EE 569: Homework #3

Issue: 2/13/2019 Due: 11:59PM 03/03/2019

Name: Wenjun Li
USC ID: 8372-7611-20
Email: wenjunli@usc.edu
Submission Date: 03/03/2019

Problem 1: Geometric Modification (50%)

(a) Geometric Transformation (20%)

(1) Motivation

Geometric property is one of the most important properties of an image. An image of different size, angle, position, etc. can express different information. Therefore, people want to control the shape, size, rotation angle, and location of images. Also, with the rising of social media, people want to edit their photos' shape, size, angle, and etc. In conclusion, this area is very important for image processing.

(2) Approach

(Note: no built-in function used for artifacts removal)

For Prob.1.(a), in order to fill the holes in lighthouse.raw, we need first find the coordinates of corners in each sub-image. i.e. lighthousr1, lighthouse2 and lighthouse3.

For lighthouse1-3, my idea is to find the first pixel that is not white (255). For example, for lighthouse 1 (the image a in Figgure.1). I write a loop to scan from top-left corner to bottom-right (in each row, from left to row), and it will return the 1st pixel that is not white. So, we can find the top-left corner pixel location for lighthouse1. Then, my program scans from top-right corner to bottom-left corner (in each column, from top to bottom), and it can return the top-right corner pixel location of lighthouse1. Similarly, my program can find the other 2 corner pixel locations in lighthouse1. And I apply this program to lighthouse2 and lighthouse3, and it finds all the corner pixels locations.

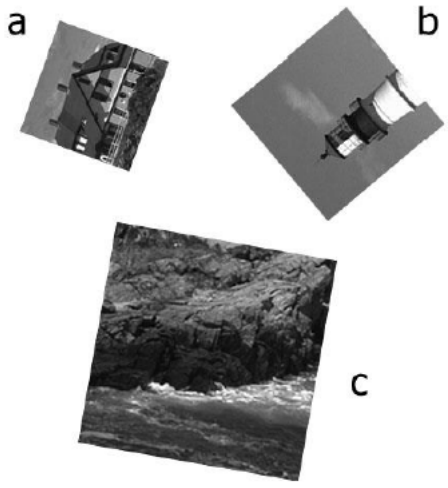


Figure 1 Lighthouse 1-3

Here, my program returns the corner pixel coordinates as below (I label each corner with their correct position):

positions of 4 corner pixel of lighthouse1 = (row, column)
top left corner pixel = (80,115)
bottom left corner pixel = (182,78)
bottom right corner pixel = (223,182)
top right corner pixel = (118,221)
positions of 4 corner pixel of lighthouse2 = (row, column)
top right corner pixel = (8,125)
top left corner pixel = (100,15)
bottom left corner pixel = (212,108)
bottom right corner pixel = (116,218)
positions of 4 corner pixel of lighthouse3 = (row, column)
top left corner pixel = (3,40)
bottom left corner pixel = (209,3)
bottom right corner pixel = (251,210)
top right corner pixel = (40,251)

Next, find the coordinates of the holes (specifically the top left corners) so that we can put the sub-images into right hole location. For convenience, I denote each hole with A, B and C as shown in Fig.1. Similar as the method I used to find corners for lightouse1-3, here, my idea it trying to find the 1st white pixel from different scanning orders (e.g. from top-left to bottom right, from top-right to bottom-left, etc.).

Here my program returns the 3 top left pixel coordinates are:

positions of 3 top left corner pixel of holes in lighthouse = (row, column)
top left corner pixel of hole A = (31,278)
top left corner pixel of hole B = (157,62)
top left corner pixel of hole C = (327,325)

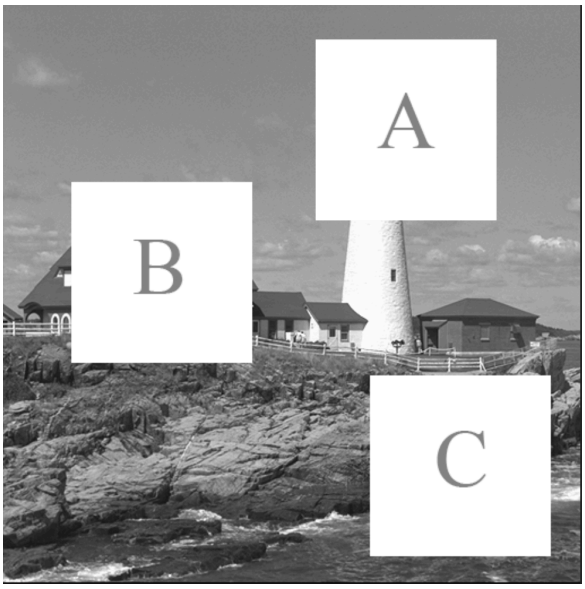


Figure 2 Holes in Lighthouse

Next, we need to find the transformation that can map lighthouse 1-3 to the correct hole location in lighthouse. Since scaling, rotation and translation are linear operation, we can find a linear transformation matrix to do the job. Below is the description of geometric transformation matrices.

- Scaling

The scaling matrix can be expressed as below:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

where (u, v) is the indexes of original image and the (x, y) is new indexes; and Sx and Sy are the scaling factors in horizontal direction and vertical direction.

- Rotation

The rotation matrix can be expressed as below:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} cos\,\theta & -sin\,\theta & 0 \\ sin\,\theta & cos\,\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

where (u, v) is the indexes of original image and the (x, y) is new indexes; and θ is the rotation angle. The rotation direction is clockwise. So, if you want to rotate the matrix by 30°, you should assign θ =30°.

- Translation

The last linear operation is translation. This operation works when you want to move a image to a new location without changing the shape or size.

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

where (u, v) is the indexes of original image and the (x, y) is new indexes; and tx and ty are

the distance you want to move the image.

When we do the mapping operation, we usually do reverse mapping rather than forward mapping. There are several advantages of reverse mapping.

First, reverse mapping will not have artifacts. As you can see from the below result images, forward mapping caused some artifacts in the result, while reverse mapping did not. This is because, when we do forward mapping, different original pixel coordinates may five the same the mapped new pixel coordinates in the result image, which will lead to pixel overlapping and finally produce some artifacts. However, in reverse mapping, we try to use the pixel coordinates in the result image and reverse mapping equation to find the corresponding pixel in the original image.

Figure 3 Forward Mapping (left) and Reverse Mapping (right)

Second, forward mapping may cause some missing pixels in the result image. For example, if we are doing the scaling and we want to enlarge the image. When you use forward mapping, we are using the original pixel coordinates to calculate the result pixel coordinates. Since the mapping function is 1-1, some pixel locations in the result image will not have corresponding pixels in the original image, and this will cause missing pixels in the result image. Here I give 2 example result images of forward mapping and reverse mapping respectively in Figure 5.

Figure 4 Forward Mapping (left) and Reverse Mapping (right)

So, how to do the reverse mapping or how to find the reverse mapping matrix?

Actually, we can solve this problem by finding the inverse matrix of forward mapping matrix. To do forward mapping, we will use several corner pixels locations to calculate rotation angel θ , translation distance tx/ty and scaling factor Sx/Sy. Since all the geometric modifications are linear operation, we can simply cascade them to generate a transformation matrix which can do rotation, scaling and translation at the same time. We can express above equation as below:

$$\text{Transformation Matrix: } A = TSR$$

where T is the translation matrix, S is the scaling matrix and R is the rotation matrix. Now, we have the forward mapping matrix, we can find its inverse matrix easily. Since the inverse matrix may not necessarily exist, so, we calculate the Pseudo Inverse Matrix here. I denote the pseudo inverse matrix of the forward transformation matrix as P, where $P=A^{+}$.

With A, we can do forward mapping: (x, y) = A(u, v); and with P, we can do reverse mapping as: (u, v) = P(x, y). Now we can find the corresponding pixel coordinates in original image and put this (u, v) into the correct coordinates in new image.

After above processing, the lighthouse1-3 are moved into correct holes in lighthouse. In the reverse mapping process, we may come up with some coordinates that are not integers. So, how to find the coordinates that are not integers in the original image? The answer is interpolation. As you can see in Figure 5, the (x, y) is the pixel coordinate in result image and the (x', y') is the corresponding pixel coordinate in the original image, which can be not integers. What we do for interpolation is to find the nearest pixel to (x', y'). For example, if (x', y') is (0.5, 0.5), then we that the average of (0, 0), (0, 1), (1, 0) and (1, 1) and assign it to (x', y'), i.e. *I(x', y') = (I(0, 0) + I(0, 1) + I(1, 0) + I(1, 1))/4.* The coordinates here represent the pixel value at its location. Another example, when (x', y') = (sqrt(2), 0), then we can interpolate I(x', y') as: $I(x', y') = \frac{2 - \text{sqrt}(2)}{2} I(2, 0) + \frac{\text{sqrt}(2)}{2} I(1, 0)$.

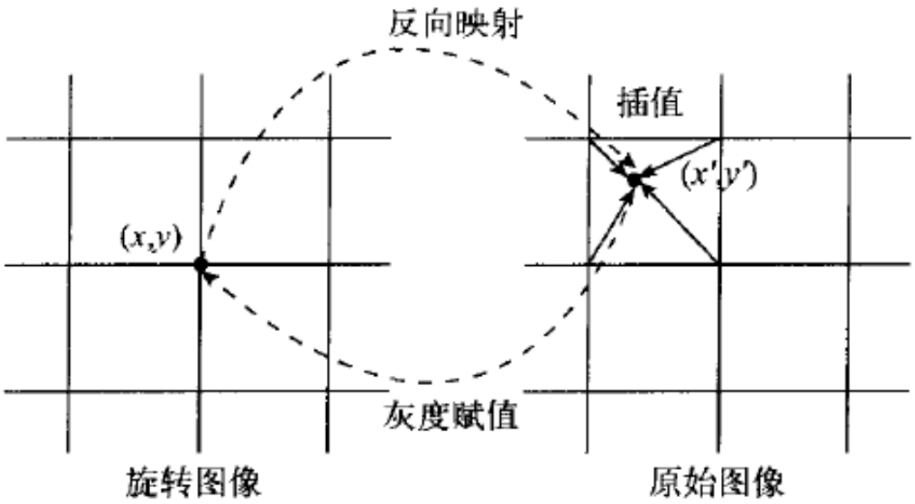


Figure 5 Reverse Mapping and Interpolation[2]

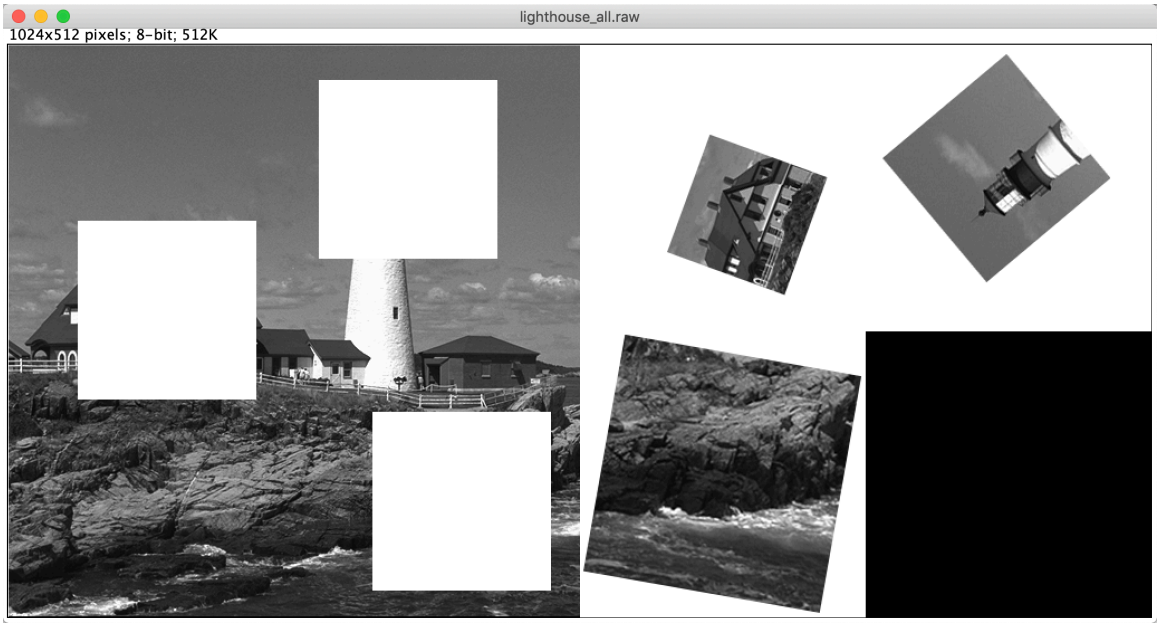
Above are all the steps to do geometric modification to images. In next part, I will show my result images.

(3) Result

a. Forward mapping of lighthouse1. (Just give as an example, this is not the final result image). As you can see, the quality of forward mapping result image is not good.

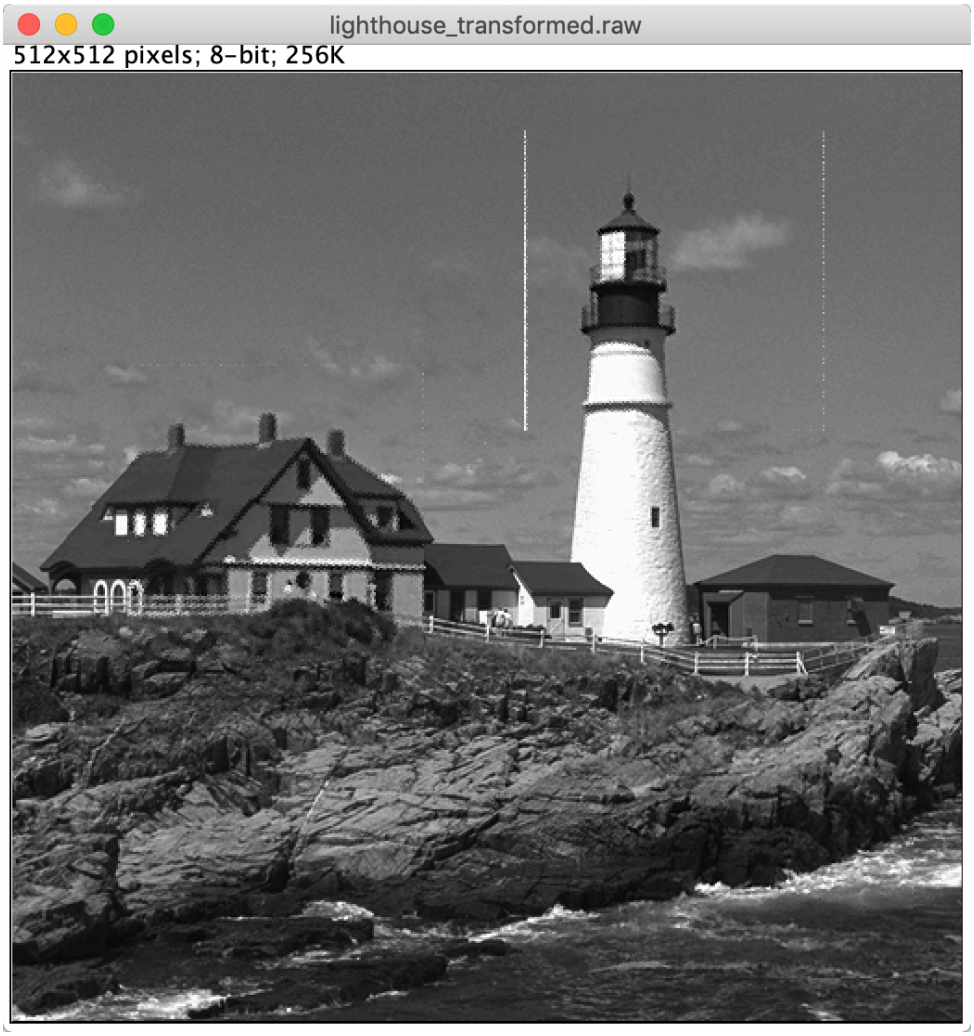
Figure 6 Forward Mapping of lighthosue1

b. Lighthouse and lighthouse1-3



c. Final result image

The final result image is very good.



(4) Discussion

There is no required discussion task for this prob.1.(a).

(b) Spatial Warping (20%)

(1) Motivation

Spatial warping is also a kind of geometric modification, but it's not linear. Spatial warping can generate fantastic images. We can modify the input image into whatever shape we want it to be. This technique is widely required and used in modern camera and vedio applications.

(2) Approach

Since spatial warping is not linear transformation, we need to add 2nd order terms into the transformation matrix. Following equation expresses spatial warping:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} a0 & a1 & a2 & a3 & a4 & a5 \\ b0 & b1 & b2 & b3 & b4 & b5 \end{bmatrix} \begin{bmatrix} 1 \\ x \\ y \\ x*x \\ x*y \\ y*y \end{bmatrix}$$

where (u, v) is the coordinate in the original image and (x, y) is the coordinate in the result image.

To do spatial warping, we need to find the coefficients matrix [a, b]. To solve this matrix we can use the following equation:

$$\begin{bmatrix} a0 & a1 & a2 & a3 & a4 & a5 \\ b0 & b1 & b2 & b3 & b4 & b5 \end{bmatrix} = \begin{bmatrix} u0 & u1 & u2 & u3 & u4 & u5 \\ v0 & v1 & v2 & v3 & v4 & v5 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ x0 & x1 & x2 & x3 & x4 & x5 \\ y0 & y1 & y2 & y3 & y4 & y5 \\ x_0^2 & x_1^2 & x_2^2 & x_3^2 & x_4^2 & x_5^2 \\ x_0y_0 & x_1y_1 & x_2y_2 & x_3y_3 & x_4y_4 & x_5y_5 \\ y_0^2 & y_1^2 & y_2^2 & y_3^2 & y_4^2 & y_5^2 \end{bmatrix}^{-1}$$

As you can see in the equation, we need to use 6 points to solve it. Since an image is a rectangle, we can segment it into 4 parts, each is a triangle and use the 6 points one the triangle to solve the equation.

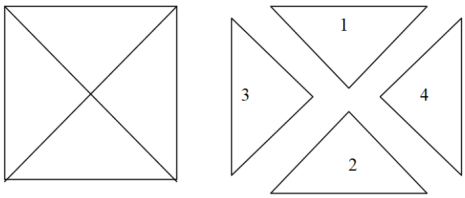


Figure 7 Rectangle Segmentation

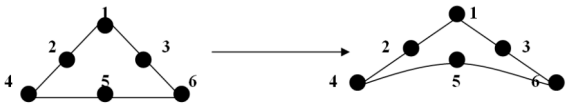
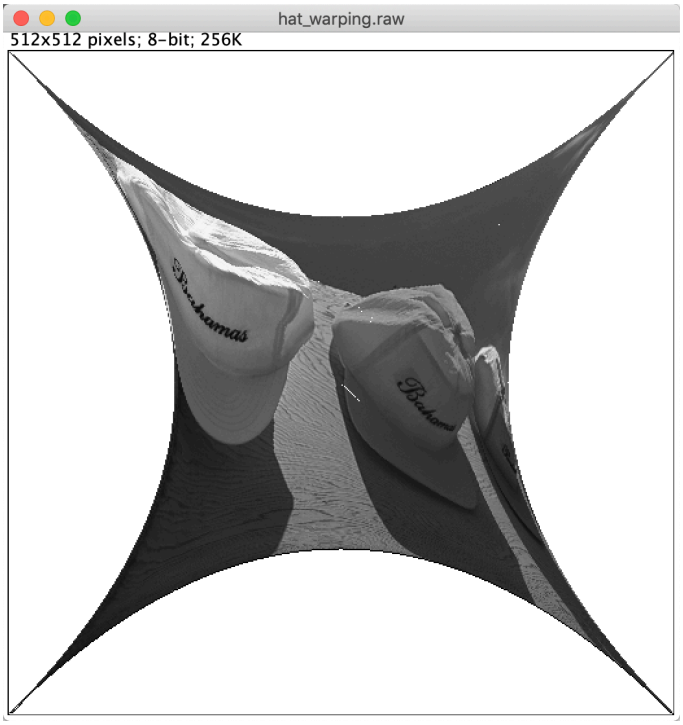


Figure 8 The 6 Points in Triangle

So, totally we will have 4 coefficients matrices [a, b] and we need to implement them to their corresponding triangle respectively.

(3) Result

Result image of spatial warping.



(4) Discussion

(5) There is no required discussion task for this prob.1.(c).

(c) Lens Distortion (10%)

(1) Motivation

Radical distortion often happens when an image is captured on a non-flat camera's focal plane. This lens distortion brings geometric distortion and reduces the image quality. A lot of smart phones companies and camera companies work on this project and want to remove this lens distortion in order to provide better photographing quality.

(2) Approach

(Note: no built-in function used for artifacts removal)

In this specific task, the relationship between the actual image and its distortion is as below:

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

where (x, y) are the undistorted pixel locations and k_1, k_2 and k_3 are radical distortion coefficients of the lens and $r^2 = x^2 + y^2$.

To recover the distorted image, we need to find function that maps undistorted image to distorted image and then find its reverse function. The reverse function satisfies:

$$\begin{cases} x = f(x_d, y_d) \\ y = g(x_d, y_d) \end{cases}$$

However, this is no such inverse function. Therefore, we could only approximate this

inverse function using linear regression. We can find all $\begin{bmatrix} x \\ y \\ x_d \end{bmatrix}$ and $\begin{bmatrix} x \\ y \\ y_d \end{bmatrix}$, and then draw a 3D plots to visualize it. Below is an example of the 3D plot. Since we want to approximate the inverse function, we can do linear regression to find a line that approximate to the inverse function most based on MSE criterion.

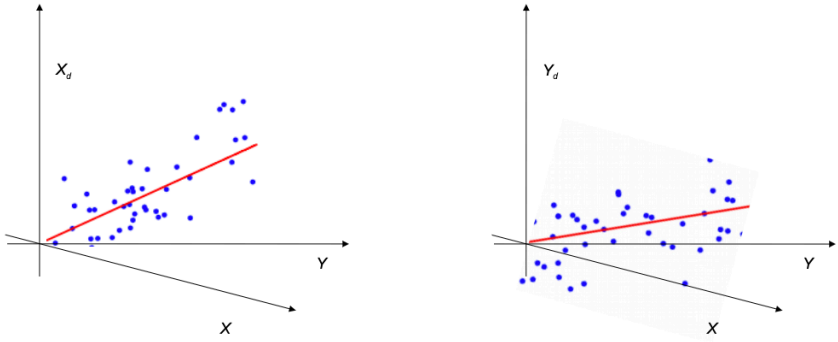


Figure 9 3D Plots and Linear Regression

Using numpy built-in function, we can find this approximated equation that satisfies above-mentioned requirements. Since it's a linear function, we can express it as the following pattern:

$$\begin{cases} x_d = \alpha \begin{bmatrix} X \\ Y \end{bmatrix} + e1 \\ y_d = \beta \begin{bmatrix} X \\ Y \end{bmatrix} + e2 \end{cases}$$

Where α and β are coefficients matrices.

Based on this approximated inverse function, we can then map the pixel locations in the distorted image to correct location in the result image. Finally, the lens distortion is removed.

(3) Result

Figure 10 Classroom Correction Result Image

(4) Discussion

In this problem, the distortion removal quality of the result image depends on the regression result. If we use the simplest linear regression to approximate the inverse function, the result is not the best, because the linear regression result function cannot express the inverse function very well. When we try to do the regression using higher order function, we can get a relatively better result compared to linear regression.

Problem 2: Morphological Processing (50%)

(a) Basic Morphological Process Implementation (20%)

(1) Motivation

When we want to extract some important information from binary images (black and white), there will usually be some redundant pixels. Also, some times, we just focus on the morphology of a certain object inside the image, and we want to remove those pixels that does not contribute to the morphology of the object. In order to solves above problems, here comes the morphological filters. Shrinking, thinning and skeletonizing are 3 fundamental morphological operations. In shrinking, objects shrink uniformly to a single pixel; in thinning, objects will erode to a line; and in skeletonizing, objects result in a stick figure representation.

Morphological operations are very useful, they can help us do some pre-processing so that we can implement complex operations afterwards. For example, when we need to count the number of objects in an image, we can first apply shrinking operation and then count the final pixel number to get the object number.

Now, we I will give the definition of Shrinking, Thinning, and Skeletonizing.

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.

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.

Skeletonizing:

Erase black pixels that are not part of the skeleton of an object. Only keep those pixels that can form a structure of an object. Skeletonizing will bridge those unconnected pixels in thinning if they satisfy certain condition.

(2) Approach

For morphological filtering, we need to define binary image connectivity first. Generally, we define the neighborhood pixel pattern as follow:

$$\begin{matrix} X3 & X2 & X1 \\ X4 & X & X0 \\ X5 & X6 & X7 \end{matrix}$$

For X0, X2, X4, X6, we define them as the 4-connectivity with regard to X and for X1, X3, X5, X7, we define them as 8-connectivity with regard to X. When we have the connectivity definition, we can calculate the pixel bound number for X using the following equation:

$$B = 2 * (X0 + X2 + X4 + X6) + 1 * (X1 + X3 + X5 + X7)$$

To do morphological operation, we can implement a 2-stage filter to determine the center pixel. The first stage Hit-or-Miss filter, it will select those candidate pixels to be erase; and the second stage Hit-or-Miss filter will finally determine whether to remove or keep the center pixel.

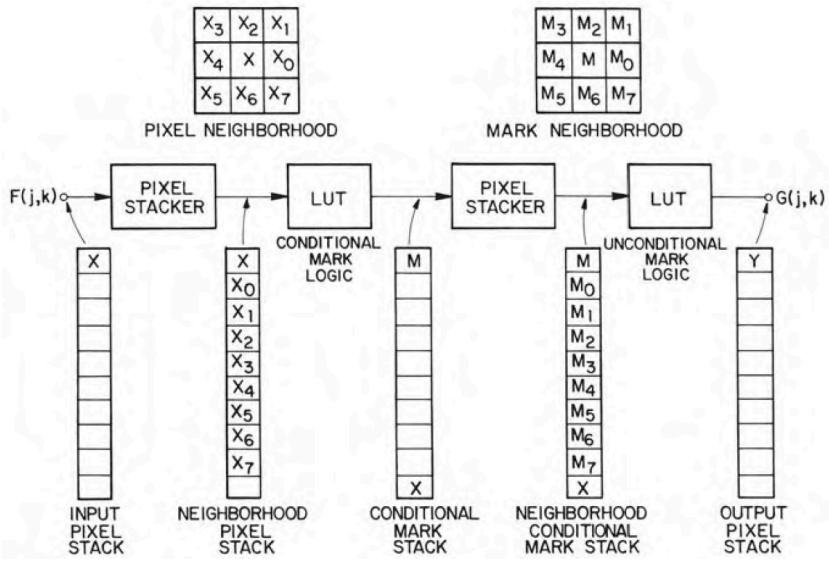


Figure 11 Flowchart for SKT Operation

In the flowchart, the F(j, k) is current center pixel and G(j, k) is the final output center pixel. The LUT is the look-up table, which has all the possible pixel patterns. Here, the conditional LUT and unconditional LUT for Shrinking, Thinning and Skeletonizing are different.

In my processing algorithm, I first stack neighborhood pixels for each center pixel inside the binary image and then send them to conditional LUT. Then store the result M in an array. To decide P, the algorithm sends M, M0, M1, M2, M3, M4, M5, M6, M7 into unconditional LUT. As last, combining current center pixel X, M and P, algorithm will return the final output G. All the above steps are 1 single loop, which will erase the outmost pixels. Adding a outmost loop can iterate all the above operation and give a fine final result image.

(3) Result

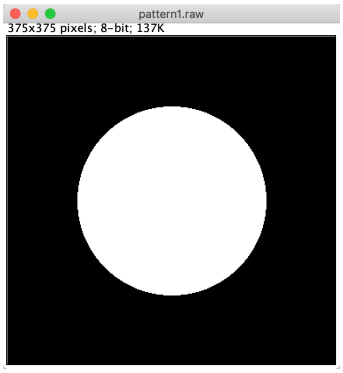
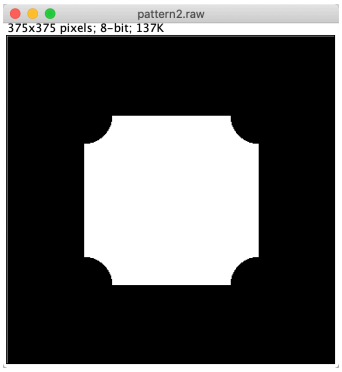
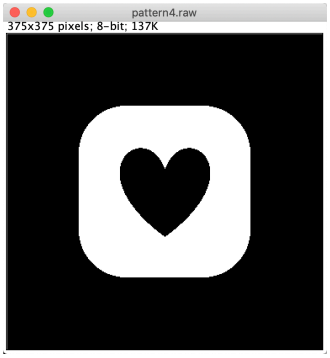
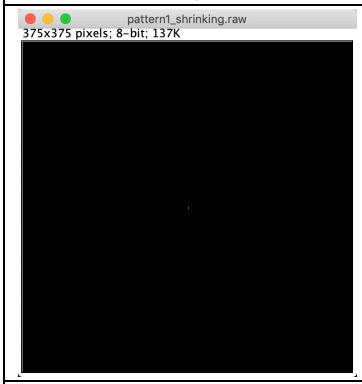
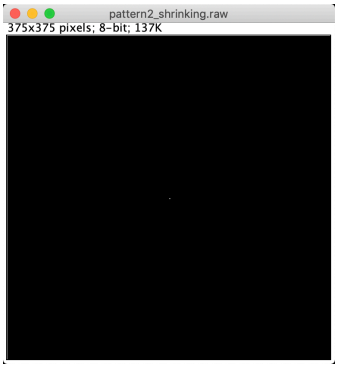
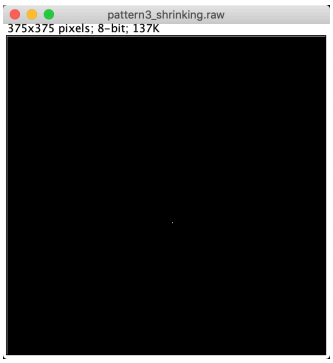
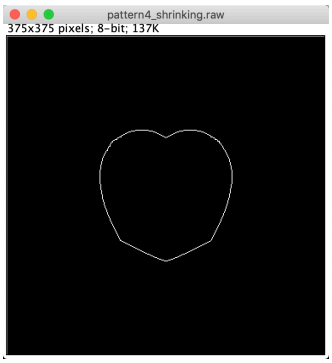
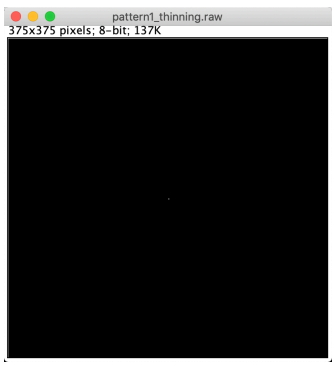
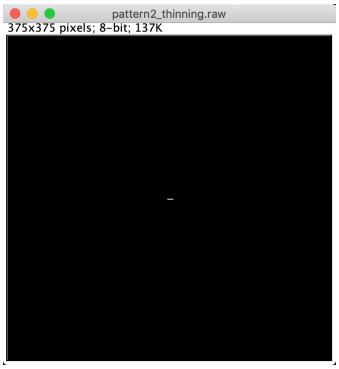
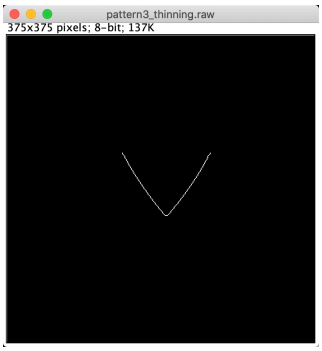
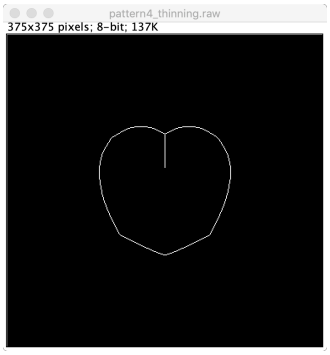
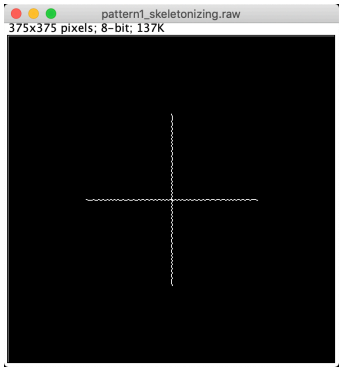
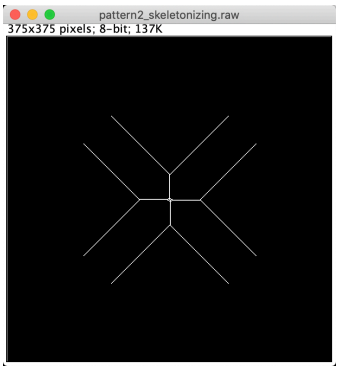
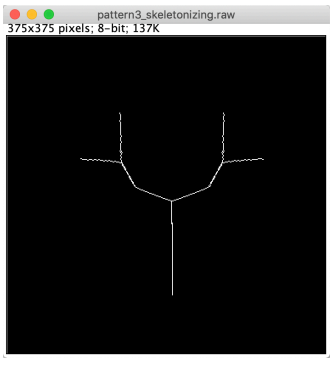
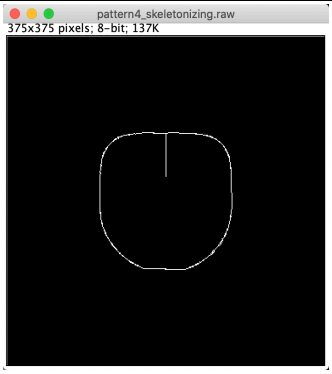
Pattern1 original image	Pattern2 original image	Pattern3 original image	Pattern4 original image
			
Shrinking Result	Shrinking Result	Shrinking Result	Shrinking Result
			
Thinning Result	Thinning Result	Thinning Result	Thinning Result
			
Skeletonizing Result	Skeletonizing Result	Skeletonizing Result	Skeletonizing Result
			

Table 1 Result Images for STK

Note: Shrinking operation for pattern1-3 shrinks to a single dot. Thinning operation for pattern1 also thins to a single dot.

(4) Discussion

Give reasonable explanation:

Explanation:

- Shrinking

  Let's look at the definition of 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[1].

  So, for pattern1, pattern2 and pattern3, the objects have no hole inside it. Therefore, the object in pattern1-3 will shrink to a single dot, just as shown in above result image table. For pattern4, the object has a heart shaped hole. According to the definition, the object will erode to a connected ring lying midway between each hole and its nearest outer boundary, which is a heart-shaped ring. Hence, the shrinking result images are all correct[1].

- Thinning

  Let's look at the definition of 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.

  Similarly, for pattern1-3 they will erode to a minimally connected stroke located equidistant from its nearest outer boundaries. For pattern1, it will still erode to a single pixel at the center of the circle area. For pattern2, since the length and width of the object are slightly different, so it will erode to a stroke at the middle. For pattern3, the object erodes to a stroke located equidistant from its nearest outer boundaries, which is a "V" shaped stroke. For pattern4, which has a hole in the middle, the object erodes to a heart shape ring similar to the shrinking result. But the thinning result has a stroke in the middle of the heart, which is caused by the slight difference in definition.

- Skeletonizing

  For skeletonizing, the objects will keep eroding until it has a thin skeleton. By drawing the skeleton of the objects inside the images, we can check the results. For pattern1 it has a square object in the middle, and the skeleton of a square is a cross in the middle. For pattern2, the object is similar to a square but with 4 corners removed. So, the final result will link those vertex up and converge in the middle with the cross. The skeletonizing result image of pattern2 matches above assumption well. For pattern3, the final result should be the skeleton of a heart-shape. When we link up those "vertex" of the heart object, we will draw a skeleton exactly like the skeleton in skeletonizing result image. For pattern4, there is a heart-shaped hole inside the square. So, the outer skeleton will be kind like a heart, but has less heart-shape. At the top of the heart, there will be a stroke since heart has a "deep valley" in this region. The skeletonizing result image is hence well verified.

(b) Defect Detection and Correction (15%)

(1) Motivation

One of the applications of morphological processing is defect detection. When printing a large binary image, we want to ensure that there is no defects (holes) inside the objects. To detect if there is holes in objects, we can use morphological filter.

(2) Approach

In the deer image, the deer is the object. We want to check if there are holes inside the deer. So, we can first do some pre-processing. Here I convert the black pixels to white and white pixels to black. In this way, the deer is transformed into background and the defects inside the deer become objects. Then, I apply shrinking operation to the image. So, if there are defects inside the deer, i.e. there are objects inside the deer, then shrinking operation will find them. The background will shrink to those defects. If there are defects, my algorithm will draw a square around it so that I can visualize the defects directly.

In the table below, the left image is the result of shrinking the converted image, and you can see there is while dots inside the black region, which are defects inside the deer. The right image is the defects detection result. You can see, I use square to highlight the defects so that we can visualize defects directly.

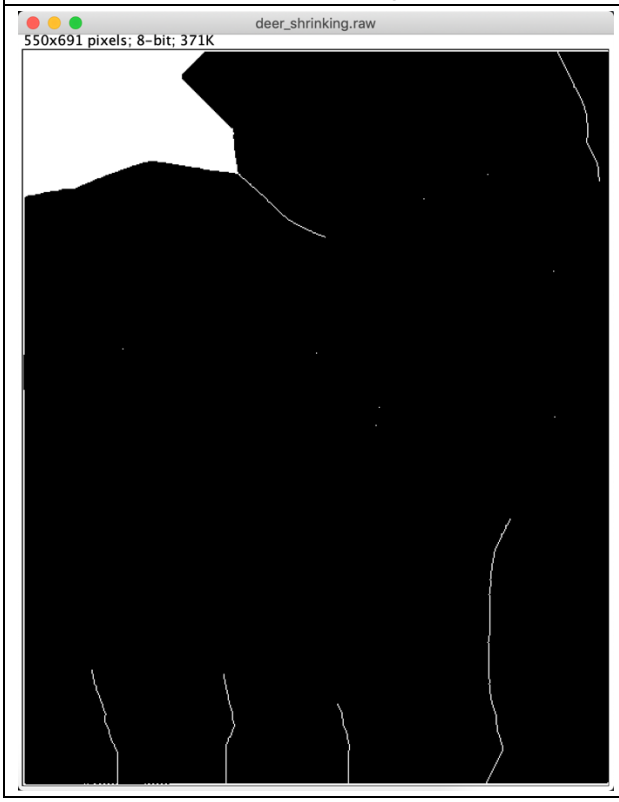
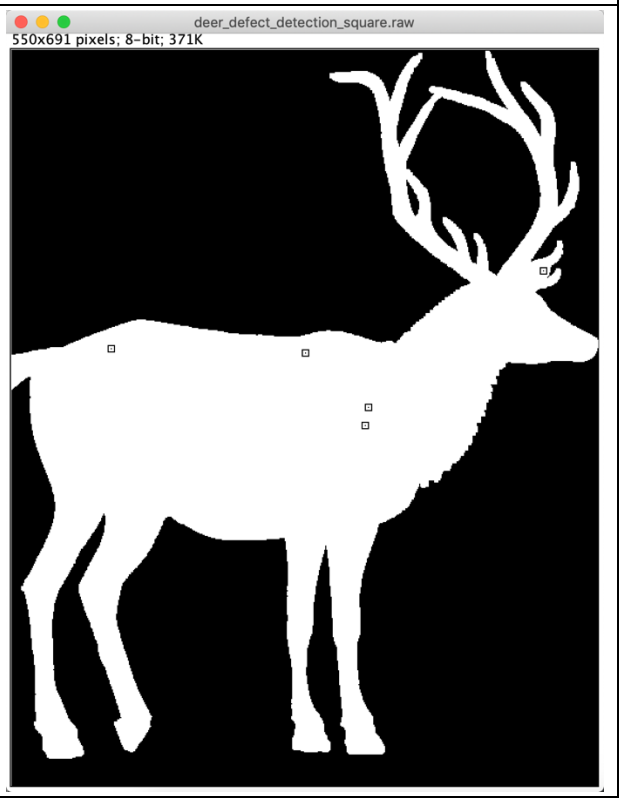
Deer Shrinking	Deer Defect De
	

Table 2 Defects Detection of deer.raw

Next, after we find the defects, what we need to do is to convert the defect pixels into white pixels. Finally, we will get a desired result image without defects inside the objects. The final result image is shown in *Result* part.

(3) Result

Below is the result image of defect correct.

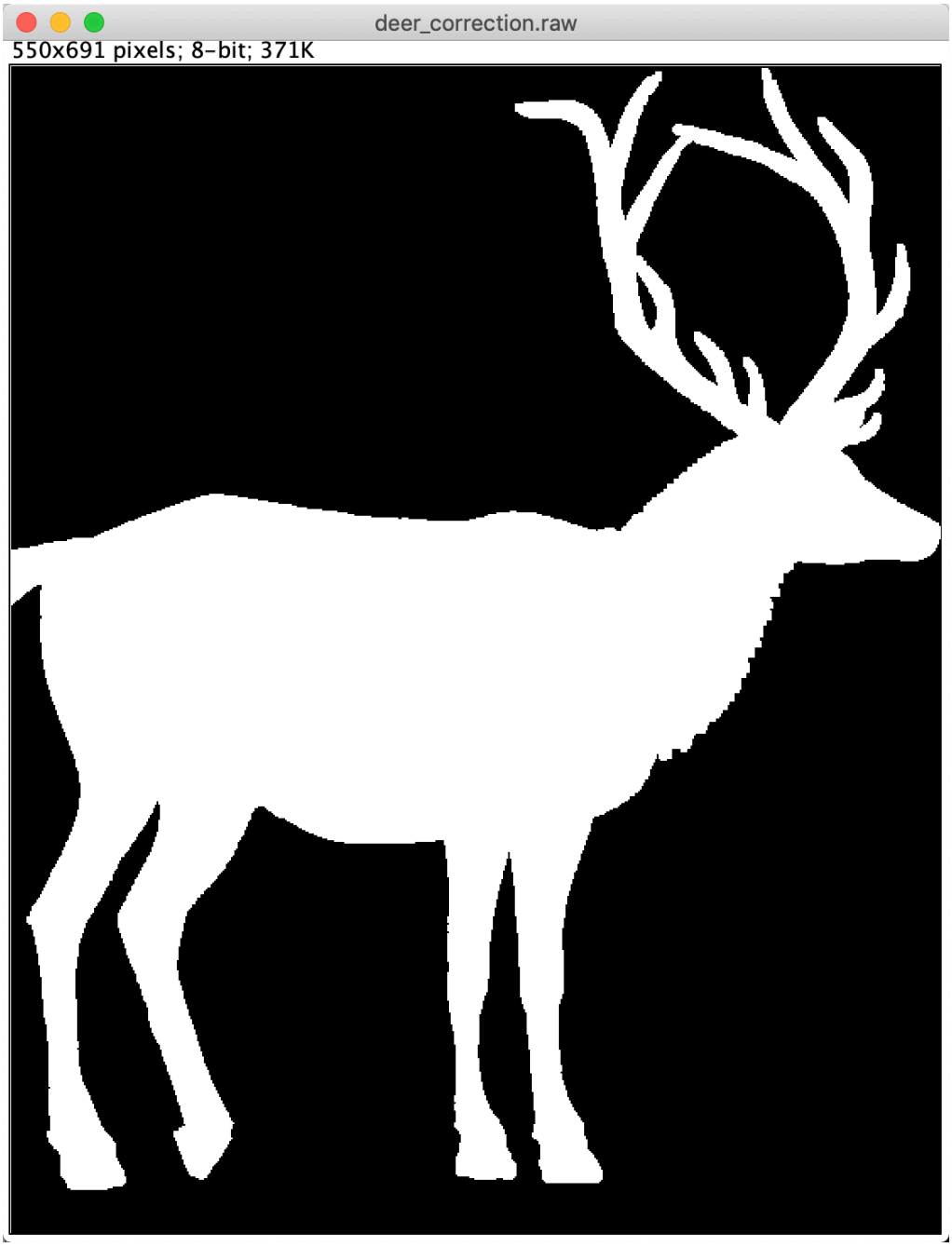


Figure 12 Defect Correction Result Image

(4) Discussion

- Whether deer image is defectless?

No, the deer image have defects.

- If it is not defectless, where are defect regions? Correct them to white dots.

  The defects are inside the deer area and they are black dots. I use squares to center and highlight those defects, which is shown above in *Approach* part. And the final result defectless image is shown in *Result* part.

(c) Object Analysis (15%)

(1) Motivation

Morphological operations are very useful, and they can be applied to count objects number and object areas. For example, we can implement morphological filters to count the number of rice grains in an image and rank those rice grains in the order of grain size.

(2) Approach

(Note: I did not use OpenCV or MATLAB built-in functions)

Since morphological operations are based on binary image, so, we need to first convert the color image to gray image and then convert the gray image to binary image. Here, I use while pixels to represent rice grains, i.e. the objects, and use black pixels to represent background.

For convenience, here I denote the type of each rice with A, B, C, D, E, F, G, H, I, J and K.

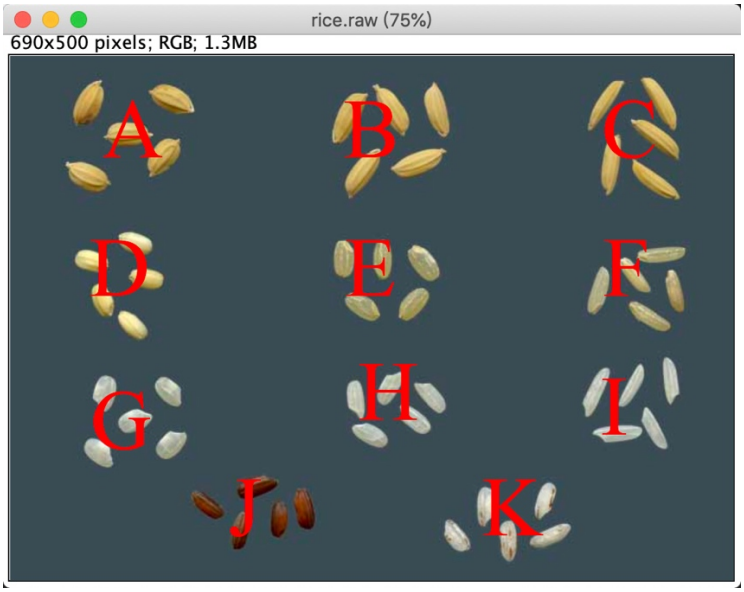


Figure 13 Rice Grain Types

In order to convert gray image into binary image, we need to set thresholds. After I convert the color image to gray image, the most pixel values inside the gray image is 72, which means 72 is the background pixel value. So, I need to assign those pixels with a value of 72 to be 0, i.e. background pixel value. Also, I note that the regions between rice grains are not considered to be background, because there are some noise-like pixel values. So, I need to set a threshold to assign these pixels to be 0. In conclusion, we want to binarize the gray image and we need to separate all the rice grains, so that we can apply morphological filters to the image to count the objects in it.

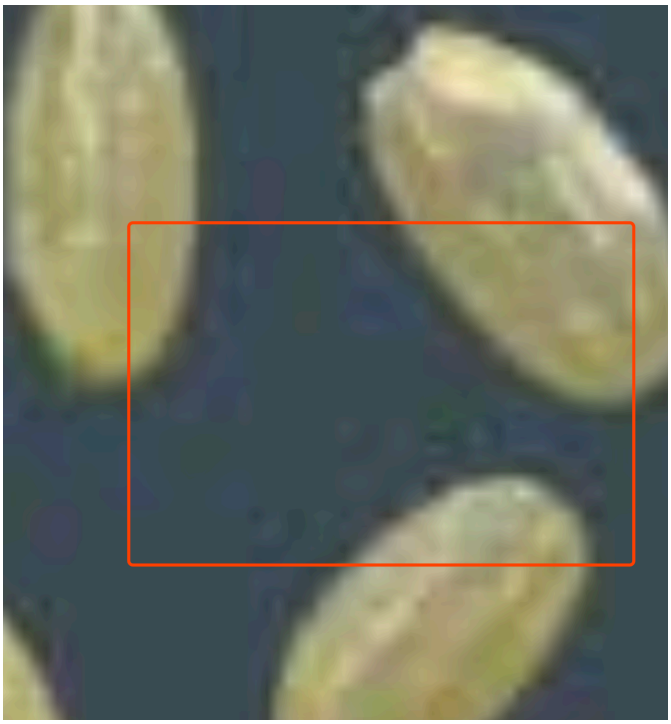


Figure 14 Noisy Region Between Rice Grains

So, how to do this?

The simplest way to realize this is to set thresholds and do thresholding to the gray image. Here, I set the low-threshold to be 60 and the high-threshold to be 80, because the pixel values of rice grains are not between this range and the background pixel value and noisy region pixel values are out of this range. The thresholding equation is as below:

$$G = \begin{cases} 0 & , \quad Low < F < High \\ 255 & , \quad F <= Low \ or \ F >= High \end{cases}$$

where F is the input pixel value and G is the output pixel value.

However, this thresholding method cannot be applied to all A~K regions, because the pixel values of rice grains J is within this thresholding range. As you can see from the below image, type K rice grains have a similar gray level with the background pixels. So, we need to narrow the thresholding range for region K. Here I set the threshold to be (70, 77) and the result is good. This thresholding can remove background pixels and keep the rice grain shape. Finally, I fill up some holes inside type J rice grains and below is the method. For those single black dots and 2-connected and 3-connected black dots surrounded with white dots inside region J, I set them to be 255.

With all above operations, we now have a good binary image.

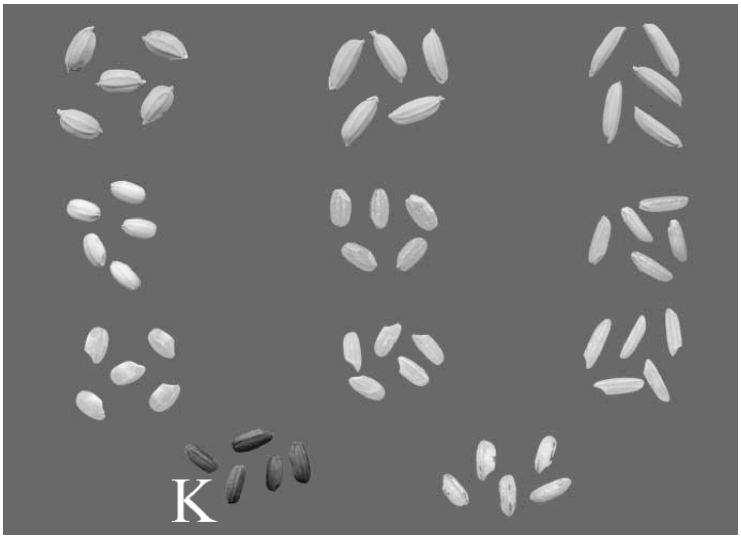


Figure 15 Gray Image of rice.raw

Below is the result of my binary image after thresholding. As you can see, most of the noise are removed and the rice grains are separated one by one.

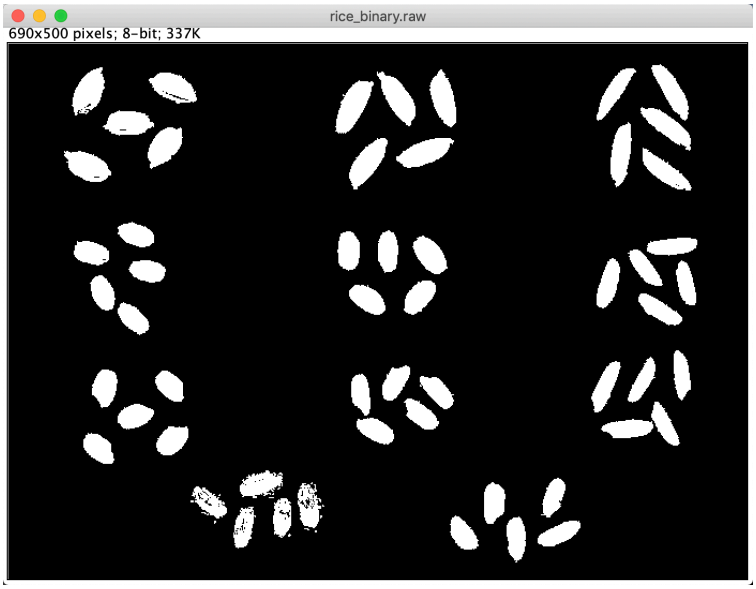


Figure 16 Binary Image of rice.raw

Next, I apply shrinking to the binary image. Since the rice grains (objects) are not connected and they have no hole inside them, so, the shrinking operation return a single white dot for each rice grain. To count the number of rice grains in the image, we just need to count the number of white dots left. And the result is 55.

To rank the rice grains in an order of grain area, we need to count the area of each rice grain and then take the average value of grain area for each type of rice grain.

Figure 17 Image Segmentation

I manually segment the image to 11 parts and each part has a single type of rice grain. Next, the program will count the number of non-background pixels in each part and we can easily see the number is the total area of 5 rice grains. Then, I take the average value of this number and rank them from small to large. And we can get the final result then.

(3) Result

The result returned by the program. The number here denotes the average size of rice grain.

```
p2_c
/Users/wenjun/ee569/hw3/cmake-build-debug/p2_c
Size_A  Size_B  Size_C  Size_D  Size_E  Size_F  Size_G  Size_H  Size_I  Size_J  Size_K
827     899     765     551     641     585     580     552     573     569     566
```

The ranked size of rice grains from small to large.

D (551) < H (552) < K (566) < J (569) < I (573) < G (580) < F (585) < E (641) < C (765) < A (827) < B (899)

(4) Discussion

- Count the total number of rice grains.
  The total number of rice grain is 55.

- Compare the size of rice grains. Rank the grain's size from small to large in terms of type.
  The ranking result is shown above in the *Result* part.

Reference

[1] William K. Pratt. 2007. Digital Image Processing: PIKS Scientific Inside. Wiley-Interscience, New York, NY, USA.

[2] Online: https://blog.csdn.net/lkj345/article/details/50555870