Name: Shreya Kate

USC ID: 2334973997

Email: shreyak@usc.edu

Submission Date: 03/03/2020

HOMEWORK #3

Issued: 02/16/2020 **Due: 03/03/2020**

Problem 1:

(a) Geometric Warping

(1) Square to Disc

Approach:

First, I converted image co-ordinates to cartesian co-ordinates, since it is easier to apply rotation, scaling and translation on cartesian co-ordinates. To convert image to cartesian, I used the following transformation matrix:

```
im_to_cart = [0  1   -0.5
             -1  0   h+0.5
              0  0     1  ]
where h is height of image
(h=512 for Hedwig, raccoon and bb8)

The image co-ordinates are:
im =   [r
        c
        1]
The cartesian coordinates are:
cart = [x
        y
        1]
To find cartesian coordinates:
cart = im_to_cart*im
```

To find the cartesian co-ordinates, the im_to_cart is multiplied with im to get cart. 'r' and 'c' are the row and column of the image co-ordinates and 'x' and 'y' are the cartesian co-ordinates.

I translated and scaled the image to avoid negative values. To do so, I subtracted (512/2)=256 from the x and y co-ordinates and divided by (512/2)=256 to normalize it.
To convert square to disk, all the cartesian co-ordinates must be inside the disk. So, I used the formula: sqrt(x^2+y^2)>1.
(since we have normalized our x and y)

If this is not true, we assign these pixels black values (0) and they form the background of the disc image.

To convert the square cartesian co-ordinates to disc co-ordinates, I used the following equations:

$$x = \frac{1}{2}\sqrt{2 + u^2 - v^2 + 2\sqrt{2}u} - \frac{1}{2}\sqrt{2 + u^2 - v^2 - 2\sqrt{2}u}$$

$$y = \frac{1}{2}\sqrt{2 - u^2 + v^2 + 2\sqrt{2}v} - \frac{1}{2}\sqrt{2 - u^2 + v^2 - 2\sqrt{2}v}$$

Now, I needed to cancel the effect of normalization, so I multiplied the image by (512/2)=256 and added (512/2)=256 to these co-ordinates.

Next, I converted the cartesian co-ordinates back to image co-ordinates by using the following transformation matrix:

```
cart_to_im = [0 -1  h+0.5
              1  0   0.5
              0  0    1  ]
```

These image co-ordinates calculated can be floating numbers so bilinear interpolation is done to calculate the row and column values in image co-ordinates and the output image is obtained.

(2) Disc to Square

To convert the disc back to the square, I used the inverse of the approach mentioned above.

First I converted image to cartesian co-ordinates by using the same transformation matrix as above.

I translated and scaled the image to avoid negative values. To do so, I subtracted (512/2)=256 from the x and y co-ordinates and divided by (512/2)=256 to normalize it.
To convert square to disk, all the cartesian co-ordinates must be inside the disk. So, I used the formula: sqrt(x^2+y^2)>1.
(since we have normalized our x and y)
If they are inside the disc, I converted the disc co-ordinates to square co-ordinates. For converting disc to square, I use the following equations:

$$u = x\sqrt{1 - y^2/2}$$

$$v = y\sqrt{1 - x^2/2}$$

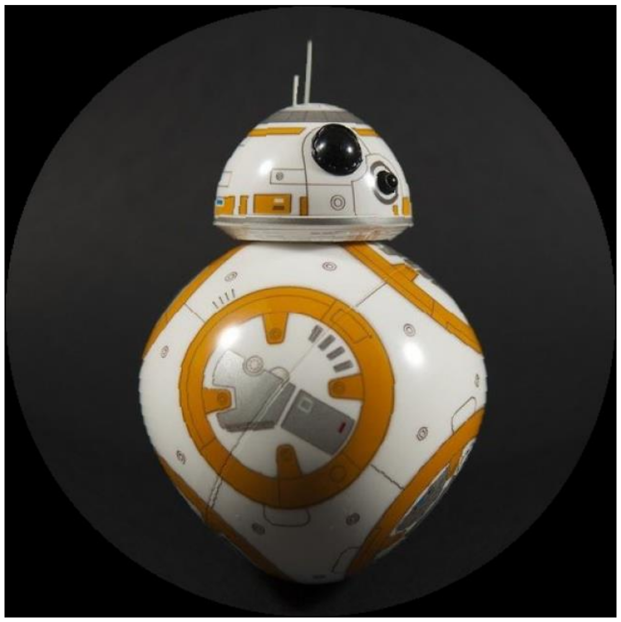
Now, I needed to cancel the effect of normalization, so I multiplied the image by (512/2)=256 and added (512/2)=256 to these co-ordinates.

Next, I converted the cartesian co-ordinates back to image co-ordinates by using the transformation matrix as above.
These image co-ordinates calculated can be floating numbers so bilinear interpolation is done to calculate the row and column values in image co-ordinates and the output image is obtained.

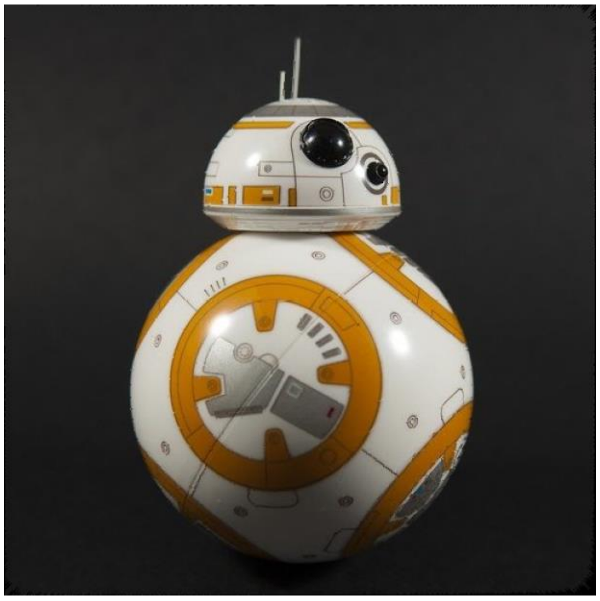
Results:

bb8.raw

Square converted to disc

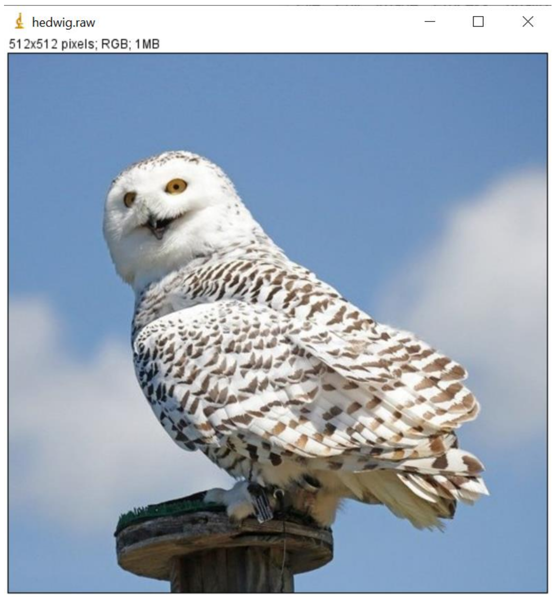


Disc converted back to square

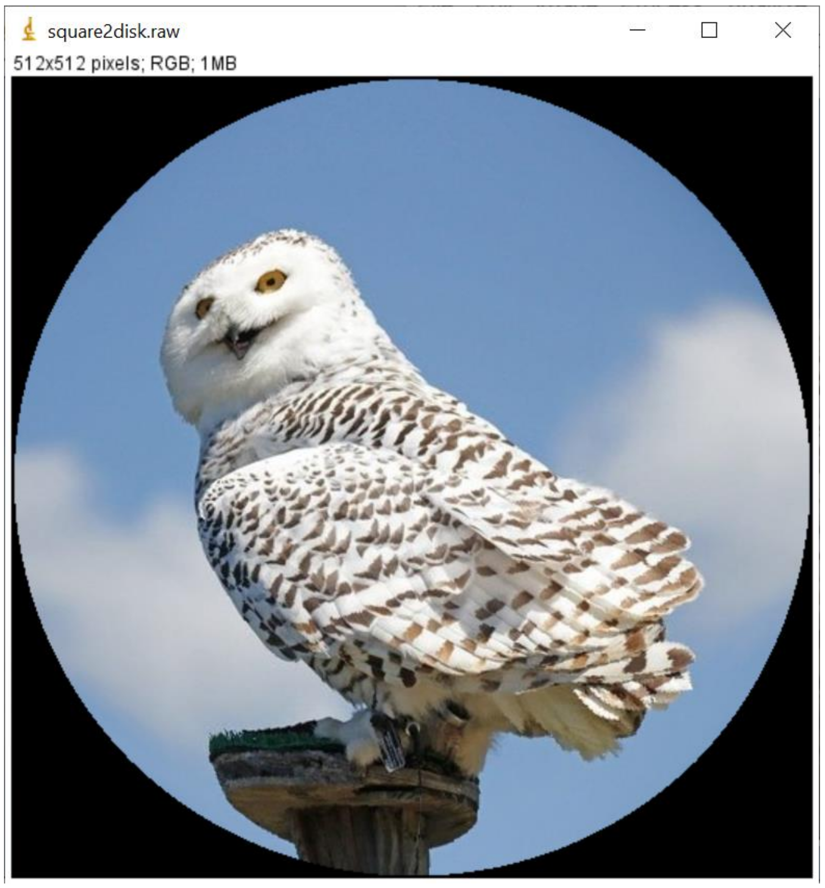


\

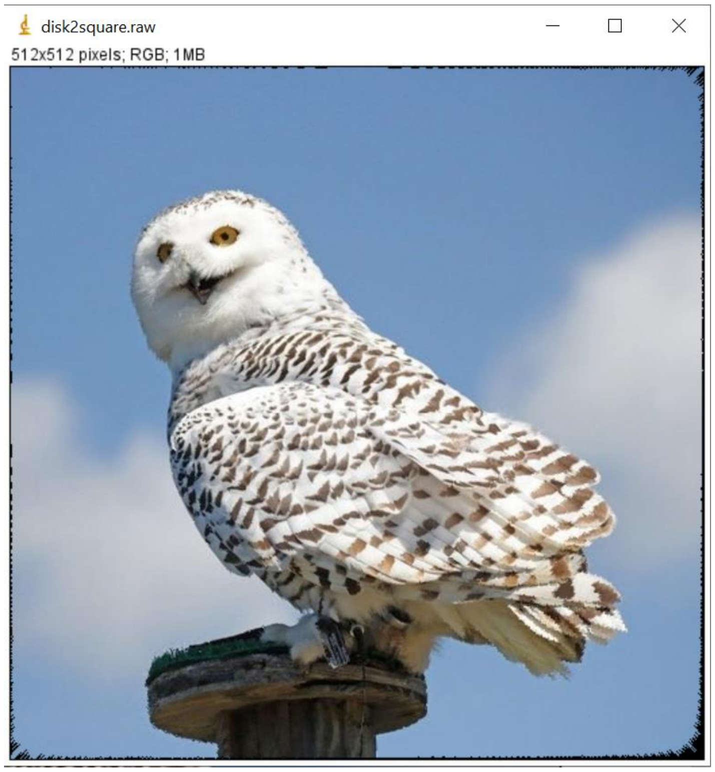
Hedwig.raw



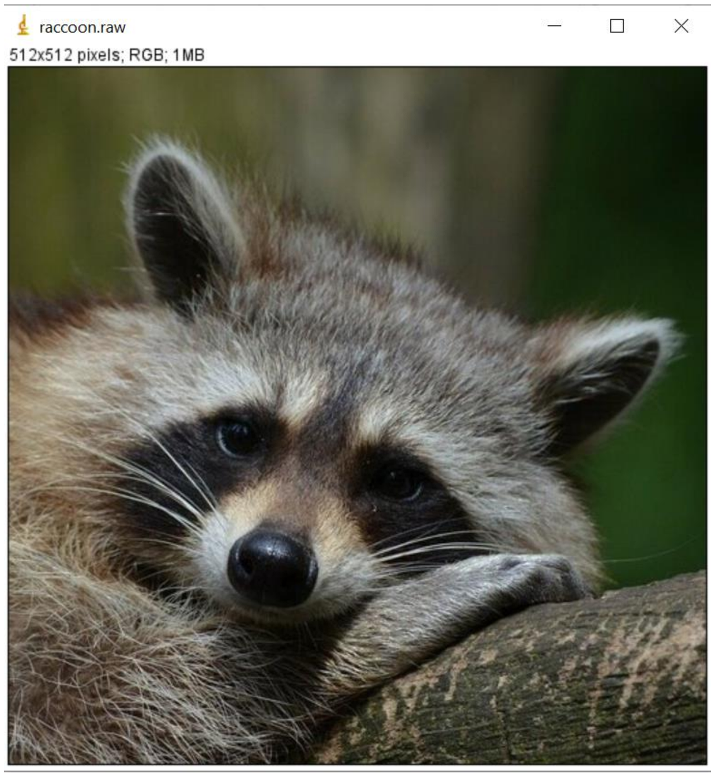
Square converted to disc



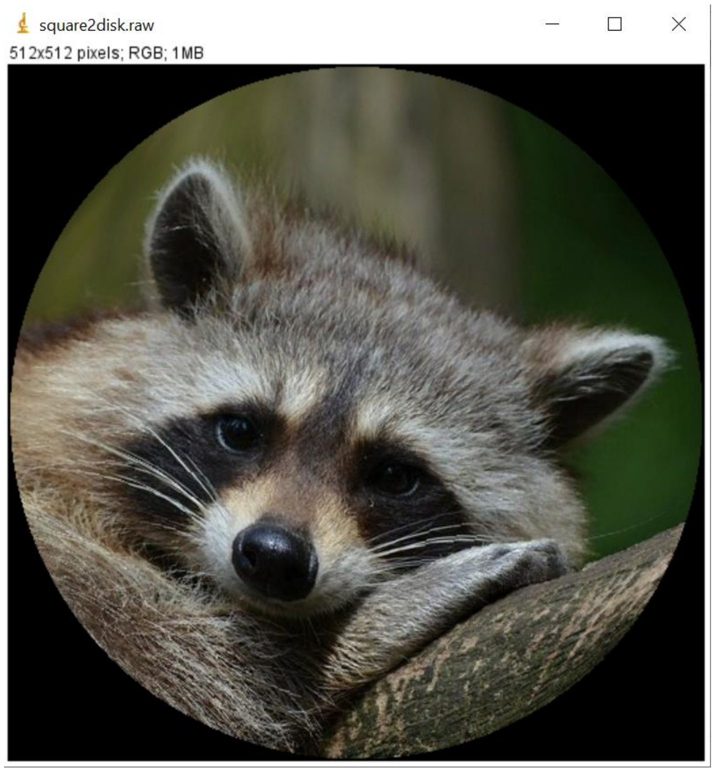
Disc converted back to square



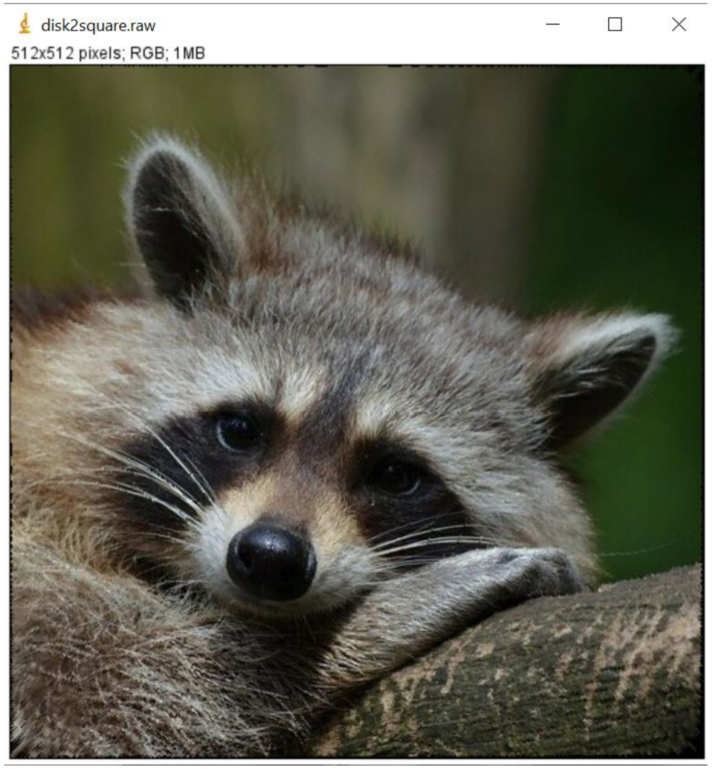
Racoon.raw



Square to disc



Disc converted back to square



(3)

Discussion:

The recovered image and the original image are different because as we can see in the output recovered image, the boundaries have black pixels. This is because when we convert square to disc, then the number of non-black pixels are less as compared to the number of pixels in the original image.

So, when we convert the image from disc to square again, we have to convert fewer number of pixels into more pixels to fit the square. We use bilinear interpolation for this and hence while doing this, the pixels at the boundaries remain black since the pixels in the recovered mage are more than in the disc image.

Problem 1:

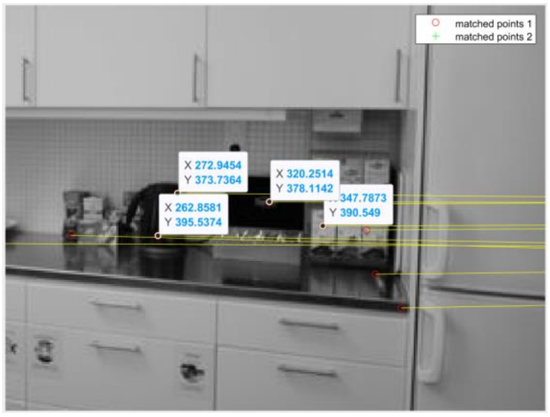
(b) Homographic Transformation and Image Stitching

(1) I used the Matlab detectSURFFeatures to detect the control points between left and middle and then middle and right images.

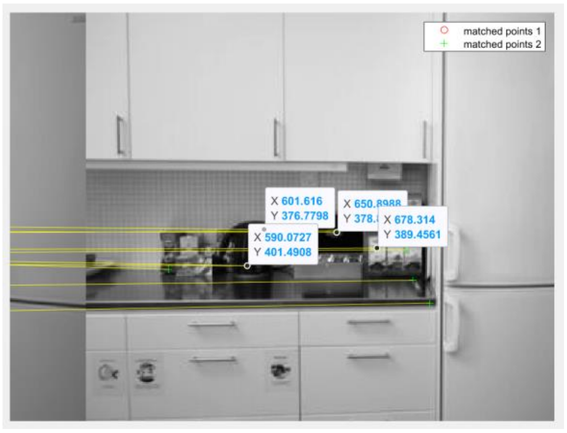
There are 4 control points in each image when 2 images are taken at a time. So, when we are dealing with left and middle images, there are 4 control points in left and the corresponding 4 control points in middle. When we deal with middle and right, there are 4 control points in right and corresponding 4 control points in middle. The images showing these control points are there below:

Selection of control points between left and middle images.

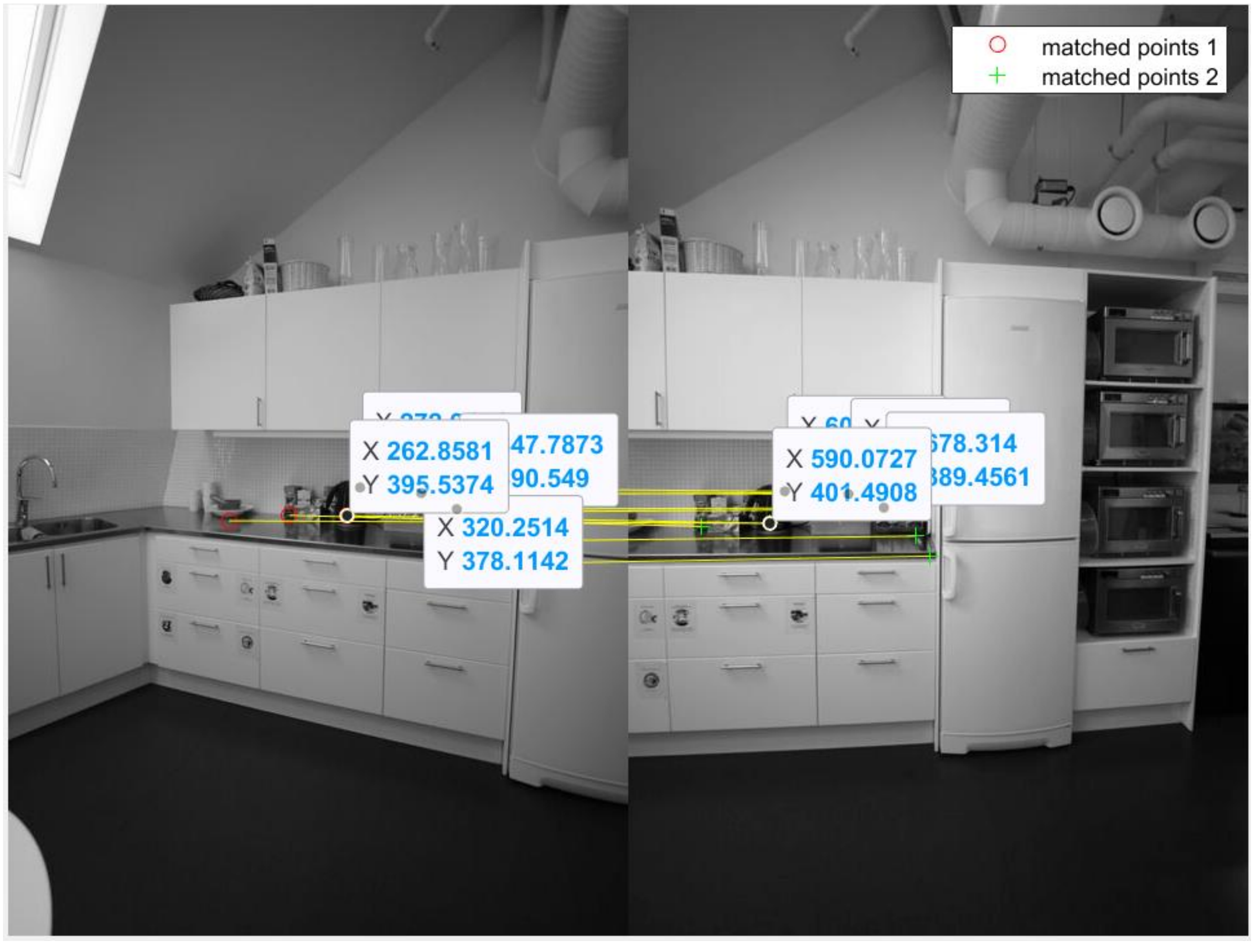
Control points of **left** image



Control points of **middle** image

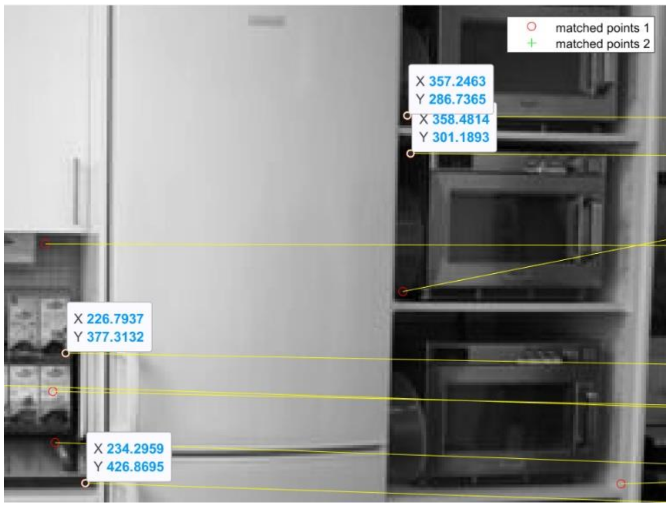


Left and middle image control points:

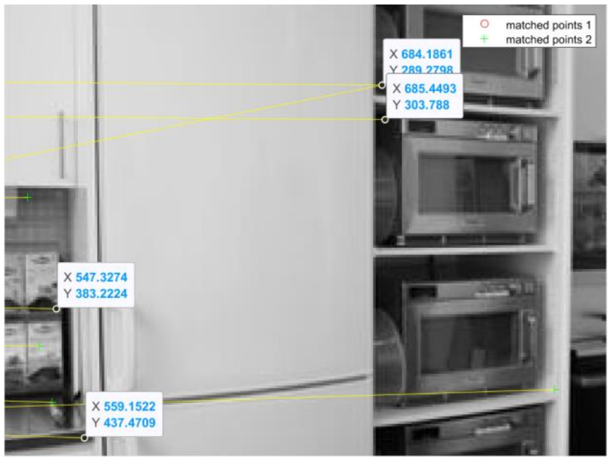


Selection of control points between middle and right images

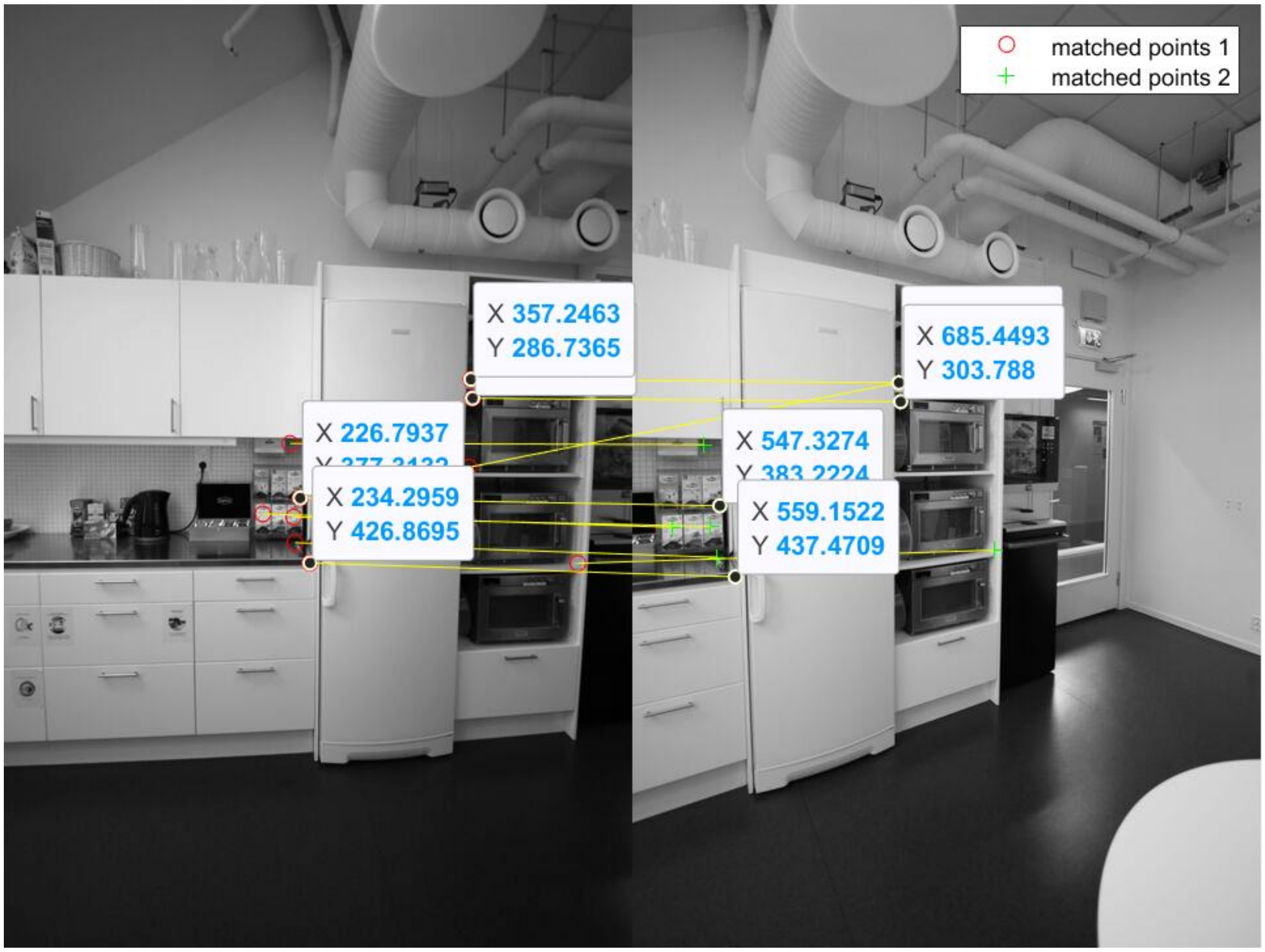
Control points of **middle** image



Control points of **right** image



Middle and right image control points:



(2)

Using the SURF feature in Matlab, I chose the first 4 control points, as the SURF feature detects the best control points for us. If the first 4 points did not work, then I tried to choose the 4 points which gave the best output out of the first 10 control points chosen by SURF. For left and middle, the first 4 control points worked for me. But for the middle and right, I had to try different combinations of points from the first 10 control points to get the best output possible.

After detecting and deciding the control points, I did the homographic transformation by calculating the 8 parameters (h1, h2, h3, h4, h5, h6, h7, h8) using the control points chosen. h9 is kept as 1.The formula for this is:

$$\begin{bmatrix} h1 \\ h2 \\ h3 \\ h4 \\ h5 \\ h6 \\ h7 \\ h8 \end{bmatrix} = \begin{bmatrix} X1 \\ Y1 \\ X2 \\ Y2 \\ X3 \\ Y3 \\ X4 \\ Y4 \end{bmatrix} \times pinv \begin{bmatrix} x1 & y1 & 1 & 0 & 0 & 0 & -x1X1 & -y1Y1 \\ 0 & 0 & 0 & x1 & y1 & 1 & -x1Y1 & -y1Y1 \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ x4 & y4 & 1 & 0 & 0 & 0 & -x4X4 & -y4Y4 \\ 0 & 0 & 0 & x4 & y4 & 1 & -x4Y4 & -y4Y4 \end{bmatrix}$$

Where x1…x4 are the control points of middle image ad X1…X4 are the control points of the left image when we take left and middle images. x1…x4 are the control points of middle image ad X1…X4 are the control points of the right image when we take left and middle images.

After finding H matrix:

H = [h1 h2 h3

h4 h5 h6

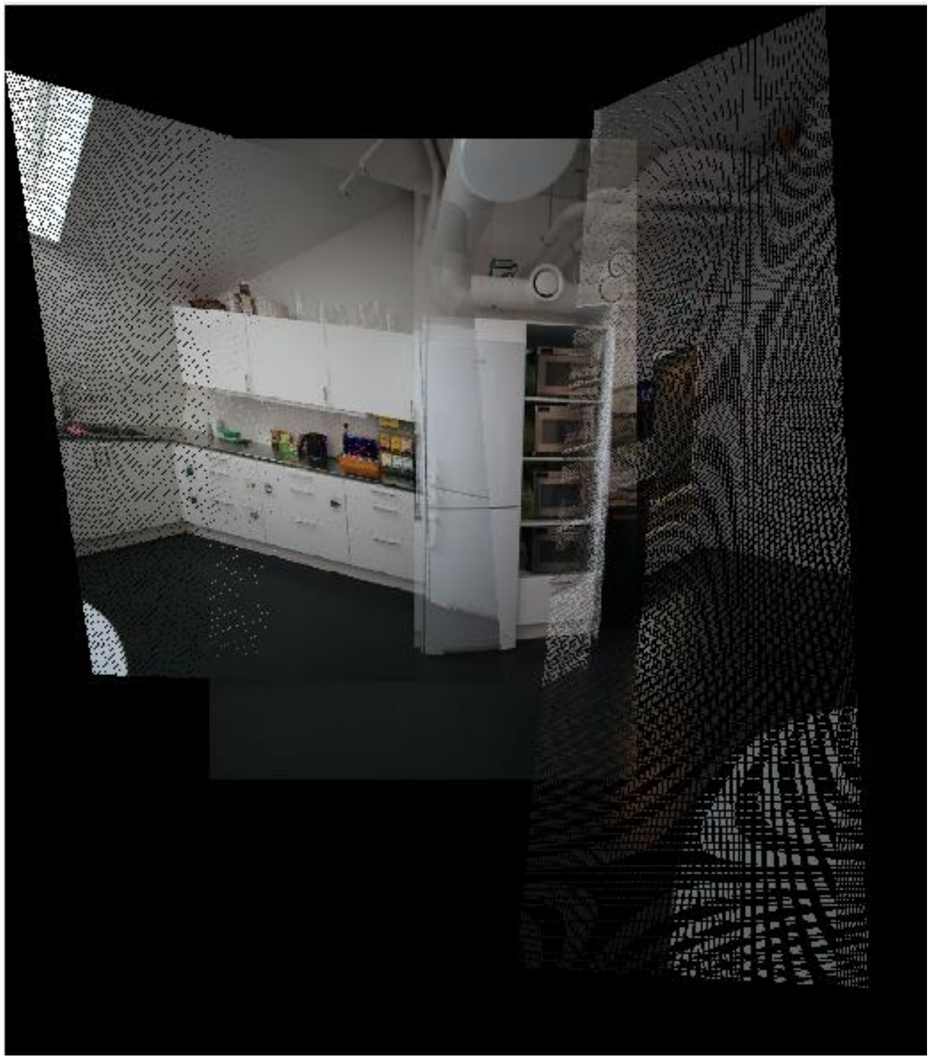
h7 h8 h9]

We find the co-ordinates of the second image and find it's corresponding co-ordinates with respect to the first image.

$$\begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = H \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} ; \; X = \frac{x'}{w} \; and \; Y = \frac{y'}{w}$$

By doing this, we have converted these co-ordinates into cartesian co-ordinates. When the 2 images overlap, we add the pixel values of the two images and divide by 2 to make the image blend together so that image stitching is good. Because the co-ordinates are cartesian, they have negative values, so we need to translate it to positive quadrant so that the image co-ordinates are positive (as required). To do this, I calculated the minimum X and Y and if it was negative, I added that number to the X and Y co-ordinates. This made X and Y positive.

I put all 3 images (left, middle and right) on the empty canvas that I created (size of canvas was 1200 by 1300). After putting all 3 of them, they overlapped, since I calculated the offset and placed them accordingly. In the overlap regions I did bilinear interpolation by adding the pixel values of the two images and dividing them by 2. Hence I obtained the final stitched image.



The stitching of left and middle image turned out to be really good, but the right and middle images' image stitching is not that clear because of the shape of the right image. Having said that, the right image is nicely stitched since the fridge looks almost perfect and it is the result of stitching all 3 images since the fridge is present in all 3 images. The only problem is that the homographic transformation of the right image has increased the size of the image a lot. Maybe selecting more apt control points could have brought a better result, but these were the best control points I could find after trying a lot of different control points.

Problem 2:

(a) Morphological Processing

In morphological processing, the boundaries of the object gets eroded. The area of the object decreases and if the object has holes inside it, then the area of the holes increases. All 3 images in this problem need to be binarized. Since these images have only 0 and 255 intensity values, 0 remains 0 and 255 is made to be 1.

Next, we use the conditional pattern table for shrinking or thinning or skeletonizing. If pixel value is 0, we let it be 0. If the pixel value is 1, then we apply the masks from the pattern table to the pixel. If it is a hit (that is, if it matches) then we set the pixel value to 1. And if it doesn't match, then we set the pixel value to 0. So, a frame is created.

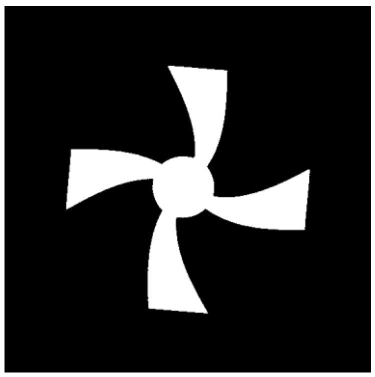
We use this image frame and pass it through the unconditional pattern table for shrinking, thinning or skeletonizing. If it is a marked pixel and it is a hit with the unconditional mask, then we assign it the original pixel value the image had. If is it a miss then we assign it a 0.

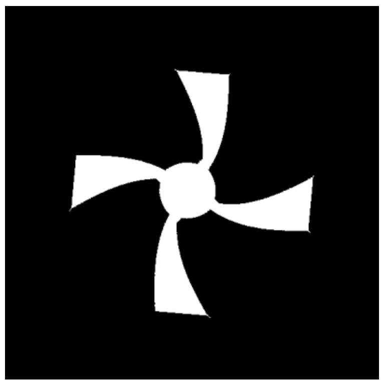
We continue this process till no modifications are there. Then we stop and display the output.

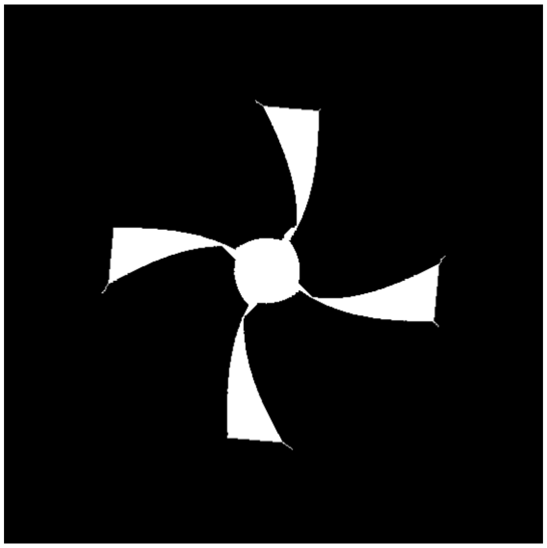
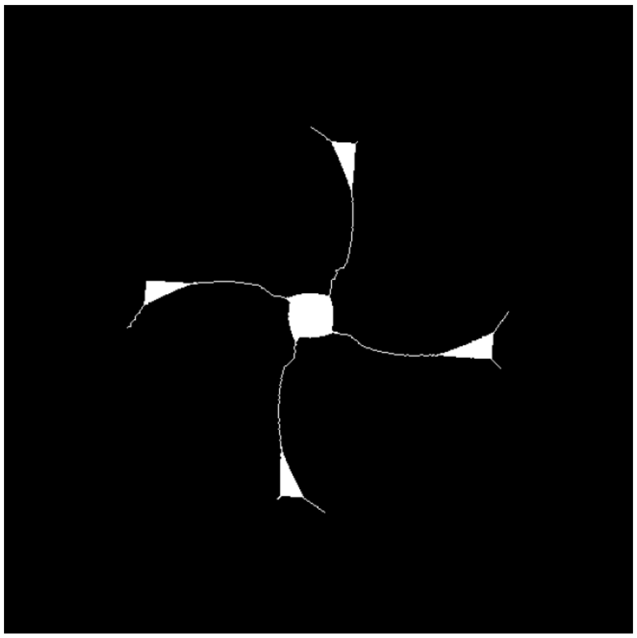
The conditional tables of all 3 are different. The unconditional table for shrinking and thinning is the same and the unconditional table for skeletonizing is different.

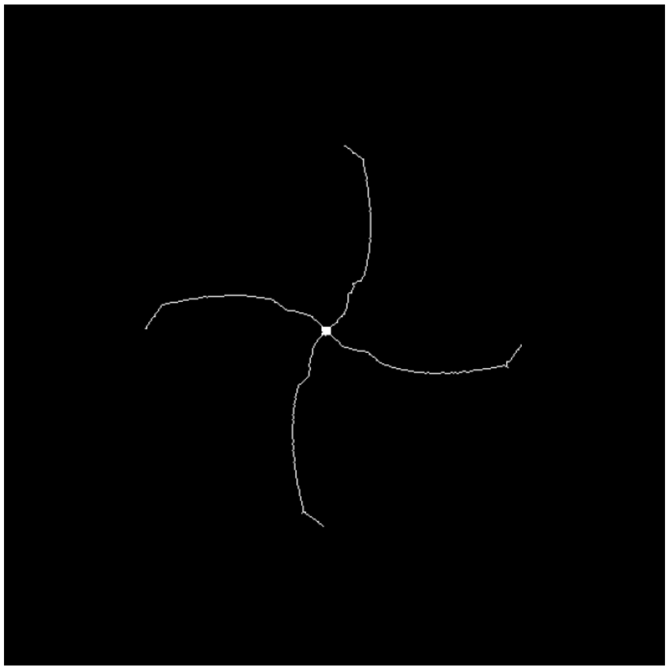
1. Shrinking

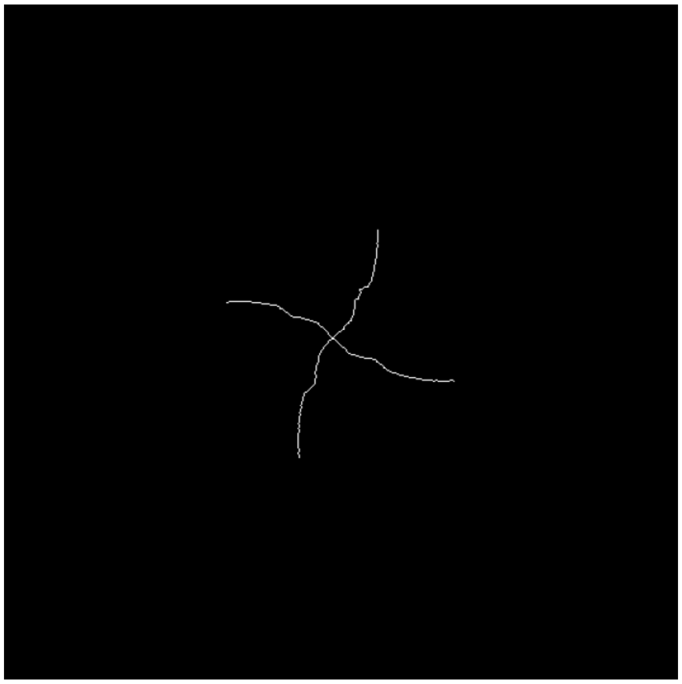
Fan.raw



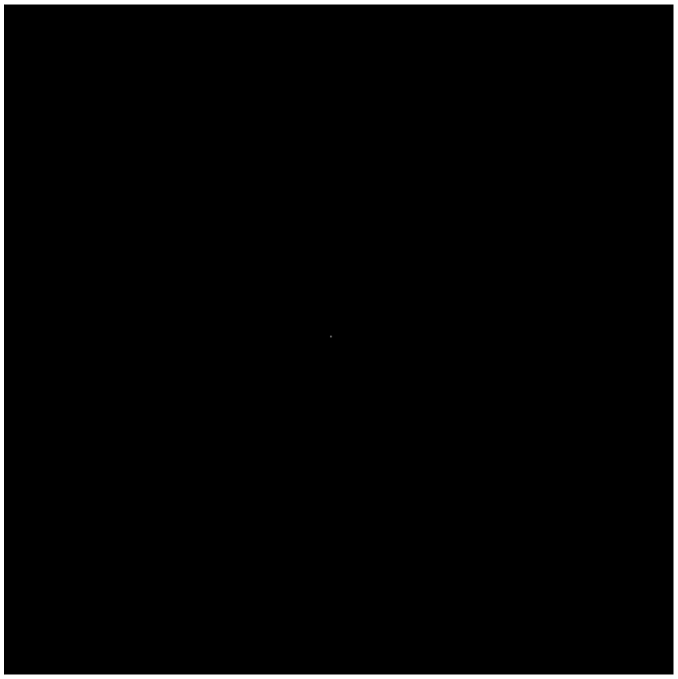




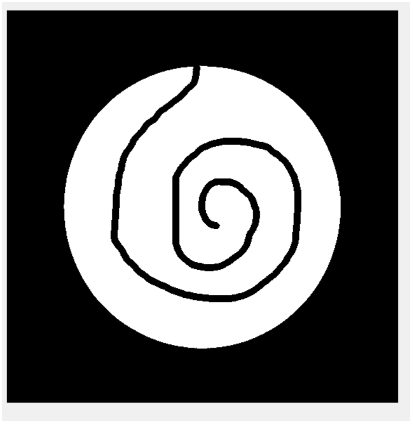


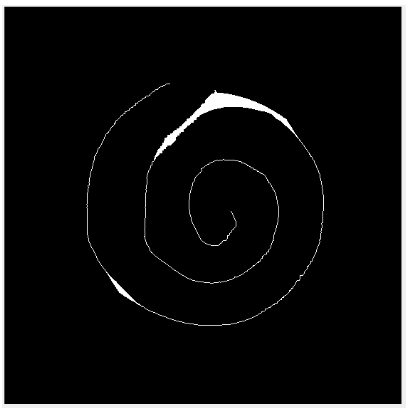
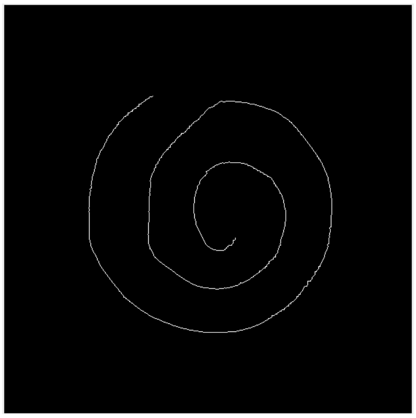
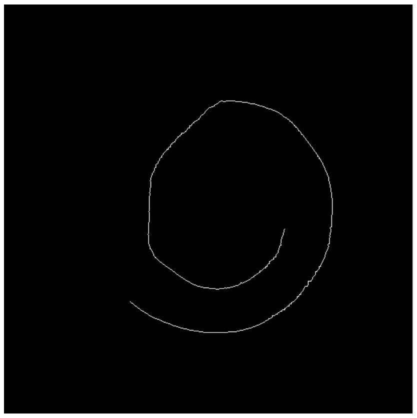
Final shrinking output for fan.raw



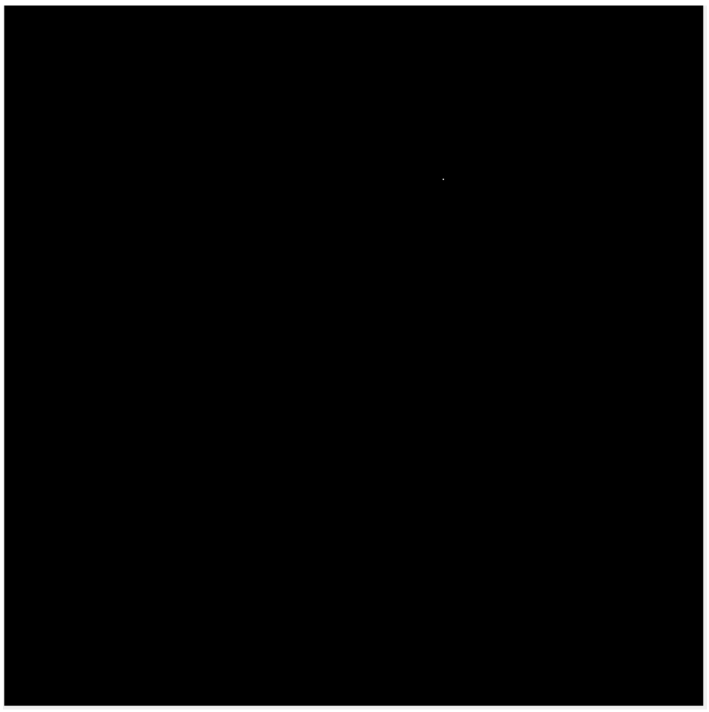
For fan.raw, we can see that the blades of the fan become thinner and thinner and then become lines. After they become lines, the start reducing in length from all 4 corners of the 2 lines. They ultimately shrink to a single point after which no further modifications are made. Some of the outputs in fan.raw look like they are discontinuous lines, but that is because I copied the imshow() figures from Matlab and their image quality might be poor.

Maze.raw



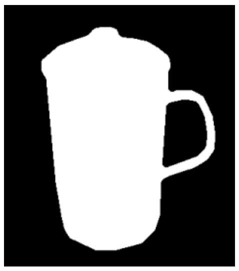
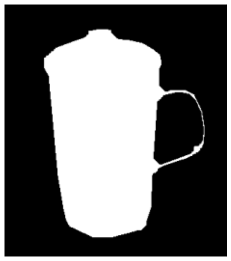
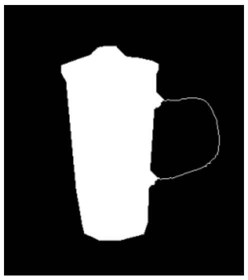
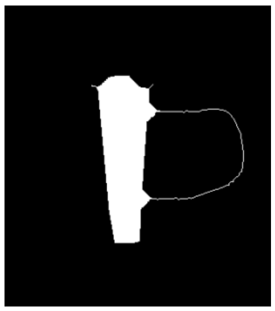
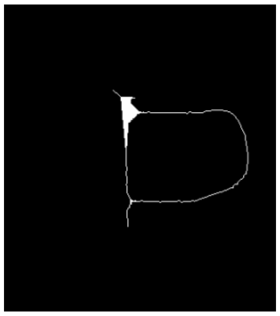
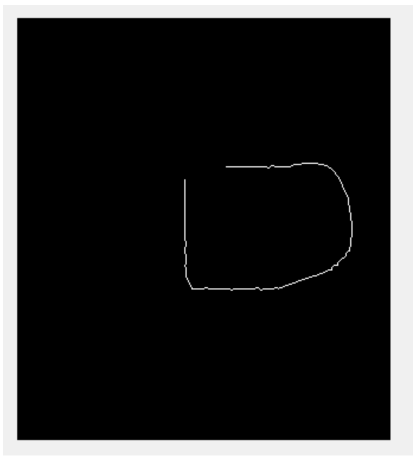
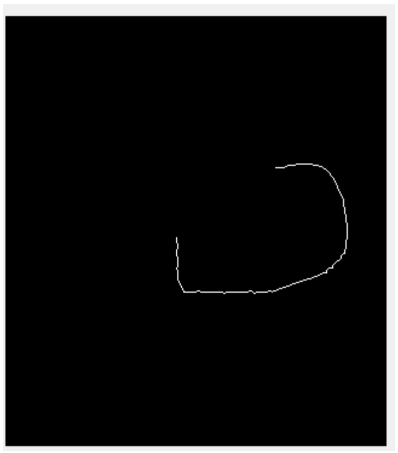
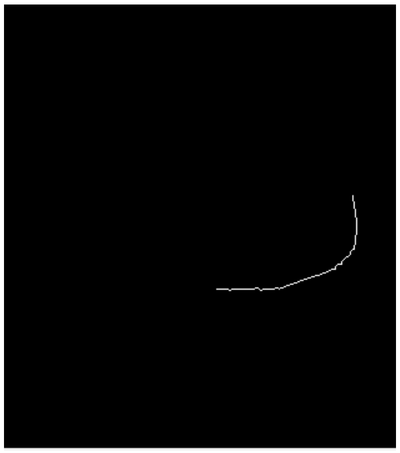




Final output of maze.raw

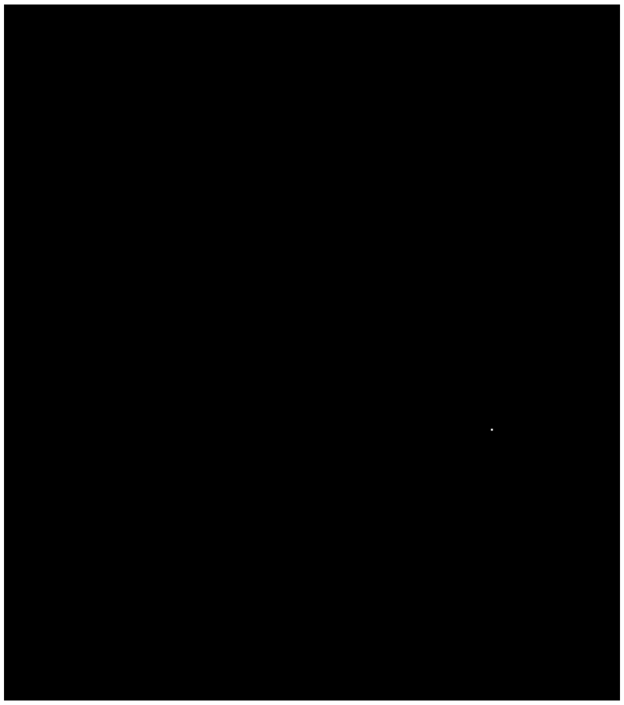


In maze.raw, the width of the maze goes on decreasing and the black spaces in between go on increasing. They ultimately go on to become a windy line which goes on becoming shorter from both ends and ultimately shrinks to a single point which is at the centre of this windy line.

Cup.raw

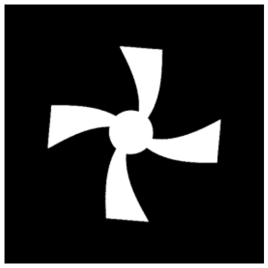
Final output for cup.raw

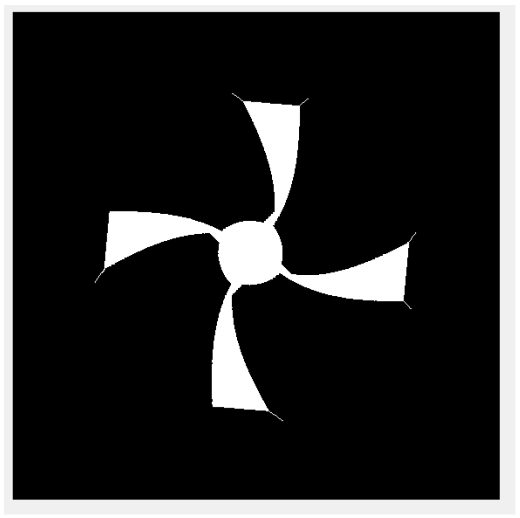


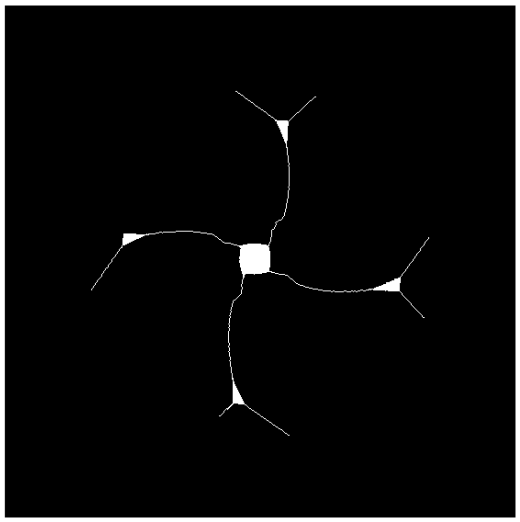
The cup.raw image goes on decreasing in width. The hole in the handle of the cup goes on increasing. Ultimately, only the handle of the cup outline is left which shrinks down to a single point as the length of this outline decreases from both sides of the outline.

2. Thinning

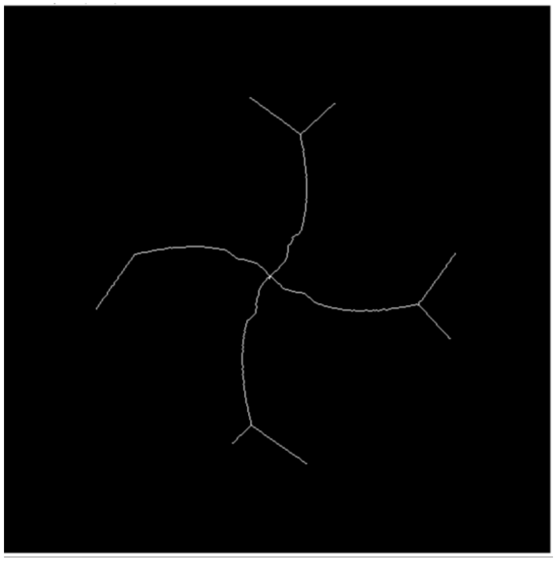
Fan.raw





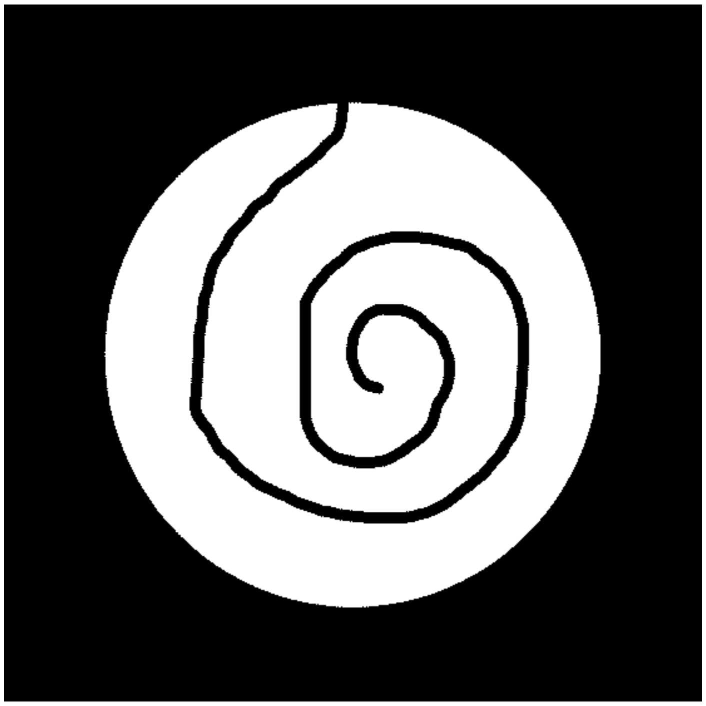


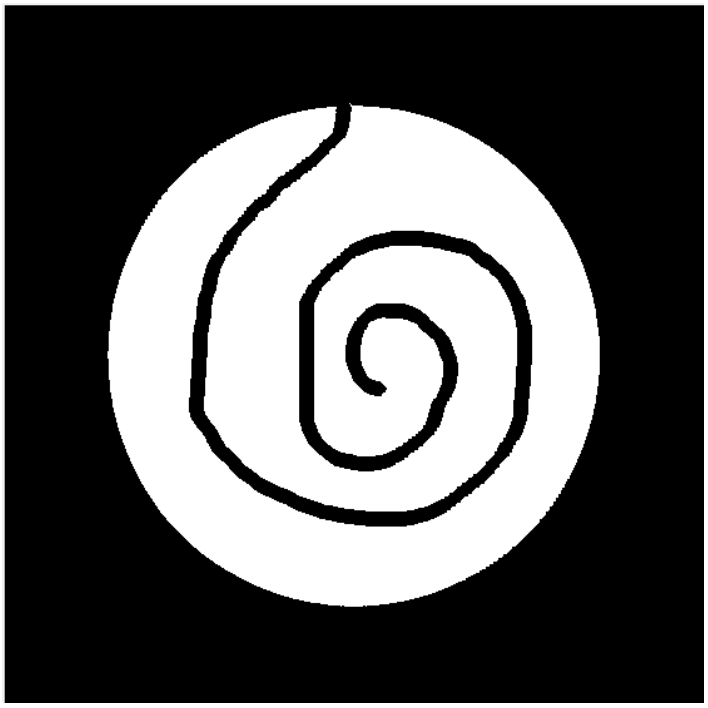
Final output image of thinning

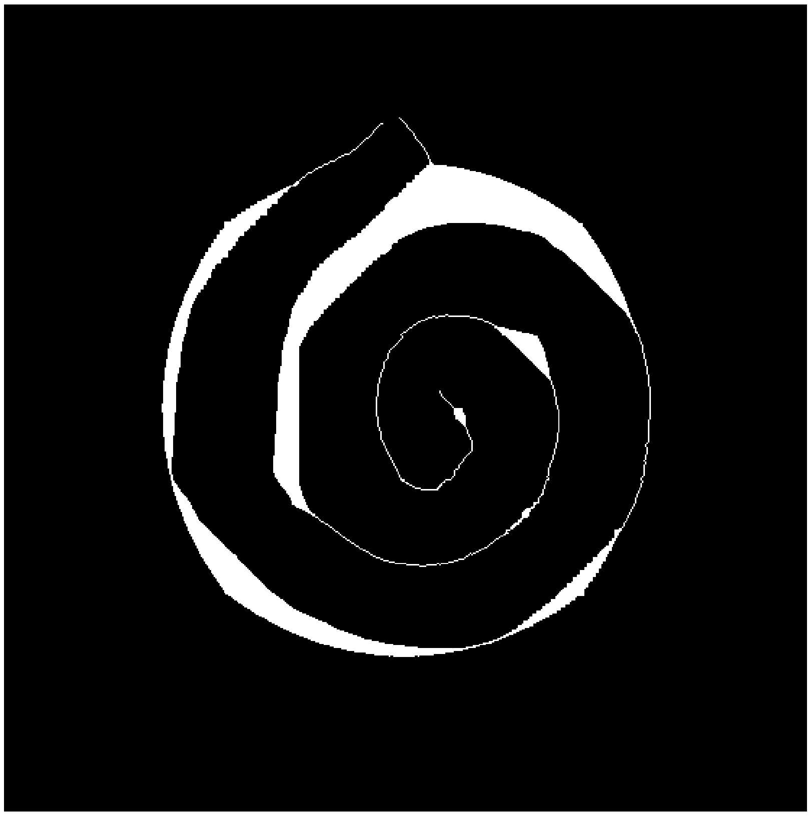


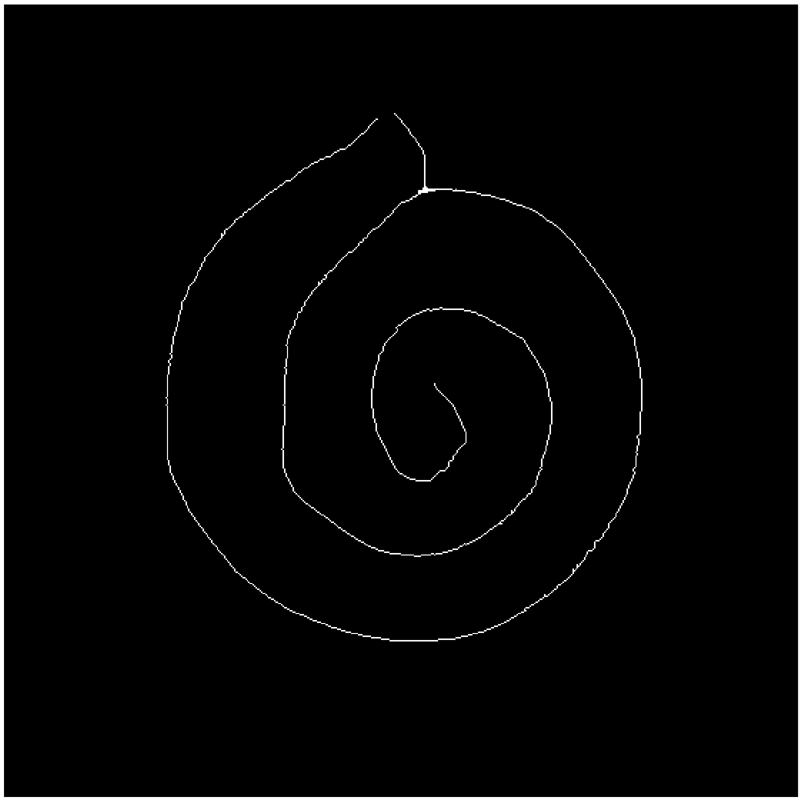
The process is the same as shrinking till only a single pixel thick line is left. After this line, there are no more modifications made to the image.

Maze.raw

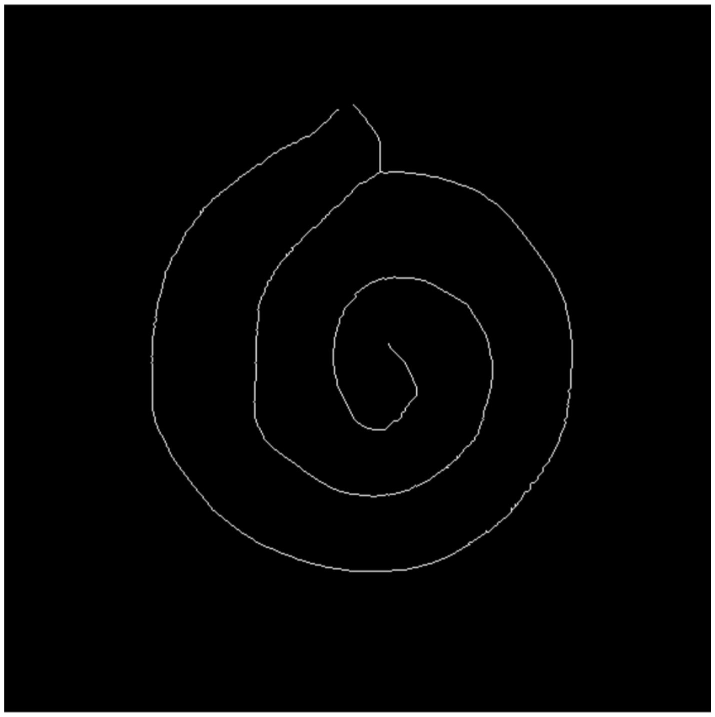






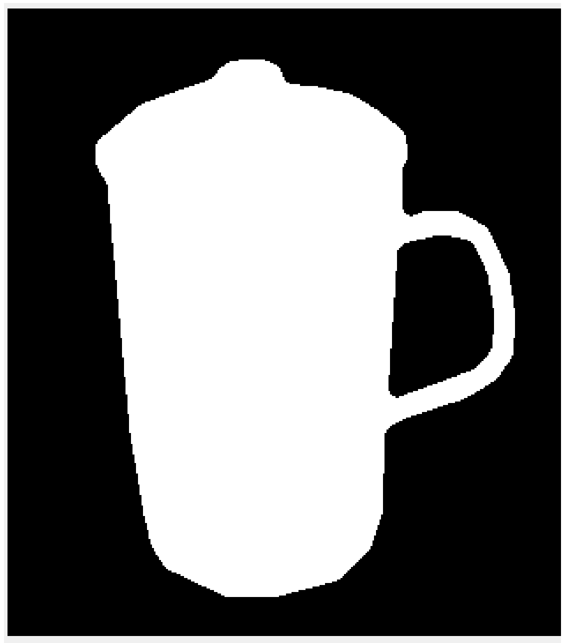


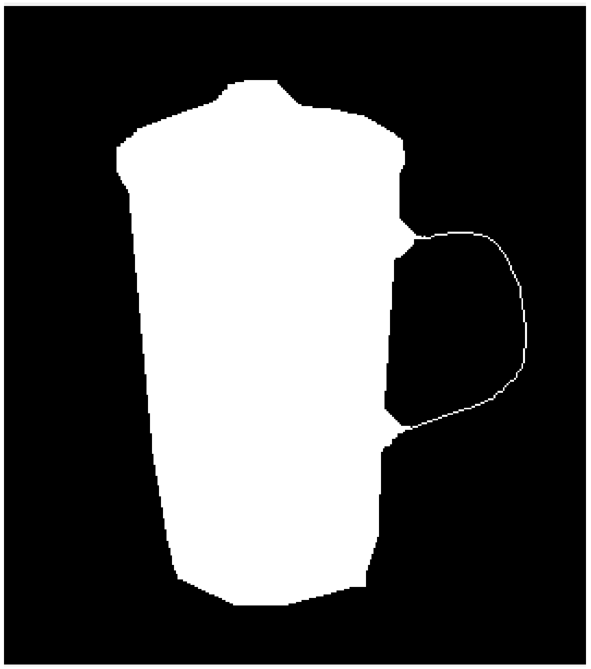
Final output image of thinning of maze.raw

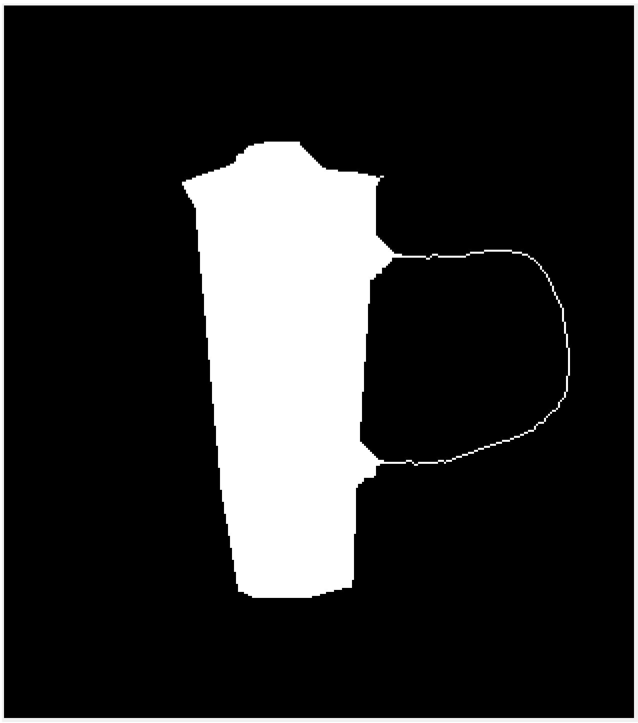


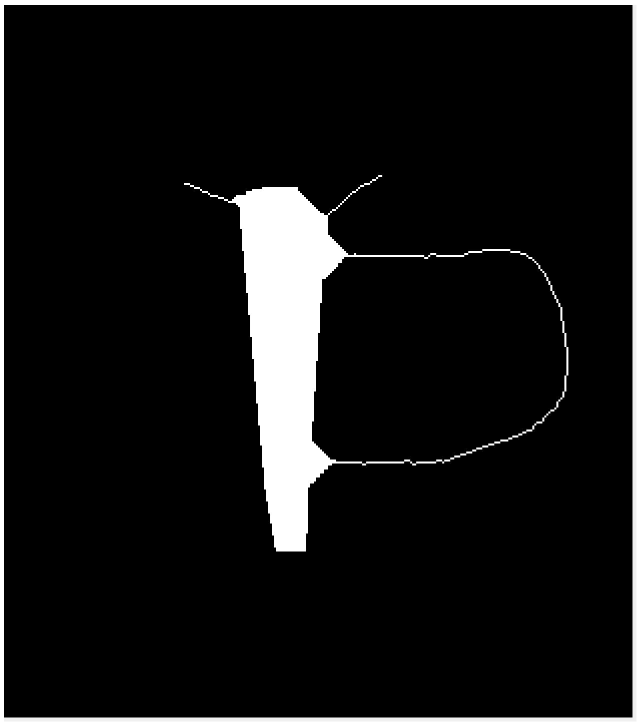
The process is the same as shrinking till only a single pixel thick line is left. After this line, there are no more modifications made to the image.

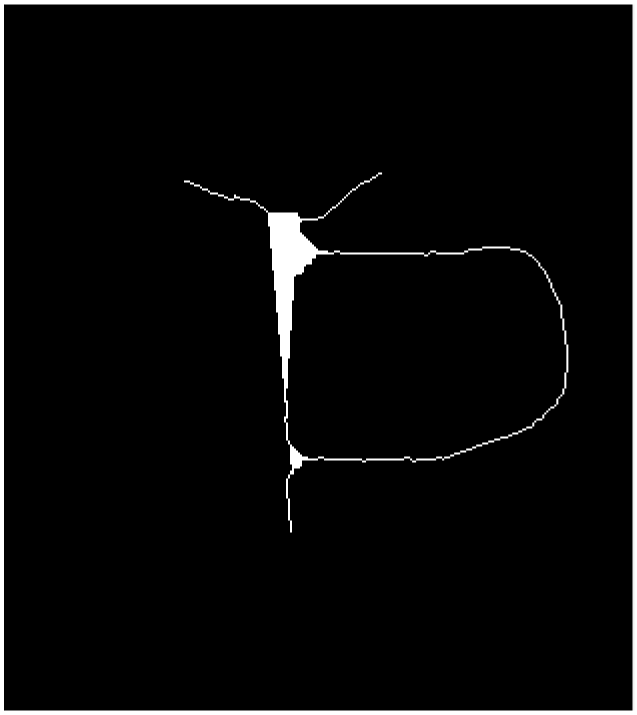
Cup.raw



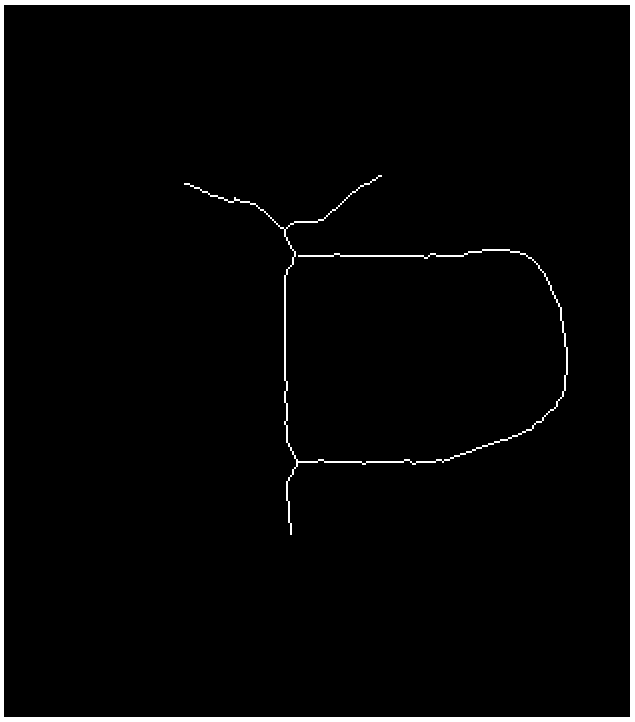








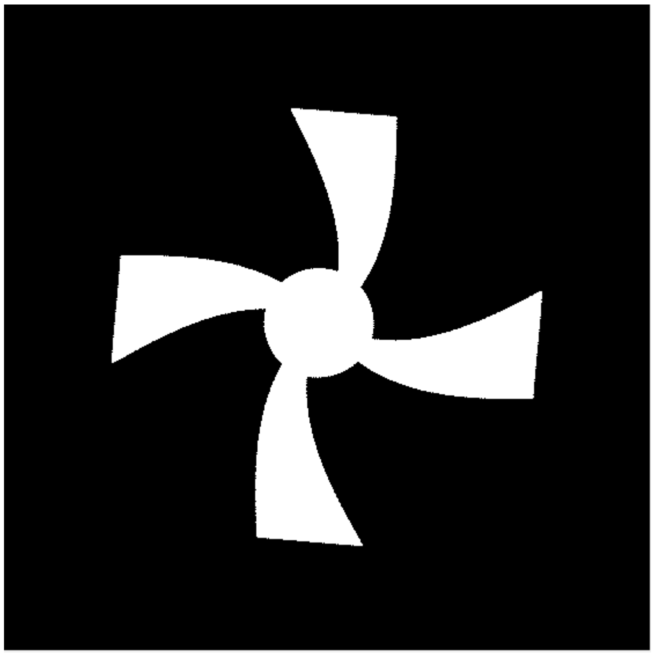
Final output image of thinning

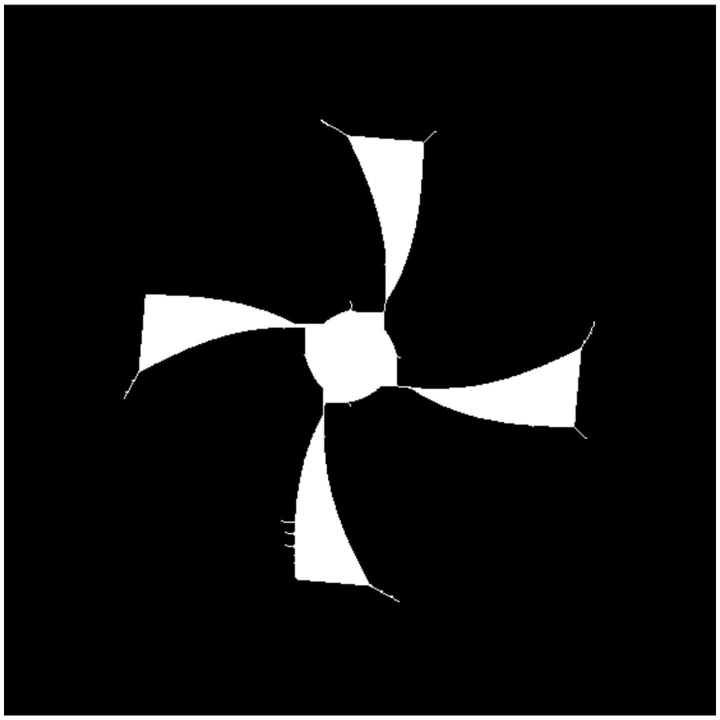


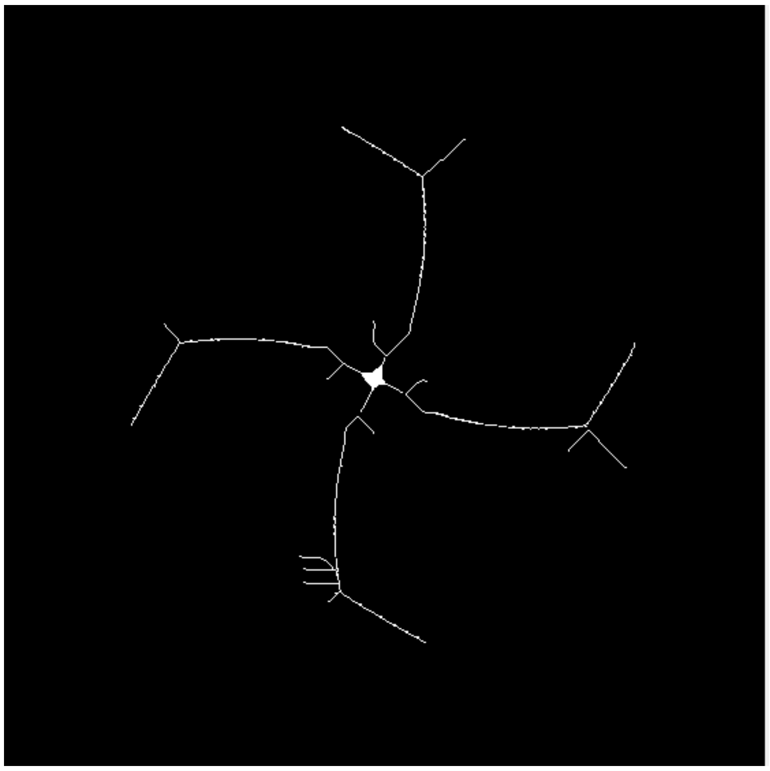
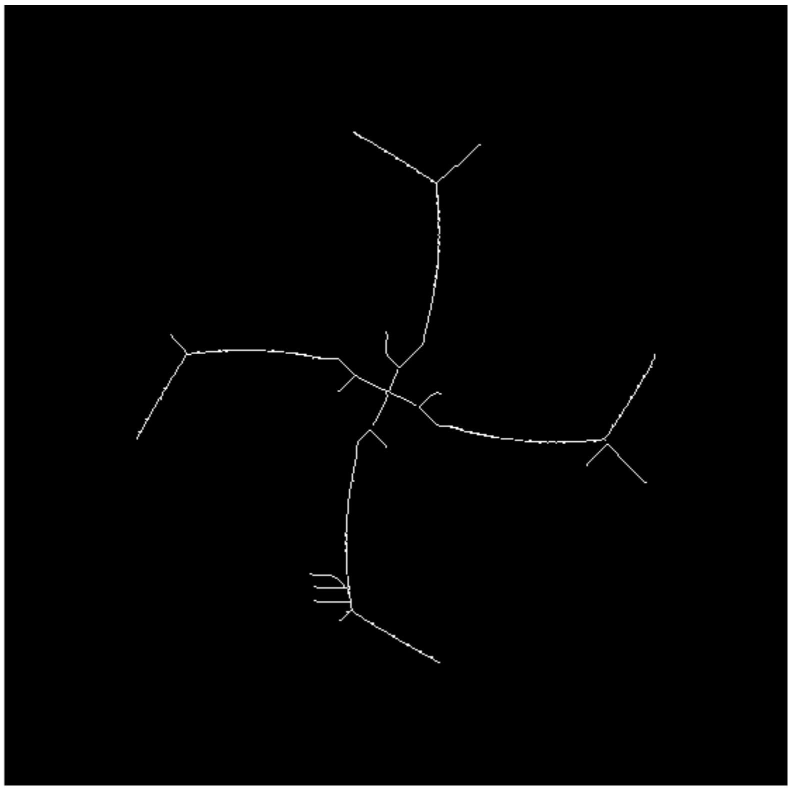
The process is the same as shrinking till only a single pixel thick line is left. After this line, there are no more modifications made to the image.

3. Skeletonizing

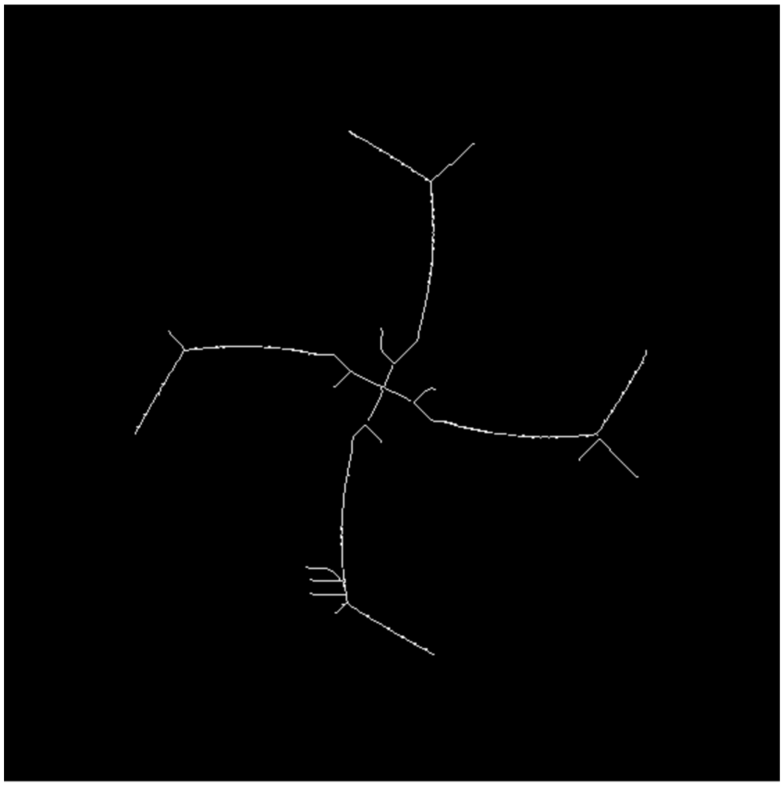
Fan.raw





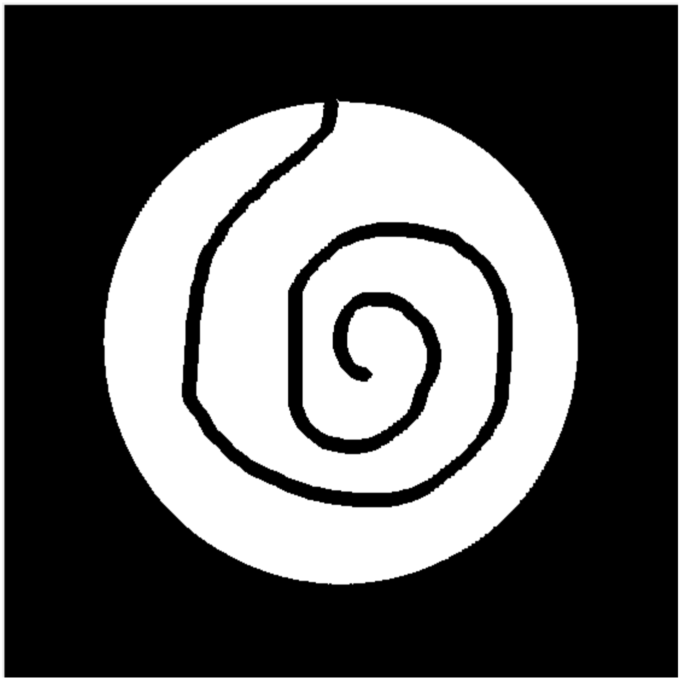



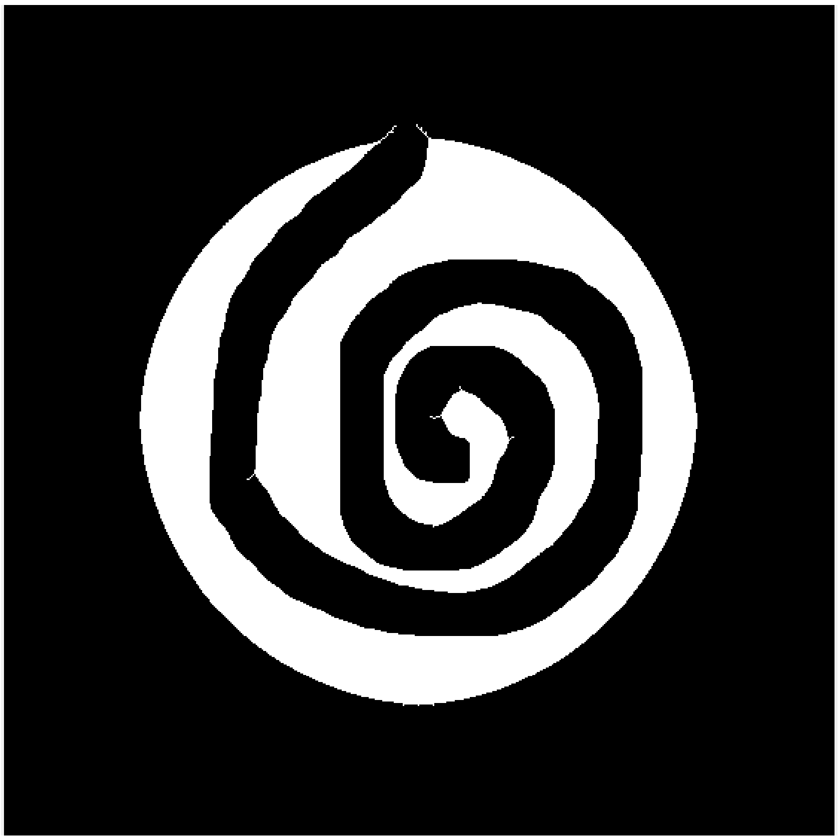
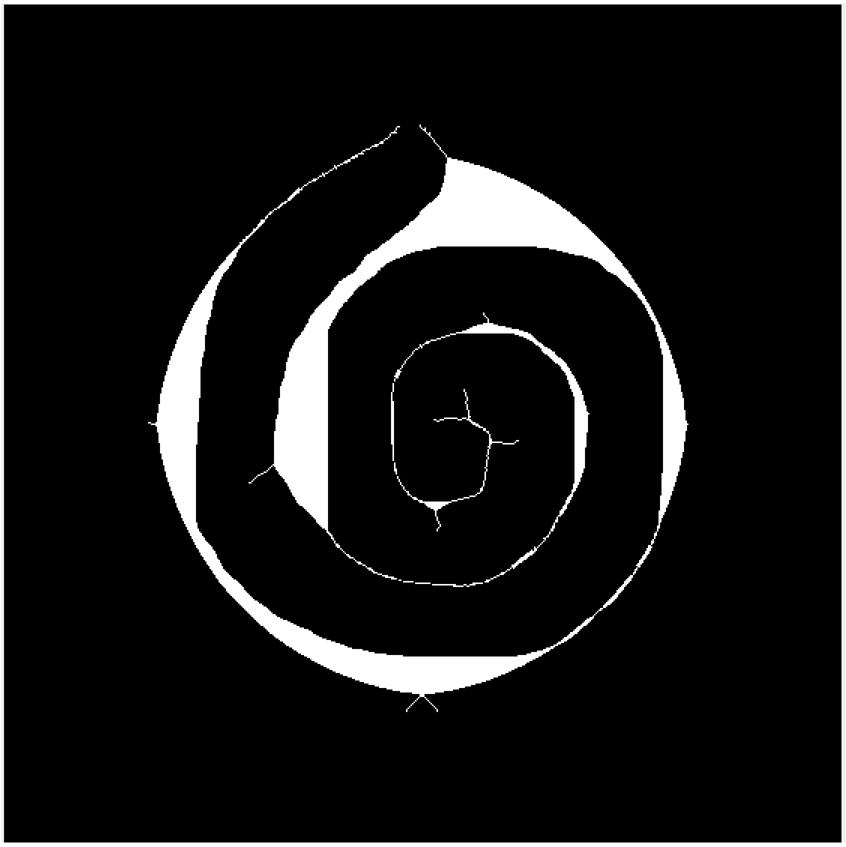
Final output for skeletonizing



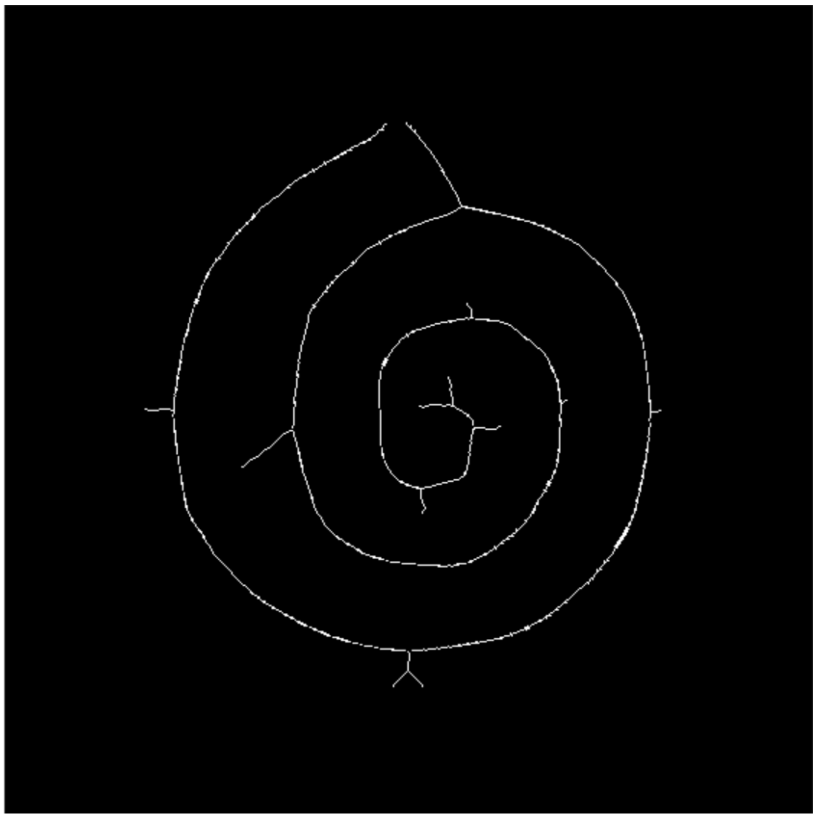
The conditional and unconditional tables for skeletonizing are used to do this operation.

Maze.raw



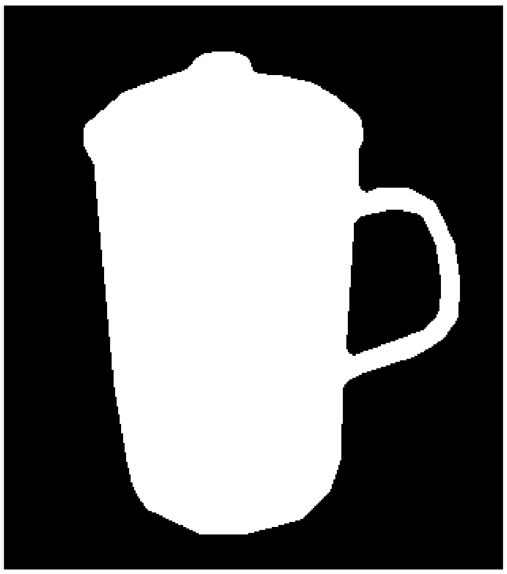



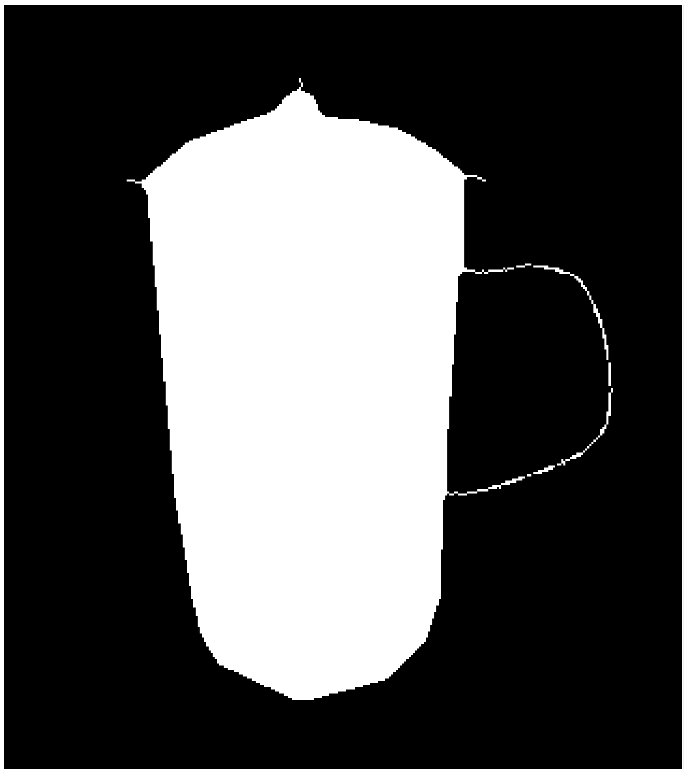
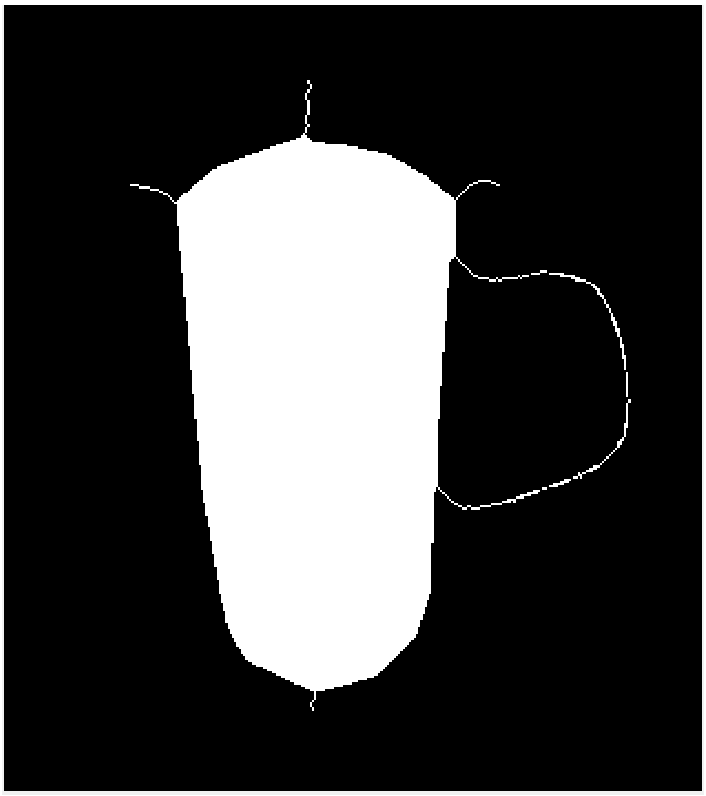
Final output for skeletonizing of maze.raw

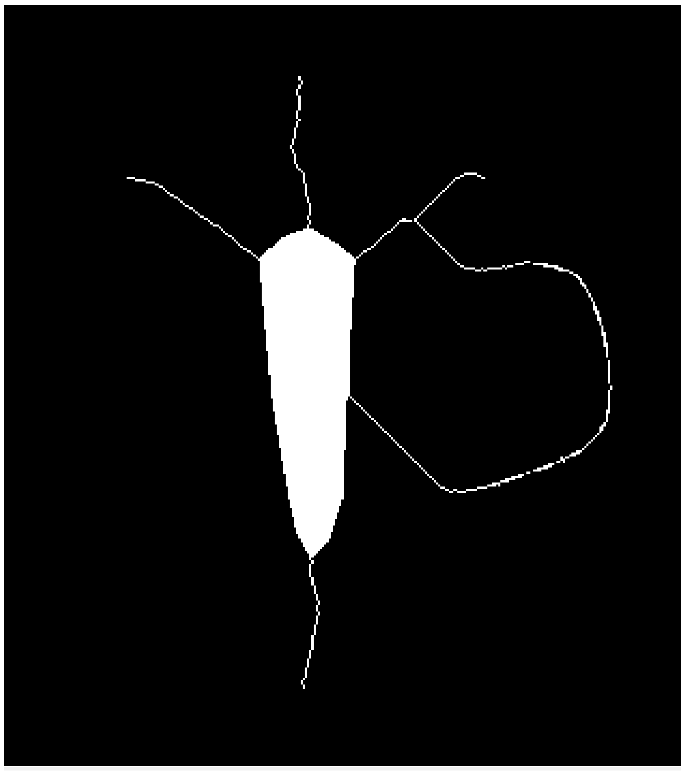
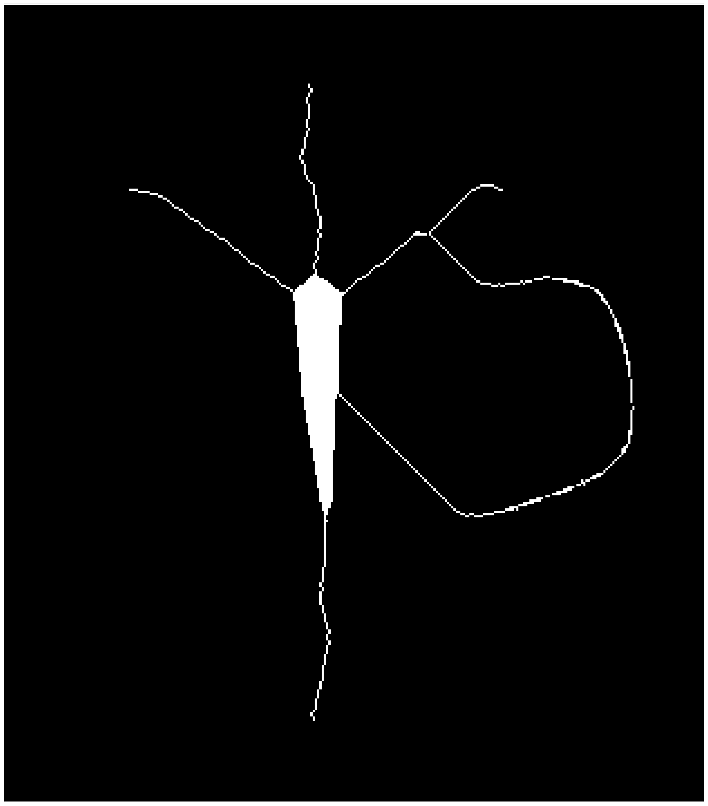


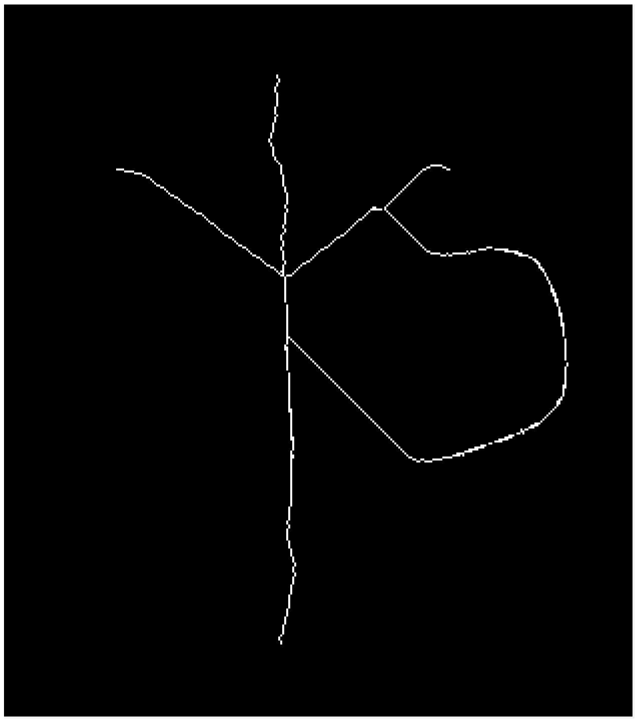
The conditional and unconditional tables for skeletonizing are used to do this operation.

Cup.raw

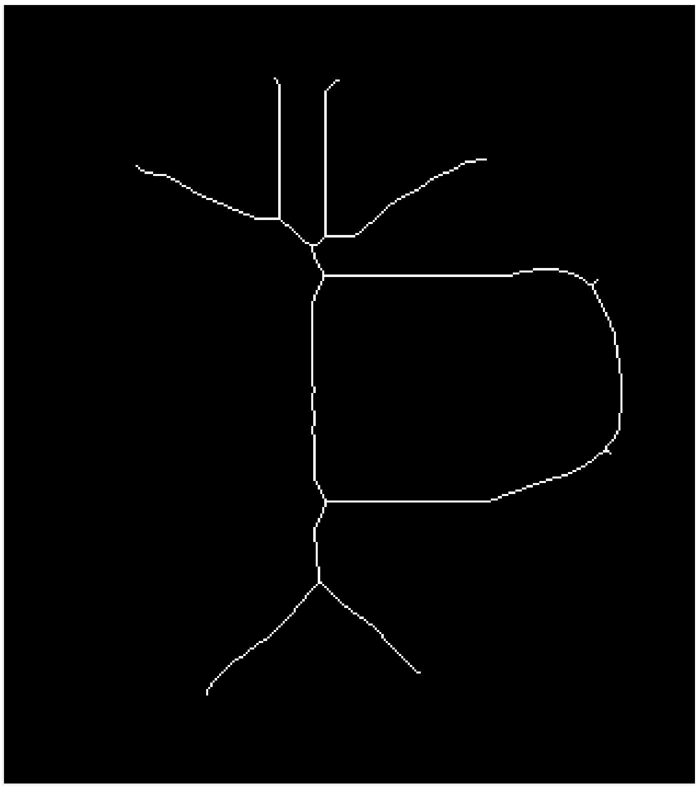




Final output for skeletonizing

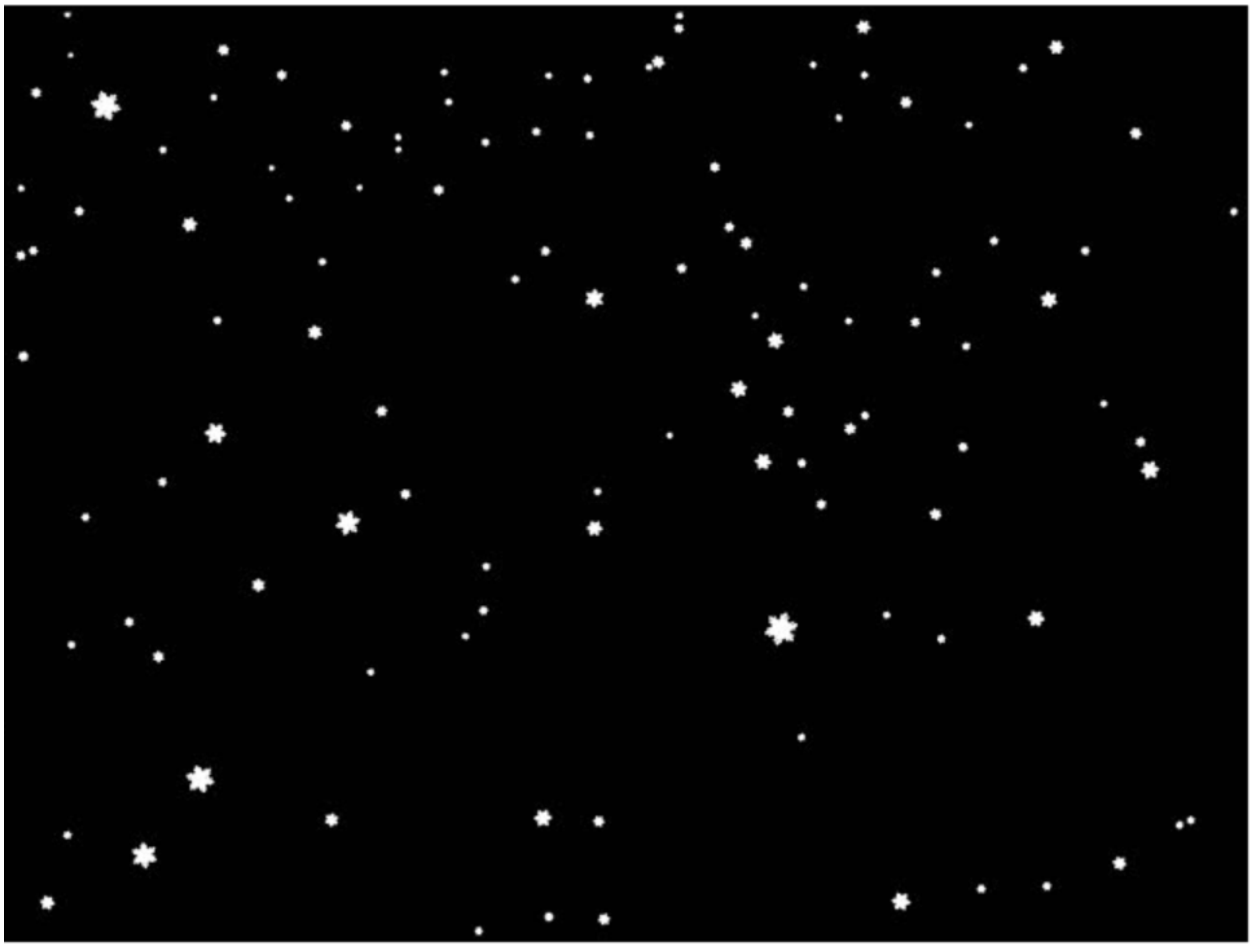


The conditional and unconditional tables for skeletonizing are used to do this operation.

Problem 2:

(b) Counting games

Stars.raw



(1) To count the number of stars in the image, I first applied the morphological operation of shrinking to the image. By shrinking, all the stars (white objects) have been shrunk down to single white pixels as can be seen in the results image.

Next, I scanned the whole image and applied this mask to all the white pixels in the image: [0 0 0; 0 1 0; 0 0 0]. If it matched, then I incremented the count of my stars. Hence, I found the number of stars in the image. The number of stars were found to be **112**.

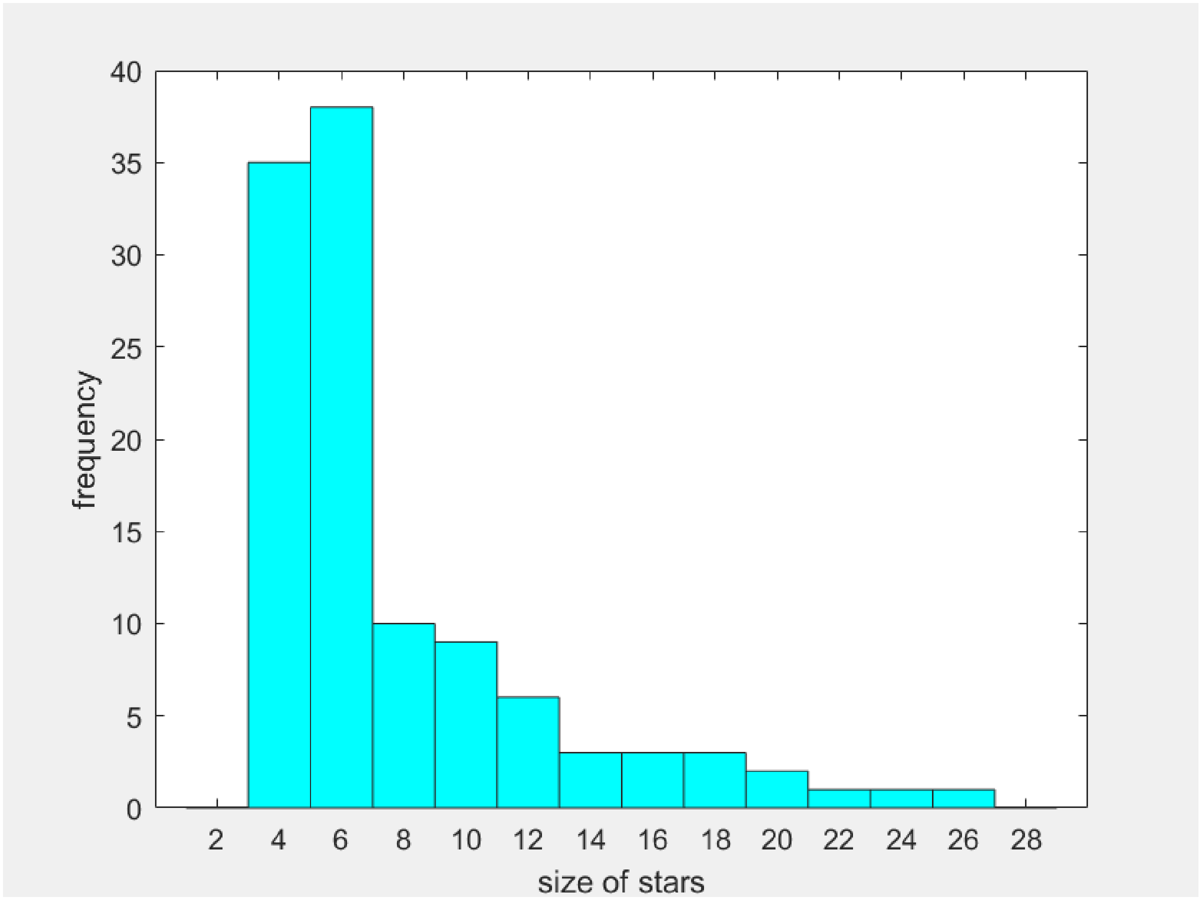
Results:

```
>> countinggames
    Retrieving Image stars.raw ...
    14


 Write image data to1.raw ...
number of stars:
   112

>>
```

(2) In every iteration of shrinking, I counted the number of stars that were fully shrunk. If at an iteration, some of the stars were fully shrunk down to 1 pixel, then I noted down the number of stars and the number of iteration (to calculate the size of the star).

The number of iterations represents **half** the size of the star, because the stars shrink 1 pixel from both sides, so the size decreases by 2 pixels in one iteration. Here I have taken size to be the diameter of the star. For circular stars, it is the diameter, and for star-shaped stars, it is the length of the point of the star from the opposite point of the star.

With this information, I have plotted the following histogram which gives the size of stars and the frequency.



(3) Another method to do this is by using **connected component labelling.**

In this, I first binarize the stars.raw image.

First pass:

I start from the first pixel and if it is a foreground pixel, then I label it as 1. Then I move to the next pixel and label it as 1 if it is a foreground pixel, otherwise I do nothing.

Then, when I find another foreground pixel after a background pixel, I label it as 2, if it is unlabeled, and so on. I check the 4 pixels adjacent to this pixel, that is, the (i-1,j-1) , (i-1,j) , (i-1,j+1) and (I,j-1). If the label of any of these pixels is less than the pixel itself, then the label of the pixel is changed to the least label value among these.

Second pass:

In this, I move through the image again and check for the 8 adjacent pixel labels. The (i-1,j-1) , (i-1,j) , (i-1,j+1) , (i,j-1) , (i,j+1), (i+1,j-1) , (i+1,j) and (i+1,j+1). If any of these pixels has a label that is less than (i,j) then I assign the label the value of the least pixel label around it.

Using these two passes, the number of label left at the end of these two loops will give the number of stars in the image.

Problem 2:

(c) PCB

(1) To find holes

To find the number of holes, I used the pcb.raw image as it was given. I binarized it so that it had values 0 and 1. Then I applied the shrinking operation on this image. As we can see, the 'objects' that we are interested in are the holes, are white in color, as they should be. So, after shrinking, the holes are shrunk to a single white pixel. After this, I applied a mask on the entire image wherever a pixel value was equal to 1 and compared it with this mask:

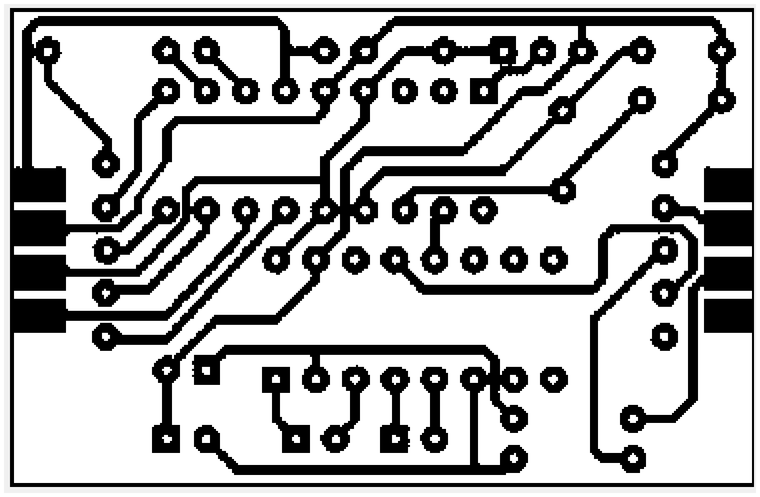
[0 0 0;

0 1 0;

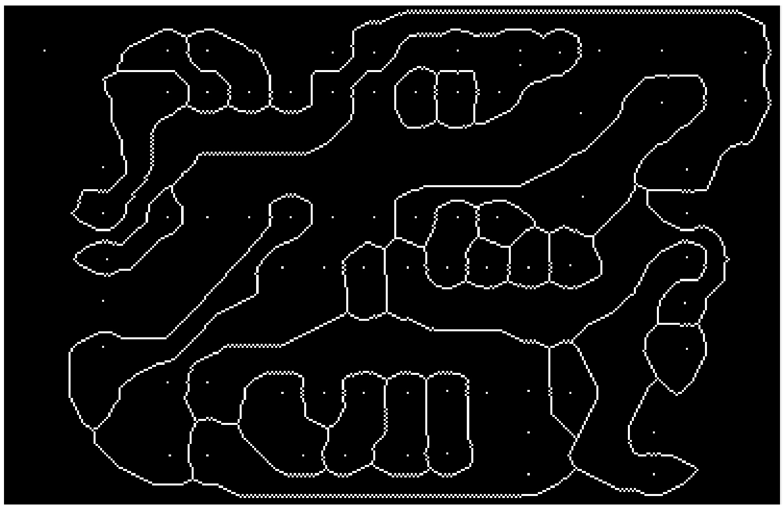
0 0 0]

If the mask and the pixel value matched, then I incremented the count of the holes by 1. And thus, I got the total number of holes in the PCB.

Pcb.raw



Result after shrinking pcb.raw



```
>> pcb
    Retrieving Image PCB.raw ...
number of holes
    71

Number of Connected Components:
    15
```

I found that there were **71** holes.

(2) to count number of connected components

To count the number of connected components, I inverted the image of pcb.raw. So, the white pixels became black and the black pixels became white. The inverted image is shown in figure (a)

Assumptions I made for this problem:

I counted one pathway to be an entire connected link through which current can pass through.

Step 1: I used shrinking to obtain the output in figure (b).

Step 2: From this image I extracted the biggest and second biggest component as can be seen in figures (c) and (d) respectively.

Step 3: I removed the smallest components, which were the holes which had no connections as can be seen in figure (e). So I needed to delete them from the final image so that I could count the pathways without errors.

Step 4: I subtracted the small holes from the output image of shrinking. I subtracted figure (e) from figure (b) to obtain the final image.

Step 5: The final image is figure (f). I applied connected component labelling to this image and got the number of connected components as the output.

I got the number of connected components to be 15.

Figure (a)

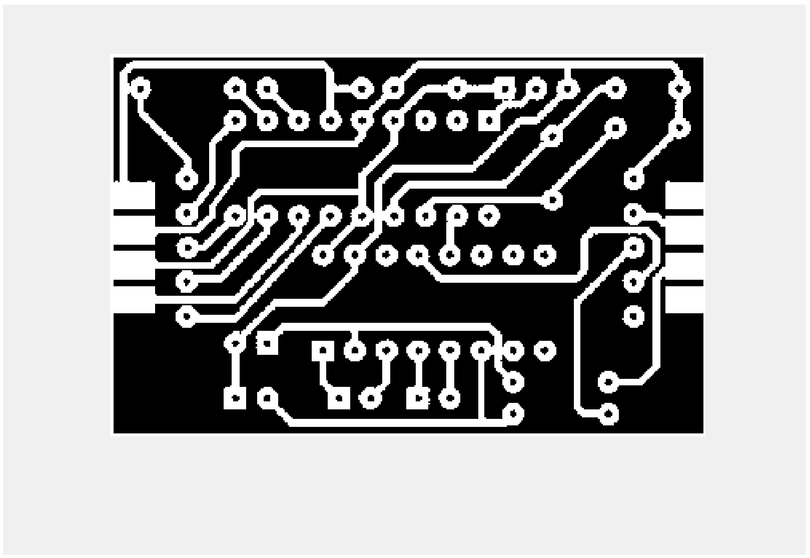


Figure (b)

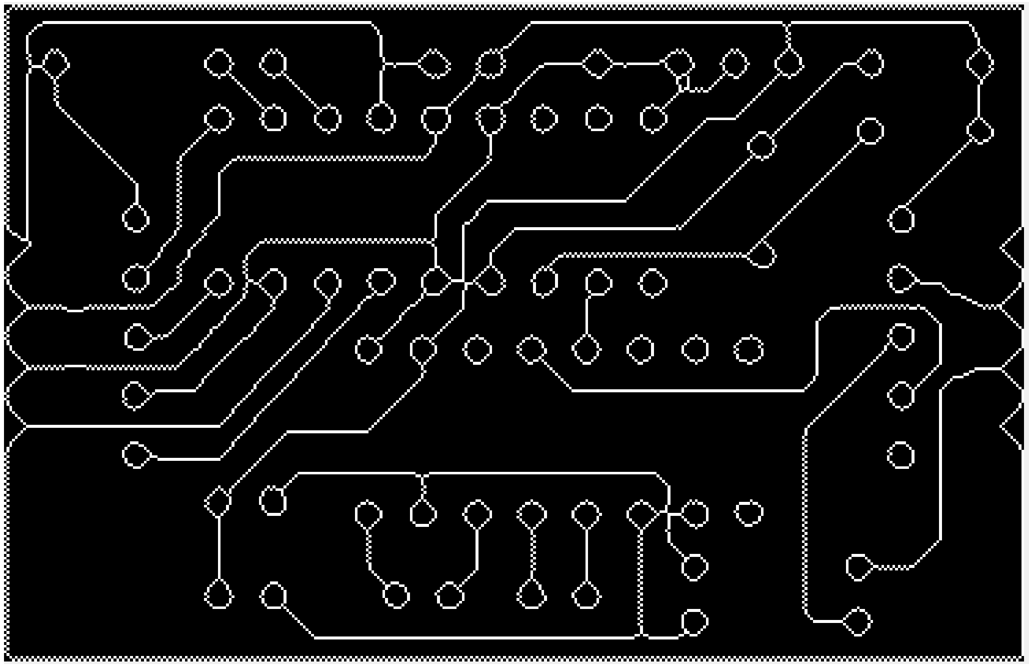


Figure (c)

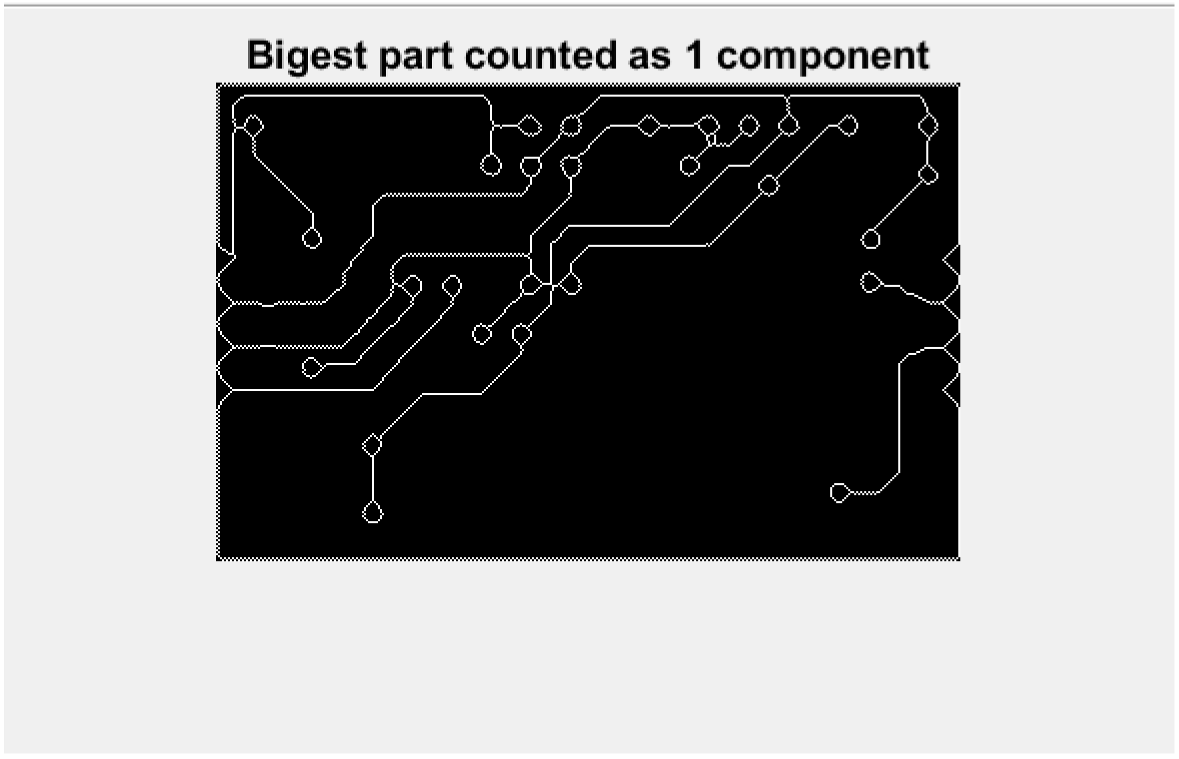


Figure (d)

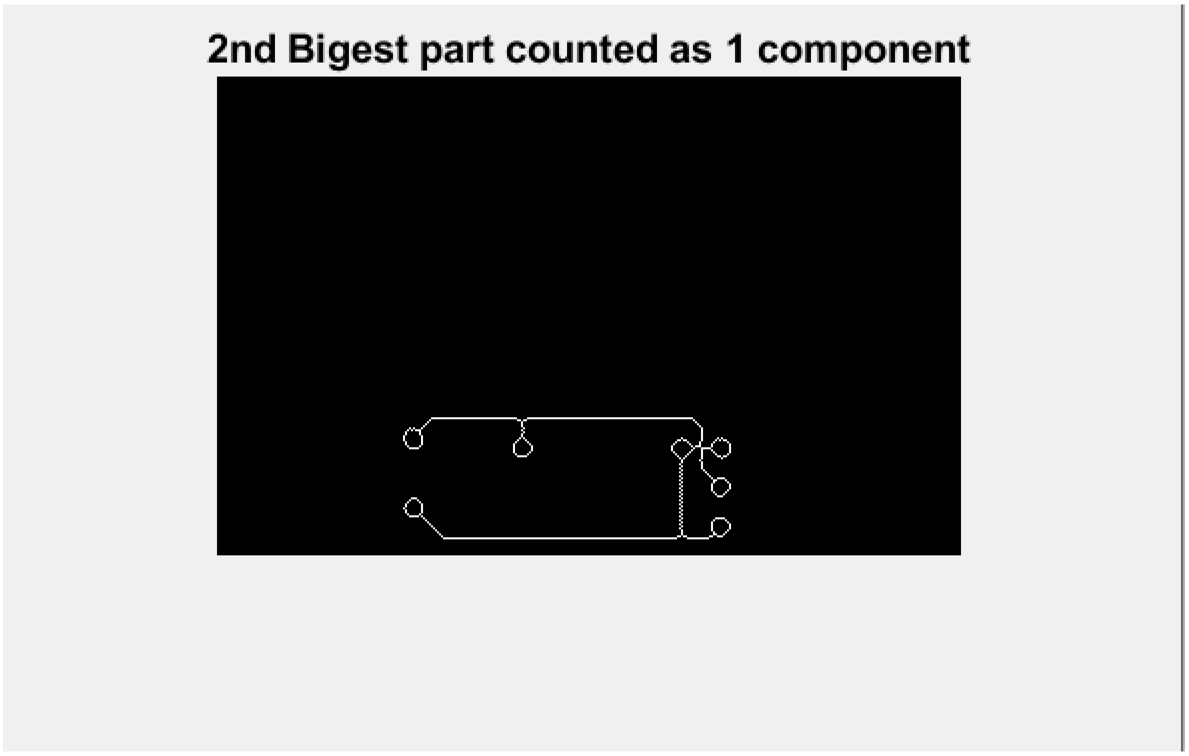


Figure (e)

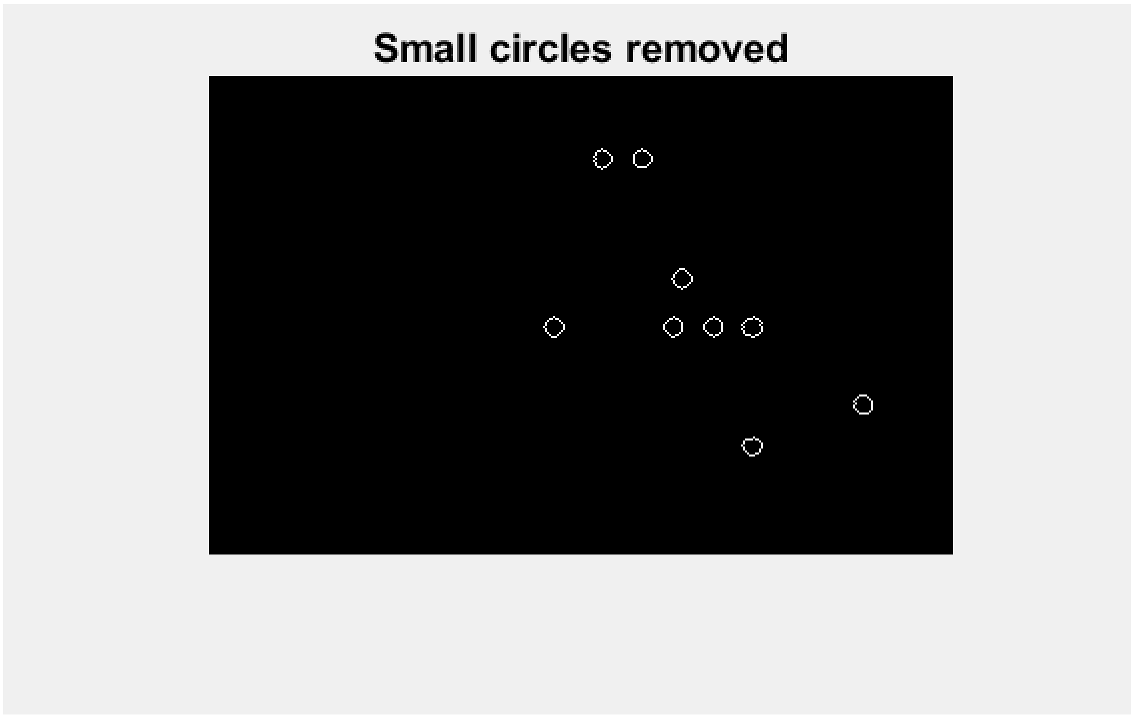
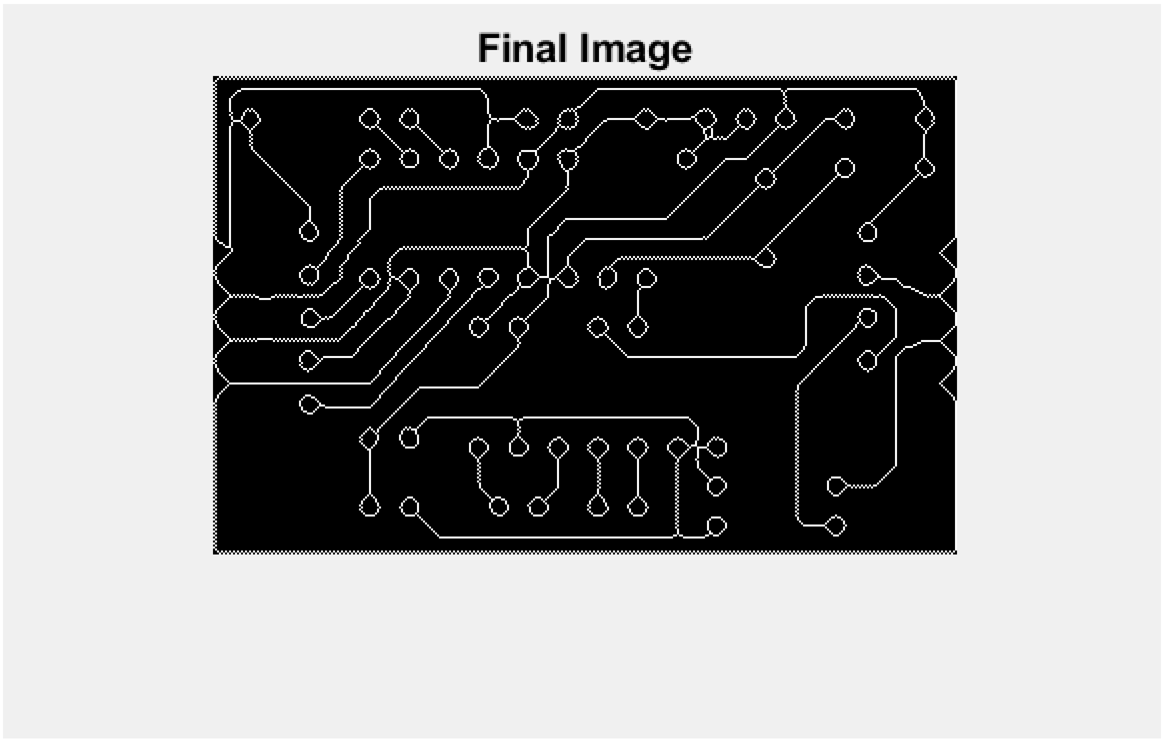


Figure (f) : Final image has the small holes removed



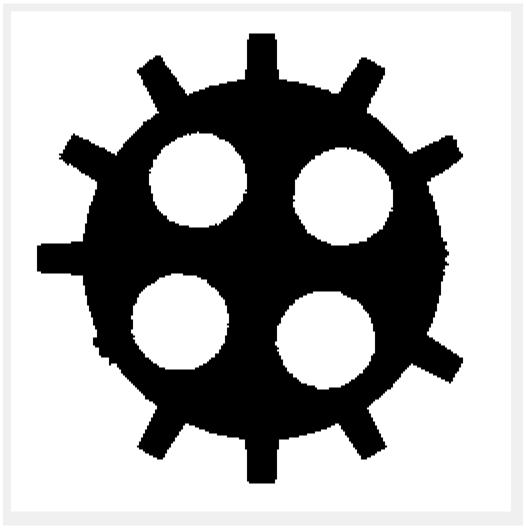
```
>> pcb
    Retrieving Image PCB.raw ...
number of holes
    71

Number of Connected Components:
    15
```

Problem 2:

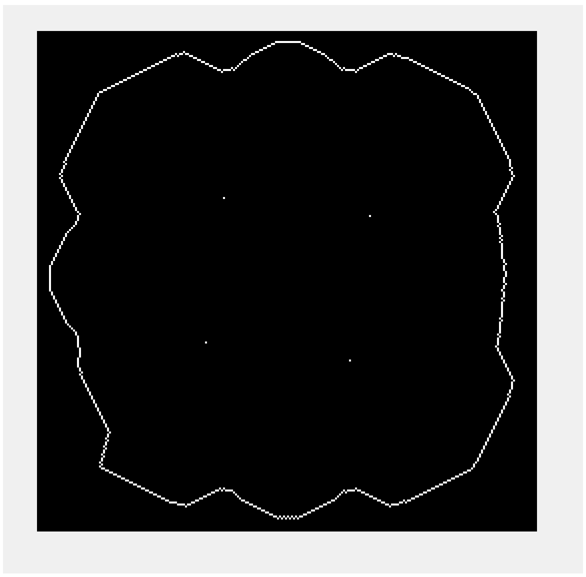
(d) Missing GearTooth Detection

Step 1: invert geartooth image (black to white and white to black) to get centres of the 4 holes

I had to find the centres of the 4 holes in the gear. So I inverted the image so that the wholes were white (since they were the 'objects' I had to find the centre of)



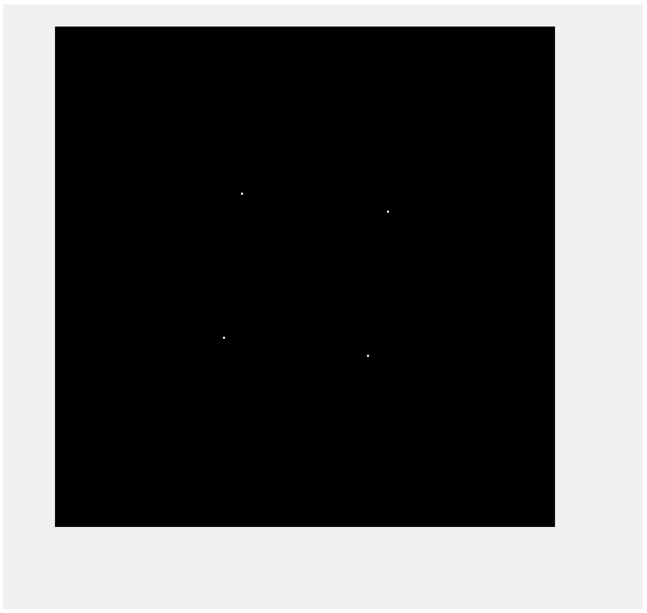
Step 2: I applied shrinking to this inverted image and got the 4 centres of holes along with a boundary.



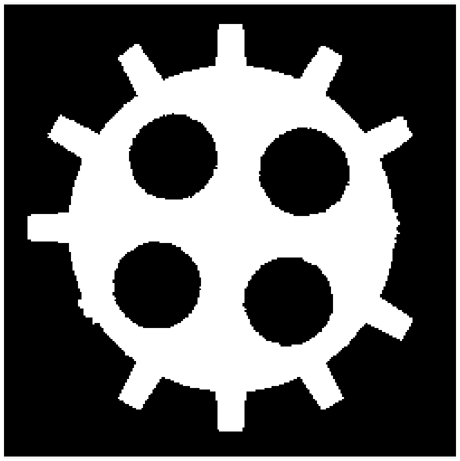
Step 3: I removed the white outline so that only the 4 centres were remaining. I did this by applying the following pattern on the image:

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

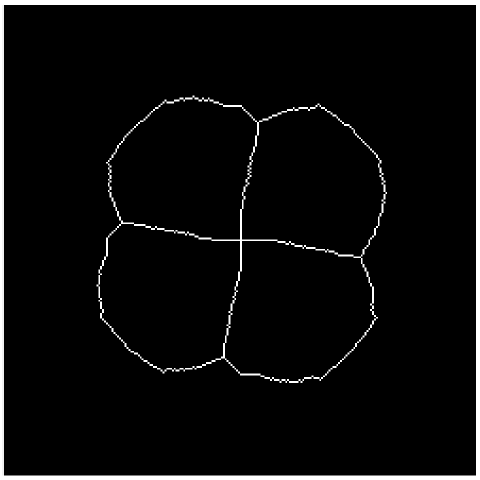
If it matched, then I kept the pixel, otherwise I made it 0.



Step 4: Read Original image



Step 5: Shrinking on the original image to get the following output.



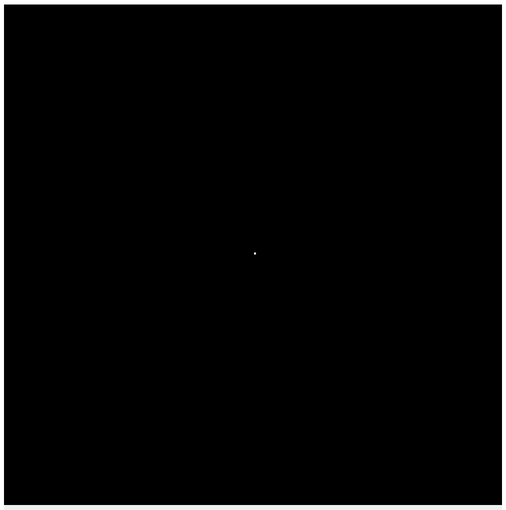
Step 6: Applied the following pattern to this image to get centre of the gear:

[0 1 0

1 1 1

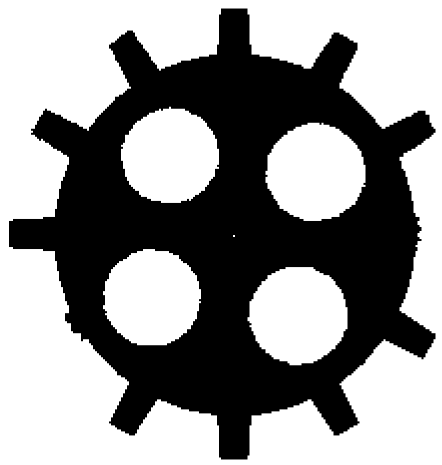
0 1 0]

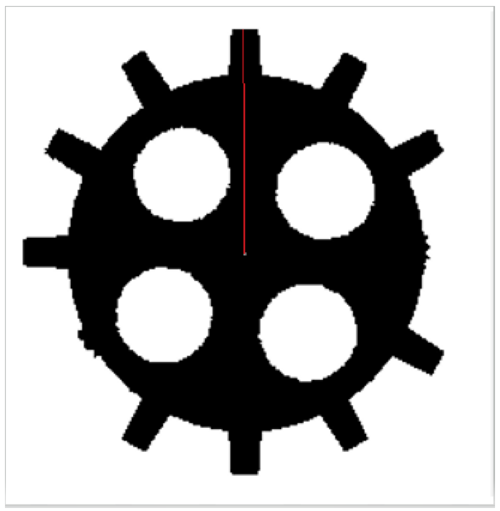
I applied this because the centre of the gear after shrinking was surrounded by 1s on the right, left, top and bottom pixels. And black pixels in the north-east, north-west, south-east and south-west pixels. So, there was only 1 pixel which would match this pattern. If it matched the pattern, I kept the pixel value as 1, otherwise 0.



Step 7: I added the image obtained in Step 6 to the inverted image in Step 1 and calculated the radius of the gear. I knew the pixel location of the centre of the gear. So, I calculated the radius of the gear by going from the centre to the top of the gear, and stopping only when another white pixel (value 1) was found. I would keep incrementing the radius till the pixel value 1 was detected.

The radius is marked in red in the image below:





```
>> geartooth

radius =

  111
```

Because the radius is 111, I take the value of r to be 105, since this would be inside the spoke of the gear.

Step 8: Mark the positions of all the 12 spokes that are supposed to be there on the gear.

Since there are 12 spokes and it is a circle, the angle between the lines joining the centre and 2 adjacent spokes is 30 degrees. I calculated the x and y values as r*cos(30) and r*sin(30) respectively, to get the image co-ordinate values, so that it moved in a circular fashion from one spoke to another and marked the position of the spoke.

centrex = 125 and centrey=126 (image coordinates of centre of circle)

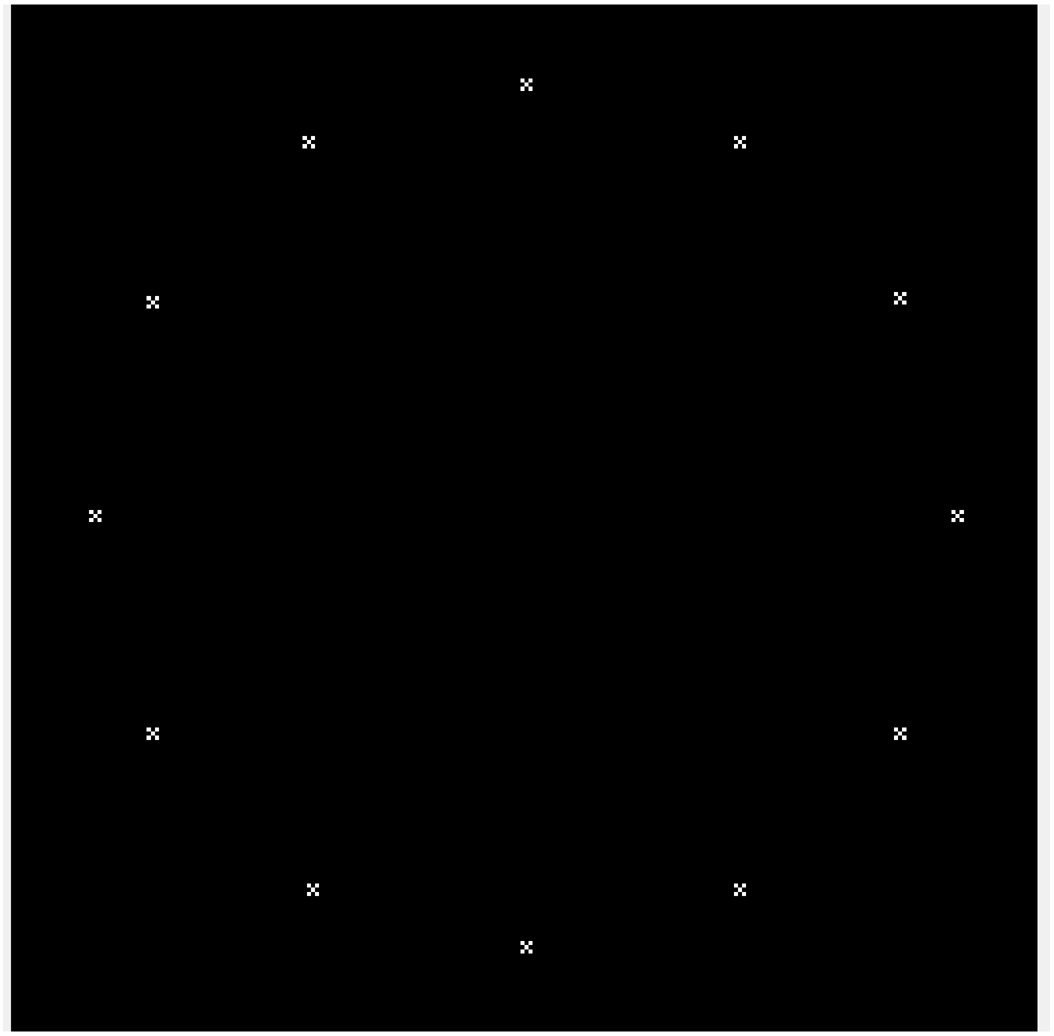
calculate image x coordinate (row)

```
if number of spoke is less than 4 or greater than
or equal to 10
        x = centrex+x;
else
        x = centrex-x;
```

calculate image y coordinate (column)

```
if number of spoke was less than or equal to 7
then,
        y = centrey - y;
if number of spoke was greater than 7
then,
        y = centrey + y;
```

The marked positions of the 12 spokes can be seen in the image below:



Step 8: Put this on the on the original image of the geartooth and I got the location of missing spokes. The two crosses represent the 2 missing spokes.

I added the previous image to the geartooth original image, and it detects the missing GearTooth locations.