EE569: Introduction to Digital Image Processing

ASSIGNMENT #3

Sonali B Sreedhar

USC ID: 1783668369

Email ID: sbsreedh@usc.edu

Problem 1: Geometric Image modification

Abstract and Motivation:

Geometric modifications are one of the most commonly used methods in image processing techniques. It is mainly used in image registration and in removal of geometric distortion. Here an image is spatially translated, scaled in size, rotated, and non-linearly warped or viewed from a different perspective. Here, we do rearrangement of the pixels on an image plane. The coordinates are transformed to the desired plane with desired geometric warping. The output of the translated image's intensity does not depend on the input of a particular location but will depend on intensity of some other pixel at a different location translated by the transformation.

In this project, we are implementing geometric warping where an input square image is converted into a disk-shaped image.

The following three main conditions are to be satisfied:

1. Pixels on the boundaries of the square image should be translated to boundaries of the disc image.
2. Pixels on the center of the square image should be translated to center of the disk-shaped image.
3. We should ensure that the mapping is reversible, thus we have to ensure that the mapping we are doing is strictly one to one mapping.

Approach and Procedure:

Here our task is to transform a square shaped image into a disk-shaped image and obtain the original back. To accomplish that, for every point in the input image, we need to find the corresponding location in the desired output shape. This can be done by forward mapping. Then, for every point in the output image we find the corresponding location in the input image by inverse mapping. Here since we are asked to convert the square shaped image into a disk shaped image we first need to work on obtaining the cartesian coordinates from the image coordinates. It is given by,

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0.5 \\ -1 & 0 & h - 0.5 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r \\ c \\ 1 \end{bmatrix}$$

Here in the matrix, r and c are the row and column coordinates taken from the image. H stands for the height of the image.

For obtaining the back the original image from the disk-shaped image, we use the same matrix inversed using inverse mapping.

The transformation carried out for converting square shaped image into a disc shaped image is provided below:

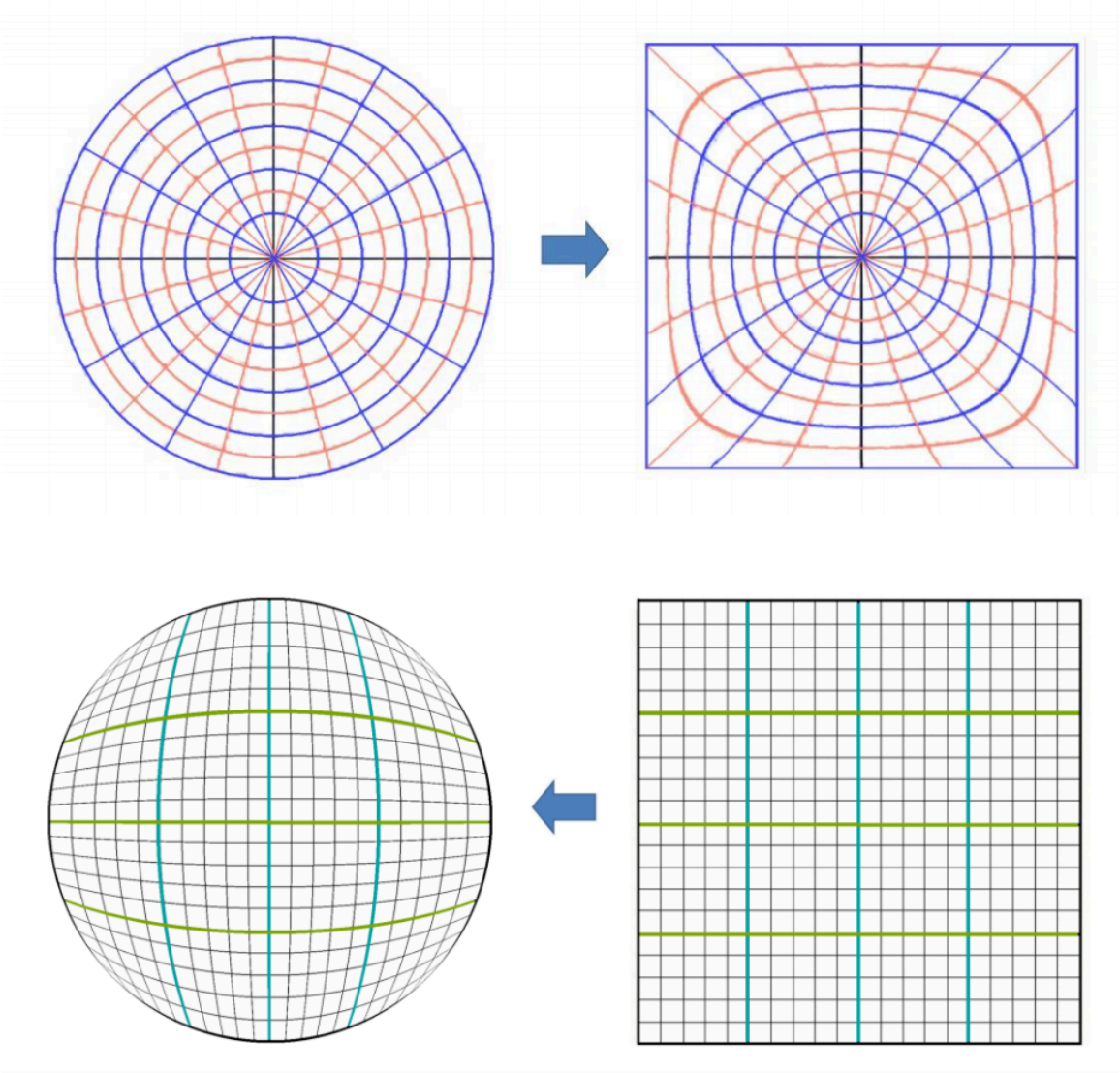


Figure 1: Illustration of Elliptical Grid mapping

The formulae used for these conversion is given below:

Disc to square mapping:

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

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

Square to disc mapping:

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

After carrying out the mapping functions, we use bilinear interpolation so as to reduce the error in pixel conversion.

Let $\mathbf{I}$ be an $R \times C$ image.
We want to resize $\mathbf{I}$ to $R' \times C'$.
Call the new image $\mathbf{J}$.

Let $s_R = R / R'$ and $s_C = C / C'$.

Let $r_f = r' \cdot s_R$ for $r' = 1, \ldots, R'$

and $c_f = c' \cdot s_C$ for $c' = 1, \ldots, C'$.

Let $r = \lfloor r_f \rfloor$ and $c = \lfloor c_f \rfloor$.

Let $\Delta r = r_f - r$ and $\Delta c = c_f - c$.

Then $\mathbf{J}(r', c') = \mathbf{I}(r, c) \cdot (1 - \Delta r) \cdot (1 - \Delta c)$
$+ \mathbf{I}(r+1, c) \cdot \Delta r \cdot (1 - \Delta c)$
$+ \mathbf{I}(r, c+1) \cdot (1 - \Delta r) \cdot \Delta c$
$+ \mathbf{I}(r+1, c+1) \cdot \Delta r \cdot \Delta c.$

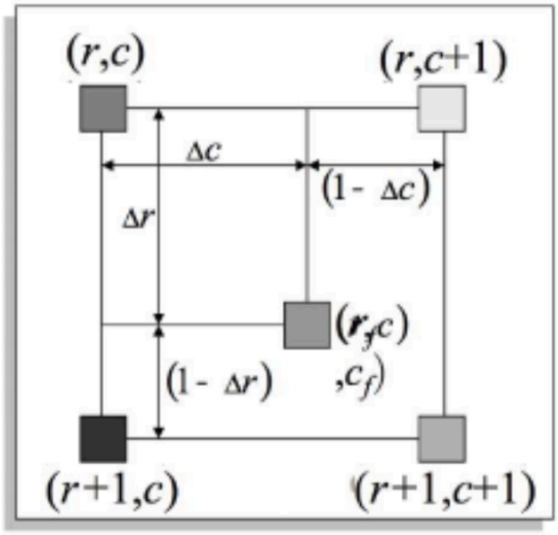


Figure 2 : Implementation of Bilinear Interpolation Algorithm

Algorithm implemented in C++:

1. Read the input image.
2. Convert the image coordinates into cartesian coordinates.
3. Use reverse mapping function to convert the cartesian coordinates of the square shaped image into its corresponding coordinates in the disk-shaped image.
4. Implement bilinear interpolation algorithm to obtain the pixel intensity and put it in the output image to be displayed.
5. Obtain the disk -shaped image mapped from the square shaped image.
6. Now obtain the cartesian coordinates by using forward mapping. And repeat step 4.
7. Obtain the square shaped image back from the disk-shaped image.

Experimental results:

Figure 3: Input images

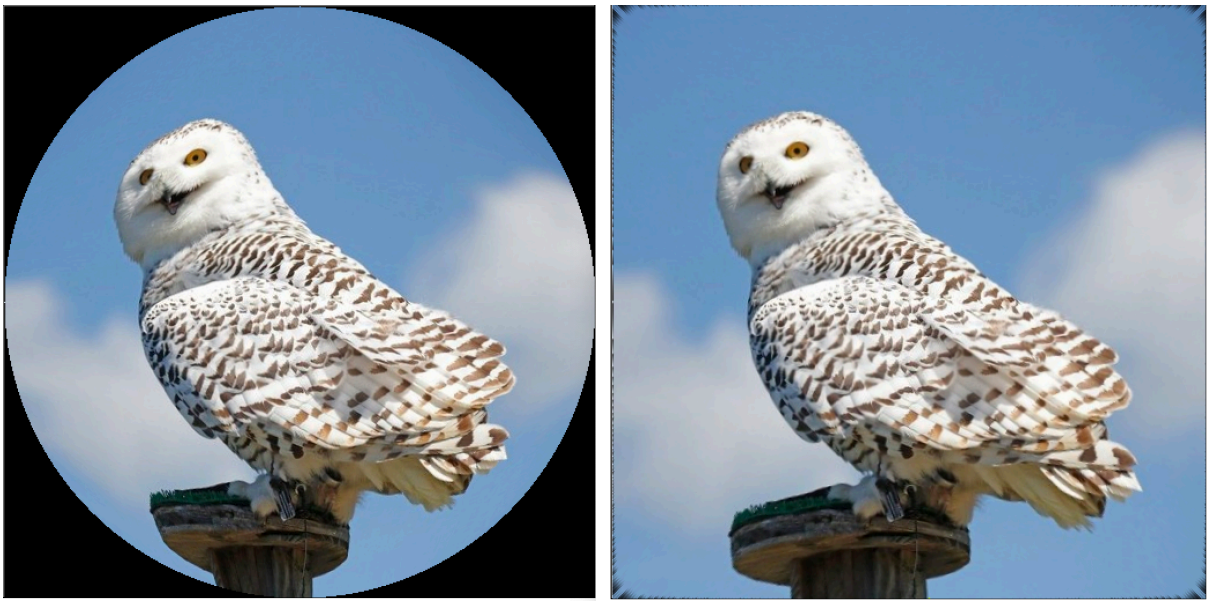


Figure 4: Disk warped and Reversed Hedwig.raw

Figure 5: Disk warped and Reversed Raccoon

Figure 6: Disk warped bb8 and Reversed image

Discussion:

Here, we can observe in the output image that it is a bit distorted at the corners. In the reverse mapping we can see that, though we try to map the center pixel of square shaped image to a disk-shaped image we see that, at the corners the output seems to be shrunk in order to be displayed in disk shape. In the forward mapping we observe that there is a black colored ray radiating outwards from the corner of the images. This distortion has occurred, may be because we were not able to recover all the pixels properly. When we transformed the input image into a disk-shaped image, we are reducing the output space, thus there might be a loss of information in the image. Hence, we will be not able to recover all the pixel information completely from the disk-shaped image.

1b) Homographic Transformation and Image Stitching:

Image stitching is mainly used to combine multiple images with different angle perspective. Here the images with matching points are overlapped to produce a high-resolution image of a segmented panorama. Here in this assignment we are given three images of a lunchroom which are taken with three different angles. All these images are stitched together by finding the strong key points.

Approach and Procedure:

The below are the basic procedure used for image stitching purpose:

The very first step is to find the control points from the sequence of images. Here I have used SURF feature detection and FLANN feature matching to identify the matching control points. Among the obtained matching control points, only four control points are selected such that they cover a very good area across the given image. These control points are used to obtain the

homographic matrix to find the points to map the images. Inverse warping is used to blend the sequence of images to one another. The new image stitched is made big enough by providing offset and the final Panorama image is written.

Algorithm implemented in C++:

1. Read the input images left.raw, middle.raw and right.raw.
2. Find the key control points.
3. Obtain H matrix by solving eight equations.
4. Use reverse mapping map left image into middle image by giving offsets.
5. Repeat the procedure for middle and right images.
6. Carry out bilinear interpolation to calculate the pixel intensity.
7. Write the output image into a file.

Experimental Results:

Figure 7: Matching points between middle and right image

Matching points between Left.raw and Middle.raw

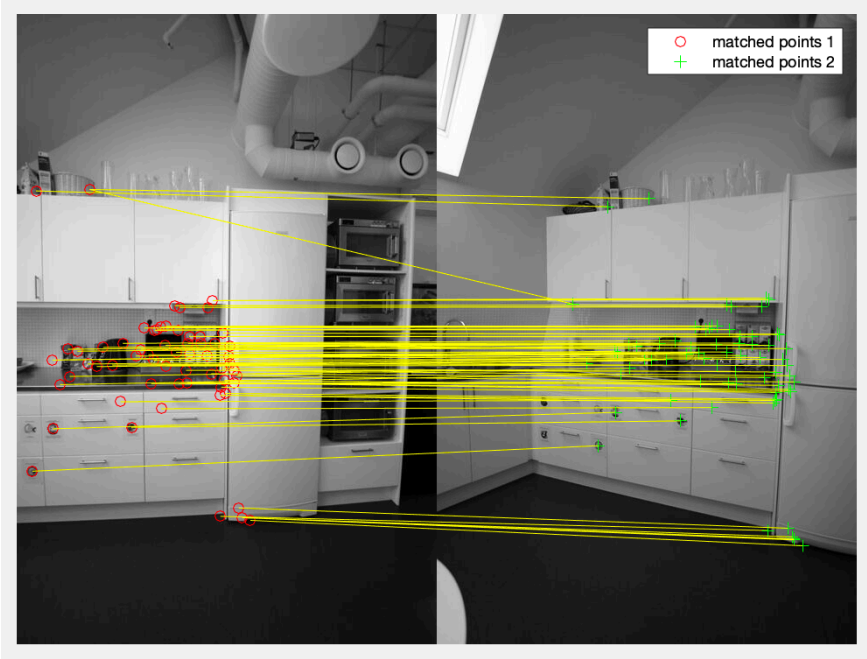
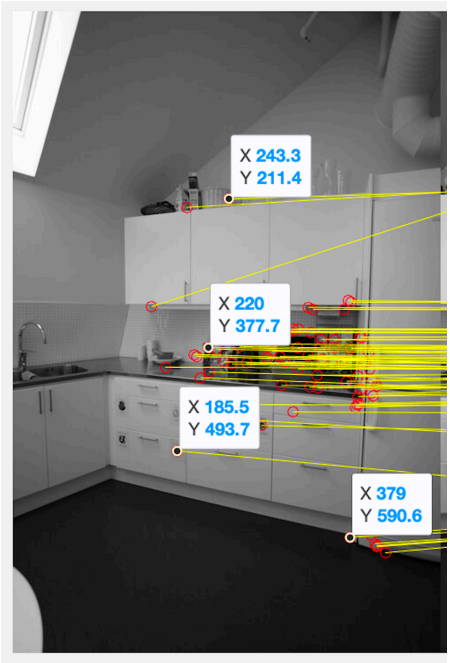


Figure 8: Matching points between left and middle image

Control points for Left.raw and Middle.raw:



The control points chosen between left and middle images are:

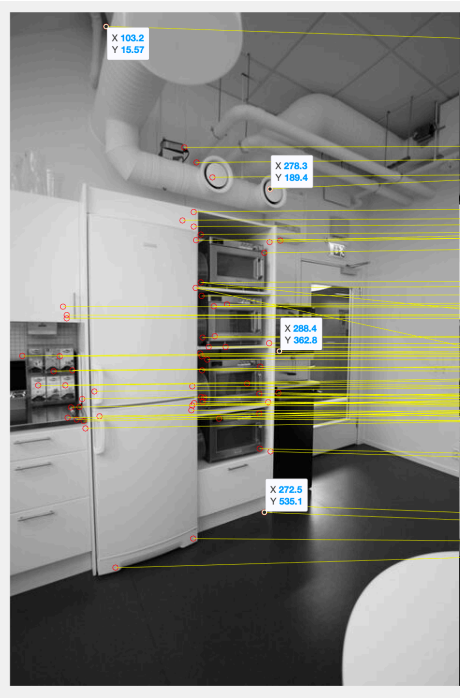
(xleft1,yleft1)=(243.3,211.4) (xleft2,yleft2)=(220,377.7)

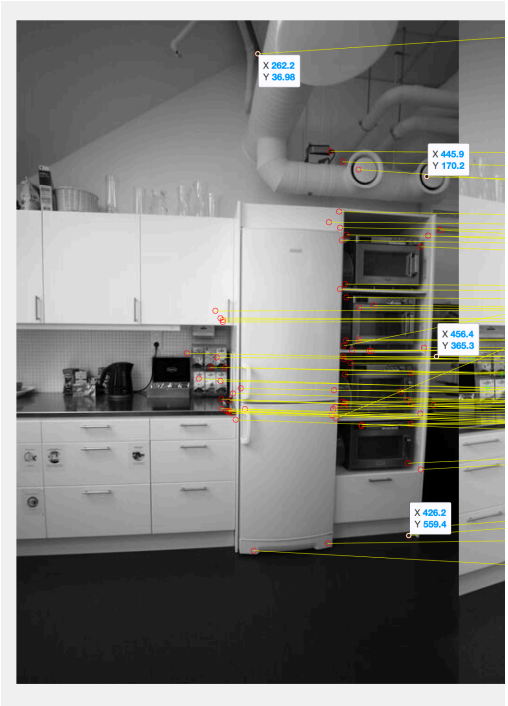
(xleft3,yleft3)=(185.5,493.7) (xleft4,yleft4)=(379,590.6)

(xmiddle1,ymiddle1)=(83.7,201)
(xmiddle3,ymiddle3)=(18.48,522.8)

(xmiddle1,ymiddle1)=(58.8,383.5)
(xmiddle4,ymiddle4)=(233.1,574.4)

Control points for Right.raw and Middle.raw:





The control points chosen between right and middle images are:

(xmiddle1,ymiddle1)=(262.2,36.98)
(xmiddle3,ymiddle3)=(456.4,365.3)

(xmiddle1,ymiddle1)=(445.9,170.2)
(xmiddle4,ymiddle4)=(426.2,559.4)

(xright1,yright1)=(278.3,189.4)
(xright3,yright3)=(103.2,15.57)

(xright2,yright2)=(288.4,363.8)
(xright4,yright4)=(272.5,535.1)

Output image:

Discussion:

The above shown image shows almost perfect image stitching of the three images provided. Here the selection of key control points plays important role in image stitching processing. While computing H matrix, we need to solve 8 equations for nine unknow variables. Here we are considering H33 to be 1 for simplification of the calculations.

The functions detectSURFFeatures() and extractfeatures() provide us the points on the images which we have to look into for matching points on sequence of the images. I have used OpenCV MATLAB for this purpose. After obtaining the matching features using showMatchedFeatures(), we select four control points. The control points are to be selected such that it covers pretty good area of the images to be stitched. Here we are selecting four control points as we have 8 equations. With the increased in the number of control points, we obtain better overlapping of the images stitched. Also, we use bilinear interpolation to reduce the distortion created while overlapping the images on one another.

Morphological processing:

Abstract and Motivation:

Morphological processing is used in identifying or modifying the structure of an image mainly. The three fundamental morphological image processing are erosion, dilation and skeletonizing. In dilation we try to make the objects in the image more visible by filling the holes in between. In erosion, we try to remove pixels so that only prominent objects are left over in the image. In skeletonizing, we try to erode the objects to its centerlines without introducing any distortion to its skeleton or the image's structure.

Here in this part we have implemented three morphological processing namely, Shrinking, thinning and skeletonizing.

Hit and Miss Transformation:

The hit and miss operation is carried out by sliding the center pixel of the structuring element which is a small binary image with 1's and 0's, to identify any particular patterns in a binary image. If we find the structuring element to match the pixels of the image, the center pixel is set to 1 or 0 otherwise.

We use pattern table for shrinking, thinning and skeletonizing operations as a look up table.

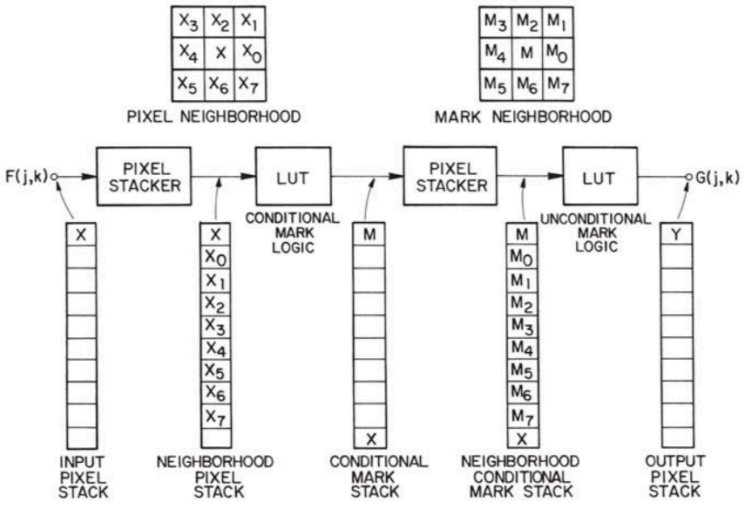


Figure 9: Algorithm implemented for morphological processes

Shrinking:

In shrinking, we shrink the objects without holes a single point and objects with holes to shrink into a connecting line/ring.

Approach and Procedure:

Algorithm implemented using C++:

1. Read the input image.
2. Convert the image into binary image.
3. Extend the boundaries of the input image as we are going to perform convolution to check for the conditional and unconditional masks.
4. Implement the Look Up table for conditional and unconditional masks.
5. The conditional mask is made to traverse all over the image for hit or miss operation. If it's a 'hit , we copy the pixel otherwise we set it to black.
6. The obtained result is to be stored in a temporary array.
7. Extend the boundaries of the temporary array again.
8. The unconditional mask is made to traverse the output obtained. If it is a 'hit', we copy the pixels, else we make them black.
9. Run these steps until the image shrink to a single point

Thinning:

In thinning, we try to reduce the thickness of the image by shrinking them to a connected line or a ring.

Approach and Procedure:

Algorithm implemented:

1. Read the input image.
2. Convert the image into binary image.
3. Extend the boundaries of the input image as we are going to perform convolution to check for the conditional and unconditional masks.
4. Implement the Look Up table for conditional and unconditional masks.
5. The conditional mask is made to traverse all over the image for hit or miss operation. If it's a 'hit , we copy the pixel otherwise we set it to black.
6. The obtained result is to be stored in a temporary array.
7. Extend the boundaries of the temporary array again.
8. The unconditional mask is made to traverse the output obtained. If it is a 'hit', we copy the pixels, else we make them black.
9. Run these steps until the image shrink to a single point

Skeletonizing:

In skeletonizing, we try to reduce pixels of the outlines of the objects without allowing the image to break apart. Thus, we get a skeleton formation.

Approach and Procedure:

Algorithm implemented:

1. Read the input image.
2. Convert the image into binary image.
3. Extend the boundaries of the input image as we are going to perform convolution to check for the conditional and unconditional masks.
4. Implement the Look Up table for conditional and unconditional masks.
5. The conditional mask is made to traverse all over the image for hit or miss operation. If it's a 'hit , we copy the pixel otherwise we set it to black.
6. The obtained result is to be stored in a temporary array.
7. Extend the boundaries of the temporary array again.
8. The unconditional mask is made to traverse the output obtained. If it is a 'hit', we copy the pixels, else we make them black.
9. Run these steps until the image shrink to a root line.
10. Break the iterations if you find the line to start breaking.

Experimental Results:

Shrinking:

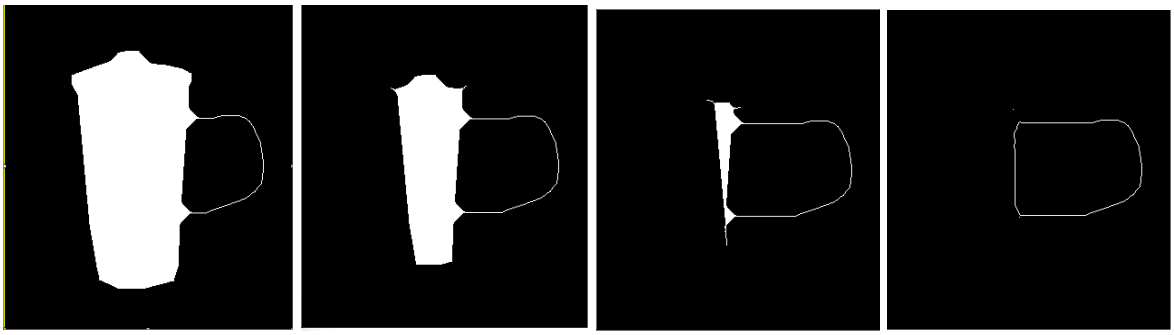


Figure 10: Shrinking of cup.raw at 25,50,75,105 iterations respectively

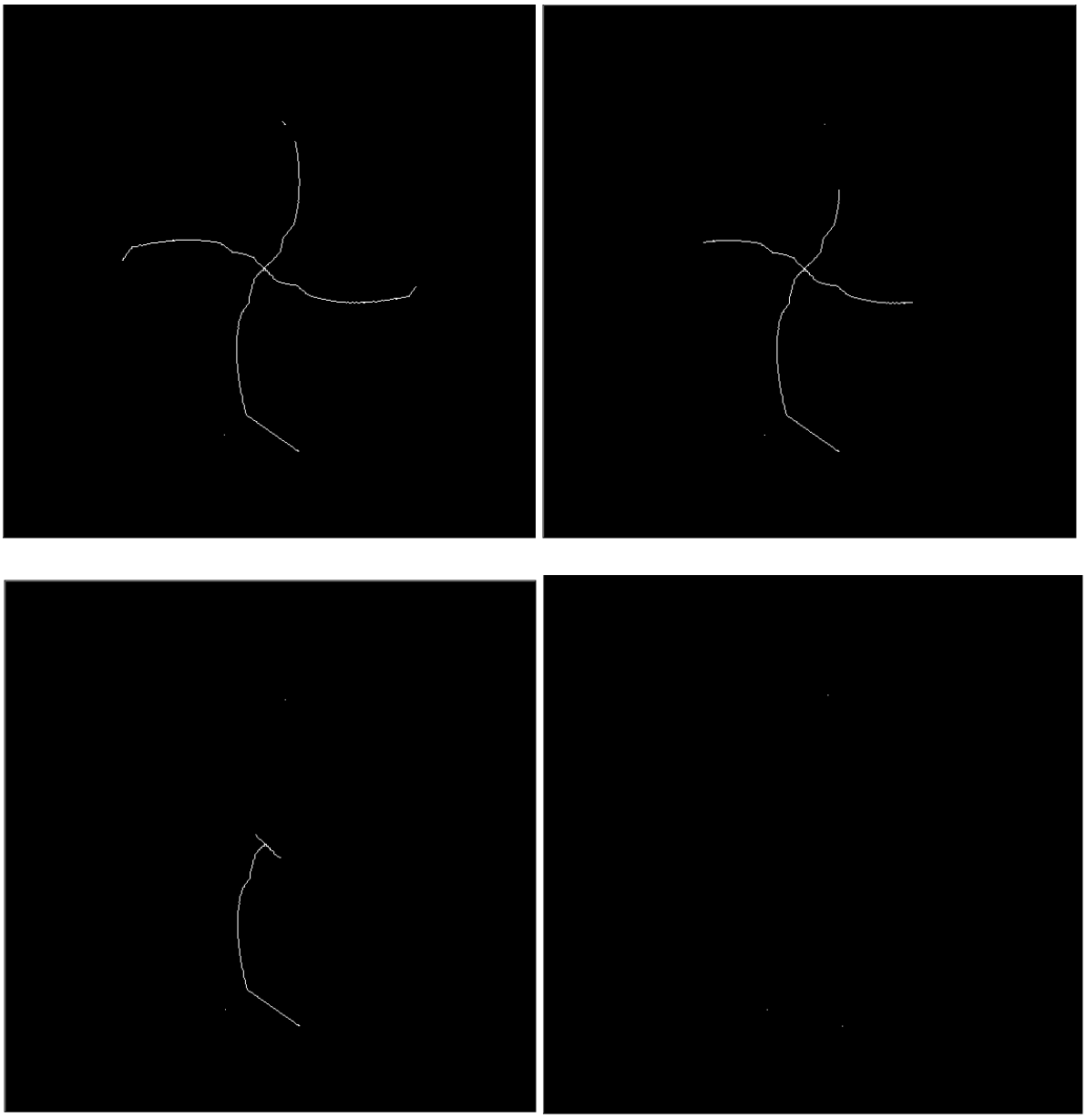


Figure 11: Shrinking of fan.raw with 50,100,200 and 437 iterations

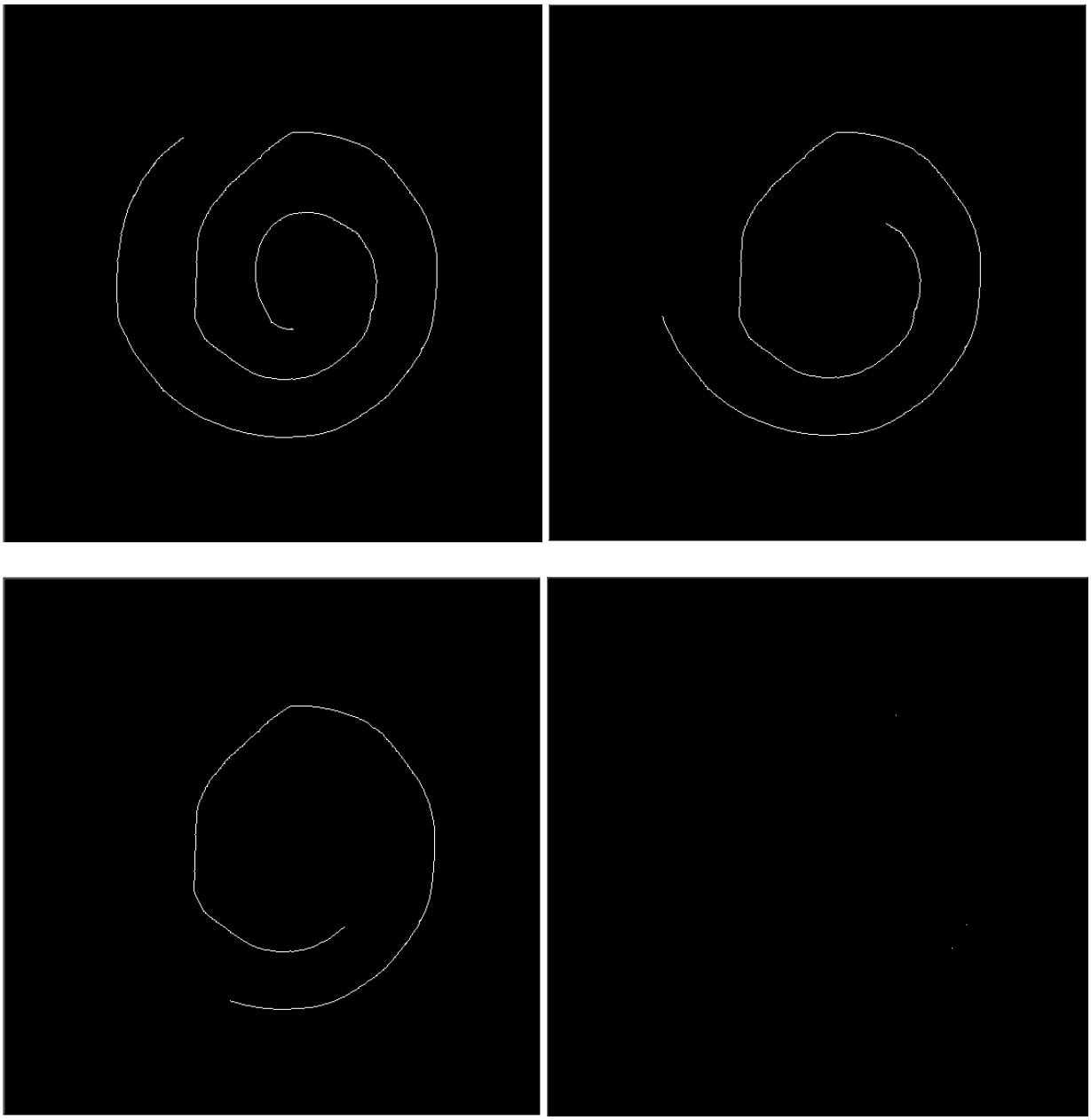


Figure 12: Shrinking of maze.raw at 100,300,450 and 902 iterations.

Skeletonizing:

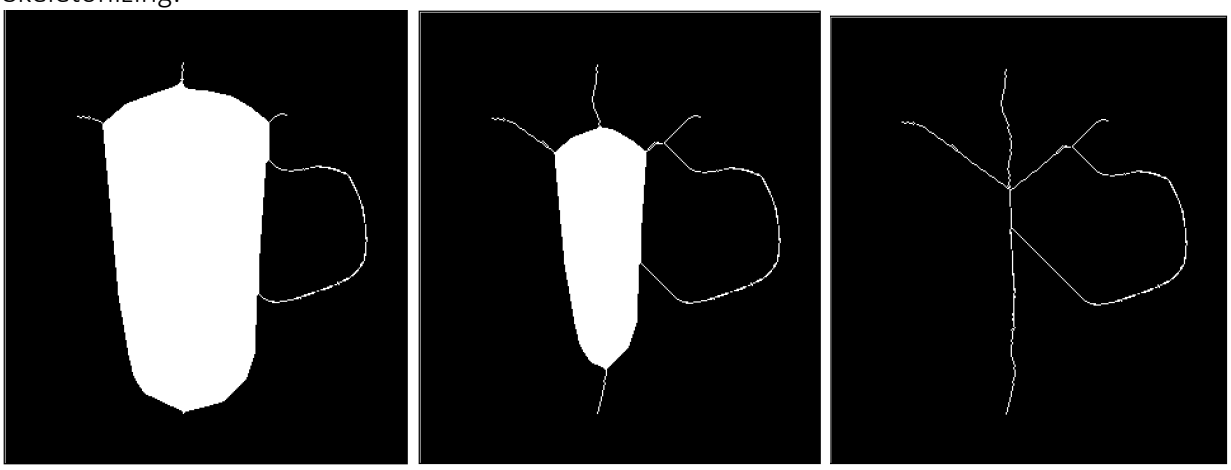


Figure 13: Skeletonizing cup.raw at 20,45 and 80 iterations respectively

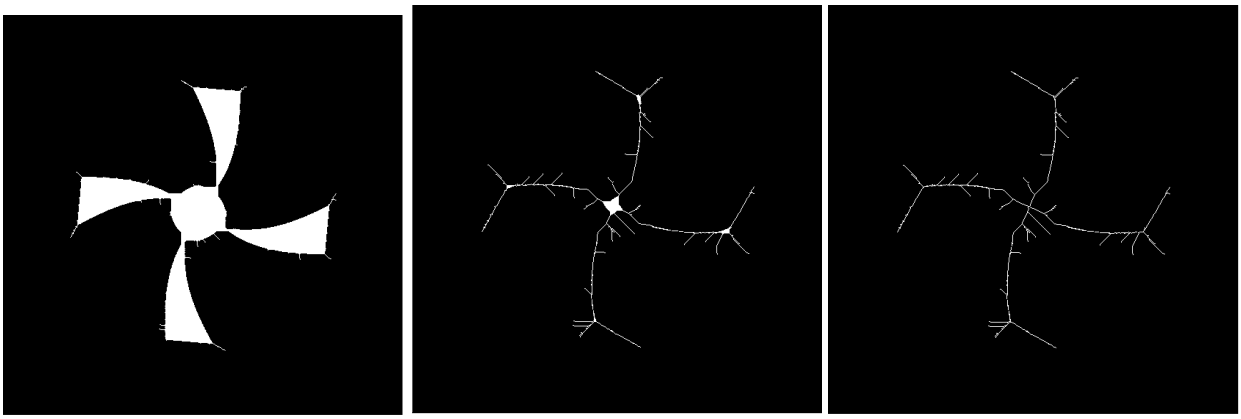


Figure 14:Skeletonizing fan.raw at 10,30 and 35 iterations respectively

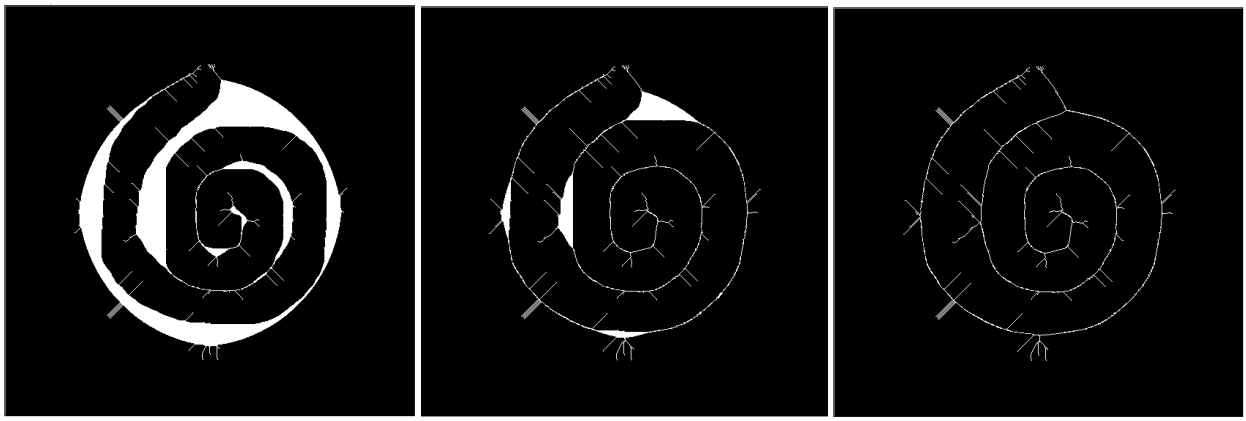


Figure 15: Skeletonizing maze.raw at 20,30 and iterations respectively

Thinning:

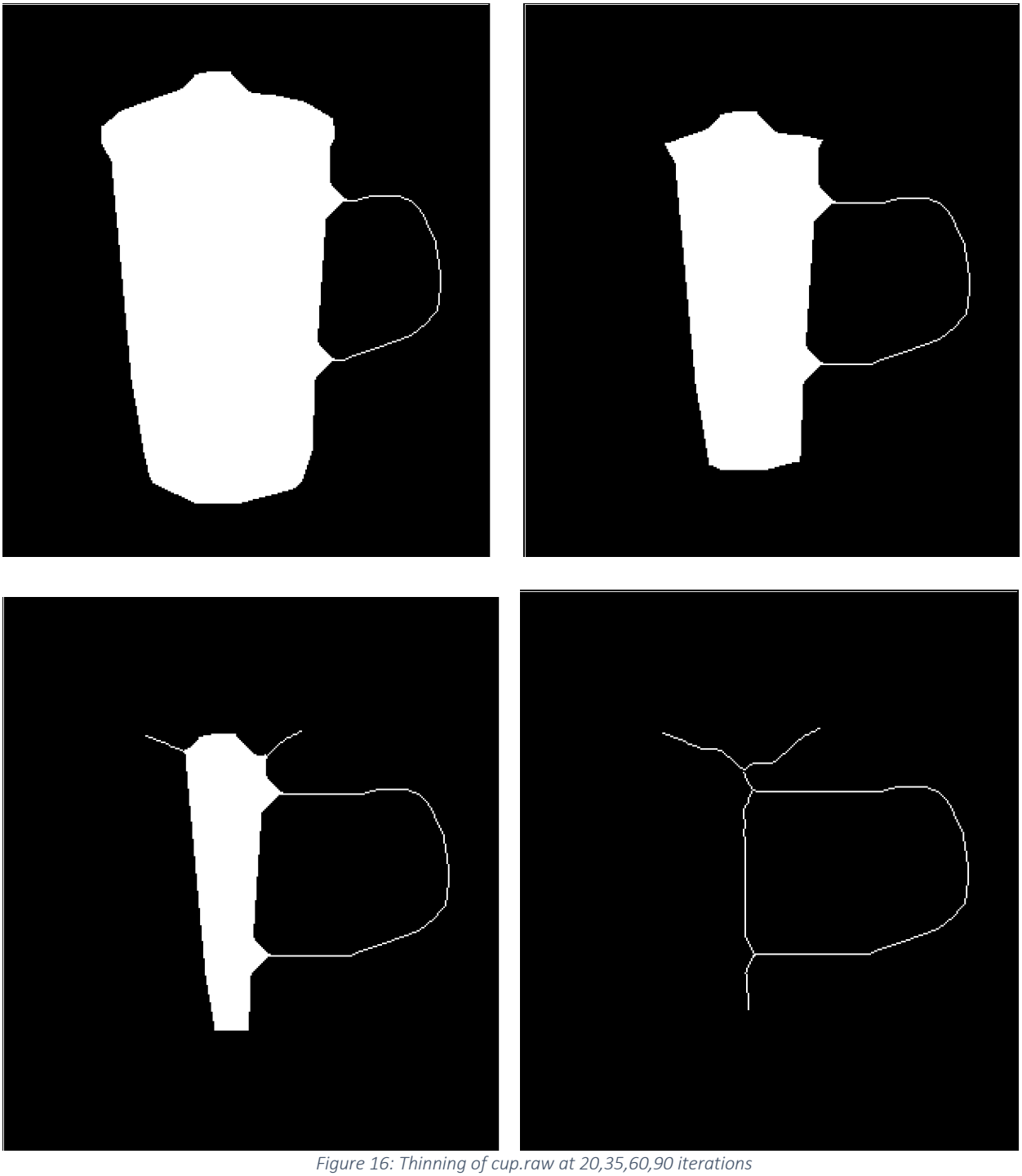


Figure 16: Thinning of cup.raw at 20,35,60,90 iterations

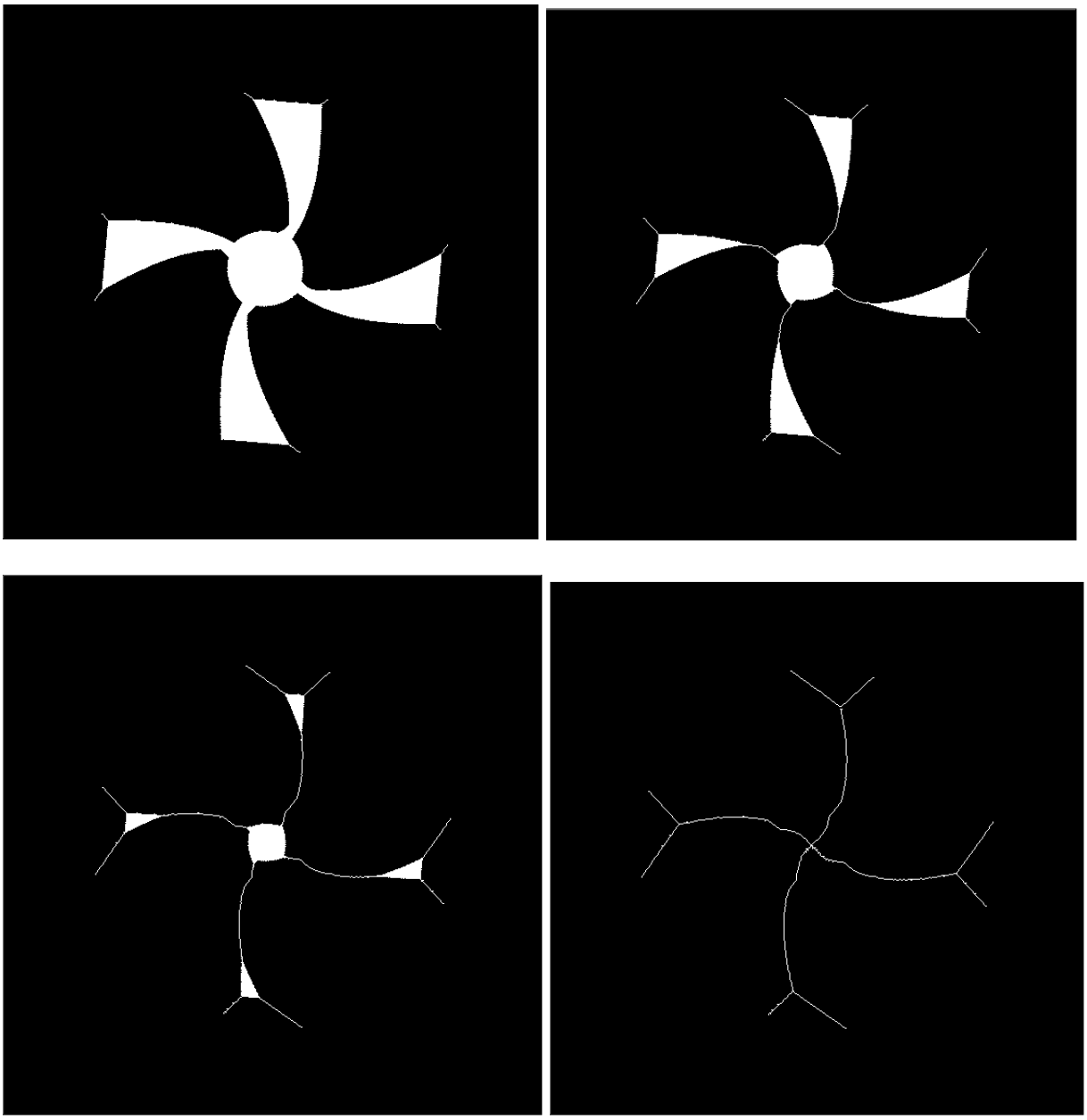


Figure 17:Thinning fan.raw at 50,40,30 and 10 iterations respectively

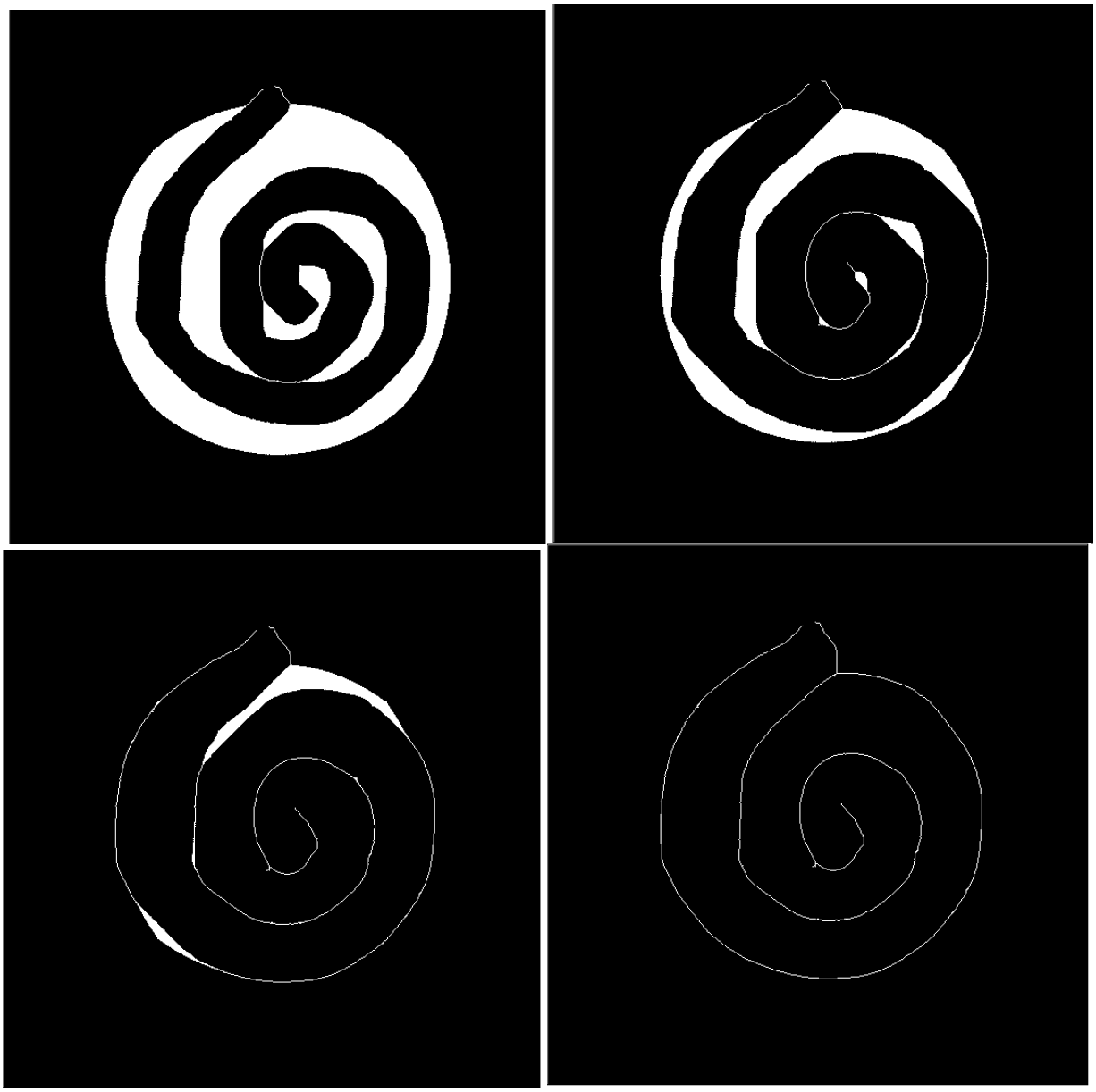


Figure 18: Thinning maze.raw at 50,40,30 and 10 iterations respectively,20, ,30,40,80 iterations

Discussion:

Thinnning:

In the previous section we can observe the thinning effect on the images at various iterations. Here the maze.raw, fan.raw and cup.raw image took a total of 54,50 and 90 to completely convert the image to a thinner version. As the number of iterations increase, we observe that the pixels erode to a thin outline which is approximated to the nearest boundaries. Iterations stop at a point where it further cannot erode the pixels and the lines do not break. Here in thinning we can observe that if there is only one white pixel stroke is left, any further application of thinning algorithm cannot further erode the pixels.

Shrinking:

In the previous section we can observe the shrinking effect on the images at various iterations. As the number of iterations increase, we observe that the pixels erode to form a single white pixel. Here since the given image is reduced to a single point, there is a huge loss of information. This method cannot be used to identify the objects but can be used for counting the number of objects in an image. The cup.raw took around 100 iterations to shrink. Likewise the fan.raw and maze .raw image took around 400 and 900 iterations to shrink. More the number of white pixels in an image, more number of iterations it takes to shrink an image into a single pixel.

Skeletonizing:

In the previous section we can observe the skeletonizing effect on the images at various iterations. The cup.raw took around 80 iterations. Likewise the fan.raw and maze .raw image took around 35 and 40 iterations . The images with more white pixels grouped together takes more time to provide the skeletonized version. Similar to the thinning effect, this method also does not further skeletonize the image once the image has been completely errored to obtain the outline of the image. This method can be used when we want to know the information of skeleton of the objects to detect.

Counting Games:

Abstract and Motivation:

The Morphological processing like thinning, shrinking and skeletonizing can be used for identifying and to know the number of objects in a given image. All these methods have a very wide range of applications from a semi-conductor company to know number of cells in a given tissue.

Here we are asked to count the number of stars in the given image and also give the frequency of the stars in the image of each size.

Approach and Procedure:

As seen in the previous section, we see that shrinking can be used to count how many objects are there in an image. Here we use shrinking method to reduce the stars to a single pixel. Thus, we can count how many white pixels are present and their respective frequency. Here to check the number of white pixels, we use a mask of 3x3 whose middle element is 1 and all its neighbors are 0s.

Alternative algorithm is to implement connected component algorithm to count the number of stars. Here we scan the whole binary image pixel by pixel to identify the connected regions. To label the connected regions, we check if the pixel is 0 or not, we don't consider it. If it is 0 or background pixel. Here we are looking for connected regions towards above and towards left of the pixels. If we find the pixel to be 1, we label it. We continue this process to label all the connected regions. After that, we implement a counter to check hoe many labels we have

obtained. This gives us the count of the connected objects we are looking for. Here , to count the number of stars, we see if the pixel has all neighbors as background pixels.

Algorithm implemented in C++ using 3x3 mask to count the stars:

1. Read the image stars.raw.
2. Convert the input image to a binary image.
3. Apply conditional and unconditional masks to shrink the image.
4. Now implement a for loop to increment the counter in every iteration of shrinking method. The counter at each iteration gives us the frequency of the stars that were left-over after each iteration of shrinking.
5. Implement a counter to count for the number of white pixels present in the image at the final iteration of the shrinking of the image.

Algorithm implemented in C++ using connected component labelling to count the stars:

1. Read the image stars.raw.
2. Convert the input image to a binary image.
3. Apply conditional and unconditional masks to shrink the image.
4. Here I implemented a connected component algorithm to count the stars after shrinking of the image.
5. This gives the number of stars in the given image.

Experimental results:

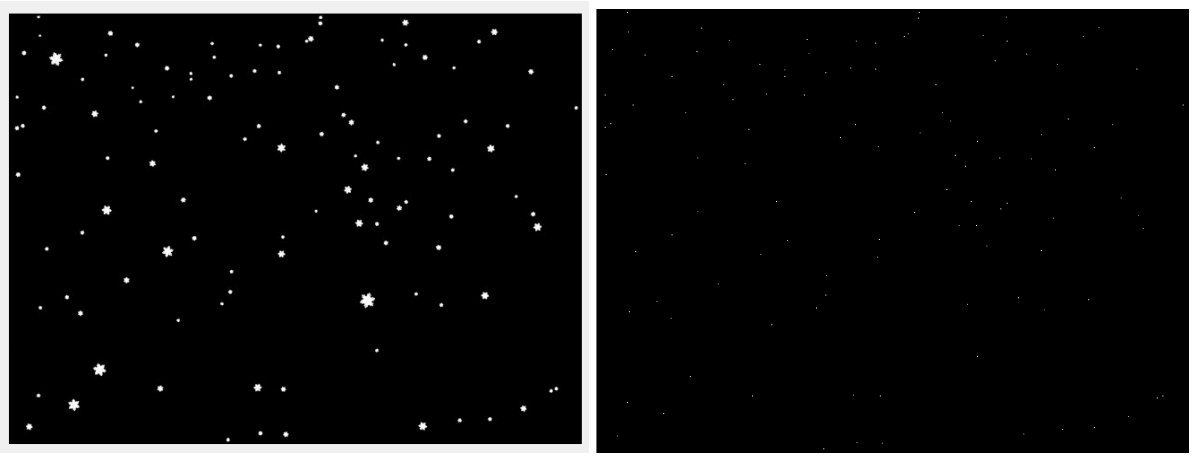


Figure 19: star input and output after shrinking

```
(base) Sonalis-MacBook-Pro:Countingstars sonalisreedhar$ ./1a stars.raw op.raw
The number of stars: 112
Number of iterations at the end of iteration 1: 7
Number of iterations at the end of iteration 2: 51
Number of iterations at the end of iteration 3: 19
Number of iterations at the end of iteration 4: 11
Number of iterations at the end of iteration 5: 7
Number of iterations at the end of iteration 6: 8
Number of iterations at the end of iteration 7: 2
Number of iterations at the end of iteration 8: 3
Number of iterations at the end of iteration 9: 1
Number of iterations at the end of iteration 10: 0
Number of iterations at the end of iteration 11: 2
Number of iterations at the end of iteration 12: 1
Number of iterations at the end of iteration 13: 0
(base) Sonalis-MacBook-Pro:Countingstars sonalisreedhar$
```

Figure 20: Frequency of stars

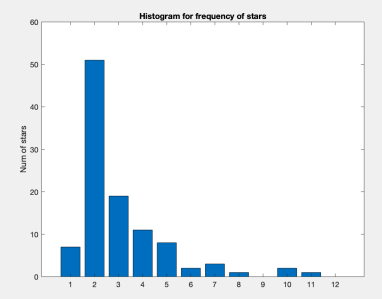


Figure 21: Histogram for frequency of stars

Discussion:

The number of stars were found to be 112. These are the number of single white pixels that remained after performing shrinking on the stars.raw image for 13 iterations. I choose 13 to be the number of iterations as I observed that after 13th iteration there were no difference between the number of stars / white pixels present in the image.

For counting the frequency of stars, I implemented a counter to check on number of pixels remaining at the end of each iterations. Here I have defined the star size to be the number of pixels stars getting reduced at each iteration. The difference in the number of pixels remained in consequent iteration is accounted for the frequency of the stars for each size.

I believe that the number of iterations is important here. If the number of iterations increase, we may tend to lose a bit of pixel information. Also, the threshold we choose to binarize the image also matters. It affects the pixels being shrunken. For increased threshold, I found the number of stars counted to be increased.

To deal with the varying number of stars being counted in the image, we can implement erode function, so that we do not loose information of very small stars.

Alternative method to count the number of stars is to implement connecting component labels algorithm to count the number of stars. Using that method also I obtained the number of stars to be 112.

PCB:

Abstract and Motivation:

In the field of Image processing we might encounter situations to identify different types of objects and count them. In such situations we can use the basic morphological processes to obtain the desired result. These methods are very simple and give out good results.

Here in this assignment we are asked to the number of holes and pathways in the image of a PCB.

Approach and Procedure:

To find the number of holes in the image, we can first shrink the inverted image of the PCB image. We can observe that the holes are shrunken to a single pixel. Now we can count the number of holes by counting the number of white pixels present in the image. To avoid counting the other white pixels as a hole, we can use a mask of 3x3 with the middle element being 1. If the location matches this mask we can consider that white pixel as a hole in the PCB. Basically, whenever we are encountering a single white pixel we have to check if it's neighboring image has 0 pixel value or not.

For calculating the number of pathways, we can do by two methods. First is that, we can use connected component function to count the number of pathways after the image has been shrunken. Second method is to apply a mask of 0 of size 14x14 at the location where we find single white pixel in the image since the hole is found to be of pixel 13x13. Thus, we ensure that we are erasing the holes from the image. Now we will have the PCB image with only pathways and no holes. Now we can use connected component labelling algorithm to find the number of pathways. The number of pathways count varies for the first and the second method mentioned above.

Algorithm Implemented using C++:

1. Read the image PCB.raw
2. Binarize the image.
3. Apply Shrink algorithm and shrink the image by taking the complement of the image.
4. Count the number of single white pixels as the number of holes present in the PCB image.
5. Now find the location of the single white pixels.
6. Apply a mask of 0s of size 14x14 to completely erase the holes from the input image.
7. Now on the obtained image of pathways, apply connected component algorithm to count the number of pathways.

Experimental Results:

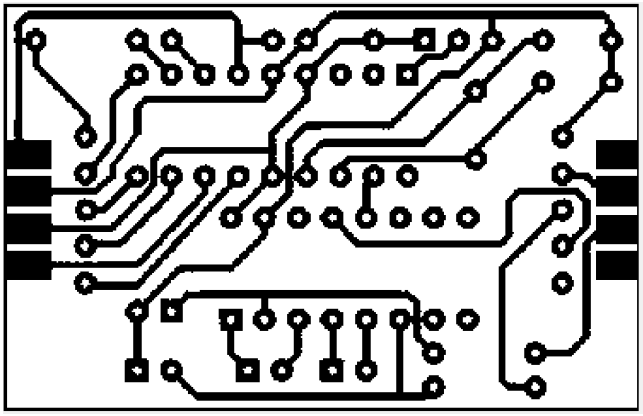


Figure 22 PCB .raw

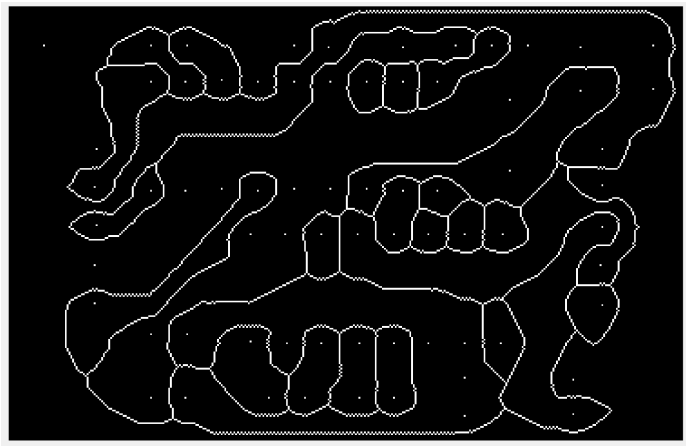


Figure 23 PCB.raw inversed and shrunken image

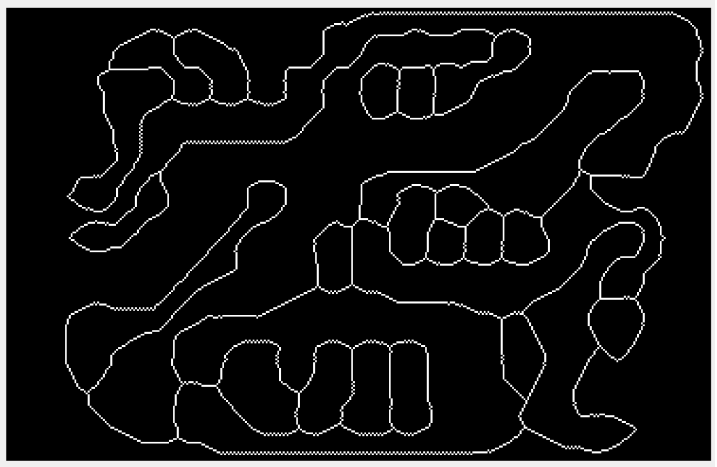


Figure 24; PCB.raw with no holes

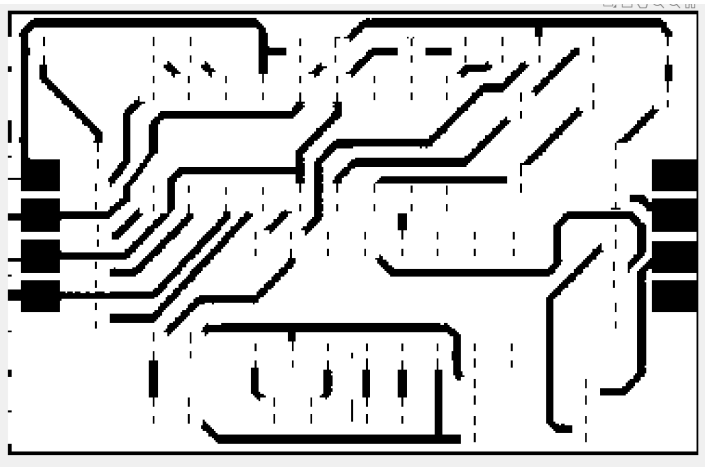


Figure 25: Input PCB image with holes erased

Discussion:

As mentioned above the image is inverted and then shrunken to get single white pixels. This can be seen in the Figure 25. The threshold chosen to binarize the image plays an important role. On increasing the threshold, the number of holes counted also increases.
We got a total of 71 holes by following the above-mentioned method.
For calculating the number of pathways using first method, we found the it to be 73. Here the pathways are not differentiated from holes. Connected Component algorithm is used in to count the number of pathways in the image given below, after the image has been shrunken and the holes have been eroded.

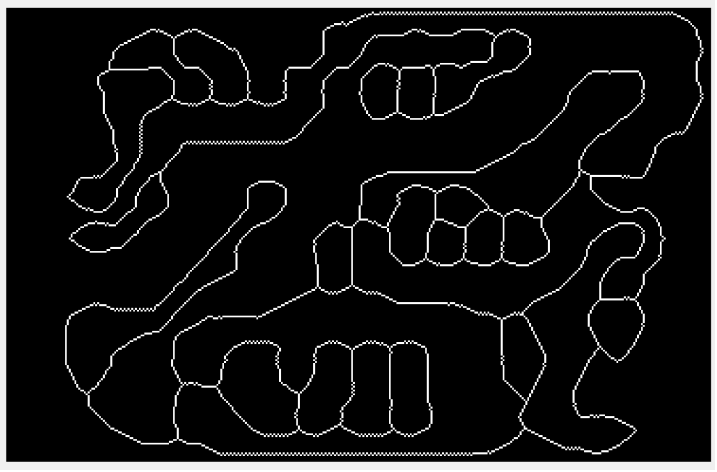


Figure 26: PCB shrunken image with no holes

Using the second method I found the number of pathways to be 35. This again varies with the choice of thresholding. Here I have erased the hole location completely and have made sure to make pathways to be separated from the holes as shown in the below image.

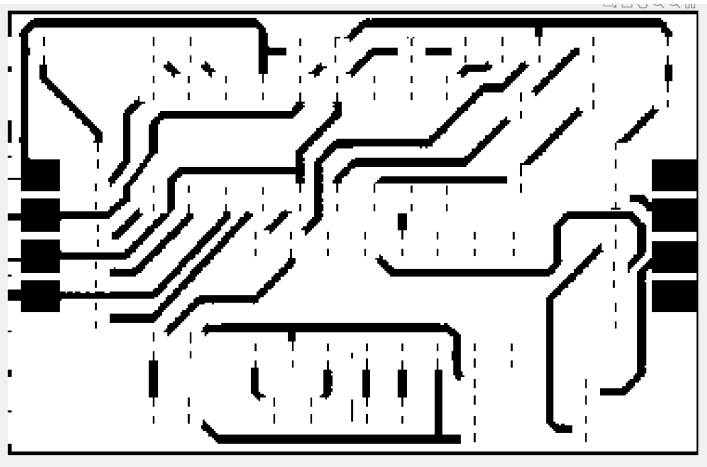


Figure 27: PCB image with no holes - erased

To the above image, when I used connected component labelling algorithm, I found the number of pathways to be 35.

Defect:

Abstract and Motivation:

Defect identification is one of the best applications of image processing. Using morphological processing, we can access and identify if the equipment or any other objects is identified to be defect piece or not. It is more efficient and easier way to detect defects rather than manually testing each object or equipment.

Here in this assignment, we will use some more morphological processing methods and logical operations on the images to identify defect in a gear tooth which has two teeth missing in it.

Approach and Procedure:

The image below is provided to us to locate the missing tooth.

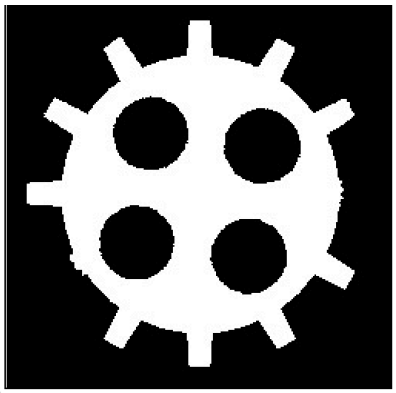


Figure 28: Defect Gear tooth

From the image we can observe that, the first quadrant of the image is perfect and has not defect inti as it has all the teeth intact. So, we will crop the image to obtain only its first quadrant of the image and rotate in -90 degree and 180 degrees to obtain the other quadrant image. Thus, we obtain a perfect gear tooth image without any defect.
After obtaining the constructed image, we use hole filling algorithm to fill the holes in both input image and the constricted image.
Then we subtract the input defect image form the constructed image. This gives the output showing locations of the missing tooth.

Experimental Results:

Figure 29: Cropped input images rotated to create a perfect Gear-tooth

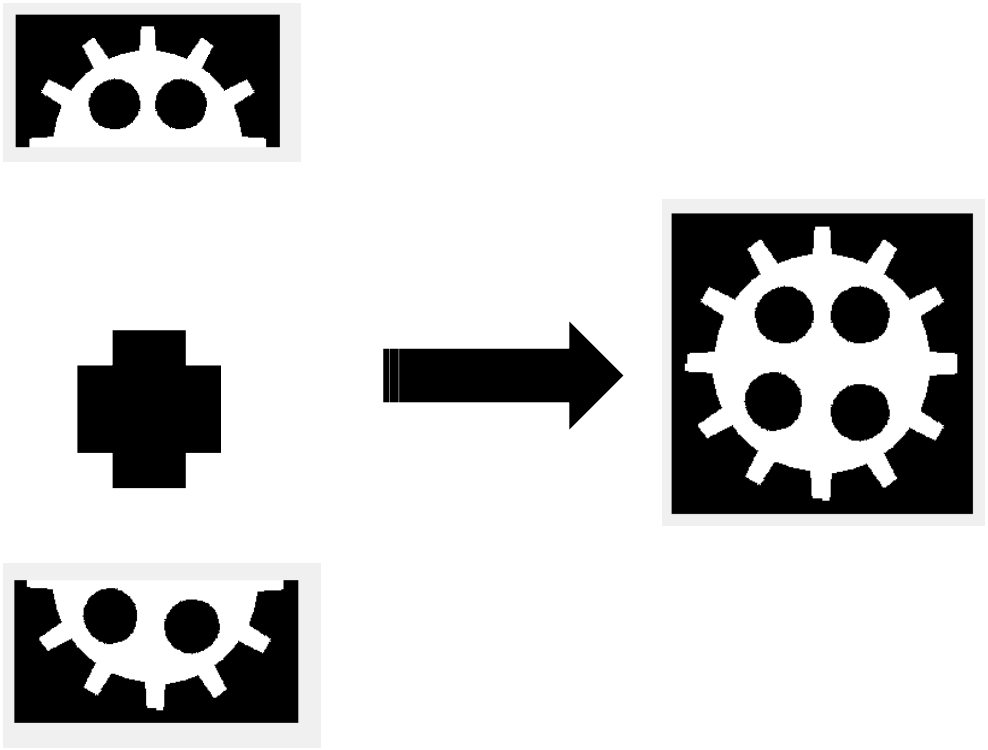


Figure 30: Concatenation of image arrays

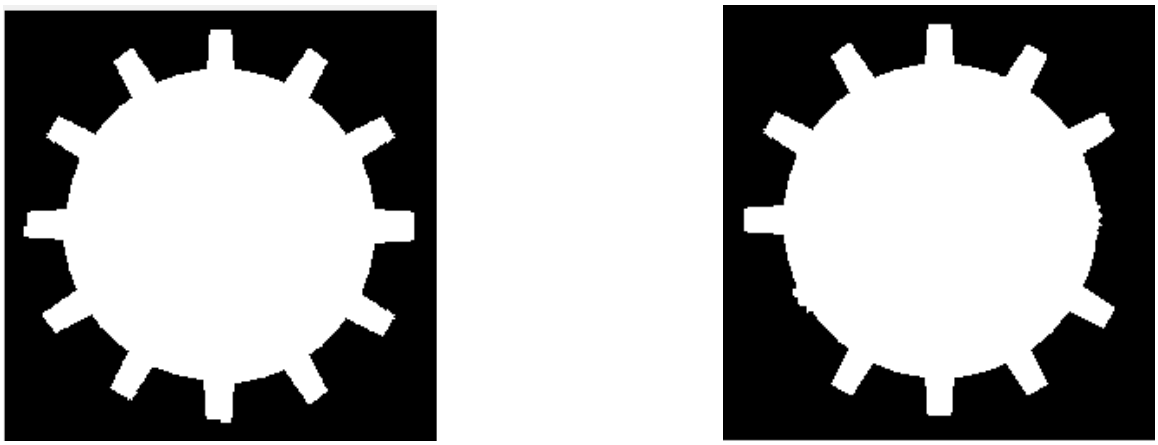


Figure 31: Output of constructed gear-tooth and the input image after hole filling

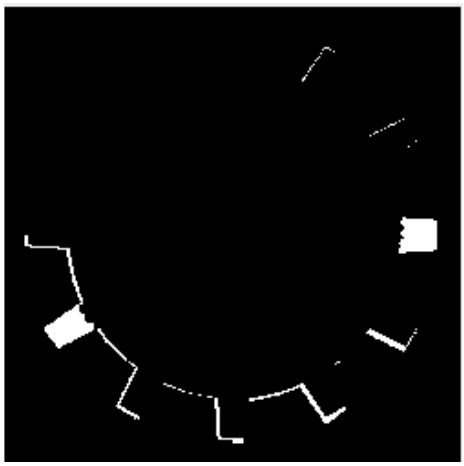


Figure 32:Output indicating the Location of missing tooth

Discussion:

I have implemented logical operations and reconstruction of the images to find out the missing tooth here.

The white pixels remaining in the output shows the location of the missing tooth. The other white pixels are also present as an outline. This may be due to the misalignment of pixels while concatenating the arrays of pixels from above and below. Also, here thresholding of pixels also be accounted as a reason for the same.

References:

[1] MATLAB Panorama example: https://www.mathworks.com/help/vision/examples/feature-based-panoramic-image-stitching.html?searchHighlight=stitching&s_tid=doc_srchtitle
[2] OPENCV feature matching example:
https://docs.opencv.org/3.4/d5/d6f/tutorial_feature_flann_matcher.html

[3] lunch rooms from "PASSTA Datasets", Cvl.isy.liu.se, 2016.
Available: http://www.cvl.isy.liu.se/en/research/datasets/passta/.

[4] https://www.pyimagesearch.com/2016/01/11/opencv-panorama-stitching/

[5] https://arxiv.org/pdf/1509.06344.pdf

[6] https://www.cis.rit.edu/class/simg782/lectures/lecture_02/lec782_05_02.pdf