HOMEWORK 3

Jiazhi Li

jiazhil@usc.edu

March 2, 2019

1 Geometric Transformation

1.1 Abstract and Motivation

Geometric transformation consists of a set of modification operations such as rotation, translation and scaling. Except for image enhancement and restoration, this procedure is also one of the most important steps in image processing. They are extensively used in computer graphics, image morphing.[1] And, in this example, it involves the following procedures:

- Find the coordinates of corners for each sub-image
- Design the generic transform for each sub-image
- Find the coordinates of corners of holes
- Scaling the sub-image to the size of 160 * 160
- Fill into the holes to get the final image

1.2 Approach and Procedures

1.2.1 Find the coordinates of corners for each sub-image

At the very beginning, the straightforward method by searching row by row and column by column is used to find the coordinates of corners. But it is not very accurate. Having observe the corner of lighthouse1 in **Figure** 1, if we just use row-by-row searching method to go through the whole pixels to find the first nonzero pixels as the corner, the pixel with red circle will become the corner, which is not what we wants.

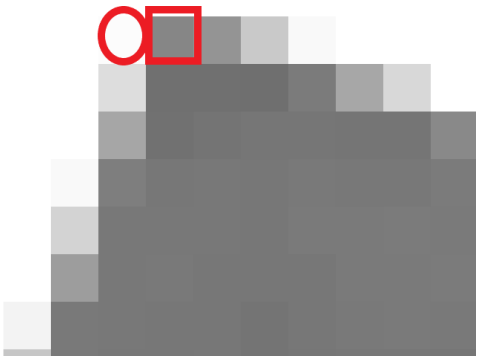
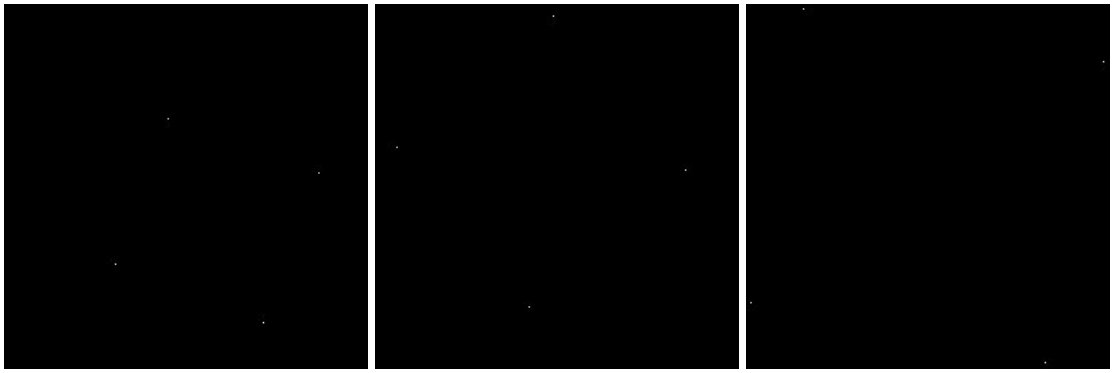


Figure 1: Details about corner in sub-image

By this way, I use the Sobel operator to calculate the x and y derivative of the image. And, we know the corner pixel have the significant changes in x and y direction so that the pixel with red rectangle will become the corner which is more accurate than the formal method.
After that, all four corner pixels have been found, which is shown in **Figure** 2.

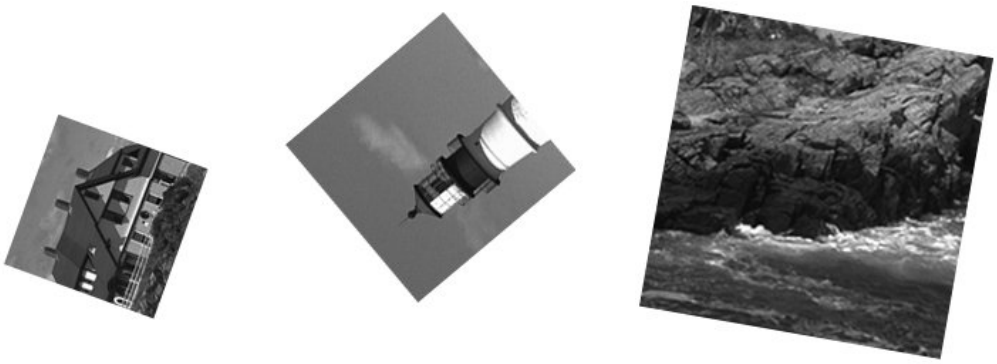


(a) lighthouse1 (b) lighthouse2 (c) lighthouse3

Figure 2: Corners in sub-image

1.2.2 Design the generic transform for each sub-image

For each origin images, though they lean in different degree shown in **Figure** 3, all of them can be define as the following model shown in **Figure** 4.



(a) lighthouse1 (b) lighthouse2 (c) lighthouse3

Figure 3: Origin sub-images

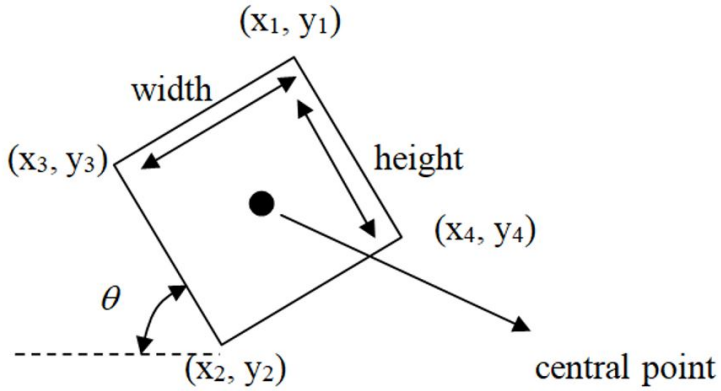


Figure 4: Coordinates model

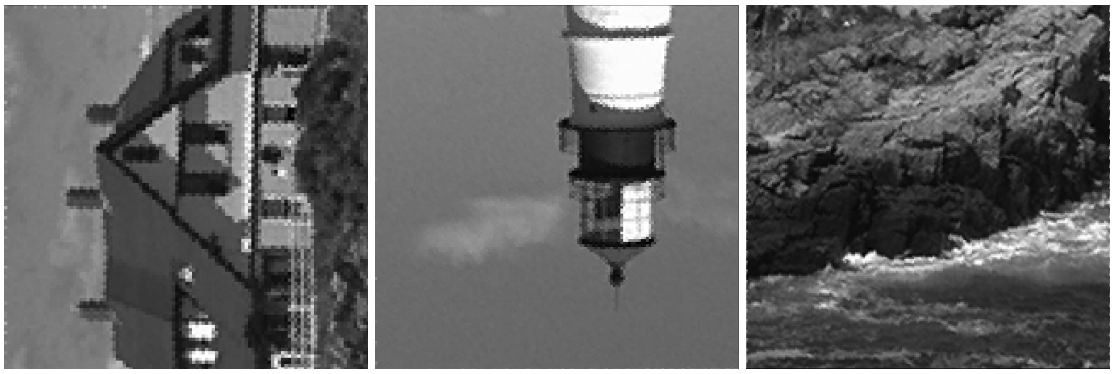
For image transformation, there are three types of basic operation to design first. They are:

- Rotation
- Scaling
- Translation

With the model shown before, the parameter of three sub-images is shown in Table1. And in this step, I just use rotation and translation to get the image with its sides aligned with the horizontal and vertical axis in the same height and width, which is shown in **Figure** 5.

Table 1: Parameter for three sub-images

	height	width	theta (Radian)
lighthouse1	113	113	0.35052
lighthouse2	144	144	0.87781
lighthouse3	214	214	0.17518



(a) lighthouse1 (b) lighthouse2 (c) lighthouse3

Figure 5: Sub-images after rotation and translation

1.2.3 Find the coordinates of corners of holes

Finding the corners of holes is very simple. Since we already know the size of holes is 160 * 160, we just need to check using a window with 255 gray-level pixel in each corner. Then the coordinates of four corners in three holes come out, which is shown in **Figure** 6

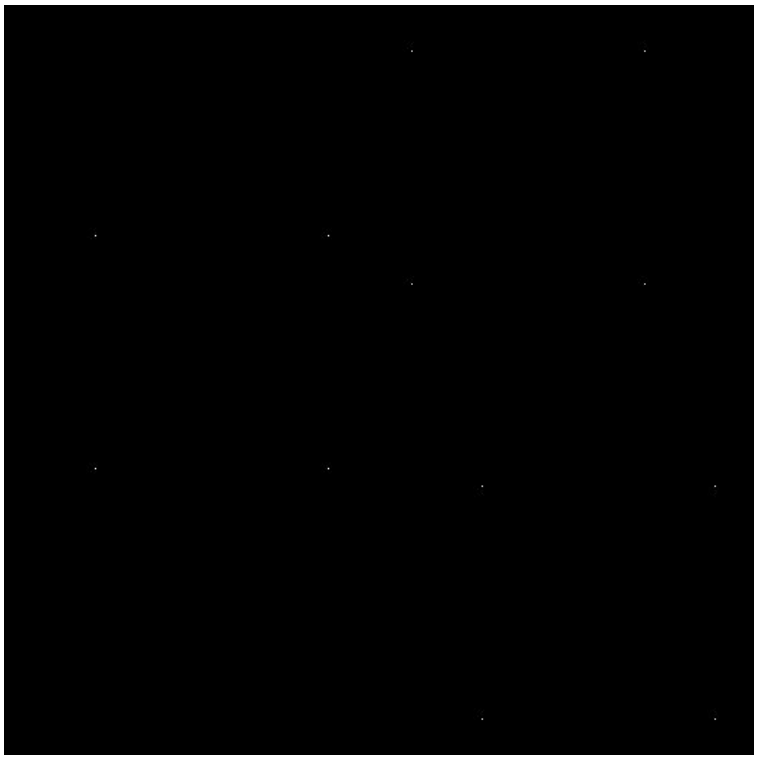


Figure 6: Corner in the image with holes

1.2.4 Scaling the sub-image to the size of 160 * 160

In this step, we need to check whether the degree of sub-images should be rotated or not firstly. Since we have already got the holes coordinates, we can get the gray-level of edge beside the holes. Then we use the similarity in gray-level of edges beside the holes and the edges of sub-image to determine if the degree is matched. If not, we make a counter-clock rotation for this image in 90 degree and check again until the direction is matched.

After that, we apply scaling to sub-images to get the image with size of 160 * 160. Since the size of lighthouse1 and lighthouse2 is smaller than the size required, the bi-linear interpolation is introduced. And the result is shown in **Figure** 7.

(a) lighthouse1 (b) lighthouse2 (c) lighthouse3

Figure 7: Sub-images after scaling

1.2.5 Fill into the holes to get the final image

The final step is to fill in the holes with the sub-images with size of 160 * 160.

1.3 Experimental Results

The result is shown in **Figure** 8.

Figure 8: Final result

1.4 Discussion

1.4.1 Automatically program design

The whole flow diagram is shown in **Figure** 9.

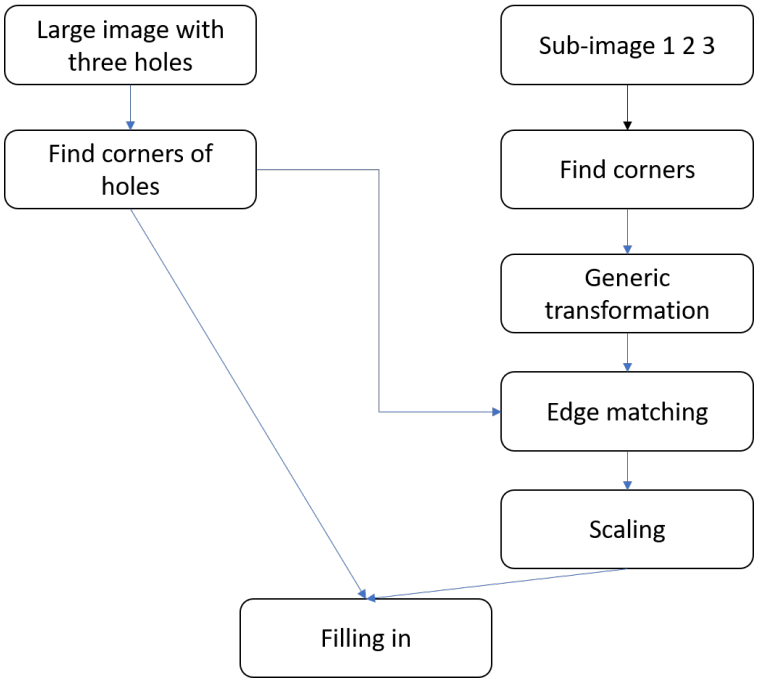


Figure 9: Flow diagram

The program makes this geometric transformation fully-automatically. The only thing you should do is to input the three sub-images and the large image with three holes. The corners are found by my program as well as the coordinates of holes. Even the information that "lighthouse1.raw" goes to the left hole do not need to be used because of edge matching by comparison of similarity of gray-level in both sides.

1.4.2 Optimization

There are two points to be mentioned in optimization about the sides of sub-images. The first one is about the results after rotation. Actually, the sides don't look like pretty well in **Figure** 5. The sides are not even shown in **Figure** 10. I just modify this uneven sides with the closest pixels.

Figure 10: Uneven sides

Also, the final image after filling is shown in **Figure** 11. There are some lines along the edges of filling images after I apply all the necessary geometric modifications. After that, I modify the sides by similarity of edge in the image with holes to get **Figure** 8.

Figure 11: Previous result after all geometric modifications

1.4.3 Reverse mapping function

For the part of scaling and rotation, the reverse mapping function is used in **Figure** 12.

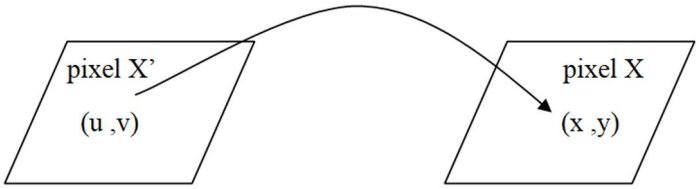


Figure 12: Reverse mapping

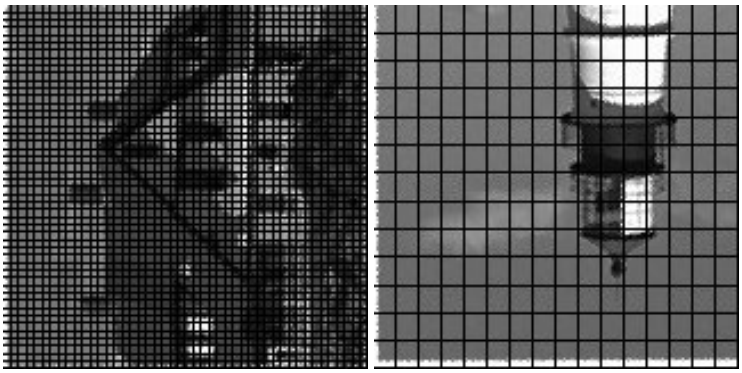
For example,the rotation result of lighthouse1 shown in **Figure** 13. If we use the forward mapping function from (x,y) to (u,v), some pixels in mapping image will not get the exactly position corresponding to origin image because of conversion of numeric data type from double to int. By this way, some information of pixel will be lost. The result can be easily found in **Figure** 13. The output by forward mapping function is just like with impulse noise. And I just use the median filter to wipe off the impulse noise. Although the result after de-noising looks fine, it loses some details compared with result obtained by reverse mapping function.

(a) Forward mapping (b) Forward mapping after median filter (c) Reverse mapping

Figure 13: Comparison between forward mapping and reverse mapping

1.4.4 Bi-linear interpolation

Since the size of lighthouse1 and lighthouse2 are smaller than the size of 160 * 160, the result after scaling is shown in **Figure** 14,



(a) lighthouse1 (b) lighthouse2

Figure 14: Result before Bi-linear interpolation

Thus, Bi-linear interpolation must be used to generate the pixel value at fractional positions. The model of bi-linear interpolation is shown in **Figure** 15.

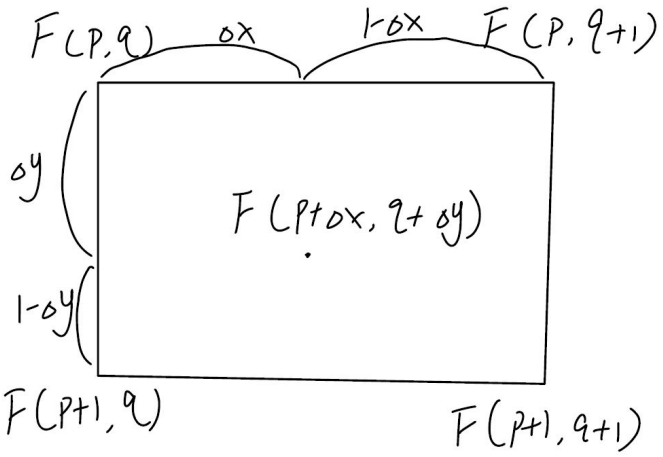


Figure 15: Bi-linear interpolation

Also, bi-linear interpolation involves the reverse mapping function. The formula is

$$F(p+\Delta x, q+\Delta y) = (1-\Delta x)(1-\Delta y)F(p,q)+\Delta x(1-\Delta y)F(p,q+1)+(1-\Delta x)\Delta y F(p+1,q)+\Delta x\Delta y F(p+1,q+1)$$

where

$$0 \leq \Delta x \leq 1$$

$$0 \leq \Delta y \leq 1$$

1.4.5 Computational complexity

The worse case for finding coordinates of corners of each sub-image is $O(m * n)$. But actually, it will not happen. By this way, the most significant part of time complexity is scaling, rotation, bi-linear interpolation

and filling. All of them are with $O(m * n)$, where m and n is the size of sub-image.

2 Spatial Warping

2.1 Abstract and Motivation

Sometimes we want to manipulate an image for considerable distortion in specific shape. By this way, it can be achieved by using the concept of spatial warping. In general, geometrical warping is used to alters the spatial configuration of an image. In this example, triangles is used to achieve such effects.[2]

2.2 Approach and Procedures

Firstly, I set the following coordinate shown in **Figure** 16 for the 512*512 image.

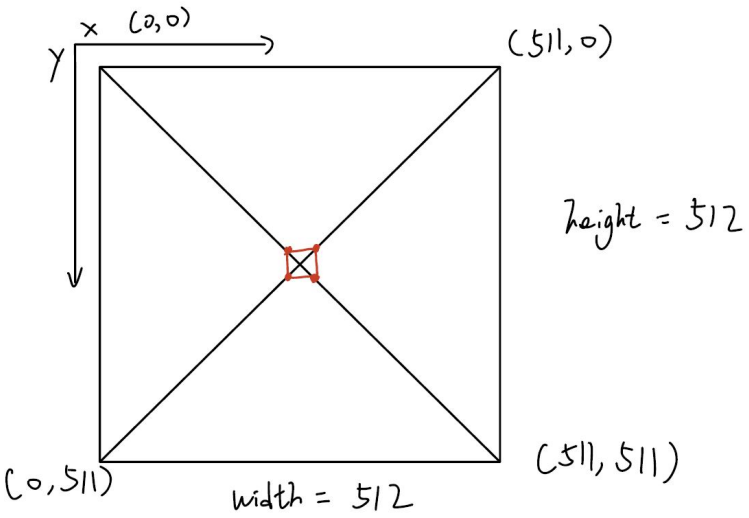


Figure 16: Coorindates

And then, I assume the center four pixels as anchor and divide the image into up, down, left and right four parts. The part of down is shown in **Figure** 17.

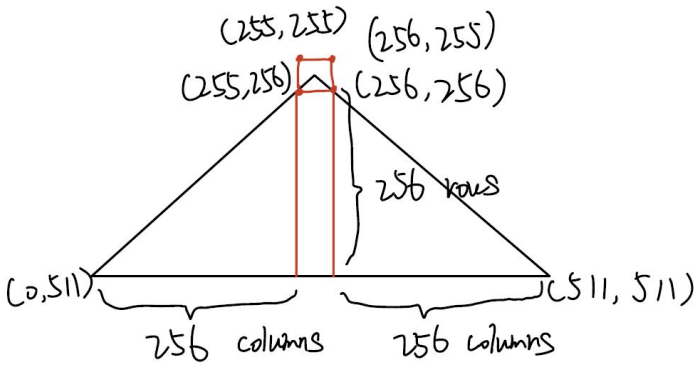
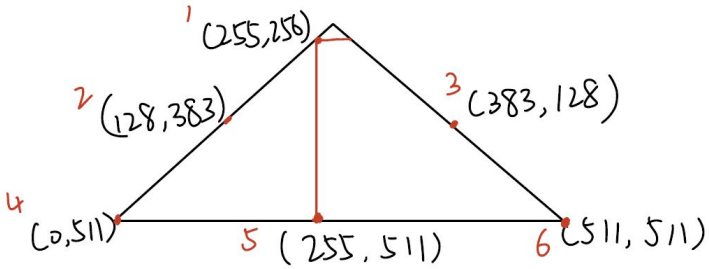
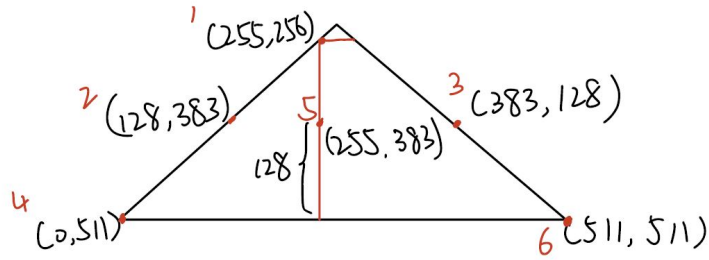


Figure 17: Quarter(down)

For each quarter, I select six points as coordinate points for the origin image and image after spatial warping, which shown in **Figure** 18.



(a) pattern1



(b) pattern1

Figure 18: Coordinates for quarters

I just use the down part as a example to illustrate the coordinate. And then, I make several clock rotation for other quarters. By this way, I can apply this coordinate for all four quarters. The clock rotation is shown in **Figure** 19.

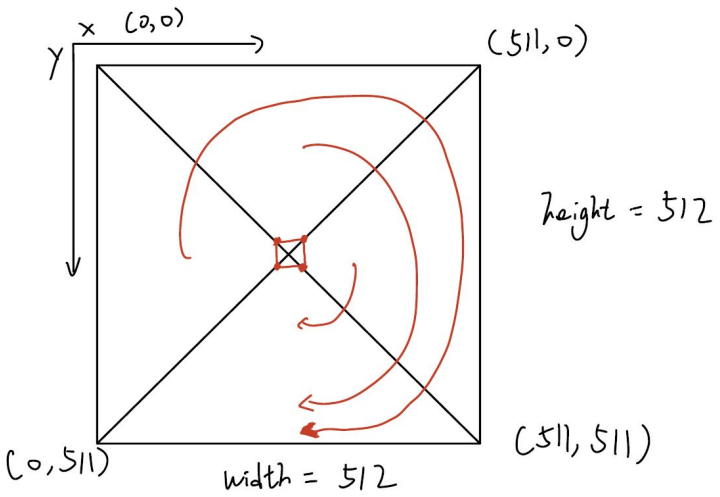


Figure 19: Clock rotation

I use (x,y) to represent the coordinate of origin image and (u,v) to represent the coordinate of image after spatial warping. The goal is to find the mapping function with the following formula shown in **Figure** 20.

$$\begin{bmatrix} a_0 & a_1 & a_2 & a_3 & a_4 & a_5 \\ b_0 & b_1 & b_2 & b_3 & b_4 & b_5 \end{bmatrix} = \begin{bmatrix} u_0 & u_1 & u_2 & u_3 & u_4 & u_5 \\ v_0 & v_1 & v_2 & v_3 & v_4 & v_5 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ x_0 & x_1 & x_2 & x_3 & x_4 & x_5 \\ y_0 & y_1 & y_2 & y_3 & y_4 & y_5 \\ x_0^2 & x_1^2 & x_2^2 & x_3^2 & x_4^2 & x_5^2 \\ x_0y_0 & x_1y_1 & x_2y_2 & x_3y_3 & x_4y_4 & x_5y_5 \\ y_0^2 & y_1^2 & y_2^2 & y_3^2 & y_4^2 & y_5^2 \end{bmatrix}^{-1}$$

Figure 20: Formula to find the forward mapping function from discussion

Finally, we get the forward mapping function in my coordinate shown as followed,

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1.002 & 2.002 & 0.002 & 0 & -0.002 \end{pmatrix}$$

By this way, I can the use this function to carry out all the pixels in image after spatial warping.

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1.002 & 2.002 & 0.002 & 0 & -0.002 \end{pmatrix} \begin{pmatrix} 1 \\ x \\ y \\ x^2 \\ xy \\ y^2 \end{pmatrix}$$

The last step is to rotate back all quarter to its correct position shown in **Figure** 21.

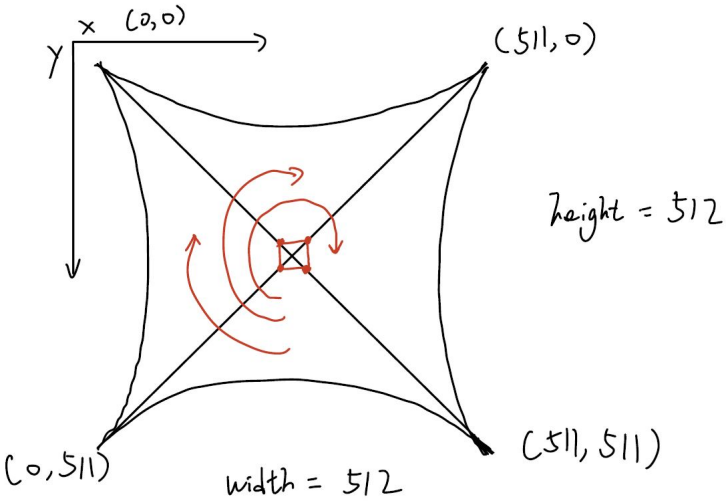


Figure 21: Redistribution by clock rotation

The above mentioned procedure is to find the forward mapping function. The method to find reverse mapping function is quite same. Just as in **Figure** 20, I replace all u with x, v with y, x with u and y with v. Then we can get the following reverse mapping function,

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 4.0236 & -3.0236 & -0.0079 & 0 & 0.0079 \end{pmatrix}$$

By this way, I can use this function to carry out all the pixels in image after spatial warping.

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 4.0236 & -3.0236 & -0.0079 & 0 & 0.0079 \end{pmatrix} \begin{pmatrix} 1 \\ u \\ v \\ u^2 \\ uv \\ v^2 \end{pmatrix}$$

2.3 Experimental Results

Figure 22: Origin image

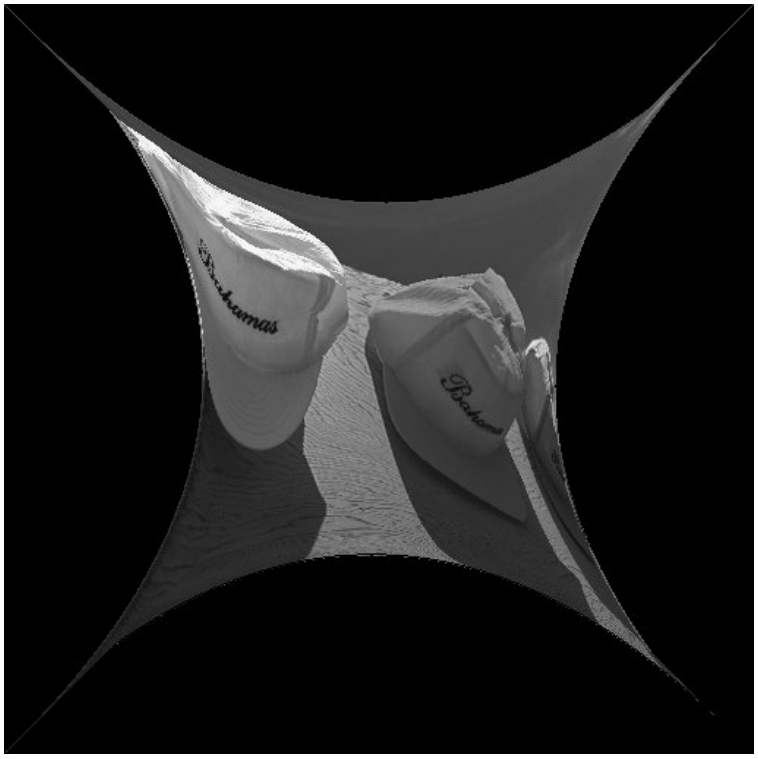


Figure 23: Forward mapping result

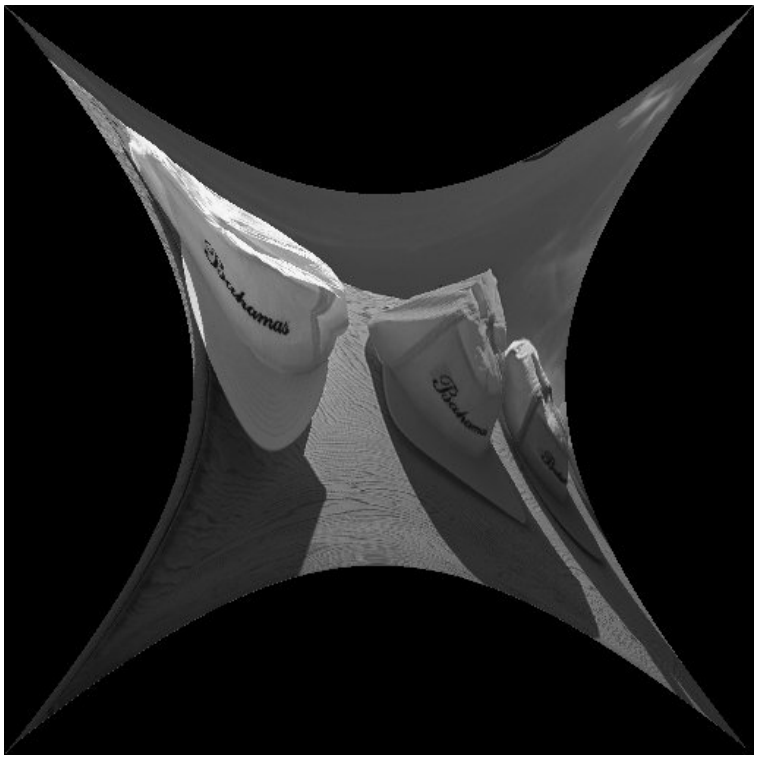


Figure 24: Reverse mapping result

2.4 Discussion

2.4.1 Image coordinates and Cartesian coordinates

To be mentioned, for most spatial transformation, the Cartesian coordinates are used. The flow chart is shown in **Figure** 25.

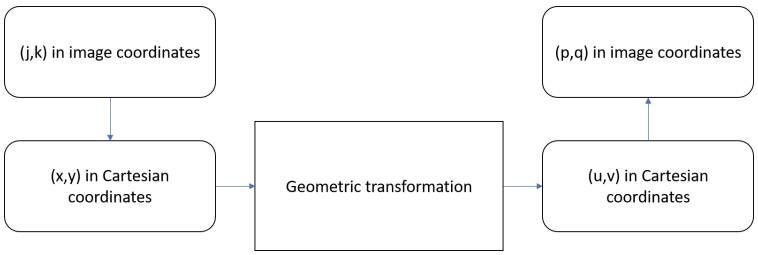


Figure 25: Transformation from iamge coordinates to Cartesian coordinates

And, the transformation formula is shown as followed,

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \begin{pmatrix} 0 & 1 & -1/2 \\ -1 & 0 & J+1/2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} j \\ k \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \begin{pmatrix} 0 & 1 & -1/2 \\ -1 & 0 & J+1/2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p \\ q \\ 1 \end{pmatrix}$$

I haven't used this transformation and I just use a more convenient coordinate in this example. What's more, for geometric transformation, the choice of coordinates doesn't influence the result, but don't forget that the last step is to transform back.

2.4.2 Reverse mapping function

Forward mapping function will create some unexpected dots in image especially in the side of image shown in **Figure** 26 because the pixels position in origin image will become fraction coordinate in warping image due to matrix multiplication.

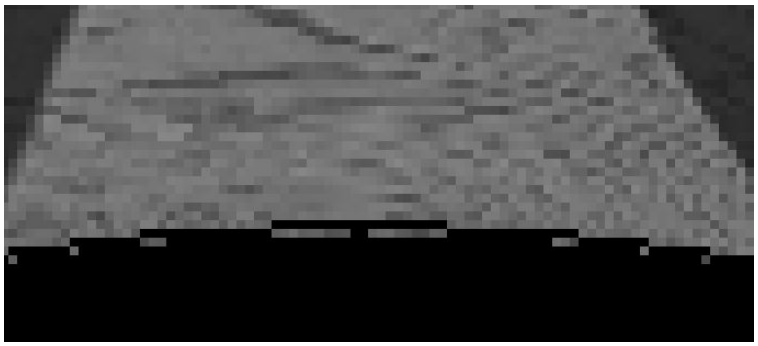


Figure 26: Unexpected dots at side

By this way, in order to avoid the black dots, I use reverse mapping in all geometric transformation to find the corresponding coordinates in origin image according to the coordinates in modification image.

2.4.3 Computational complexity

Without using MATLAB to do Matrix multiplication, I implement it by C++ on my own. So, it may be slower than MATLAB. And the time complexity will be $O(m * n)$, where m and n is the size of image.

3 Lens Distortion Correction

3.1 Abstract and Motivation

In geometric optics, distortion is a deviation from rectilinear projection; a projection in which straight lines in a scene remain straight in an image. It is a form of optical aberration.[3]
"Radial distortion" is the change in the length of vector endpoint along the length, that is, the change in the radial diameter. Radial distortion is the positional deviation of the image pixel at the center of the distortion along the radial direction, which causes the image formed in the image to be deformed.

3.2 Approach and Procedures

Firstly, I set up the following coordinate in **Figure** 27.

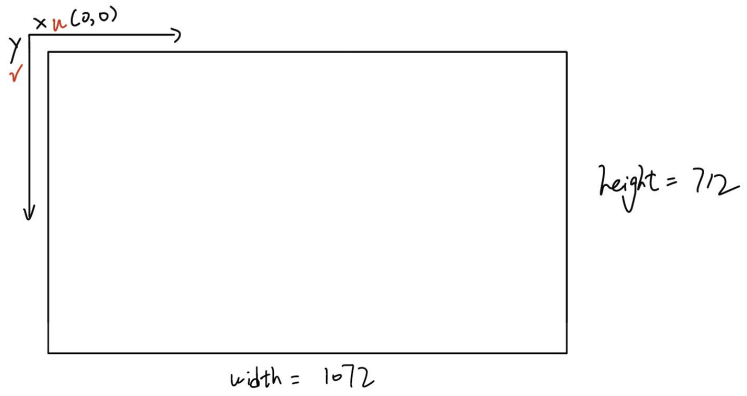


Figure 27: Coordinates

And then, I use the following formula to convert image coordinate to camera coordinates.

$$x = \frac{u - u_c}{f_x} = \frac{u - 536}{600}$$

$$y = \frac{v - v_c}{f_y} = \frac{v - 356}{600}$$

When we got the result after correction, we need to use the following formula to convert camera coordinate to image coordinate.

$$u = 600x + 536$$

$$v = 600y + 356$$

The whole flow diagram is shown in **Figure** 28. Let's come to the most important part about linear regression.

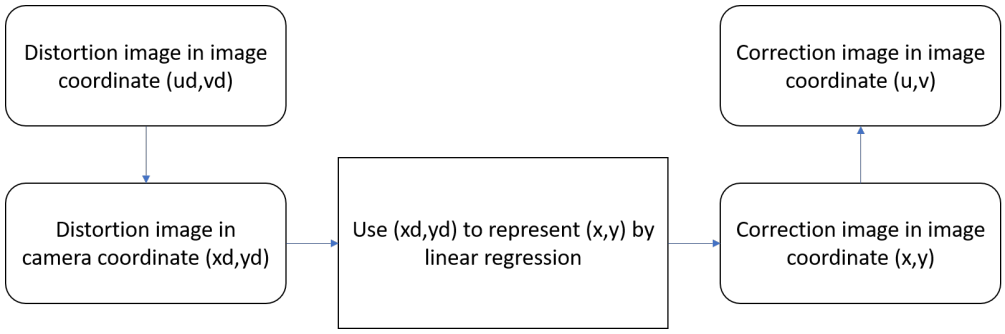


Figure 28: Transformation from Image coordinate to camera coordinate

I want to use (xd,yd) to represent (x,y) but the forward mapping function is non-linear. So, we need to use some methods to fit the reverse mapping function and linear regression is one of them. Having using the forward mapping function and scanning the whole image, I get 1072*512 pairs of (xd,yd). And I project xd and yd to the space of (x,y). Then we get two 3-D figure and apply linear regression which in MATLAB is polyfit method to fit the linear line to obtain the smallest least square errors. The linear regression result goes to,

$$x_d = 0.8601x - 0.000686$$

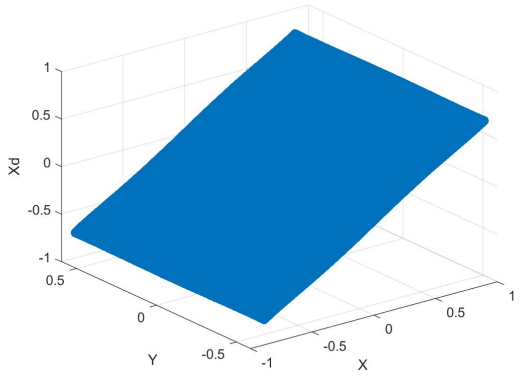
$$y_d = 0.8819y + 0.000714$$

We can easily get reverse mapping function,

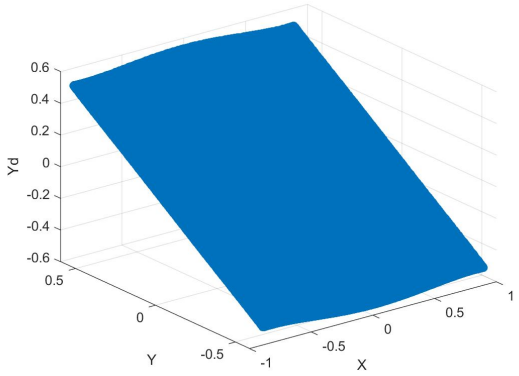
$$x = (x_d + 0.000686)/0.8601$$

$$y = (y_d - 0.000714)/0.8819$$

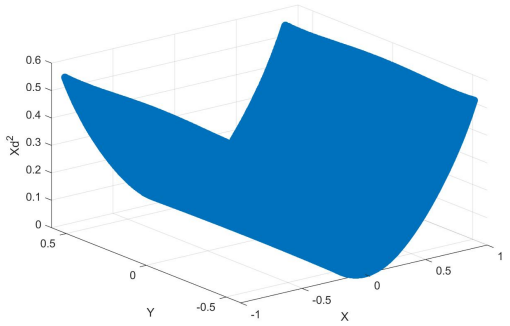
From these reverse mapping functions, there are just scaling and translation so that the distortion will not be removed. Thus, I add some high dimension term such as x^2, y^2 and xy to the function and project them to (x,y) space to find the linear regression result. These projection image is shown in **Figure** 29.



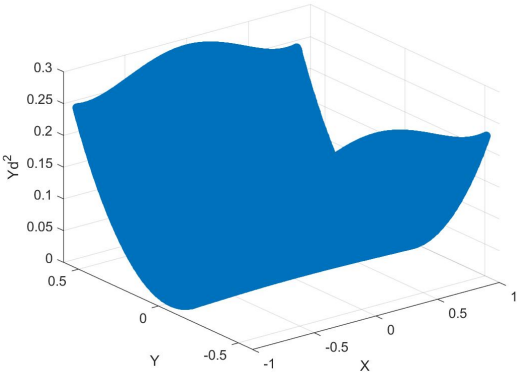
(a) xd



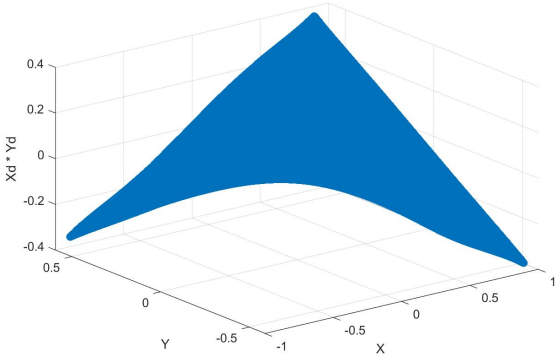
(b) yd



(c) xd^2



(d) yd^2



(e) xdyd

Figure 29: Different terms projection in (x,y) space

By this way, I get the following transformation equation,

$$x = (x_d + 0.000686)/0.8601 + 0.0088x_d^2 + 0.0079x_d y_d$$

$$y = (y_d - 0.000714)/0.8819 + 0.0191y_d^2 + 0.0033x_d y_d$$

Using these inverse function, we get the undisturbed image without aliasing artifacts.

3.3 Experimental Results

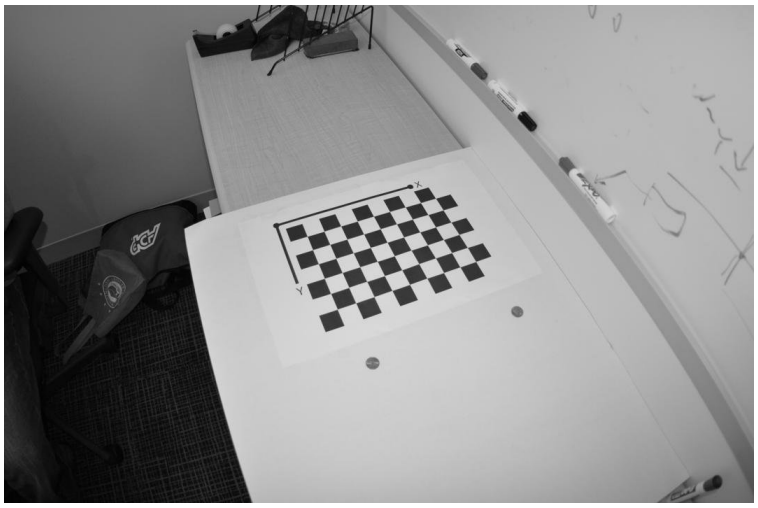


Figure 30: Origin image

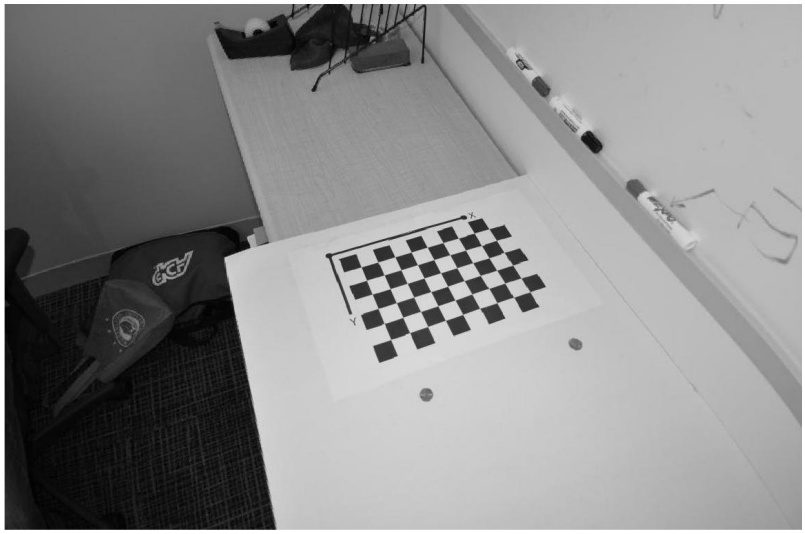


Figure 31: Undisturbed image

3.4 Discussion

3.4.1 More about linear regression

In statistics, linear regression is a linear approach to modeling the relationship between a scalar response (or dependent variable) and one or more explanatory variables (or independent variables). The case of one explanatory variable is called simple linear regression. For more than one explanatory variable, the process is called multiple linear regression. This term is distinct from multivariate linear regression, where multiple correlated dependent variables are predicted, rather than a single scalar variable. [4]
For linear regression, we want to find a linear fitting line which has the least sum of Euclidean Distance to all other sample points.

3.4.2 Forward mapping

Actually, using non-linear forward mapping function, we can also get the result shown in **Figure** 32. But because the pixels in undisturbed image and the pixels in disturbed image are not one-to-one mapping. It tend to make some pixels in undisturbed image set for several time, which causes aliasing artifacts.

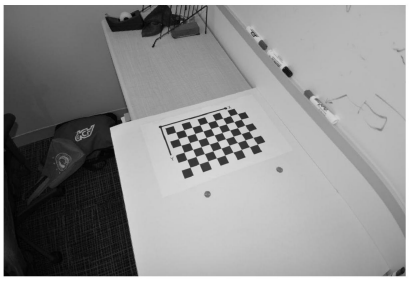


Figure 32: Undisturbed image by forward mapping

3.4.3 Improvement

In this example, we can see the forward mapping function is definitely non-linear so that using linear regression to approximate these non-linear function may be not accurate because linear regression can only create linear component which is not enough for approximation. But the advantage of linear regression can make us to see the reverse mapping function in a explicit formula. However, maybe other methods like Newton's method only give the implicit formula of reverse mapping function is better.

4 Basic Morphological Process Implementation

4.1 Abstract and Motivation

Morphological image processing is a collection of techniques that are used for analyzing geometrical structures in an image.[5] In this part, we use three kind of basic morphological operation. They are:

- Shrinking
- Thinning
- Skeletonizing

The basic of all three morphological process is hit-or-miss transformation. It can be considered as if the binary-valued pattern of the mask matches the state of the pixels under the mask(hit), an output pixel is spatial correspondence to the center pixel of mask is set to some desired binary state. And if it doesn't match the pattern(miss), there is no changes in this pixel.

4.2 Approach and Procedures

It is not possible to perform the operation using a single-stage 3*3 pixel hit-or-miss transform because the 3*3 window does not provide enough information to prevent total erasure and to ensure connectivity. Although a 5*5 hit-or-miss transform could provide sufficient information to perform proper operation, such an approach would result in excessive computational complexity. Instead, we use two stage 3*3 filter, which is shown in **Figure** 33.

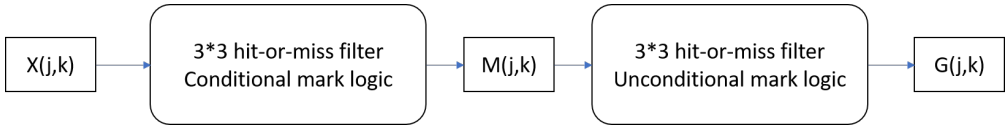
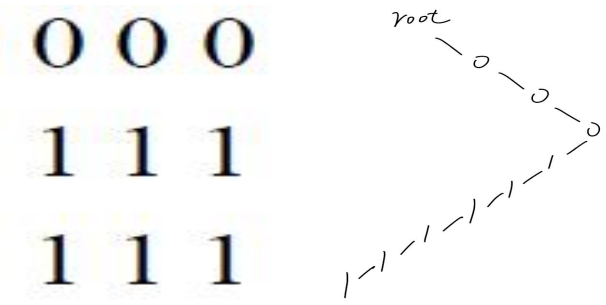


Figure 33: Two stage

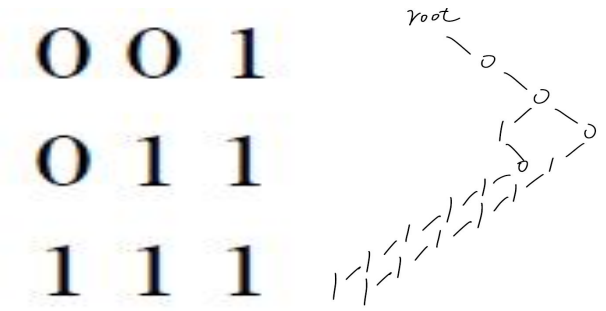
There are two stage 3*3 hit-or-miss filters. First stage is coordinates for subtraction and the second stage is to make confirmation of subtraction operation. Inside my implementation, I use the data structure of binary tree to represent each condition and un-condition pattern. For example, the pattern matrix and its tree is shown in **Figure** 34.



(a) pattern1 (b) pattern1

Figure 34: Binary tree construction

Having added a new pattern to this tree, we get the tree shown in **Figure** 35.



(a) pattern1 (b) pattern1

Figure 35: Binary tree addition

Since there are 58 patterns in the condition pattern table of shrinking,after construction there are 58 leaf nodes in the condition pattern tree of shrinking construct by my algorithm.

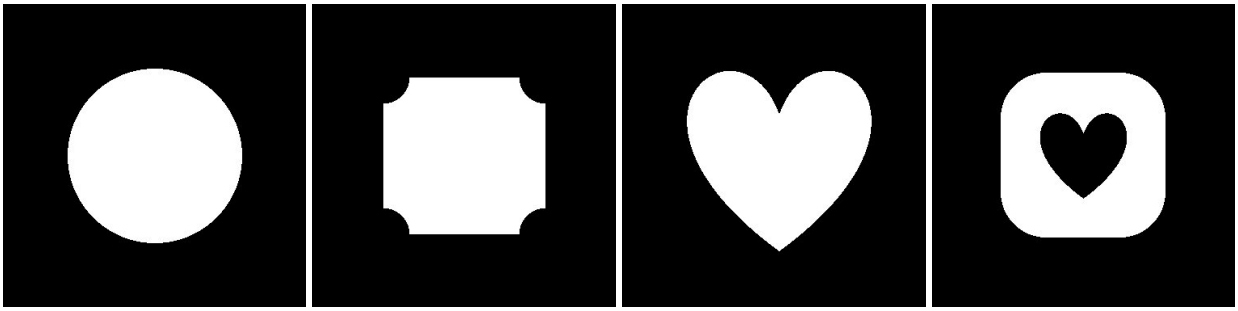
For un-condition patterns, they are a little complicated because there are some terms like "A B C D". For term "ABC", I make a conversion that "ABC" can be interpreted as "MDD" , "01D" and "001". And then, when I construct the binary tree, if we meet the term "D", we make recursion for both right and left child nodes.

By this way, all patterns are transformed into the binary tree. Then we use the 3*3 check window to go through the whole image, and make decision in each node of the pattern tree. Both constructing and comparison use recursion.

After that, according to the following formula, we can get the result after operation.

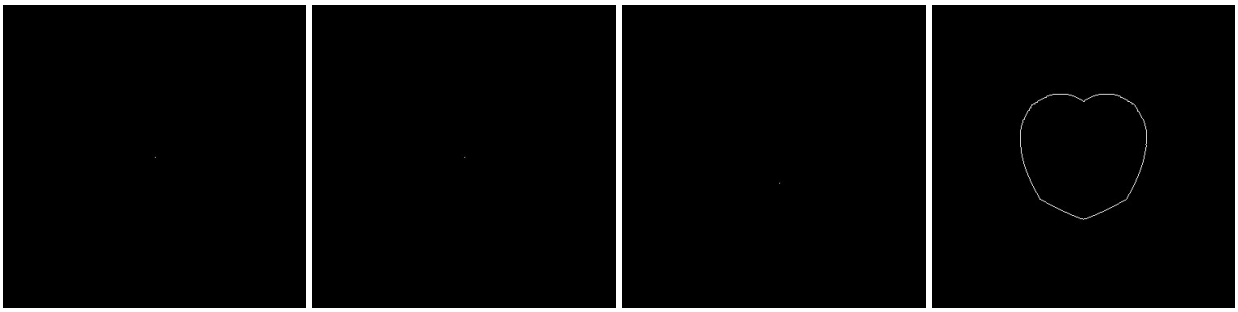
$$G(j,k) = X \cap (\bar{M} \cup P(M, M_0, M_1, M_2, M_3, M_4, M_5, M_6, M_7))$$

4.3 Experimental Results



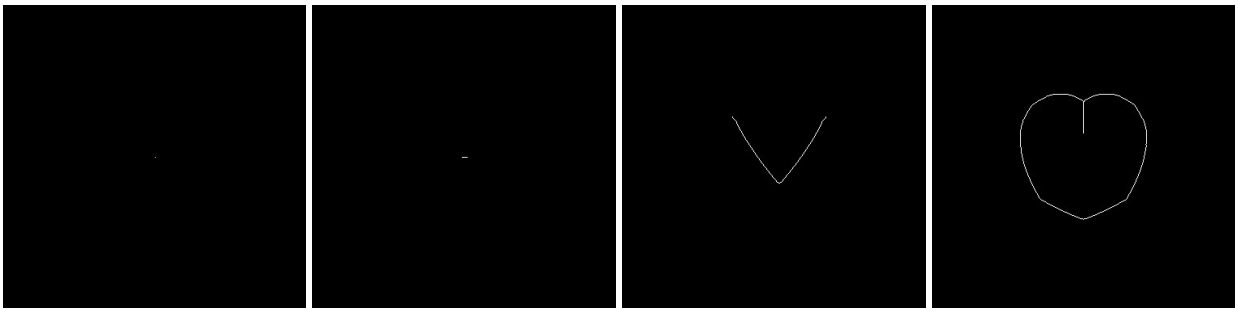
(a) pattern1 (b) pattern2 (c) pattern3 (d) pattern4

Figure 36: Origin images



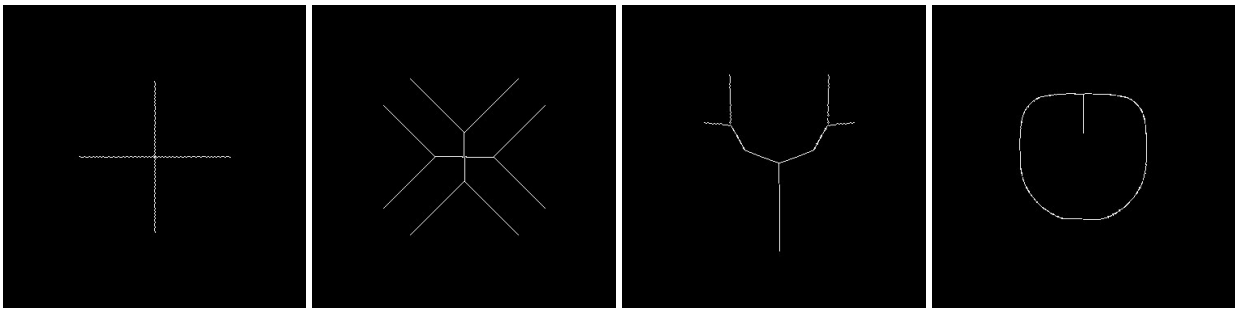
(a) pattern1 (b) pattern2 (c) pattern3 (d) pattern4

Figure 37: Results after shrinking



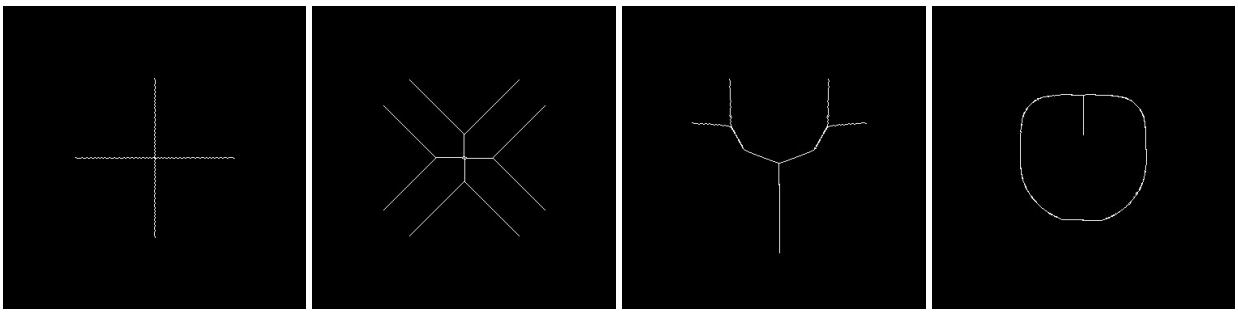
(a) pattern1 (b) pattern2 (c) pattern3 (d) pattern4

Figure 38: Results after thinning



(a) pattern1 (b) pattern2 (c) pattern3 (d) pattern4

Figure 39: Results after skeleton



(a) pattern1 (b) pattern2 (c) pattern3 (d) pattern4

Figure 40: Results after shrinking and bridging

4.4 Discussion

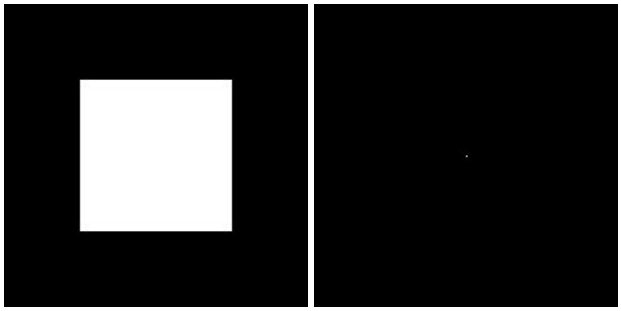
4.4.1 Shrinking

Erase black pixels such that an object without holes erodes to a single pixel at or near its center of mass, and an object with holes erodes to a connected ring lying midway between each hole and its nearest outer boundary. By this way, the shrinking result of a 3*3 pixel object will be a single pixel at its center. And for 2*2 pixel object will be a single pixel at its lower right corner after shrinking.
The first three pattern have no holes in their body. Pattern1 is a central symmetric circle so that the result is just its center. Pattern2 look like "almost" central symmetric shapes so that the shrinking result is just its center which is also the center of image. And for pattern3, the mass center is a little bit lower. And for the last pattern, the inside heart is like a big hole so that it will be shrunk to a connected ring lying midway between the side of the heart and its nearest outer boundary.

4.4.2 Thinning

Erase black pixels such that an object without holes erodes to a minimally connected stroke located equidistant from its nearest outer boundaries, and an object with holes erodes to a minimally connected ring midway

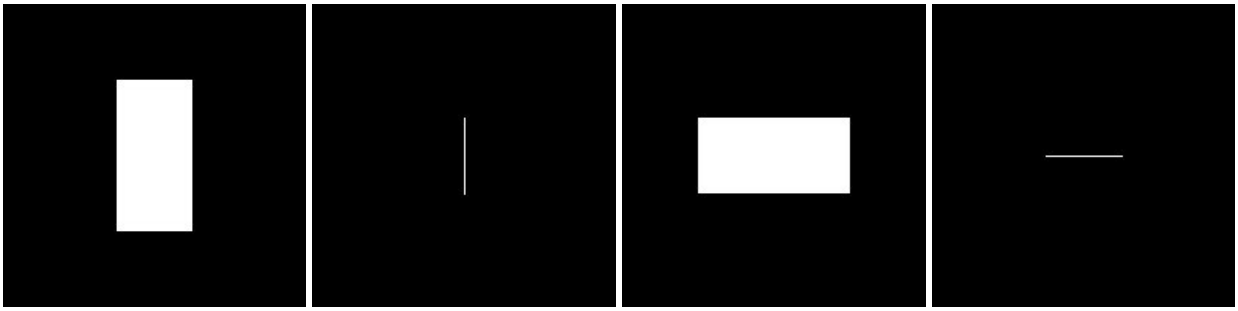
between each hole and its nearest outer boundary. For example, the thinning result of rectangle and square is shown in **Figure** 41. We can easily find that if the shape is a square without holes, the thinning result is also one single pixel.



(a) pattern (b) pattern after thinning

Figure 41: Thinning example (square)

The first three pattern have no holes in their body. The result of pattern1 is a single pixel because its center symmetry. But for pattern2, the result is a horizontal line instead of one single point so that the pattern have longer width than height, which is interpreted in **Figure** 42.



(a) pattern (b) pattern after thinning (c) pattern (d) pattern after thinning

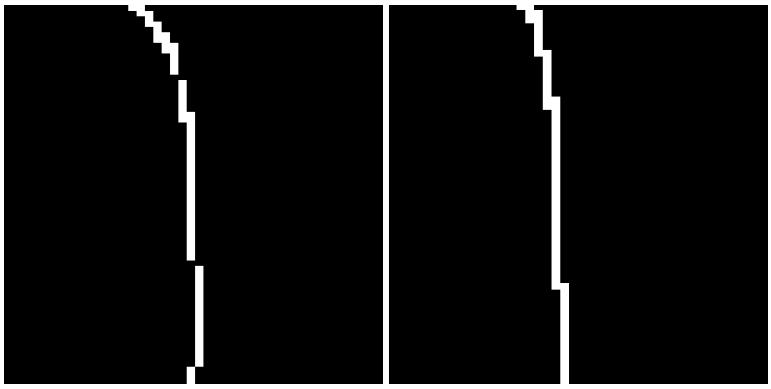
Figure 42: Thinning example (rectangle)

For pattern3, the result of thinning is just the connected stroke located equidistant from it nearest outer boundaries. And for pattern4, it is a little bit complicated. We should consider the pattern from up and down direction separately. From the down direction, the tip of heart is like a holes so that it erodes to a minimally connected ring midway between each hole and its nearest outer boundary. But from up direction, the groove of heart construct two sides which make a vertical connected stroke located equidistant from the two sides.

4.4.3 Skeletonzing

Skeletonzing is to erase the redundant pixels and only maintain the medial axis skeleton of shape. The medial axis skeleton consists of the set of points that are equally distant from two closest points of an object boundary.

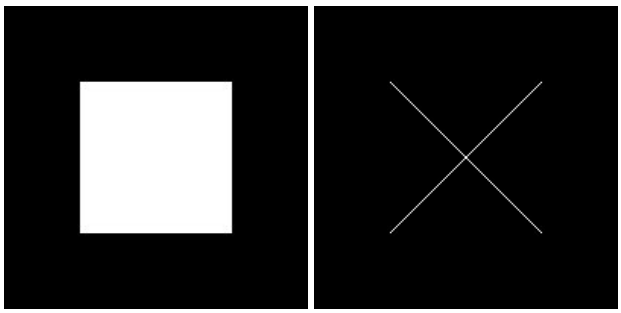
From the medial axis skeleton of an object, it is possible to reconstruct the object boundary. Also, at the conclusion of the last iteration, it is necessary to perform a single iteration of bridging to restore connectivity. The detail about the bridging is shown in **Figure** 43.



(a) pattern1 (b) pattern1

Figure 43: Results after shrinking and bridging

The skeleton of square is two diagonals shown in **Figure** 44. For pattern1, the skeleton of circle is just two diameters, which are quite simple because the circle is a center-symmetric shape. And for pattern2, the cross in the center is the base of shape and other 8 lines represent the detail of origin shape. For pattern3, the lines upward represent the kind of half circle of the heart. For pattern4, there is a heart-kind holes inside. The result is more like the result after thinning. From down direction, the tip of heart is like a holes so that it erodes to a minimally connected ring midway between each hole and its nearest outer boundary. But from up direction, the groove of heart construct two sides which make a vertical connected stroke located equidistant from the two sides.



(a) pattern (b) pattern after thinning

Figure 44: Skeleton example (square)

4.4.4 Comparison

The shrinking for the shape without holes inside let us get the center of mass. The thinning can help us the extension direction of shape. Compared with thinning, there are more details in the result after skeletonizing

especially for the special shape in the end of shape. And for the shape with holes, the results from these three operation are quite similar. The objects with holes will be eroded to a connected ring lying midway between each hole and its nearest outer boundary.

4.4.5 Computational complexity

If we just use normal kernel to compare with the checking window, the time complexity will be $O(m*n*k^2*s)$ in comparison with condition pattern, where m and n is the size of image, k is the size of pattern and s is number of patterns. Then we need to compared the result with un-condition pattern. There are even more patterns so that it will really take more time. In order to improve this situation, we use tree for comparison. The time complexity of constructing pattern tree is constant. And then with the tree, time complexity will be $O(m*n*k^2*\log_2 s)$.

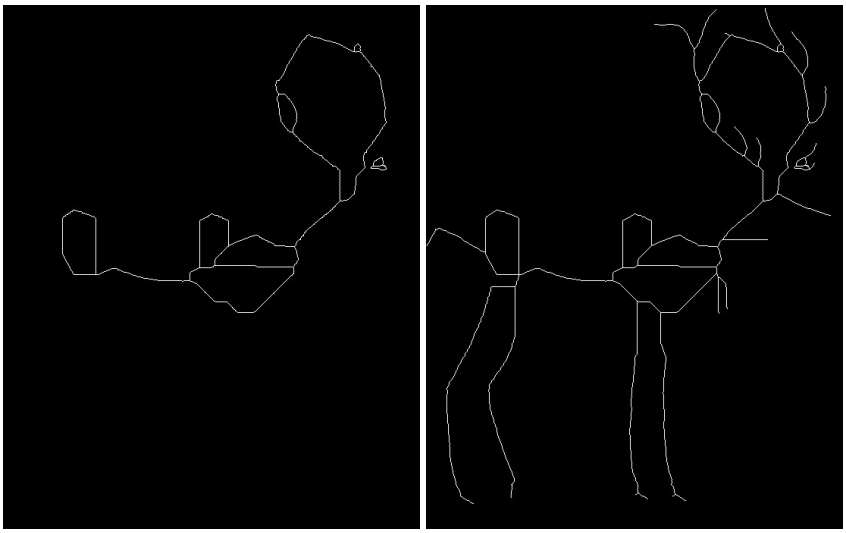
5 Defect Detection and Correction

5.1 Abstract and Motivation

Sometimes, there will be some defects like several black pixels. They are kind of "island" in the body of figure and some of them are difficult to detect. By this way, we need find some ways to detect them and correct them.

5.2 Approach and Procedures

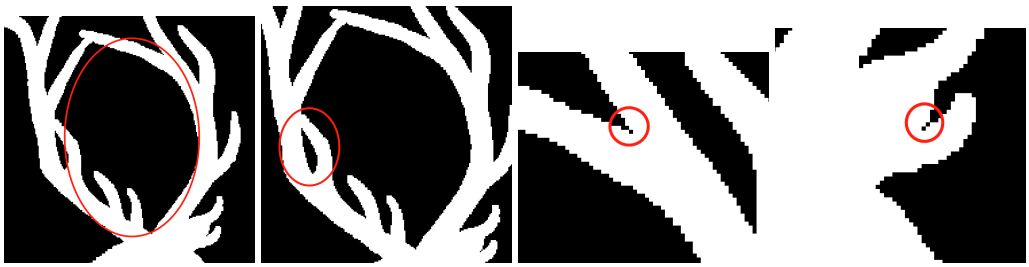
First, we apply the shrinking and thinning introduced above to this figure, the result is shown in **Figure** 45.



(a) Shrinking result with defect (b) Thinning result with defect

Figure 45: Shrinking and thinning result

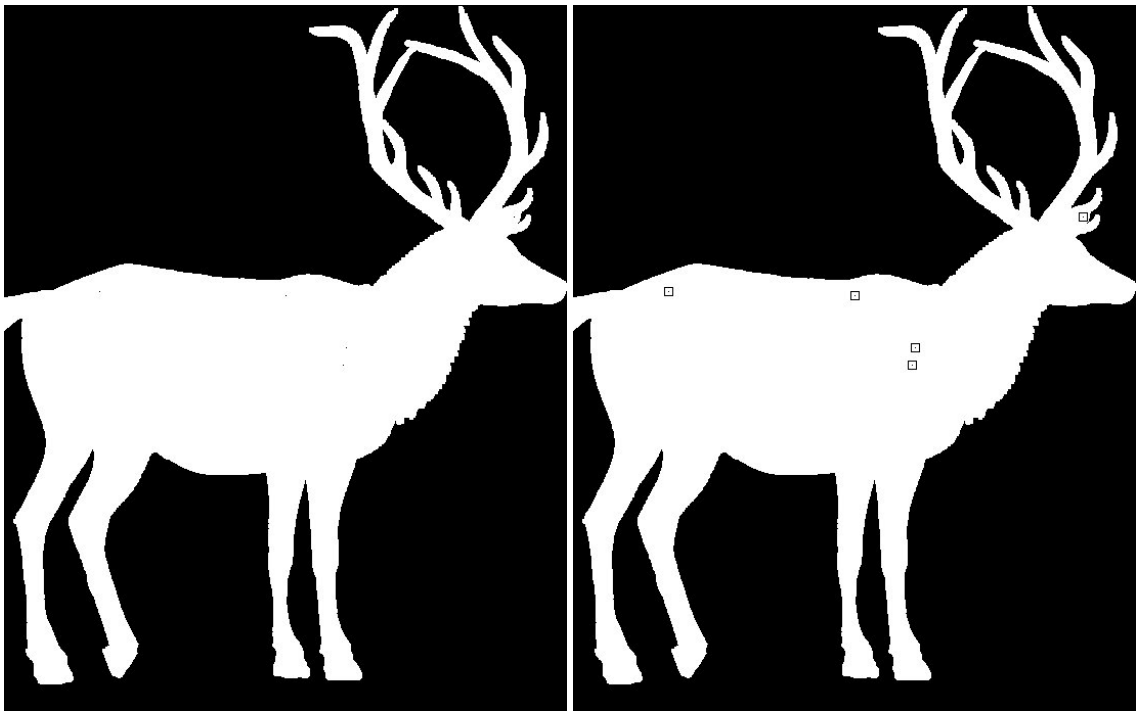
Although there are 9 "circle" in the result of shrinking and thinning, actually, there are only five defects needed to be correct. Right there, some explanation must be made. From the **Figure** 46, we can see that although (a) and (b) are definitely black holes, they are just the special pattern of the deer. Also, like in (c) and (d), some pixels are eight connected neighbors to the connected domain. So, we must ignore them when I want to correct other real defects.



(a) Special pattern1 (b) Special pattern2 (c) 8-connectivity neighbor1 (d) 8-connectivity neighbor2

Figure 46: "fake" defects

Having found that, I must design some extra algorithm to filter these "fake" defect. By this way, I use the area of connected domain to filter the first type of "fake" defect. If the area of the connected domain is greater than a threshold(given 5 pixels in this report), it will regarded as special pattern instead of defect. And for the second type of "fake" defect, I use detection of eight connected neighbor to filter them. With the previous condition pattern, eight-connectivity neighbor shown in **Figure** 46.
After that, I got the real defect which is shown in **Figure** 47.



(a) Image with defects (b) Image with defects labeling

Figure 47: Five real defects

Having located defects in the shrinking circles, I just reverse the gray-scale from 0 to 255 and 255 to 0. By this way, in this circles, the defect becomes a white shape without holes inside it. Shrinking can erode it to be one single pixel. The flow chart about these procedures is shown in **Figure** 48.

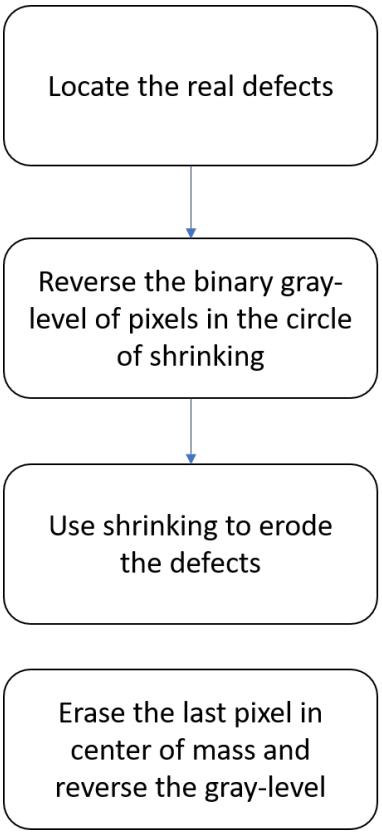


Figure 48: Flow diagram for defects erasing

And then, the following 3*3 filter can be used to erode this single white pixel.

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

5.3 Experimental Results

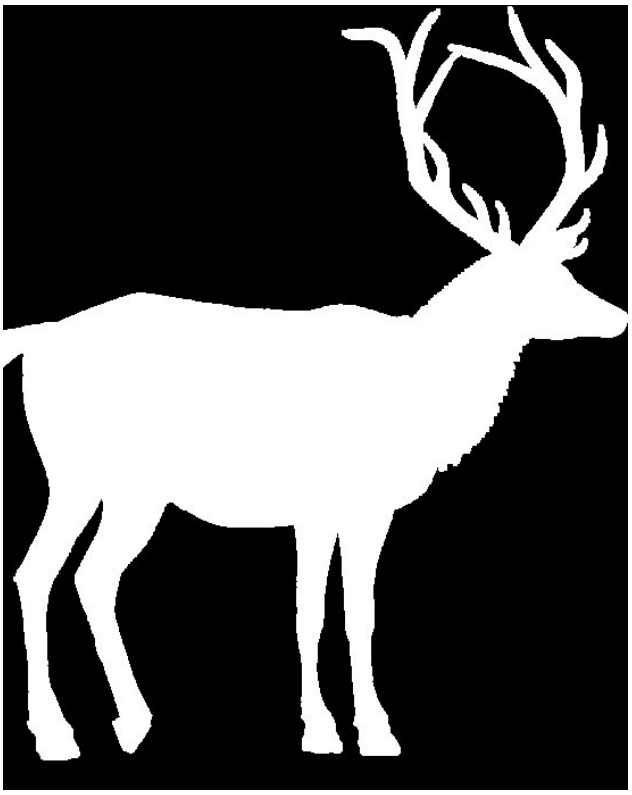
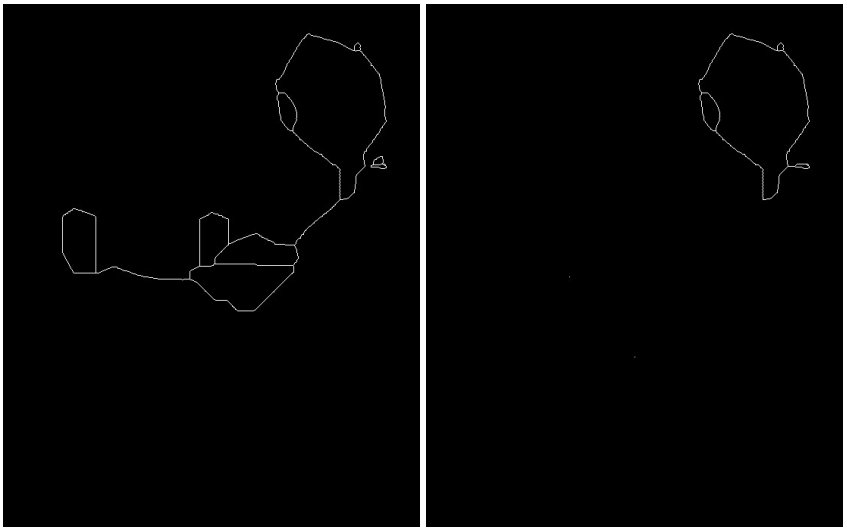
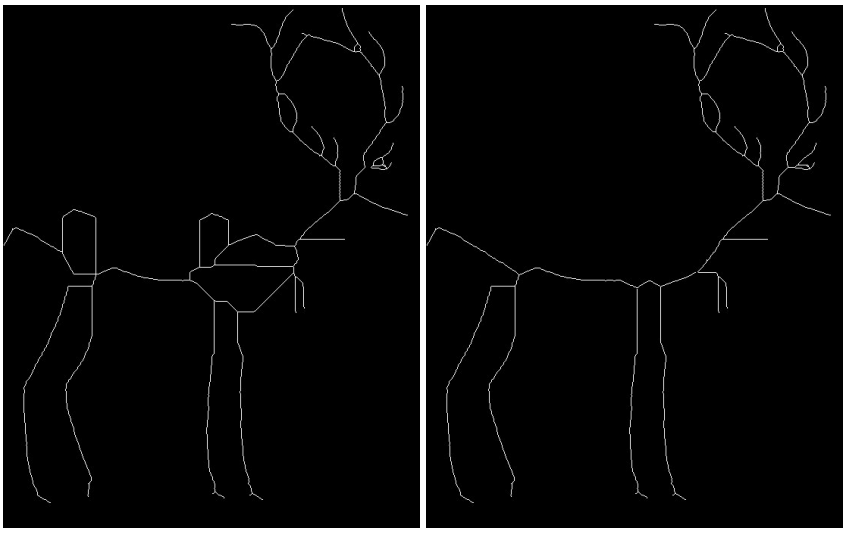


Figure 49: Result after defect correction



(a) Shrinking result with defect (b) Shrinking result without defect

Figure 50: Shrinking result



(a) Thinning result with defect (b) Thinning result without defect

Figure 51: Thinning result

5.4 Discussion

5.4.1 Automatically program design

The whole flow diagram is shown in **Figure** 52.

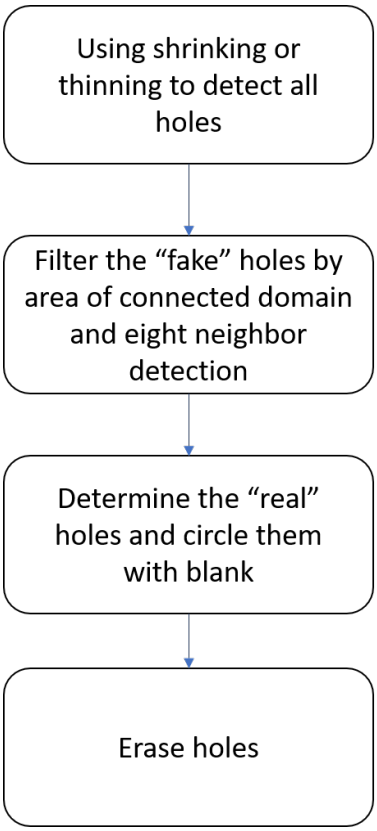


Figure 52: Flow diagram

The program of defect detection and correction is fully-automatically. The only thing you should do is to input the deer.raw. These defects are located by my program as well as the distinguishing between real and fake defect. After that, the erasing is finished using the result after shrinking.

5.4.2 Robustness

Having observed the figure, I found the all five defects are just one single black dot. But I don’t use any shortcut to detect the single black dot using the following filter.

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

I just treat them as a connected domain whose area is smaller than the special pattern in deer so that the method is very robust.

5.4.3 Computational complexity

First, I haven’t used any built-in method in OpenCV like erosion and so on. The methods using for defects detection and correction is shrinking which I implemented in previous question. And, the method for

connected domain area detection is Breadth first search, which I will introduce in details to calculate the area of rices in next problem. Since I use binary tree in shrinking so that time complexity is $O(m * n * k^2 * \log_2 s)$, where m and n is the size of image, k is the size of pattern and s is number of patterns.

6 Object Analysis

6.1 Approach and Procedures

To solve the problem as described, there are some steps to follow shown in **Figure** 53.

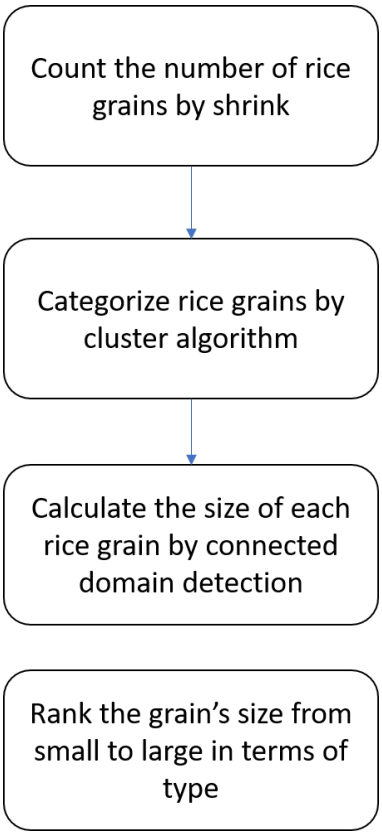


Figure 53: Flow diagram

And, before counting the number of rice grains by shrink, there are more discussion about how to get the clean image when we convert the RGB image to gray-level image and then apply thresholding to obtain the result. First, I use the following formula to get the gray-level image.

$$Graylevel = 0.299 * RED + 0.587 * GREEN + 0.114 * BLUE$$

And then, I just apply single thresholding method with threshold 77 to get the result shown in **Figure** 54 because 77 is the gray-level of background.

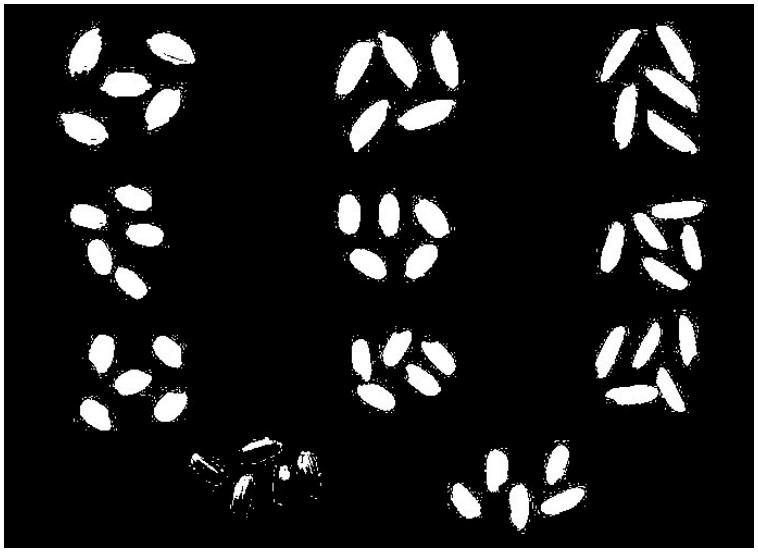


Figure 54: Binary image using single threshold 77

There are two main problems. The first one is that beside the rice grain and inside the body of rice grain, there are some unexpected dot. In order to solve this problem, I use a 3*3 median filter to take of these impulse noise and the result is shown in **Figure** 55.

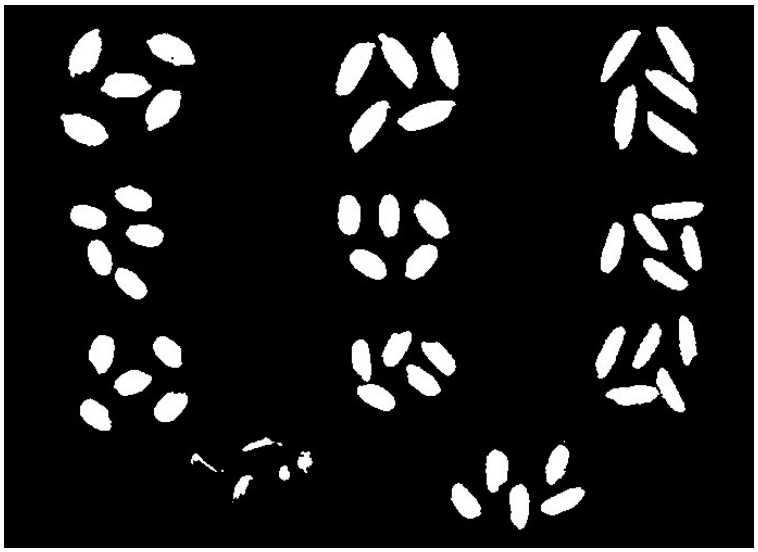


Figure 55: Binary image using single threshold 77 and median filter

It really looks fine for most of grains, but the grain rices which are darker than background are taken off with the threshold 77. So, we need another way to figure out the these dark rices.

In the discussion part, I will introduce more about other methods using to solve the problem. And, at this section, I want to pay more attention to the method chose to get the best result. The method I choose is not one mixed threshold anymore. Instead, I give a offset to origin background gray level because the edge between background and grain rice contain some pixels whose gray level is in the range of

$[background - offset, background + offset]$. The detail about the detour of grains is shown in **Figure** 56.

Figure 56: Details for grains contour

Having done some trials with different offset chosen, I determine 14 as the best offset value. By this way, I have the final flow chart shown in figreffig:flow5.

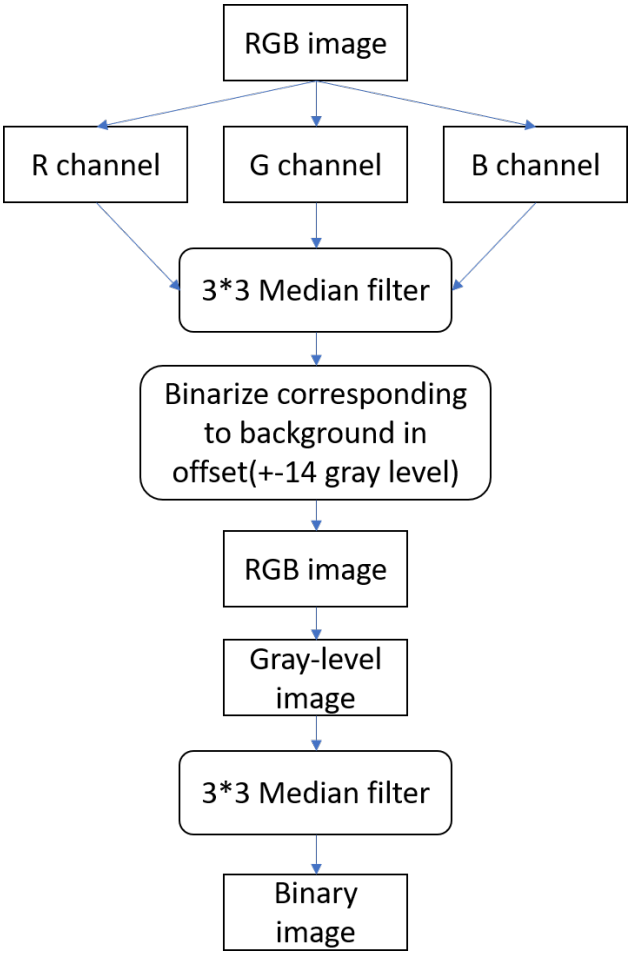


Figure 57: Flow diagram

By this way, the result is shown in **Figure** 58.

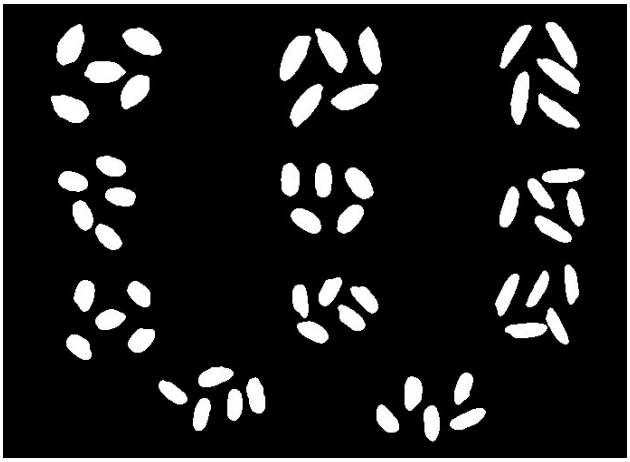
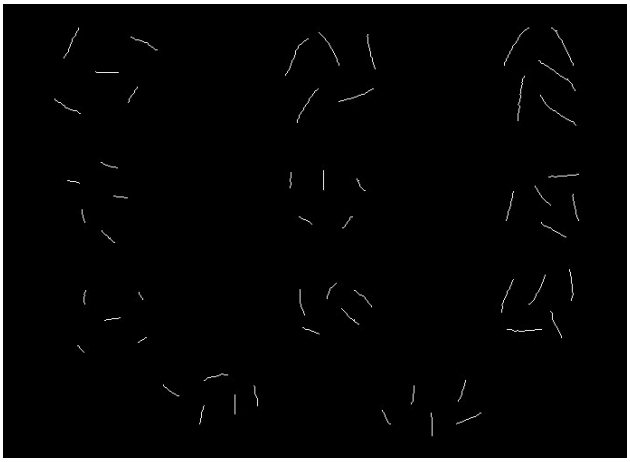
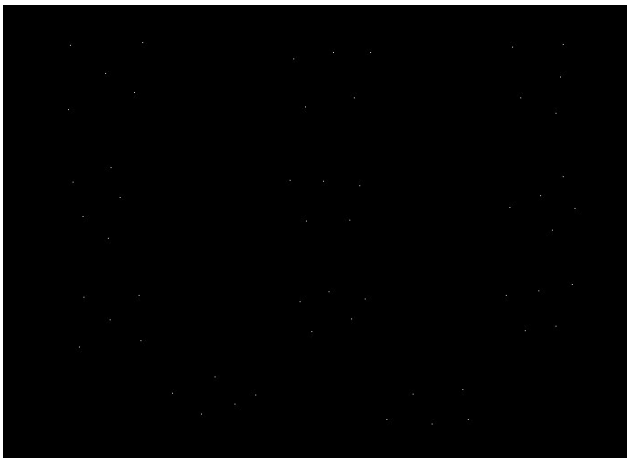


Figure 58: Binary image using background offset threshold

Having removed unwanted dots and holes caused by binarization and got the best binary result, we can use shrink technique to count the number of rice grains. The shrink and thinning result is shown in **Figure** 59.



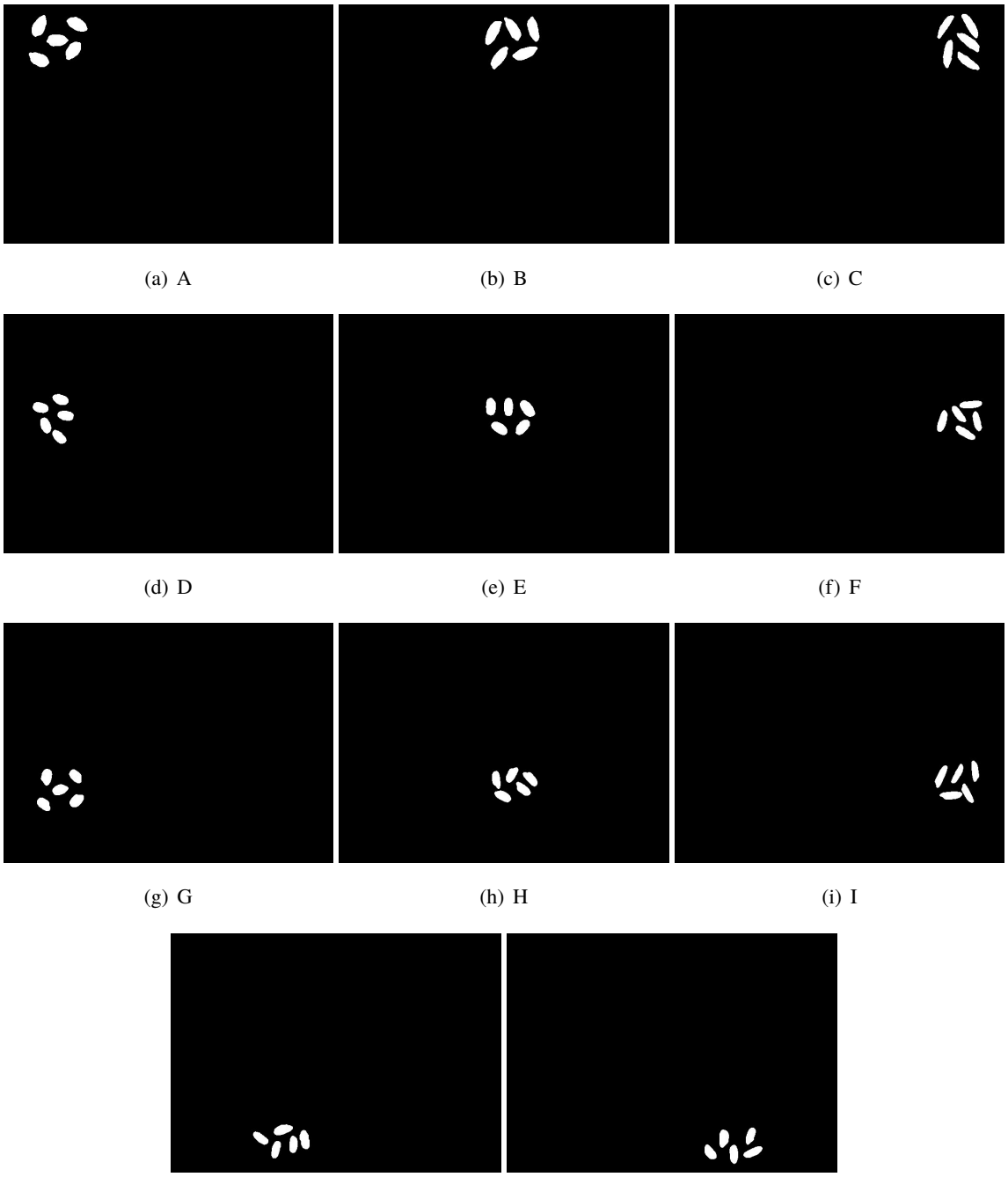
(a) Thinning result



(b) Shrinking result

Figure 59: Thinning and shrinking result

Having got the position of the single white dot after shrinking, we use the cluster algorithm to category these grains into 11 classes according to their distance from each other. At this time, the basic assumption of cluster algorithm is that the grain rices gather together are the same type so that there are 11 classes shown in **Figure** 60.



(a) A (b) B (c) C

(d) D (e) E (f) F

(g) G (h) H (i) I

(j) J (k) K

Figure 60: 11 clusters of grains

After getting these 11 classes separately, we label them from A to K as class name for further presentation. The labeling rule is just from left to right and from up to down, which is shown in **Figure** 61.

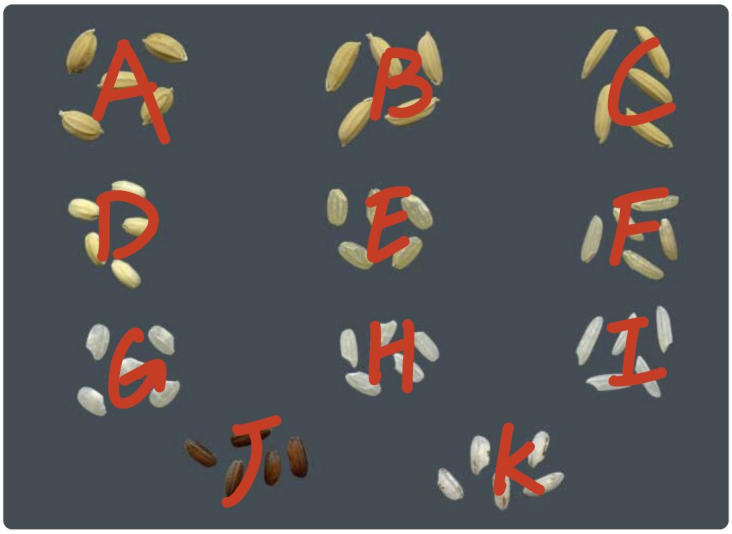


Figure 61: Labeling rule

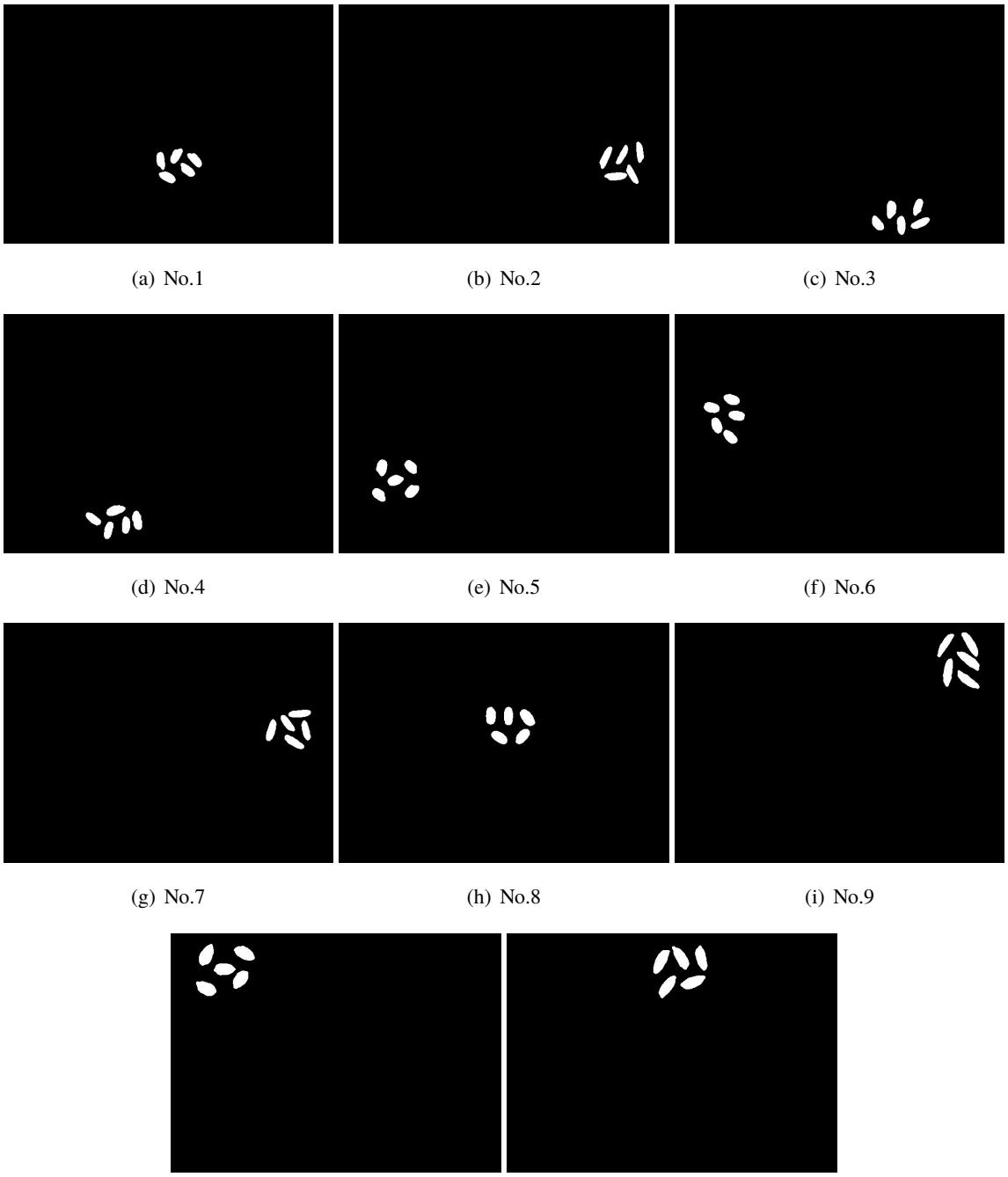
Eventually, the last step is to calculate the average size of each classes and rank them from small to large. This time, the single point after shrinking can be considered as the mass center of each grains. I just use this point as start point and label this point. After that, Breadth first search algorithm is used to diffuse this label from this point to its neighbor until coming to edge of grain or all its neighbor labeled. At this process, the labeling pixels are counted and the number of labeling pixels is regarded as the size of this grain. The size of all other grains are measured by this way and the average is calculated by taking mean of each classes. Finally, we rank them and get the following results.

6.2 Experimental Results

```
The total number of rice grains is 55
```

Figure 62: Print result in command window

The rank from small to large in terms of type are shown in **Figure** 63 and **Figure** 64. All of them is output automatically by my program.



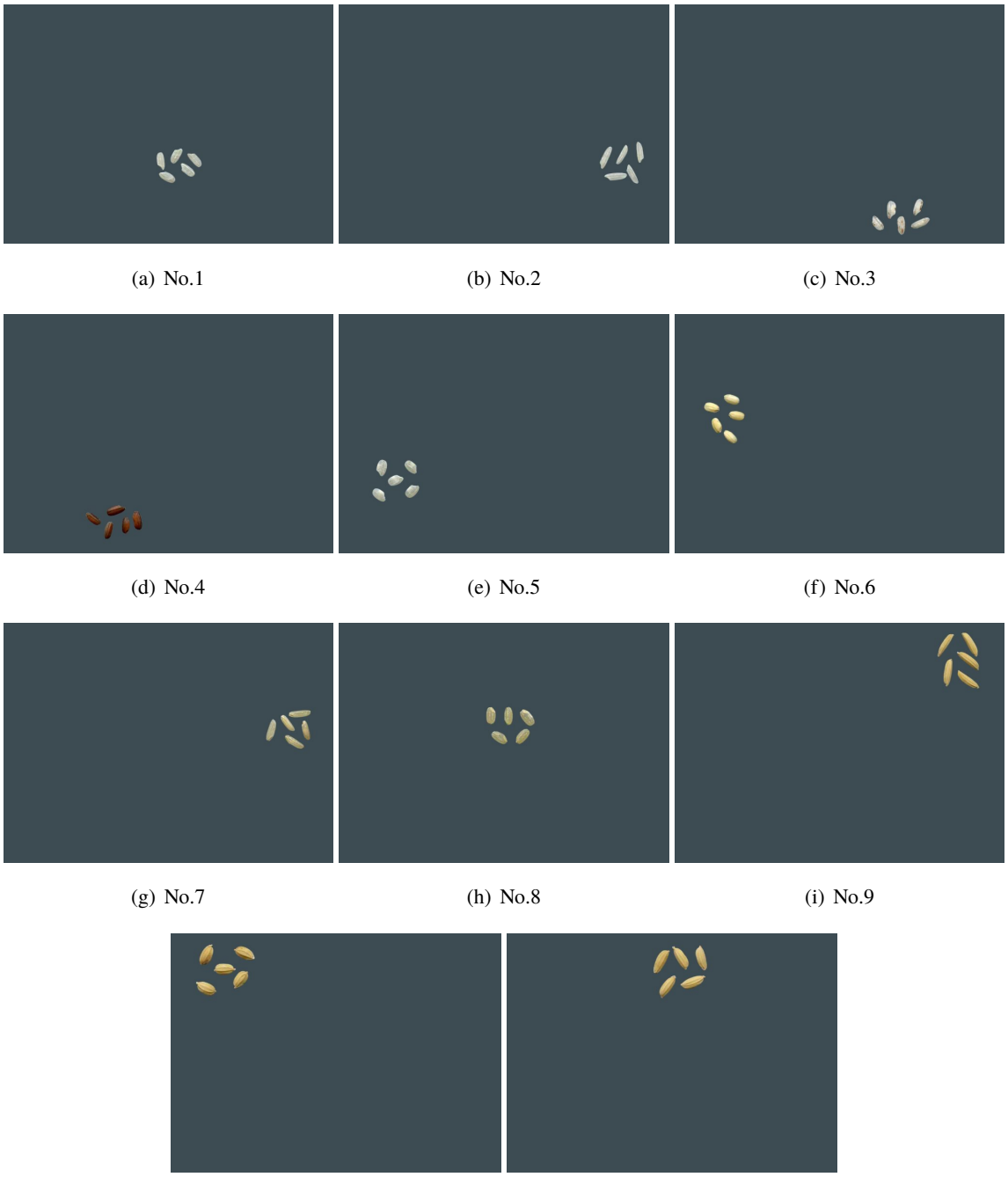
(a) No.1 (b) No.2 (c) No.3

(d) No.4 (e) No.5 (f) No.6

(g) No.7 (h) No.8 (i) No.9

(j) No.10 (k) No.11

Figure 63: Rank from small to large in terms of type(Binary image)



(a) No.1 (b) No.2 (c) No.3

(d) No.4 (e) No.5 (f) No.6

(g) No.7 (h) No.8 (i) No.9

(j) No.10 (k) No.11

Figure 64: Rank from small to large in terms of type(RGB image)

The average size of each cluster os grain is shown in Table 2.

Table 2: Average size of each cluster of grains

Rank	Label	Size(number of pixels)
1	H	538
2	I	555
3	K	558
4	J	565
5	G	568
6	D	569
7	F	589
8	E	647
9	C	813
10	A	904
11	B	941

6.3 Discussion

6.3.1 Automatically program design

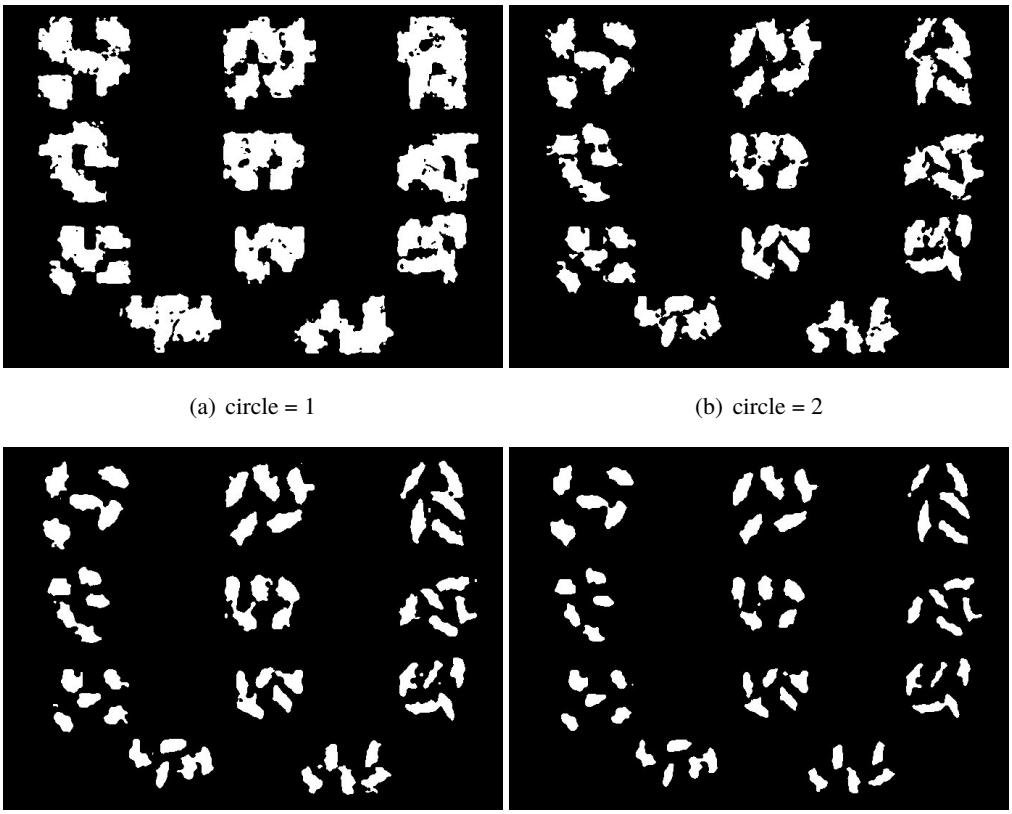
The whole procedures are done fully-automatically. The only thing you should do is to input the rice.raw shown in **Figure** 65.

Figure 65: Origin image

The total number of rice grains is obtained by adding up several single white points by searching them in whole image and the number is printed out in command window. After that, the grains are categorized as 11 classes according to distance cluster algorithm. Also, the average size of each classes is calculated by connected domain detection with BFS. Finally, 11 images with separable class of grains are output in the order of size from small to large.

6.3.2 Other method

Before I find the best solution for getting clean grains rice, I use another way to think about this problem. The method is to think about the 4-connected neighbor and 8-connected neighbor of background pixel and I named it as 1 circle. Also, I enlarge the range of searching window into two circles, three circles and 4 circles. The results are shown in **Figure** 66.



(a) circle = 1 (b) circle = 2

(c) circle = 3 (d) circle = 4

Figure 66: Binarize with different size

However, I just find this method destroy the shape of rices so that it was abandoned.

6.3.3 Computational complexity

First, I haven't use any built-in method in OpenCV like erosion and so on. The methods using for counting number of rice grains is shrinking which I implemented in previous question. Since I use binary tree in shrinking so that time complexity is $O(m * n * k^2 * \log_2 s)$, where m and n is the size of image, k is the size of pattern and s is number of patterns. And, the method for counting the size of each grain is connected domain area detection by Breadth first search and queue. The time complexity is $O(n)$. The method for categorizing the rice grains is cluster by position. The time complexity is also $O(n)$.

Reference

[1] Gielis J. A generic geometric transformation that unifies a wide range of natural and abstract shapes.[J]. American Journal of Botany, 2003, 90(3):333-338.

[2] Alajlan N, El Rube I, Kamel M S, et al. Shape retrieval using triangle-area representation and dynamic space warping[J]. Pattern Recognition, 2007, 40(7):1911-1920.

[3] Fitzgibbon A W. Simultaneous linear estimation of multiple view geometry and lens distortion[C]// IEEE Computer Society Conference on Computer Vision and Pattern Recognition. 2001.

[4] Montgomery D C, Peck E A. Introduction to linear regression analysis[M]. 1982.

[5]Schreuder R, Baayen R H. Modeling morphological processing.[M]// Encyclopedia of Cognitive Science. 1995.

Appendix

In this part, I want to introduce the structure of program in details. There are three classes for the homework, Imagedata, kernel and algorithm.

Class Imagedata

In class Imagedata, there are six kinds of function such as basic function, Image operation, edge detection, dithering, error diffusion and color halftoning with error diffusion.

Basic function

```
//basic function
Imagedata(int h, int w, int p); //constructor
~Imagedata(); //destructor
void set_doubledata(); //initialize double data for shot noise
void initialize(int d); //initialize all data with value d
void read(unsigned char* buff); //read data from buff
void load(string path); //load data from path
unsigned char* write(); //write data to buff
void save(string path); //save data to path
int convert(int h, int w, int p); // convert the position in a
```

Image operation

```
//image operation
Imagedata Boundaryextension(int ex); //Boundary extension
Imagedata Crop(int N); //crop operation
Imagedata Convolution(Kernel k); //convolution operation with kernel
Imagedata Boundaryextension_double(int ex); //Boundary extension for double data
Imagedata DN_Gaussion_double(int N, double sigma); // Gaussian filter for double data
Imagedata Convolution_double(Kernel k); // convolution for double data
void double2usignedc(); //change double data to unsigned char data
Imagedata Crop_double(int N); // crop operation for double data
Imagedata Save_double(string path); // save double data to path
double* write_double(); // write double data to buff
void pure(unsigned char color); // set the whole image to the specific color
void set_color_data(int i, int j, vector<unsigned char> c); //set color data
vector<unsigned char> get_color_data(int i, int j); //get color data
Imagedata RGB2GRAY(); //convert RGB image to gray-level image
Imagedata Binary(int threshold); // convert gray-level image to binary image in 0 and 255
Imagedata Binarize(); //convert range from 0 and 255 to 0 and 1
Imagedata RBinarize(); //convert range from 0 and 1 to 0 and 255
Imagedata label_pixel(vector<pair<int, int>> v); // label some specific pixels
Imagedata DN_Median(int N); // Denoise filter (Median)
void Color_merge(Imagedata R_component, Imagedata G_component, Imagedata B_component); //
Imagedata get_RGB(int i); //get single channel data
```

Geometric transformation

```
// Geometric Transformation
vector<pair<int, int>> find_corner(); // find the corner for sub-image
Imagedata rotation(); // wrap the rotation
Imagedata modify_edge(int hei, int wid); // modify the edge after rotation
Imagedata counter_clock(); // make a 90 degree counter clock rotation
Imagedata scale(int hei, int wid); // shrink or enlarge the image into a specific size
Imagedata insert(Imagedata A, Imagedata B, Imagedata C); // Insert three sub-image to
Imagedata test_find_corner(vector<pair<int, int>> v); // test the corner found
```

Spatial warping

```
// Spatial warping
vector<Imagedata> seperate(); // seperate RGB image into three separable images
Imagedata Spatial_warping(); // wrap for spatial warping
void Spatial_combine(Imagedata down, Imagedata up, Imagedata left, Imagedata right);
Imagedata Sobel_X(); // X Sobel edge detector
Imagedata Sobel_Y(); // Y Sobel edge detector
```

Morphological processing

```
// Morphological Processing
void square(int size); // generate a square with size
void rectangle(int h, int w); // generate a rectangle with height(h) and width(w)
Imagedata Shrinking(); // wrap for shrinking
Imagedata Skeleton(); // wrap for skeleton
Imagedata Bridge(); // wrap for bridge after skeleton
Imagedata Thinning(); // wrap for thinning
```

Defect detection and correction

```
// Defect Detection and Correction (deer)
vector<pair<int, int>> find_defect(); // find the defect in the body of deer
void emphasize_defect(vector<pair<int,int>> defect,int size); // Circle the defect with a rectangle
void delete_defect(vector<pair<int, int>> defect); // Delete the defect area
void boundary_modify(); // Set the boundary of the image to 0 to get the frame
```

Object analysis

```
// Object analysis (Rice)
// Segment the grains in gray scale image to black(0)-and-white(255) image where grains is 255 and background
Imagedata Grain_segmentation_value_offset(int offset);
// Segment the grains in gray scale image to black(0)-and-white(255) image where grains is 255 and background
Imagedata Grain_segmentation_position_offset(int offset);
vector<pair<int, int>> find_grain(); // find grains in shrinking image
int area_grain(pair<int, int> centre); // calculate the area of grains by the centre in segmentation image
vector<pair<int,double>> average_area(vector<vector<pair<int, int>>> cluster); // calculate the average area
Imagedata cluster_label(vector<pair<int, int>> cluster_single); // Generate a gray-level image with the single
Imagedata cluster_label_RGB(Imagedata cluster_label_Binary); // Generate a RGB image with the single cluster
```

Class Algorithm

```
// Basic function
int correction(double a);
void combine(vector<Kernel> &ori, vector<Kernel> add); // Combine two sets of Kernel to be a bigger one
int comp(int i); // 1 to 0 and 0 to 1

// Morphological processing
vector<Kernel> Condi_Shrink(); // Generate conditional shrink pattern set
vector<Kernel> Condi_Thin(); // Generate conditional thin pattern set
vector<Kernel> Condi_Skeleton(); // Generate conditional skeleton pattern set
vector<Kernel> UCondi_ST(); // Generate unconditional shrink and thin pattern set
vector<Kernel> UCondi_K(); // Generate unconditional skeleton pattern set
vector<Kernel> extendABC(int arr[]); // Convert ABC label to D label in uncondition pattern
Bnode* PATTERNS_TREE(vector<Kernel> k); // Generate the tree according to a set of kernel given

// rice
vector<vector<pair<int,int>>> cluster(vector<pair<int, int>> v); // Divide the grains into 11 cluster according
double distance(pair<int, int> a, pair<int, int> b); // Calulate the distance between two points
```

Class Kernel

```
Kernel(int h, int w); //constructor
~Kernel(); //destructor
void set_data(double d, int i, int j); //set elements(i,j) as value d
void set_wholedata(double d);
double sum(); //get sum
vector<vector<double>> get_wholedata(); //return a two dimension vector as kernel
double get_data(int i, int j); //get the value of element(i,j)
int get_height();
int get_width();
Kernel EQ(); //equalization
void Print(); // Print the kernel
void resize(int h, int w); // Resize the kernel
void Mark_Pattern(int a[]); // convert a array to a kernel
void generate(int a[]); // convert a array to kernel
Kernel Multiplication(Kernel other); // this * other (Matrix multiplication)
```

Class Patterntree

```
Bnode *create_tree(queue<int> s);
void reconstruct(Bnode* b);
bool compare(queue<int> q);
Tree(Bnode* r);
~Tree();

Bnode *root;
```