int(tmpj)];
    }
```

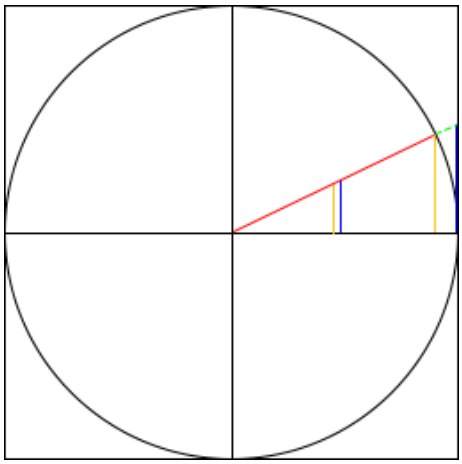

Figure 22: Projection from Square to Circle

Figure 23: D_{circle}.png

After this step, we might be able to start doing the operation, swirling! The im-

portant concept of "swirling" is to rotate the image by different angle based on the length between the point and the center. Therefore, I set up the angle to be $0.5 * \pi * (1 - \sqrt{i^2 + j^2}/128)$. In addition, I compute the new coordinate location via rotation matrix.

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \tag{2}$$

We can calculate and simplify the equation, and transform them into the format that we can represent and calculate by code. Notice that we also choose to compute the value backward!

```
for (int i = 0; i < size; i ++)
    for (int j = 0; j < size; j ++){
        // length between center and the point
        double tmpi = i - 127.5;
        double tmpj = j - 127.5;

        double t = 0.5 * pi * (1 - sqrt(tmpi * tmpi +
           tmpj * tmpj) / 128);

        double resi = tmpi*cos(t) - tmpj*sin(t) + 127.5;
        double resj = tmpi*sin(t) + tmpj*cos(t) + 127.5;

        res[i][j] = tmp[int(resi)][int(resj)];
    }
```

Eventually, we might be able to get the similar resultant image with Fig.2(b) which is displayed below.

Figure 24: D_{circle}.png

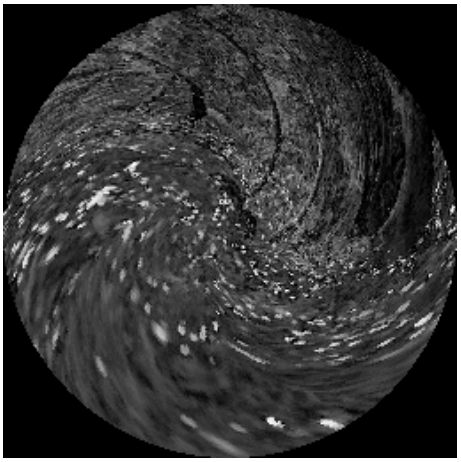

Figure 25: D.png

Appendix

- `Problem1_a.cpp` to execute edge detection by 1^{st} order gradient. Compile by `g++ Problem1_a.cpp -o Problem1_a`;
  execute by `./Problem1_a` and input K (Prewitt or Sobel) and the threshold value T then get the output images three `E_1.raw` and the output file `Problem1_a.csv`.
  * Plot the histograms by using the function of excel.

- `Problem1_b.cpp` to execute edge detection by 2^{nd} order gradient. Compile by `g++ Problem1_b.cpp -o Problem1_b`;
  execute by `./Problem1_b` and input the threshold value T then get the output images three `E_2.raw`.

- `Problem1_c.cpp` to execute edge detection by Canny order gradient. Compile by `g++ Problem1_c.cpp -o Problem1_c`;
  execute by `./Problem1_c` and input the two threshold value T_H and T_L then get the output images three `E_3.raw`.

- `Problem2_a.cpp` to execute edge crispening.
  Compile by `g++ Problem2_a.cpp -o Problem2_a`;
  execute by `./Problem2_a` and input the parameter b for Low Pass Filtering and c for unsharp masking then get the output image `C`.raw.

- `Problem2_b.cpp` to do warping, more precisely, swirling.
  Compile by `g++ Problem2_b.cpp -o Problem2_b`;
  execute by `./Problem2_b` and get the output image `D`.raw.