DIP Homework Assignment #2

資工三 b05902115 陳建丞

PROBLEM 1: EDGE DETECTION

(a) Perform 1^{st} order edge detection

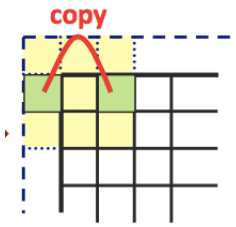
- **Motivation**

  As taught in the lecture, I calculate the gradient of rows and columns. I use **9 points approximation** because it might be more sensitive. All the eight points adjacent to the center point are taken into consideration.

A_0	A_1	A_2
A_7	$F(j,k)$	A_3
A_6	A_5	A_4

- **Approach**

  1. Store the pixel values in a 2d array.
  2. Expand the original image to deal with the border condition.



  3. Set up mask constant K ($K = 1$: Prewitt Mask, $K = 2$: Sobel Mask)
  4. Calculate the gradient (use two for loop2 to run through)
     - $G_R(j,k) = \frac{1}{K+2}[(A_2 + KA_3 + A_4) - (A_0 + KA_7 + A_6)]$
     - $G_c(j,k) = \frac{1}{K+2}[(A_0 + KA_1 + A_2) - (A_6 + KA_5 + A_4)]$
     - $\Rightarrow G(j,k) = \sqrt{(G_r^2(j,k) + G_c^2(j,k))}$
     - The mathematical operation can be implemented with the function in "math.h" library in c++
  5. Pick up a threshold T
     - if $G(j,k) < T \rightarrow$ Set pixel value to 0 (Non-edge point)
     - if $G(j,k) < T \rightarrow$ Set pixel value to 255 (Edge point)

- **Original image (sample1.raw) // no noise**

- **Result**
  - $K = 1$ with different threshold T

T=10 T=30 T=50 T=70

  - $K = 2$ with different threshold T

T=10 T=30 T=50 T=70

- **Original image (sample2.raw) // uniform noise**

- **Result**
  - $K = 1$ with different threshold T

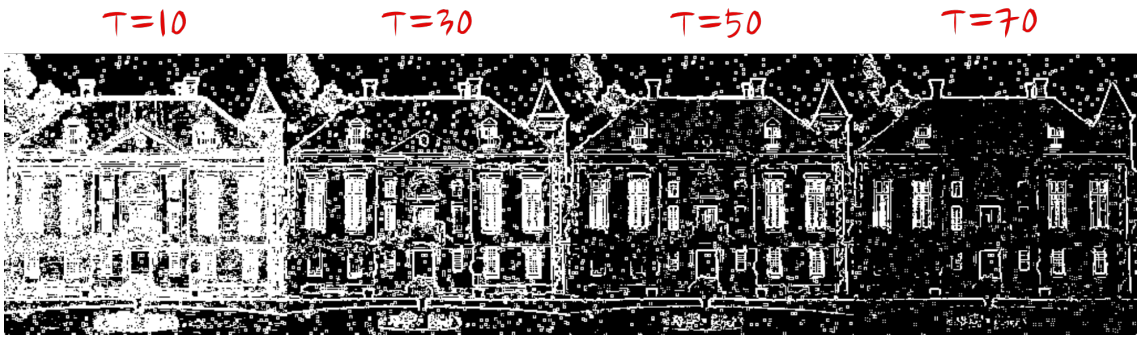
T=10 T=30 T=50 T=70

  - $K = 2$ with different threshold T

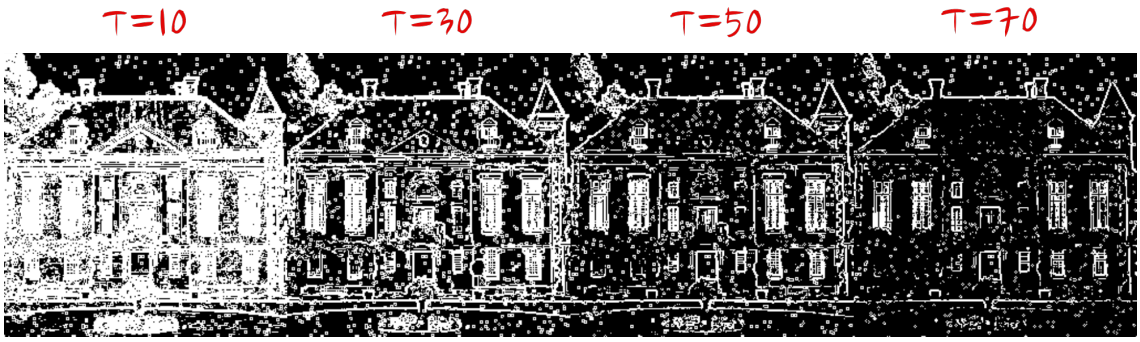
T=10 T=30 T=50 T=70

- **Original image (sample3.raw) // impulse noise**

- **Result**
  - $K = 1$ with different threshold T



  - $K = 2$ with different threshold T



- **Discussion**

  From the result, we can see apparently that 1^{st} order edge detection method is sensitive to noise. Also, since I use a 9 points approximation, the noise is even worse. That said, it can generate a clear edge image if there's no noise, like sample1. The results of sample2 & sample3 have lots of noise remains in the picture.

  The Prewitt mask and Sobel mask have no prominent difference. They might have more difference if the mask size is enlarged to 5x5 or 7x7.

  The threshold T decides the numbers of edge points that we should keep. According to the result, set T between 30 to 50 can output the most clear result.

(b) Perform 2^{nd} order edge detection

- **Motivation**

  Since 1^{st} order edge detection and Canny edge detection sections have focused on the threshold adjustment, in this section, I try to test the difference among each Laplacian mask. I will select suitable thresholds and see the results of different mask.

- **Approach**

  1. Store the pixel values in a 2d array.
  2. Expand the original image to deal with the border condition.
  3. Apply different Laplacian impulse mask H on the input image.

  four-neighbor

  $$H = \frac{1}{4}\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

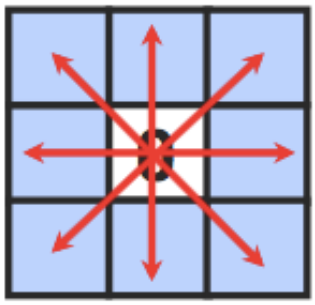
  $$H = \frac{1}{8}\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

  non-separable

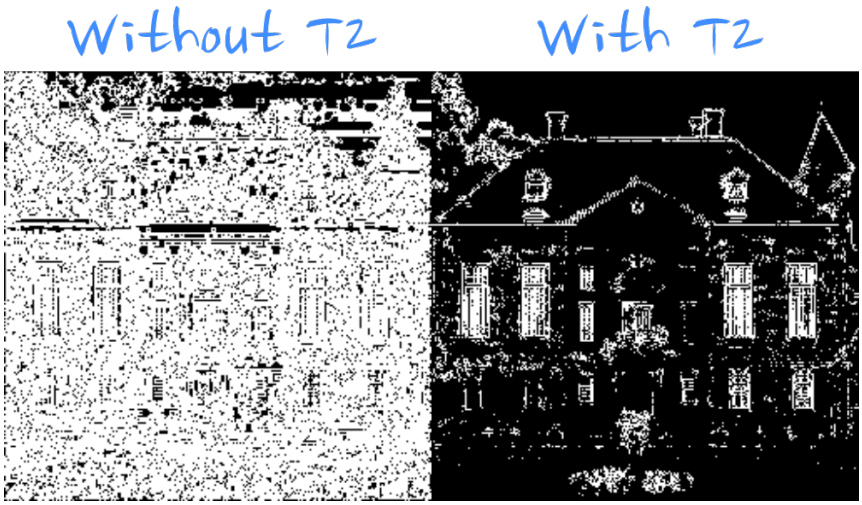
  $$H = \frac{1}{8}\begin{bmatrix} -2 & 1 & -2 \\ 1 & 4 & 1 \\ -2 & 1 & -2 \end{bmatrix}$$

  separable

  4. Pick up a threshold T , if $|G(j,k)| \leq T \Rightarrow$ label the pixel as zero point .
  5. For each $G(j,k) \in$ zero points, multiply the two neighbor values on a red edge line to check if it's a zero-crossing point

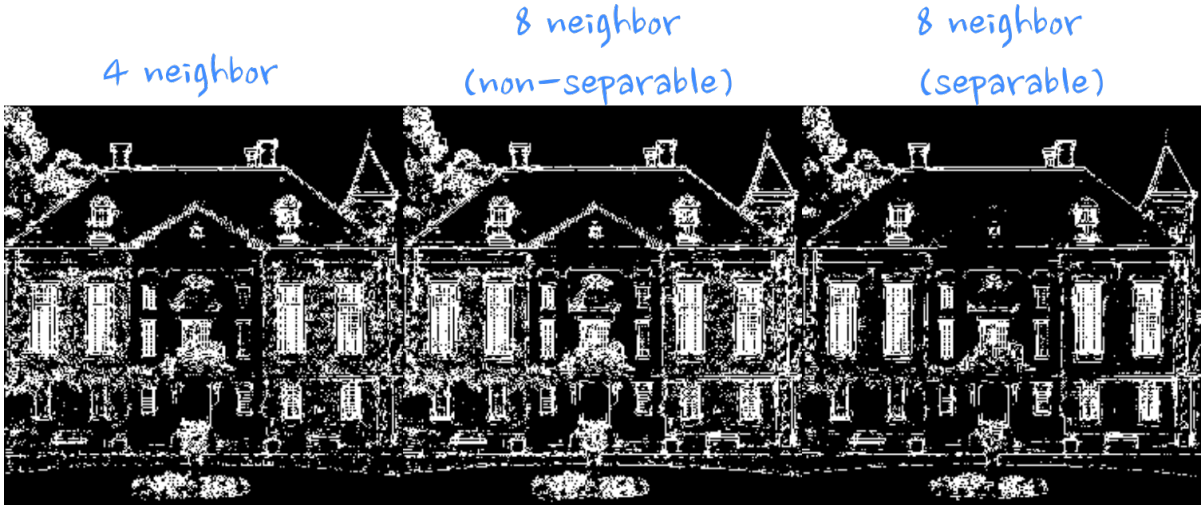
  

     - To control the zero-crossing point numbers, I add another threshold T_2
     - If the multiplied results $\leq -T_2 \Rightarrow$ Set the pixel value to 255 (Edge point)
     - Otherwise, set the pixel value to 0 (Non-edge point)
     - Discuss with B05902115 婁敦傑 (zero-crossing point threshold method)

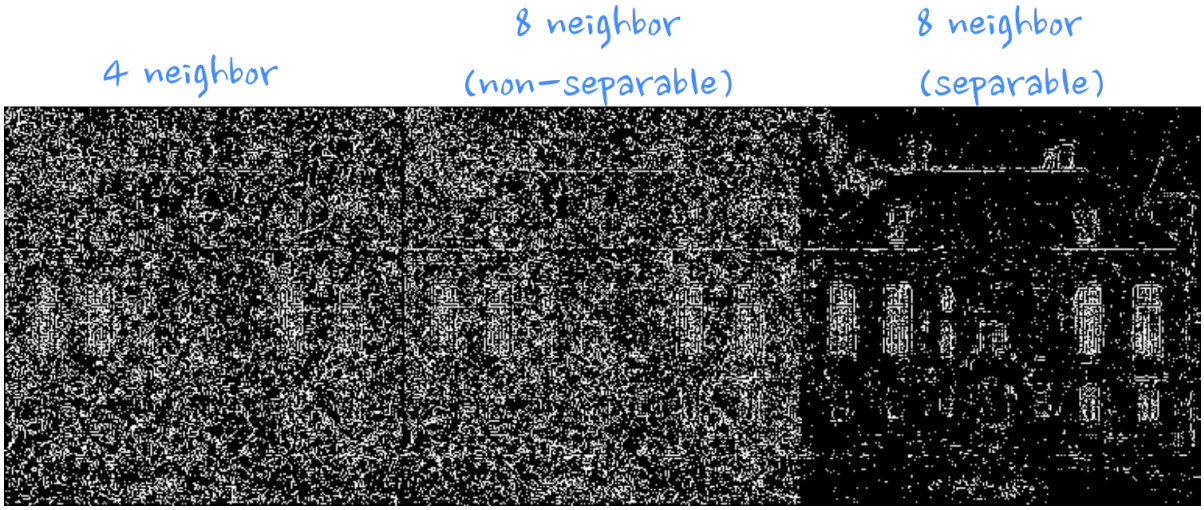
- **Original image (sample1.raw) // no noise**

- **Result (**$T = 70, T_2 = 150$**)**



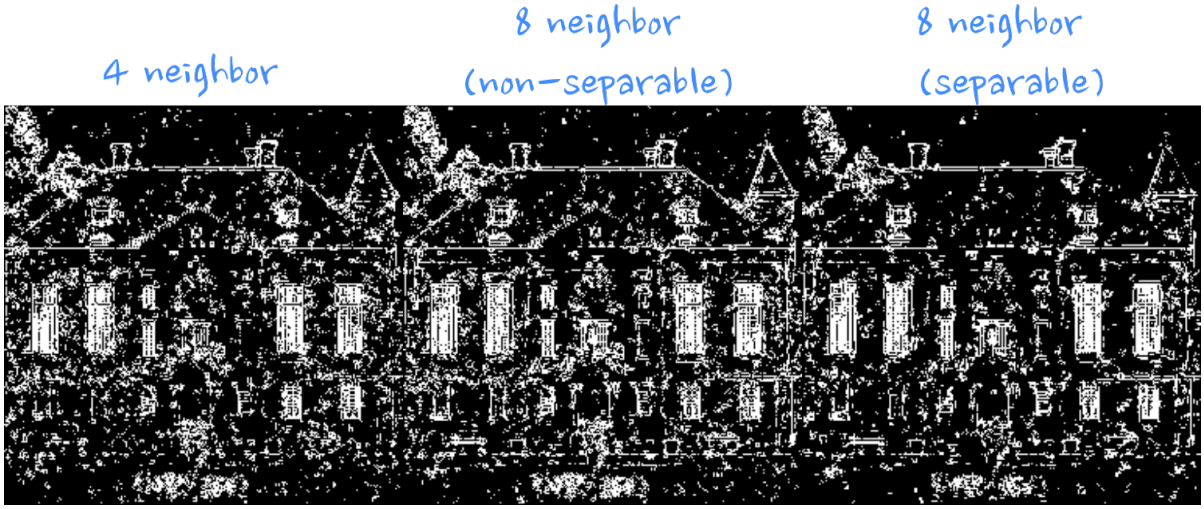
- **Original image (sample2.raw) // uniform nosie**

- **Result (**$T = 30, T_2 = 500$**)**



- **Original image (sample3.raw) // impulse noise**

- **Result ($T = 70, T_2 = 300$)**



- **Discussion**

  The same as 1^{st} order edge detection, $2^n d$ order edge detection is sensitive to noise, too. All the results of sample2, and sample3 are affected by the noise greatly. There must be some noise removal preprocess to cope with such input image. Especially in the case of uniform noise input, the whole result output is covered with noise.

4 neighbor mask and 8 neighbor mask have little difference. With proper thresholds set up, all of these masks can generate clear output in the non-noise case. 8 neighbor separable mask will have a bit less edge points than the others, but the edges are still clear.

(c) Perform Canny edge detection

- **Motivation**

  Since the main purpose of this assignment is to see the edge detection results, I don't put lots of efforts on the noise removal process. I just simply apply the 5x5 Gaussian filter as taught in the lecture slide to smooth the original image. In my implementation, I adjust different thresholds to see the relations between the thresholds and results.

- **Approach**

  1. Store the pixel values in a 2d array.
  2. Expand the original image to deal with the border condition.
  3. Apply a 5x5 Gaussian filter to smooth the image

  $$F_{NR} = \frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} * F$$

  4. Compute the gradient magnitude and orientation
     - The gradient magnitude is the same as 1^{st} order edge detection
     - The orientation $\theta(j,k) = tan^{-1}(\frac{G_c(j,k)}{G_R(j,k)})$

     ```
     theta[j][k] = atan(GC/GR)*180/M_PI
     ```

  5. Search the nearest neighbors along the edge normal

A_0	A_1	A_2
A_7	$F(j,k)$	A_3
A_6	A_5	A_4

     - Since $-90 \leq \theta \leq 90$, we can simplify the direction problem.
     - if $-22.5 \leq \theta(j,k) \leq 22.5 \Rightarrow$ neighbor: A_3, A_7
     - if $22.5 < \theta(j,k) \leq 67.5 \Rightarrow$ neighbor: A_2, A_6
     - if $-22.5 < \theta(j,k) \leq -67.5 \Rightarrow$ neighbor: A_0, A_4
     - if $67.5 < \theta(j,k) \leq 90$ or $-67.5 > \theta(j,k) \geq -90 \Rightarrow$ neighbor: A_1, A_5
  6. Set up two threshold T_L, T_H

$$G_N(x,y) \geq T_H \quad \textbf{Edge Pixel}$$
$$T_H > G_N(x,y) \geq T_L \quad \textbf{Candidate Pixel}$$
$$G_N(x,y) < T_L \quad \textbf{Non-edge Pixel}$$

7. Connected component labeling
    - Use a c++ std queue to implement BFS.
    - Push all the edge pixel into the queue.
    - Pop a pixel and check the eight pixels adjacent to it. If a neighbor is a candidate pixel, then label it as an edge point and push it into the queue.
    - Keep the process until the queue is empty.
8. Set all the edge pixels and labeled candidate pixel to 255 (edge point). And set the other pixels to 0 (non-edge point)

- **Original image (sample1.raw) // no noise**

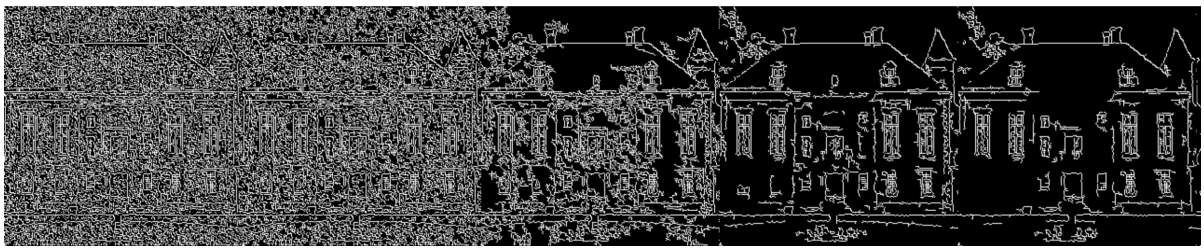
- **Result**

LH = 20 LW = 10 | LH = 30 LW = 10 | LH = 70 LW = 10 | LH = 70 LW = 30 | LH = 90 LW = 30

- **Original image (sample2.raw) // uniform noise**

- **Result**

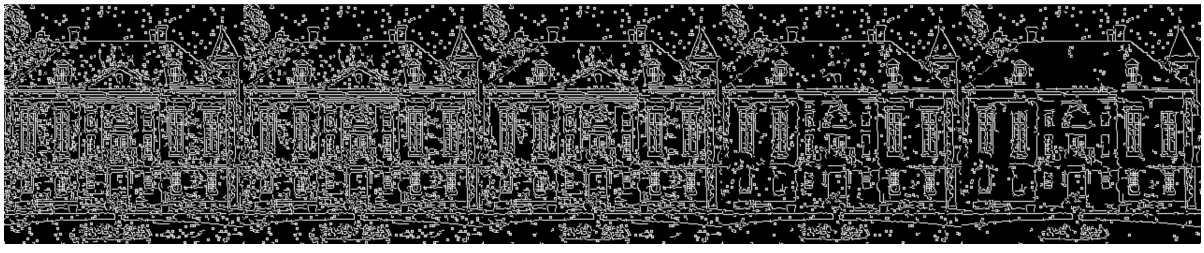
LH = 20 LW = 10 | LH = 30 LW = 10 | LH = 70 LW = 10 | LH = 70 LW = 30 | LH = 90 LW = 30



- **Original image (sample3.raw) // impulse noise**

- **Result**

LH = 20 LW = 10 | LH = 30 LW = 10 | LH = 70 LW = 10 | LH = 70 LW = 30 | LH = 90 LW = 30



- **Discussion**

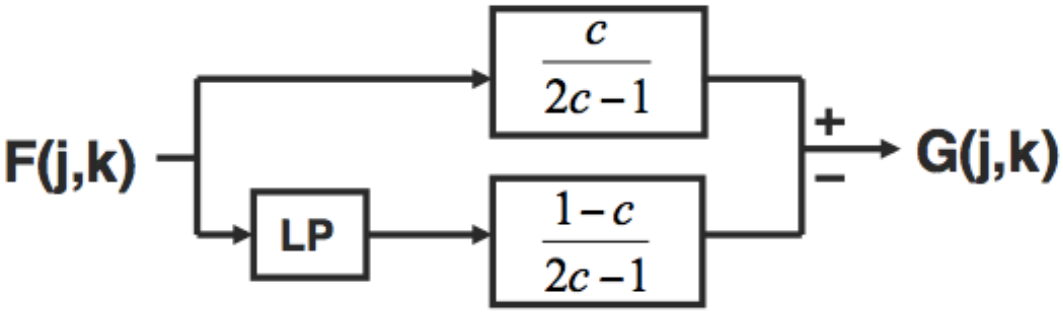
  Since Canny edge detection has a filtering process, it can handle the noise better, like sample2. However, it's still sensitive to impulse noise, and there's still lots of noise remains in the result of sample3. It's probably that we should apply other noise removal preprocess to clean the noise before we do the edge detection.

  The results vary widely according to different threshold values, and each image's suitable thresholds are widely different, too. It's flexible to adjust the two thresholds to determine the intensity of edge we want.

PROBLEM 2: GEOMENTRICAL MODIFICATION

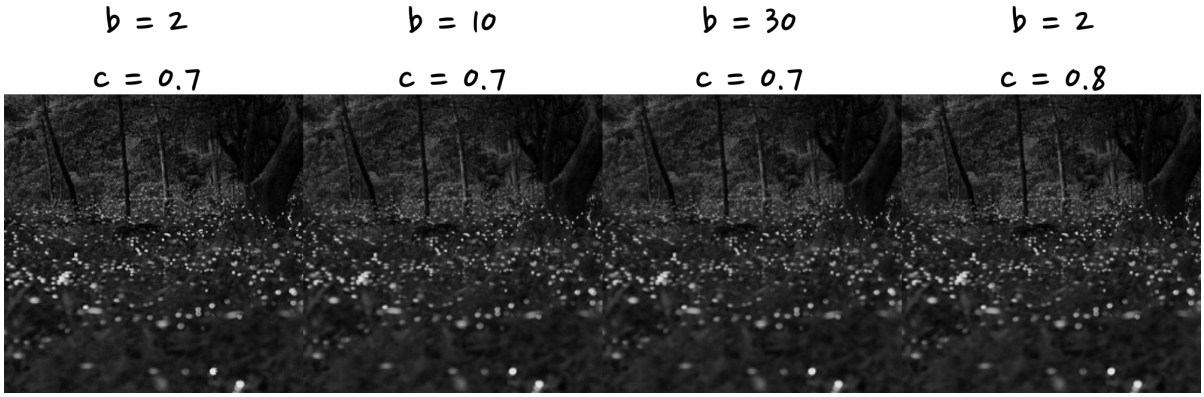
(a) Edge crispening

- **Approach**
  1. Store the pixel values in a 2d array.
  2. Expand the original image to deal with the border condition.
  3. Apply unsharp masking
     - $LP = \frac{1}{(b+2)^2}\begin{bmatrix} 1 & b & 1 \\ b & b^2 & 1 \\ 1 & b & 1 \end{bmatrix}$



  4. Adjust b & c
- **Original image**

- **Result**



- **Discussion**

  There is no prominent change between the result images and the original images. The upper portion becomes a little sharper than the orginal one, but it's really not very clear from macroscopic observation. Besides, I can't tell from my eyes if there's any difference among different parameters.

(b) Warping

- **Motivation**

  The swirled disk effect is like segmenting the image into many concentric circles. All the pixels on the circle will rotate. The rotation angle on a same circle is the same while different circle will have different rotation angle, proportional to the radius. The rotation angle is greater if the radius is bigger.

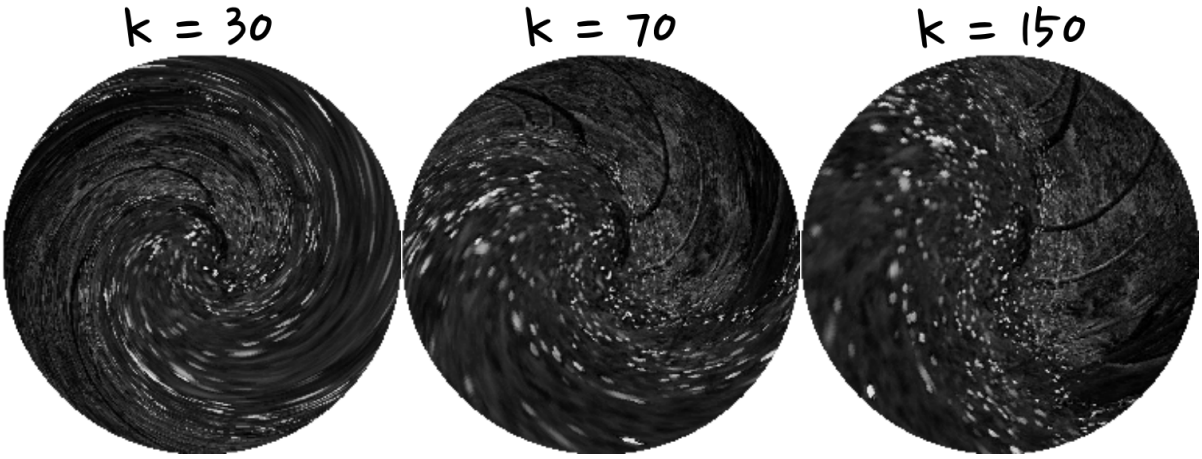
- **Approach**

  1. Shift the image onto a polar coordinate system ($F(j,k) = P(j-128, k-128)$)
  2. Remove the pixel that is out of the inscribed circle of the image.
     - Calculate the distance of the pixel to the $P(0,0)$
     - If the distance $\geq$ 128, set the pixel value to 0
  3. Rotate the pixel according to the distance.
     - Since the image are in a polar corrdinate system, the new coordinate after rotation is $P'(j,k) = (distance * cos\theta,\ \ distance * sin\theta)$

    - I set the rotation angle θ as $2 - \frac{distance}{k}(rad)$
    - Adjust k to have different swirl effect.
4. Use the new coordinate to backward trace the original image.

- **Original image (edge crispening with b = 2, c = 0.7)**

- **Result**



- **Discussion**

  We can observe from the result or from the θ forumla above, when k is smaller, the swirl effect is stronger. I use a linear function to detemine the rotation angles of the concentics circles, maybe more differenct function can be applied to generate special swirl image. In this case, the linear function I choose with k = 70 is the most indentical one to the expected result.

Electronic version README

- **The input image should be in the directory raw**
- call **make –f README**
- The programs will be compiled and executed automatically after the command.
- Output images will be stored in the folder **output**.