

# Problem 1: Geometric Modification

## I. Abstract and Motivation

Geometric modification operations are basic digital image processing operations. It not only has been widely used in image processing in 2D case, but also computer vision in 3D case. In this problem, I implemented the 2D image geometric modification algorithms to solve several engineering problems. They all involve the use of matrix multiplication for coordinate changes, so mathematical calculations are the basis of this technology.

## II. Approach and Procedures

### *Bilinear Interpolation*

Bilinear interpolation is used in this problem. For this problem, after we get the mapped coordinate by matrix coordinate operations. The coordinate will always be decimal not integer. To find the mapped pixel, if we round the coordinates to nearest integer, the error will be large, and the visual effect will be bad. Bilinear interpolation is used in this situation. It is the two-dimensional version of linear interpolation. It uses the four surrounding pixels and the distance between the coordinate and pixels' integer coordinates to compute the final interpolated pixel value. Nearer pixels will be assigned with larger weights. Bilinear interpolation is combination of two linear interpolation among horizontal direction, and one linear interpolation among vertical direction based on the result of first two linear interpolation operations. Images obtained using bilinear interpolation, the gradient is smoother, and the visual effect is good. Bicubic interpolation may give a better result but difficult to implement.

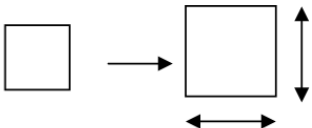
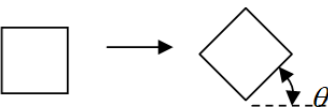
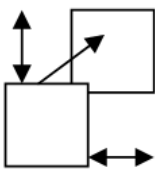
### *Forward Mapping and Inverse Mapping*

When we do geometric modification, there are two kinds of mapping methods. The first one maps the pixels in original image into warped output image. This is called forward mapping (forward warping). And mapping pixels in output image into original image, is called inverse mapping (inverse warping). One pixel's coordinate after forward mapping may not be integers, it can assign its value to nearest neighbor pixel or separate its value to all neighbors in term of the distance between each neighbor. However, this mapping method may lead to having holes in the warped output image. Since, adjacent pixels in original image may not mapped to adjacent pixels in the warped output image, there may be gaps between mapped pixels. No pixel values are assigned to these gaps, so it seems like black holes in the output image. Inverse mapping does not have this defect. All the pixels in the output image are mapped to somewhere in the original

image, so there will be no holes in the output image. One interpolation method should be used to decide the pixel value for those mapped pixels with decimal coordinates. Although, inverse mapping is good, an invertible warp function is not always possible. As a result, in this problem, all the sub problems are implemented using inverse mapping. Even in sub problem c, it is impossible to find an inverse mapping, we can still use an approximate inverse function to get a result.

## Geometric Transformation

In this sub problem, we need to implement 3 transformations, scaling, rotation and translation. They can be done by following matrix operations

Name	Demonstration	Equation	Matrix Operation
Scaling		$x = s_x u$ $y = s_y v$	$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$
Rotation		$x = u \cos \theta - v \sin \theta$ $y = u \sin \theta + v \cos \theta$	$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$
Translation		$x = u + t_x$ $y = v + t_y$	$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$

A geometric transformation is a combination of these 3 operations in order. And changing the order of operations will lead to different results.

Sub problem a require us to filling 3 white regions in lighthouse.raw with 3 pieces in lighthouse1.raw, lighthouse 2.raw and lighthouse 3.raw.

First, I find the left top corners and the sizes of three white regions by traversing all the pixels. There are some pixels whose value is white but not in any white region, the program should distinguish this kind of pixels. Second, I also need to find the coordinates of 4 corners in each piece. This is challenging, since the pixel values in piece images are interpolated, and the real corner coordinates are decimal values. My solution for finding corners has 3 steps to get better

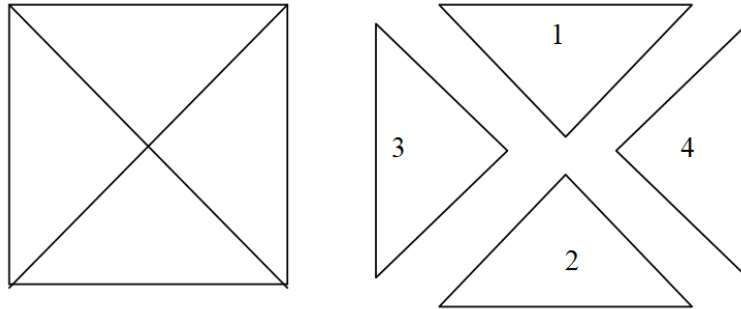
result. First, after the program find the first / last row / column that contains nonwhite pixels, then find the darkest pixel in this row / column. Since the background is white and pieces are dark area in original image, this method works. But we can still find white edge in the output image. That is because these corner pixels are the result of interpolation between white background and nearby image pixels. So, I set a parameter to shrink the corner coordinates called piece boundary shrinking, and make the bounding box smaller. This is the second step. However, this procedure does not always work well, since it is applied to all 4 corners, although white edges may disappear, some pieces may zoom in and the image will slightly be misplaced. For the third step, the program slightly adjusts the result by making the rake ratios of both sides become equal (and included angles become right angles). This is done by searching around neighborhood pixels and try every possible corner coordinate combinations and record the combination with minimal error. In my implementation, I did not do it so complicated, each corner only tried 2 possible positions, one for original position and one for one-pixel inner position, that leads to 16 combinations.

Then, find the angle for rotation, the scale for scaling and the shifting for translation. I need to find the angle for rotation at first, that is because other values' calculation is based on this angle to determine which corner is the left top corner. Computing the  $\theta$  is by using arctan function. However, there will be 2 (for rectangle) or 4 (for square)  $\theta$  that match the requirement in  $2\pi$  range. I did not hard code the range for angles for each piece. I just let the program try all possibilities for each piece and get a geometric transformation result, then record the output image with minimal pixel value error along the hole's boundary. Scale can be found by comparing each corresponding sides' length. Scaling factors for two directions may be different, so it needs the knowledge of  $\theta$  to match the edges. However, in this problem, all pieces are square, I still implemented a scaling operation with full function. There are two translation procedures, one for moving the white region's left top corner to origin of coordinate, one for moving the origin of coordinate to pieces' left top corner. Shifting values are just the coordinates of left top corners.

Then, translate, scale and rotate these white regions to match the pieces. The first operation for each white region's geometric transformation is translation, it translates the white region's left top corner to origin of coordinate. Second operation is rotation. The third operation is scaling. The fourth operation is translation, it translates the white region to the position that the piece shows. Finally, multiply these matrices for each operation in order to get a geometric transformation matrix. Traverse each pixel in that white region, using this matrix to compute the inverse mapping coordinate, and using bilinear interpolation to find the right pixel value.

### *Spatial Warping*

In sub problem b, I map pixels in output image to pixels in original image. To make the warping easier, I divide the image into 4 regions, north, south, west and east.

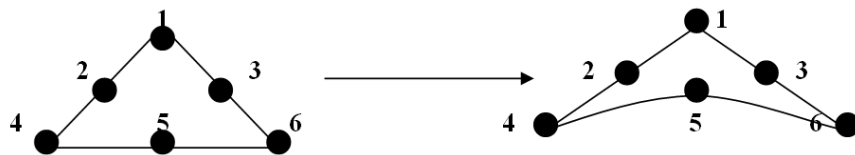


Each region is a triangle. The following equation is the mapping function.

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 & a_4 & a_5 \\ b_0 & b_1 & b_2 & b_3 & b_4 & b_5 \end{bmatrix} \begin{bmatrix} 1 \\ x \\ y \\ x^2 \\ xy \\ y^2 \end{bmatrix}$$

Since I do not know the first matrix in right hand side, I need to set 6 pairs anchor points, then solve an equation to get factors in that matrix.

Anchor point pairs for each triangle are defined like this



For anchor pair 5 in figure above, the magnitude of warping is given in the problem's statement.

Then the factors can be computed by

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

The inverse of matrix can be computed by  $A^{-1} = \frac{A^*}{|A|}$ , I implemented this equation in my program. Finally, for each triangle, map pixels in output image inside that triangle to original image by using these factors, then calculate the pixel value by using bilinear interpolation.

### *Lens Distortion Correction*

In this sub problem, an image is distorted by following equation to emulate lens distortion

$$x_d = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

$$y_d = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

$$r^2 = x^2 + y^2$$

Where  $k_1 = -0.3536$ ,  $k_2 = 0.1730$ ,  $k_3 = 0$ ,  $x$  and  $y$  are pixel coordinate in undistorted image under camera coordinate system. Image coordinate system can be converted to camera coordinate system as follows

$$x = \frac{(u - u_c)}{f_x}$$

$$y = \frac{(v - v_c)}{f_y}$$

where  $u_c$  and  $v_c$  is the center of the image, and  $f_x = f_y = 600$  are the scaling factors. Only distorted image is given, I need to recover it and get undistorted output image. However, the inverse mapping function does not exist. Here I use both linear regression and non-linear fitting to estimate the inverse function. Estimating are based by samples generated by using the forward mapping equation. Inverse function is separated into horizontal and vertical directions, and they are estimated separately. For horizontal direction, the independent variables are  $x$  and  $y$ , the dependent variable is  $x_d$ . For vertical direction, the independent variables are  $x$  and  $y$ , the dependent variable is  $y_d$ . After I get the estimate inverse function, do the inverse mapping to get an undistorted image. For non-linear fitting, I use 4 order polynomial function as its model to find a fitting using Gauss-Newton method. The inverse mapping equation has following formation

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\ b_0 & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 & b_8 & b_9 & b_{10} & b_{11} & b_{12} & b_{13} & b_{14} \end{bmatrix}^*$$

$$\begin{bmatrix} 1 & u & v & u^2 & uv & v^2 & u^3 & u^2v & uv^2 & v^3 & u^4 & u^3v & u^2v^2 & uv^3 & v^4 \\ 1 & u & v & u^2 & uv & v^2 & u^3 & u^2v & uv^2 & v^3 & u^4 & u^3v & u^2v^2 & uv^3 & v^4 \end{bmatrix}^T$$

Fitting for higher order maybe difficult. So, I only tried up to 4 order for demonstration. The linear regression can be seen as 1 order fitting.

Then, I found that the camera coordinate system can be easily convert to polar coordinate. Since the  $\theta$  are the same in input image and output image, so the independent variable for fitting reduced to one, only radius. Then I do the nonlinear fitting again. The inverse mapping equation has following formation


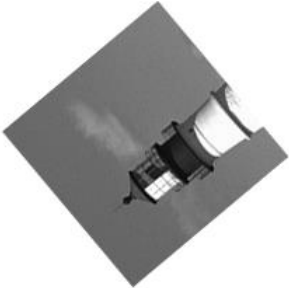

$$\mathbf{r} = [a_0 \ a_1 \ a_2 \ a_3 \ a_4][1 \ r \ r_d^2 \ r_d^3 \ r_d^4]^T$$




(Only program for this sub problem in this homework is written in MATLAB)

There is another way to deal this problem. Since the forward mapping equation is given, I can build a look-up table without knowing the inverse mapping equation by traverse all pixels, the key is coordinate of mapped pixel, and the value is the coordinate of corresponding original pixel. Then send pixels in distorted image to its original position. However, I just implemented this method for reference. I do not do any post processing for that result. (This approach is implemented in C++)


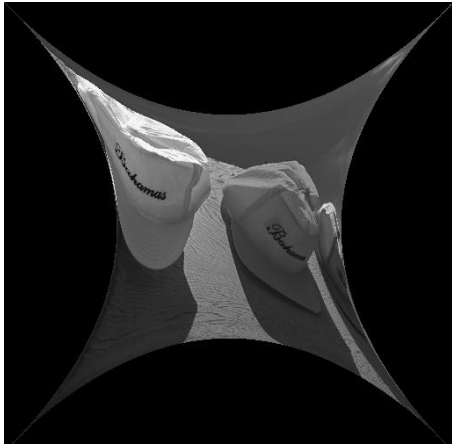
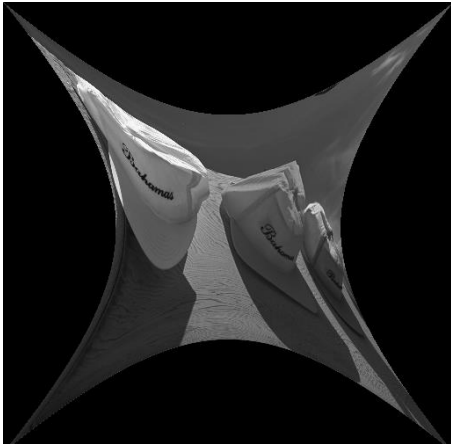
### III. Experimental Results

#### *Geometric Transformation*

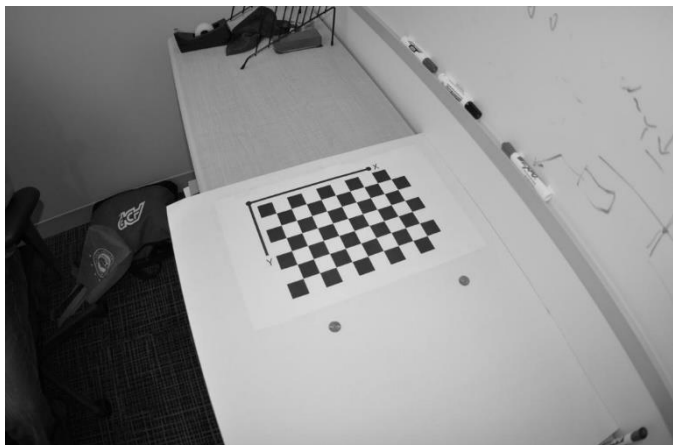

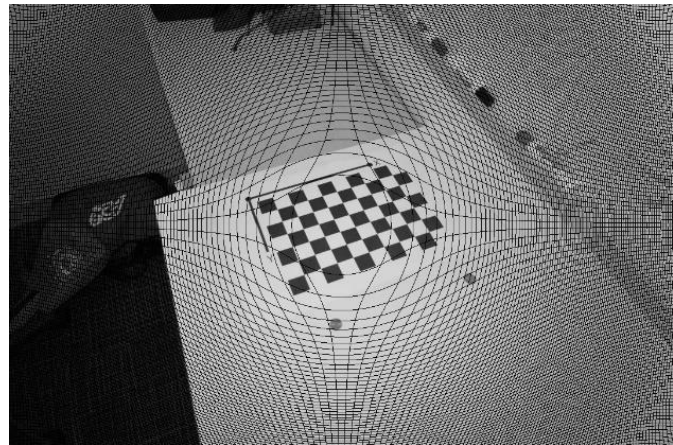
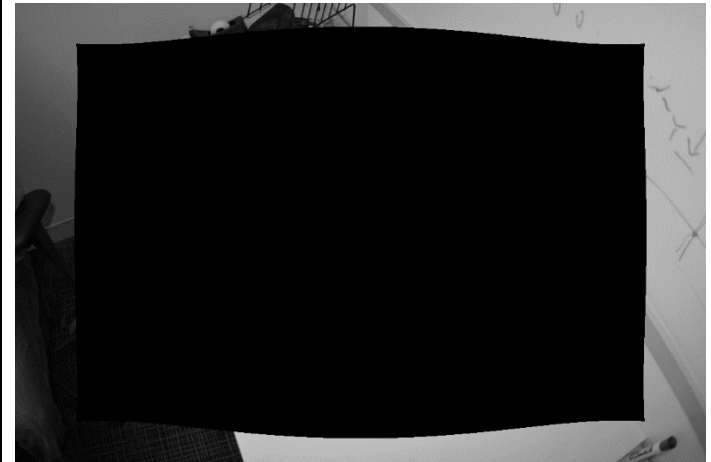
		
Piece 1 (lighthouse1.raw)	Piece 2 (lighthouse2.raw)	Piece 3 (lighthouse3.raw)

		
<p>Input image (lighthouse.raw)</p>	<p>Output image (piece boundary shrinking disabled)</p>	<p>Output image (piece boundary shrinking enabled (1 pixel))</p>

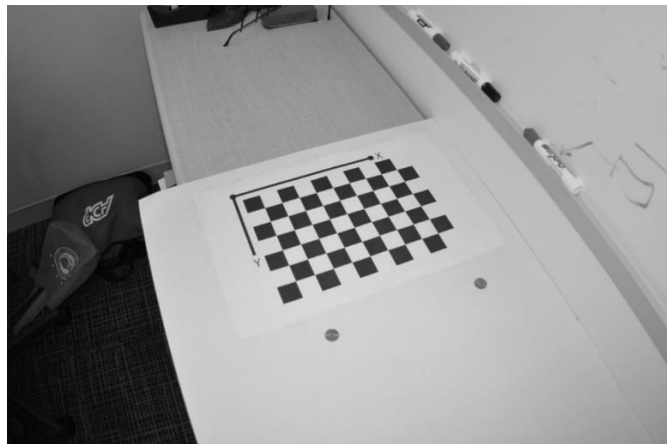
## *Spatial Warping*

		
<p>Input image (hat.raw)</p>	<p>Reference output image (forward mapping)</p>	<p>Output image (inverse mapping)</p>

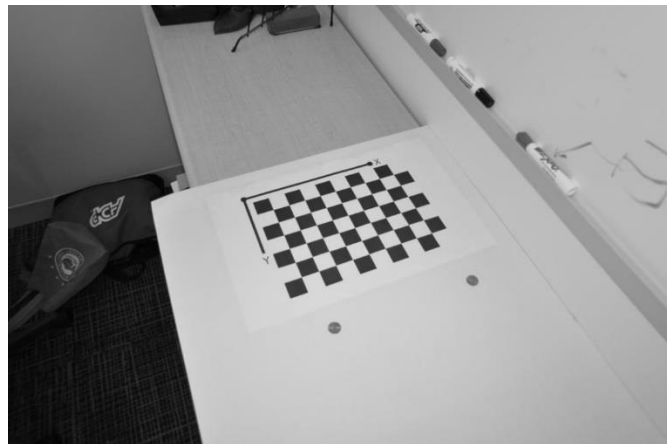
# *Lens Distortion Correction*

	
<p>Distorted image (classroom.raw, 1072*712)</p>	
	
<p>Reference undistorted image (forward mapping look-up table, 1072*712)</p>	<p>Reference unmapped pixels (forward mapping look-up table, 1072*712)</p>

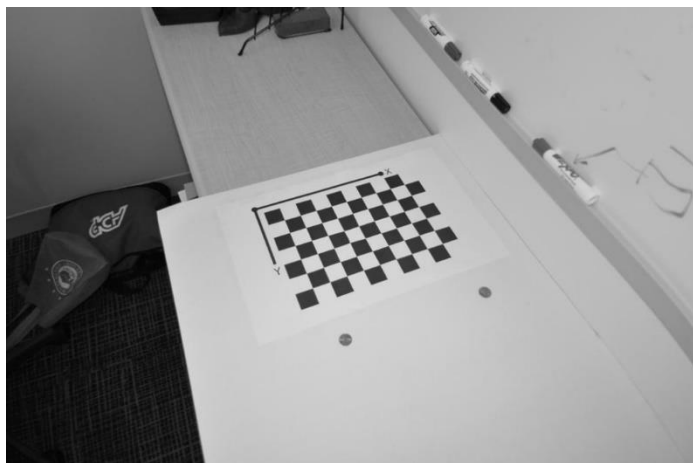




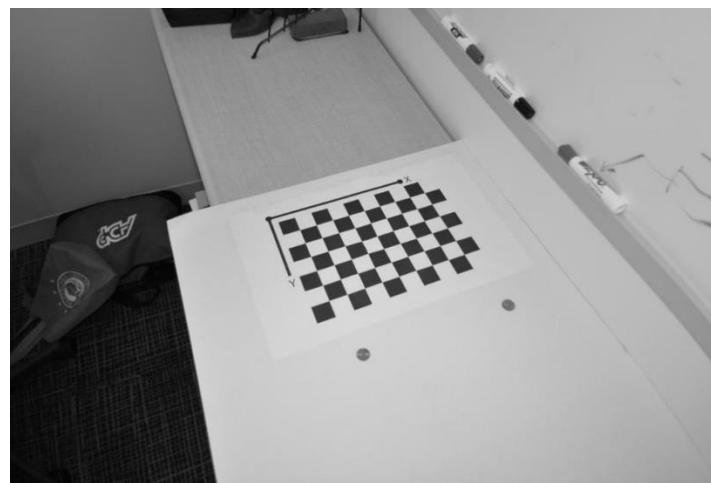
Undistorted image (rectangular coordinates, linear regression, 1072\*712)



Undistorted image (rectangular coordinates, non-linear fitting: 4 order polynomial, 1072\*712)



Undistorted image (polar coordinates, linear regression, 1072\*712)



Undistorted image (polar coordinates, non-linear fitting: 4 order polynomial, 1072\*712)

## IV. Discussion

### *Computing Complexity*

For basic geometric transformation, the computing complexity is  $O(mn)$ , where  $m$  and  $n$  are the height and width of the input image. Computing transformation matrix does not require much computing resources. However, for sub

---

problem c, non-linear fitting is done by Gauss-Newton method, it needs multiple iterations of trials. So, if the order is high, the fitting needs more time.

### *Geometric Transformation Result Analysis*

I have two results for sub problem a, piece boundary shrinking enabled and disabled. All results show white edges, however enabling shrinking gives a better result. However, when we zoom in transformed piece 1, the house's roof does not match very well. The scaling factor zoom that piece too much after enabling shrinking function. This problem for other pieces are not as serious as that for piece 1. I think that is because piece 1 in lighthouse1.raw is the smallest one among 3 pieces, that lead to a large error after scaling it up.

### *Compare Forward Mapping and Inverse Mapping*

In sub problem b, I tried forward mapping and inverse mapping both. The result of inverse mapping is better than that of forward mapping. The curve edges of result are smoother, and the detail of original image does not lost. On the right top of original image, there is a small black region, this region is lost in the result of forward mapping while reserved in the result of inverse mapping. That is because forward mapping only assigns values to nearest pixels, newer values will cover the older value for squeezed regions. And it is also very hard to implement a scheme that assign pixel value to all neighbor pixels, since the value may overflow if multiple pixels are mapped to its surrounding area. As a result, inverse mapping is better than forward mapping.

### *Image Size of Lens Distortion Correction Results*

In problem c, if keeping the size of output image as the same size as input image. All the results show a smaller field of vision. That is because the forward mapping function compresses the space of image. We can deal this by enlarging the output images' resolution. Such as

<p>Undistorted image (rectangular coordinates, linear regression, 1322*962)</p>	<p>Undistorted image (rectangular coordinates, non-linear fitting: 4 order polynomial, 1322*962)</p>
<p>Undistorted image (polar coordinates, linear regression, 1322*962)</p>	<p>Undistorted image (polar coordinates, non-linear fitting: 4 order polynomial, 1322*962)</p>

We can see that the distortion of edge is larger. That means only distortion around center can be corrected well.

### *Comparison Between Linear Regression and Non-linear Fitting*

Results from polar coordinates give better correction results, the straight lines become straight. For results from rectangular coordinates, non-linear fitting gives better result than linear regression. However, for polar coordinates the

difference is not obvious, that means the inverse function may have simple formation under polar coordinate system and a lower order polynomial is enough.

The boundary of reference image's unmapped pixels, it has a corresponding law with the results' boundary obtained from the polar coordinate system. The prominent place corresponds to the recessed place. This is another evidence to prove that using polar coordinate may give an estimated inverse function close to the true inverse function.

## Problem 2: Morphological Processing

### I. Abstract and Motivation

Morphological processing operations focuses on the shape of objects in digital images. Two sorts of basic morphological processing operations are dilation and erosion. Dilation make the object grow larger by adding pixels at object's boundary, and erosion do the opposite thing. In this problem, I implemented three erosion operations, shrinking, thinning and skeletonizing. I use these operations together with other algorithms to some several engineering problems.

### II. Approach and Procedures

#### *Zero Padding*

In this problem, objects are marked as white, and background is marked as black. In my implementation, I use Boolean data type to store pixels. Objects are assigned to true, and background is assigned to false. To perform erosion correctly, I have to enlarge the image by adding black background pixels. So, the new image will have a larger size, with 2 more pixels both in horizontal and vertical directions. This is called zero padding. The following operations are done by using this larger image. After that, the extra pixels are removed to recover the original size of the input image.

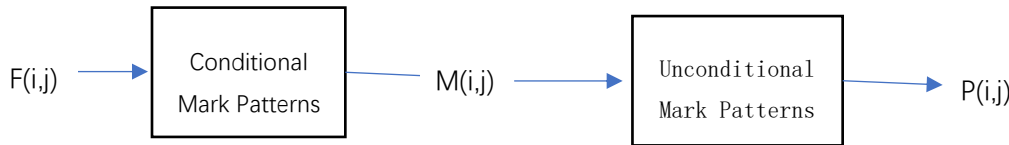
If I do not do zero padding, when we met images with objects reach the boundaries of images. Morphological processing operations will not get correct results. Parts that objects touch the boundary will not be eroded.

#### *Basic Morphological Process Implementation*

There are three basic morphological processing operations in this problem, shrinking, thinning and skeletonizing. They all share the same processing procedures, while skeletonizing needs one more step of post-processing called bridging.

Operations should be done multiple iterations, until no pixel can be eroded any more. For each iteration, image is

compared with conditional mark patterns and unconditional mark patterns with following order



$F(i,j)$  is the input image for that iteration,  $M(i,j)$  is the result after applying conditional mark patterns,  $P(i,j)$  is the result after applying unconditional mark patterns. For each kind of operation, conditional mark patterns and unconditional mark patterns are different, but the processing procedures are the same. We use two step procedures, so that 3 by 3 patterns are enough. Although, these operations can be done by one step using 5 by 5 patterns, the number of 5 by 5 patterns are large and is hard to be hard coded in to the program. So, in my implementation, all the patterns are 3 by 3, and will be compare with the pixel in the middle and its 8 neighbors. If all 9 pixels is the same with one of patterns, the output value for this compare will be true, otherwise false. After we get  $F(i,j)$ ,  $M(i,j)$  and  $P(i,j)$ , we can get the result for this iteration by

$$G(i,j) = F(i,j) \cap (\overline{M(i,j)} \cup P(i,j))$$

A flag variable is set to see whether in an iteration, any pixel value has been changed. If no change, then exit the loop. All the patterns are hard coded, they are used to judge whether an object pixel is at the outer boundary of the object. If yes, it will be eroded in that iteration. While, different operations have different sets of patterns, so the result will be different. For shrinking, objects without holes inside will finally become a dot in the middle of that object, objects with holes inside will become rings around those holes. Thinning will erode the pixels on each direction equally, until one of the directions cannot be erode any more. Skeletonizing is quite different, it can show the structure of objects with stick figure representations. Sometime, results from thinning and skeletonizing are similar. The additional procedure for skeletonizing is bridging, it is performed after the iterations and only once, and it connects the disjointed pixels with one-pixel gap together.

### *Defect Detection and Correction*

In this sub problem, there is a deer image. We need to detect whether there are defects in that image. Defects here are defined as unwanted black dots inside the object. Since the shrinking operation will show different results with object with and without holes inside the object, it is suitable for this task. So, for the first step, shrinking is applied, then the number of rings in the output will be the number of holes inside the object. While, some holes are not defects, such as holes formed by deer's antlers. So, we need to judge whether a hole is defect. The key is the area size of holes, since defects are only small holes, if we know the area size of each hole, we can distinguish them by applying a threshold. Holes with smaller area will be counted as defects. The next problem is how to measure the area. 4 directions flood-fill algorithm is suitable for this task. Since we already have the rings, we can find black pixels in original image when filling rings using flood-fill on the output image. Then we know the coordinates of black pixels in each ring. Since in each ring, there will be

only one black hole, so we randomly pick one coordinate and do the flood-fill again to count the size of connected region, but this time on the copy of original image not the output image. Then we know the area of each hole in the object, so the threshold is applied to determine whether that hole is a defect. The next task is to correct defects to white pixels. This can be done by keep a copy of original input image, as long as a hole is confirmed as a defect by applying the threshold, fill that hole in the copy using flood-fill. Then we will get a defect less result. Setting the threshold is important, but in the given deer image, we cannot see defects at one glance, so even if it has defects, the area will be very small, and will vary greatly differ from the normal hole area. As a result, all detection and correction procedure are done automatically, no manual intervention required.

## *Object Analysis*

In this sub problem, we are required to find all the grains, then classify them into types, and sort classes by their size. One pre-processing procedure is to binarize the color image. Background will be set to black, objects will be set to white. I compare each pixel with sample background pixel color, if its color distance is smaller than a given threshold, it will be set to black. To reduce misjudgment for rice pixels which will have detrimental effect on subsequent computation of rice grains' size, this threshold should not be very large. Finding a good threshold is a challenge, it cannot be too big or too small. While, it can be done automatically by run the whole algorithm multiple times with different thresholds and find the thresholds that will have same stable number of rice grains result. Then find the smallest threshold which will give the same number of grains as the stable result. Furthermore, with a small threshold, sometime background pixels will be misjudged as object pixels. For this sub problem, the given raw image seems like be converted from a jpeg image, since there are wavelet form distortions appeared in the background where should be solid color. This fact leads to a more serious situation of misjudgment. So, the in the next step, we need to remove unwanted object pixels. The principle is the same as the previous sub problem. A threshold is needed to determine whether a connected object area is a defect by its size of area. Then we can correct those defects, and get a clean image with white pixels representing the rice grains and black pixels representing the background. Counting the number of rice grains is easy, use flood-fill to find the number of unconnected regions. At the same time, we can compute the area size of each rice grains, or we can do it latter.

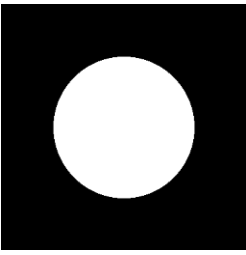
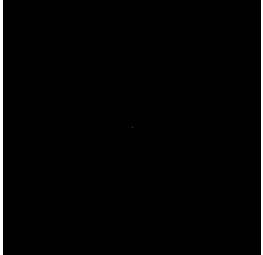

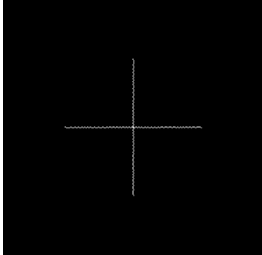
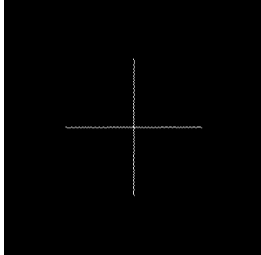

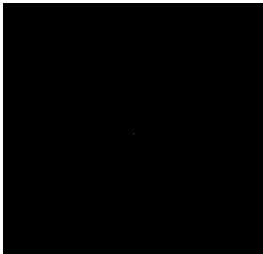
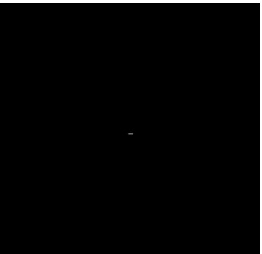
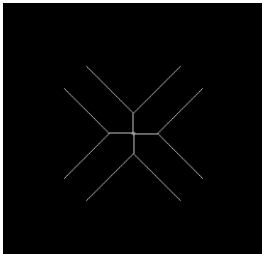
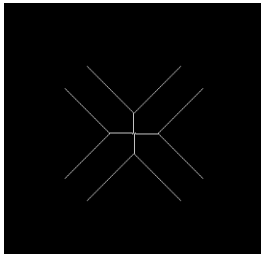
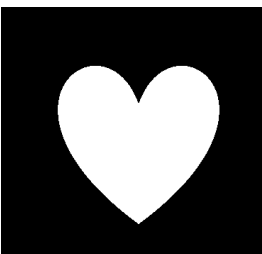
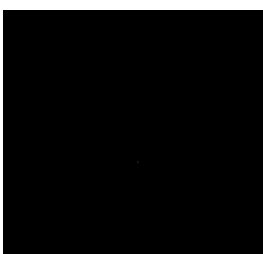
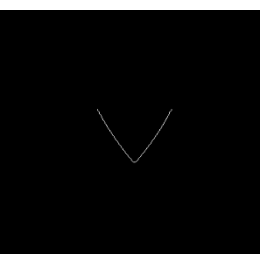
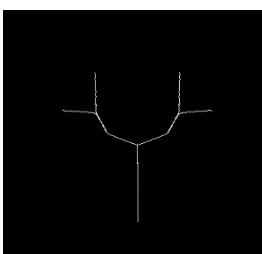
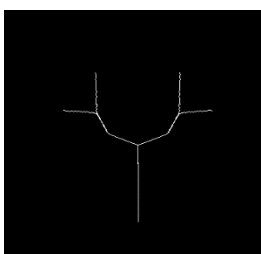
Next, we want to classify these rice grains, we can do this by its shape and its color. While the information give by color is more important than the information given by shape, since all rice grains are oval and have individual differences. Thinning and skeletonizing do not come in handy here. We can distinguish the type from the histogram pattern of the rice pixel colors, while it can be a little difficult to implement. Then, I noticed the fact that the same kind of rice grains will be put together in the image. So, I use K-means written by myself to classify rice grains automatically. Since we have already known there are 11 types of rice, so the K for K-means is set to 11. No other parameters or hand coded features are required. K-means should be run multiple times and pick the result which has the minimum overall error. However, to perform K-means, I need to give a coordinate for each rice grains. This can be done by traverse the image pixels, and use the fist come out pixel coordinate to represent that rice grain. But it is not the best choice since I already have shrinking operation. I use shrinking to find the geometric center of each rice grain. This will improve the accuracy of K-means. Al

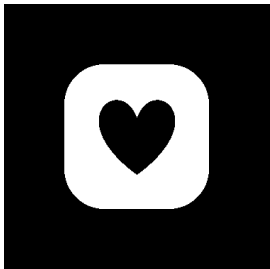
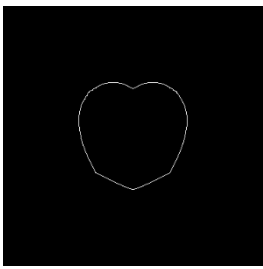
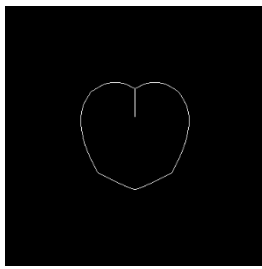
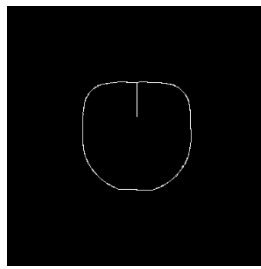
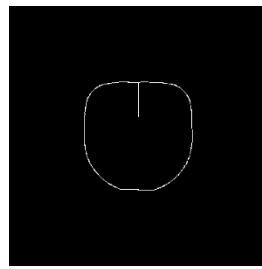
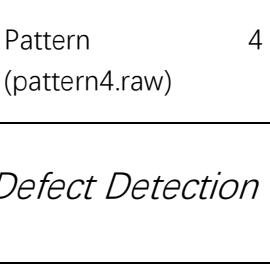
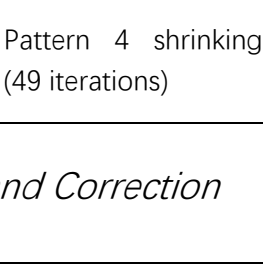
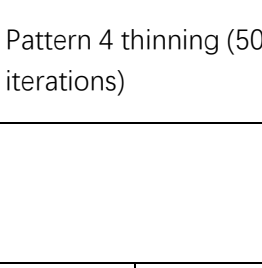
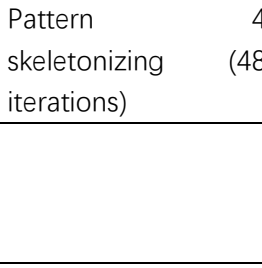
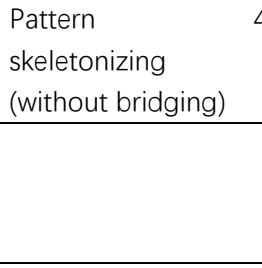
last, computing and order the types with their average size is simple, just perform a sorting algorithm.

(Problem 2c hint is posted after I finished this problem. The program is written in C++ and no external library is used.)

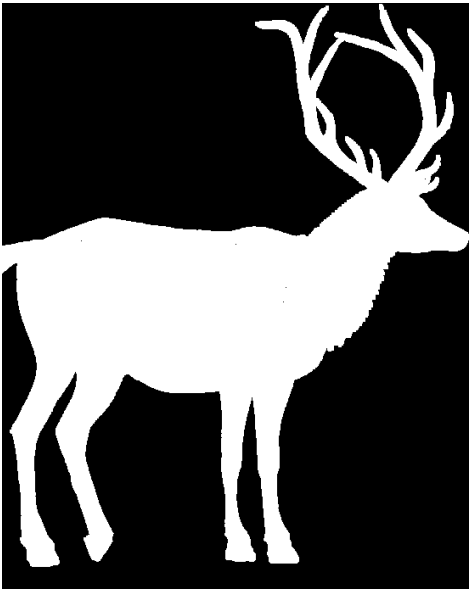
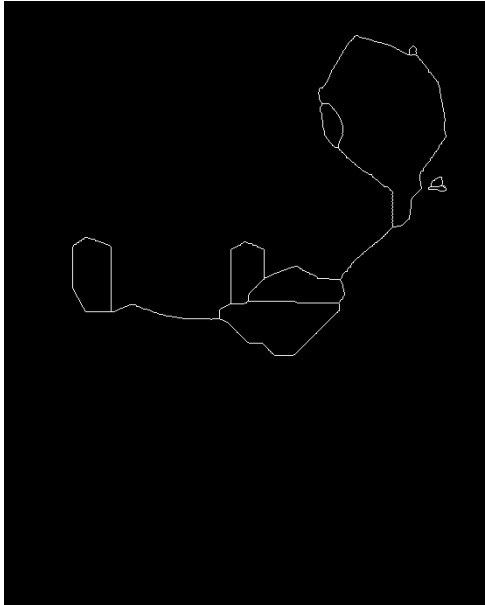
### III. Experimental Results

#### *Basic Morphological Process Implementation*

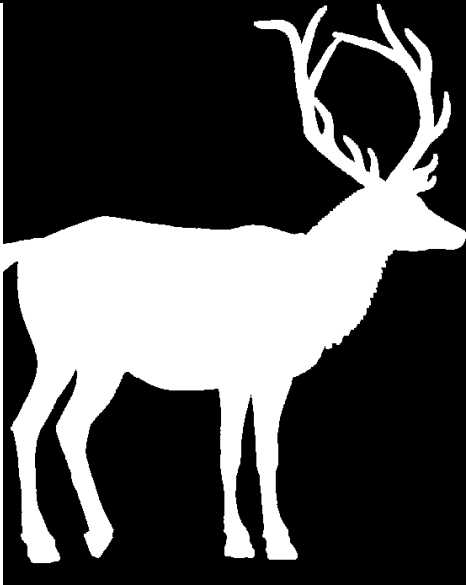

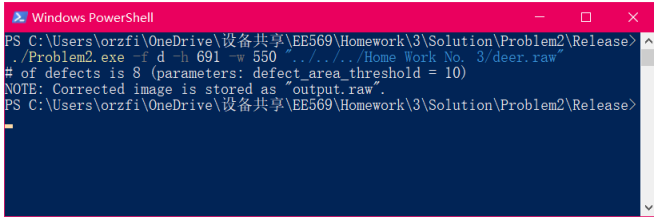
				
Pattern 1 (pattern1.raw)	Pattern 1 shrinking (109 iterations)	Pattern 1 thinning (109 iterations)	Pattern 1 skeletonizing (77 iterations)	Pattern 1 skeletonizing (without bridging)
				
Pattern 2 (pattern2.raw)	Pattern 2 shrinking (101 iterations)	Pattern 2 thinning (98 iterations)	Pattern 2 skeletonizing (97 iterations)	Pattern 2 skeletonizing (without bridging)
				

Pattern 3 (pattern3.raw)	3	Pattern 3 shrinking (143 iterations)	Pattern 3 thinning (89 iterations)	Pattern 3 skeletonizing (85 iterations)	Pattern 3 skeletonizing (without bridging)
					
Pattern 4 (pattern4.raw)	4	Pattern 4 shrinking (49 iterations)	Pattern 4 thinning (50 iterations)	Pattern 4 skeletonizing (48 iterations)	Pattern 4 skeletonizing (without bridging)
					

### Defect Detection and Correction

	
Input image (deer.raw)	Shrinking (354 iterations)



	
<p>Corrected defect less image</p>	<p>Defect area (threshold 10, defect less image minus input image using minus.m)</p>
	
<p>Command line output screen shot (# of defects is 8)</p>	

*Object Analysis*

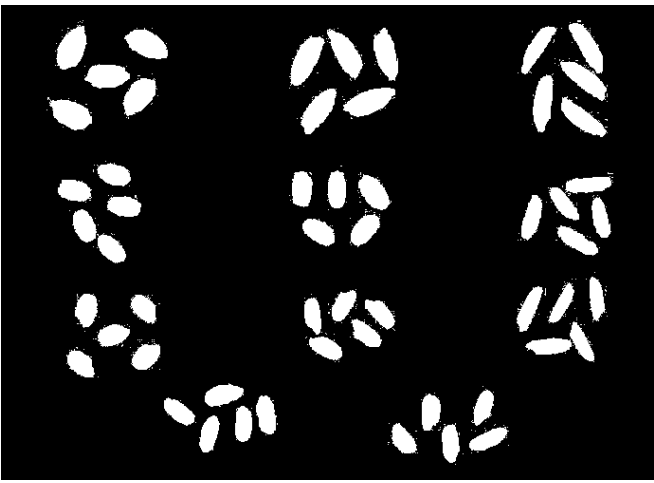
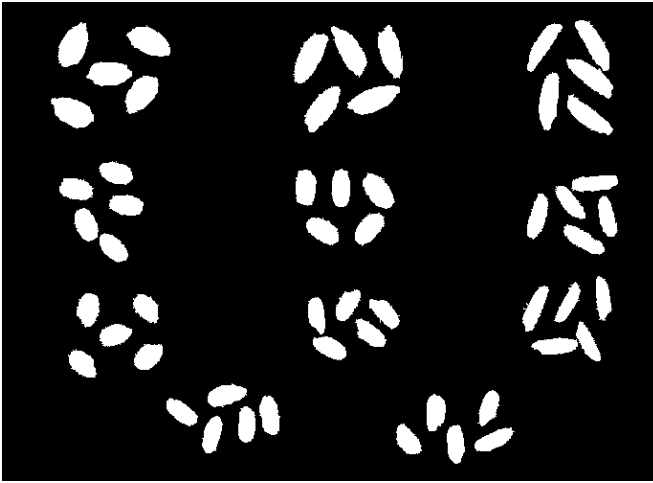

Input image (rice.raw)

```

Windows PowerShell
PS C:\Users\orzfi\OneDrive\设备共享\EE569\Homework\3\Solution\Problem2\Release>
./Problem2.exe -i r -h 500 -w 690 -c 3 -t 10 -k 11 -n 100 -f rice.raw
# of rice grains is 55 (parameters: color_distance_threshold = 16, defect_area_
threshold = 10, K = 11, #_of_trial = 100)
All rice grains are classified automatically. Sorted results are shown below.
Type  Row  Col  Area
-----
1      327  328  526
1      313  362  562
1      346  384  565
1      324  399  574
1      360  341  581
2      425  507  539
2      457  424  555
2      461  474  595
2      456  515  596
2      429  453  620
3      313  593  519
3      354  611  537
3      310  629  549
3      359  576  644
3      311  560  671
4      318  150  565
4      346  126  582
4      377  84   584
4      370  152  608
4      321  90   634
5      439  256  565
5      428  187  566
5      453  219  576
5      431  279  664
5      410  234  689
6      255  116  595
6      212  127  615
6      232  88   627
6      178  118  629
6      195  78   636
7      208  593  568
7      223  632  580
7      189  618  662
7      223  560  672
7      247  606  683
8      192  354  644
8      239  336  670
8      193  317  677
8      236  384  680
8      192  389  791
9      43   565  765
9      120  613  880
9      44   620  909
9      78   617  935
9      104  571  938
10     75   111  906
10     97   145  926
10     40   153  968
10     114  70   988
10     44   74   1015
11     51   405  945
11     113  332  983
11     51   364  1010
11     102  388  1040
11     59   321  1088
PS C:\Users\orzfi\OneDrive\设备共享\EE569\Homework\3\Solution\Problem2\Release>

```

Command line output screen shot (# rice grains is 55)

	
<p>Binarized image</p>	<p>Defect less image (55 connected regions)</p>
	
<p>Average size ranking from small to large in terms of type. (average size difference: 19.4, 3, 10.6, 25.8, 12.6, 59.4, 193, 75.2, 52.6)</p>	

## IV. Discussion

### *Computing Complexity*

All these morphological processing take a long time to process. The computing complexity for these task is  $\theta((mn)^2)$ , where  $m$  and  $n$  are the height and width of the input image. For Shrinking deer image, under debug mode it takes me more than 7 minutes, and a few seconds under release mode. We can see that optimizations for this task is important.

### *Shrinking, Thinning and Skeletonizing*

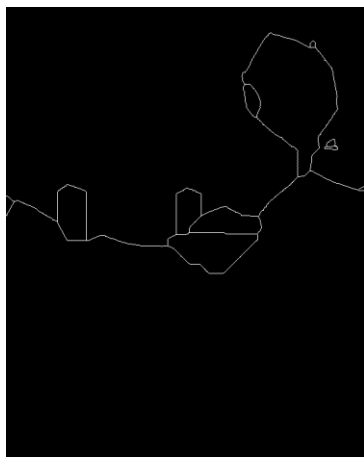
As I mentioned above, shrinking shrink object to a dot or a ring. If there is no hole inside the object, shrinking result is a dot, otherwise a ring at midway of its nearest boundary. Thinning shrink the object's boundary until one of the directions has shrink to one-pixel height / width in each small region. Skeletonizing draws the skeleton of the object.

### *Bridging*

For sub problem a, only pattern 4 has different output before and after bridging. We can see some broken strokes are connected by bridging.

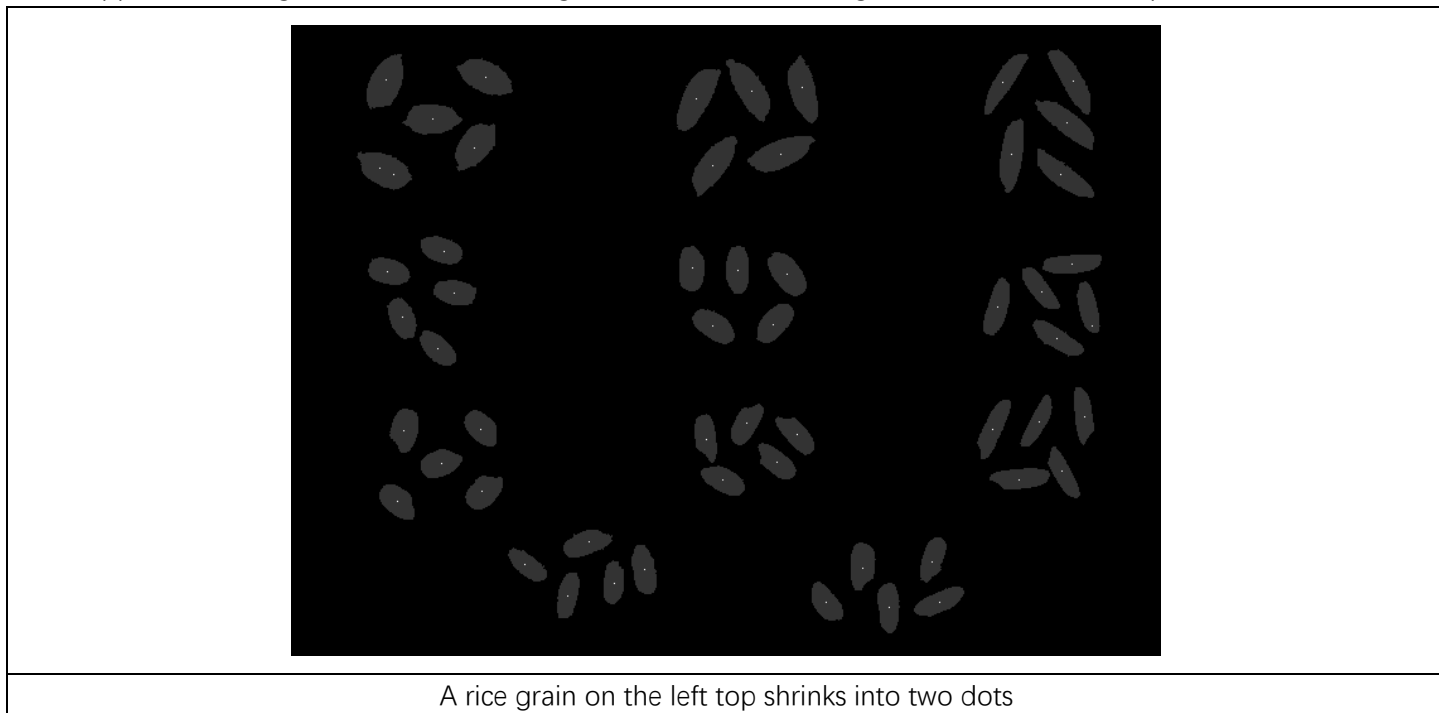
### *Shrinking Deer Image without Padding*

If there is no zero padding, the result may be different. For deer image, the object touches the boundary. The result is shown below and it is not correct.



### *Defect of 3\*3 Mark Patterns Approach*

After I applied shrinking to the defect less image, I found that one rice grain shrinks to two independent dots.



I'm sure that my implementation is the same as the book's and has no bug. So, I think the book's implementation has a defect. However, this defect will not influence the final result after adding some judgments. The book says that these 2 steps 3\*3 mark patterns approach has less patterns than the approach of 1 step 5\*5 mark patterns. I think this problem is caused by missing some patterns in the book's pattern tables, otherwise this approach should be discarded. Since I implemented this by pure C++, there is no convincing way to compare my result with other implementations.

### *Ranking for Average Size of Rice Types*

As I mentioned above, the threshold for binarizing the input image influences the final size of rice grain objects. After I ordered the rice types by average size, I found that the 2<sup>nd</sup> and 3<sup>rd</sup> type have very small difference between their average size. Then I saw the binarized image and defectless image and realized the threshold did not influence the size very much. So, these two types may have similar size of grains.