Homework 3 for 20 spring EE569
Geometric Transformation and Morphological Processing of Images
Chengyao Wang
69615991816
chengyao@usc.edu

Problem 1, Geometric Image Modification:

1. Motivation & Approaches:

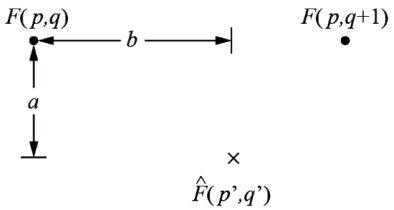
Overall Description:
Geometric modification of images can be used in many real-world applications, where an image is rearranged into different shapes, rotations & pixel distributions, common ones include construction of mosaics, geographical mapping, stereo & video. Pixel relocation rules plays the most important role in those algorithms, and its can be expressed as a spatial mapping function:

$$\emptyset: (x, y) \rightarrow (x`, y`)$$

The domain & range of $\emptyset$ are often 2D / 3D Euclidean spaces, is often a bijective mapping meaning it's invertible and isn't necessarily defined only on integers – however, digital images are. The derivation of $\emptyset$ can either be explicitly conducted by humans analytically, or data driven.

Another important part of such algorithms is interpolation techniques. It mainly serves to address the last problem when we actually want two digital images on two sides of the mapping. Thanks to its second characteristic, we have two ways of getting the transformed image we want: forward mapping & reverse mapping. Like said, $\emptyset$ only relocates the pixels and doesn't change the pixel values. In forward mapping, we map all the pixels of the source image to the target space and use interpolation techniques to fill in the missing values of interest with its nearest with-value neighborhood. Reverse mapping acts more like a query system where for every pixel of interest in the target image, we use $\emptyset^{-1}$ to find its location in the source image. In cases where it lands in between pixels in the source image, we again apply interpolation techniques.

Bilinear interpolation is an easy yet powerful interpolation technique in reverse mapping and is also widely used in the rasterization stage in computer graphics. Since pixel values in source digital image have a grid layout, the mirror image of the pixel of interest in the output image is guaranteed to land in between 4 pixels and bilinear interpolation technique can be expressed as the following formula[1]:



$$\hat{F}(p + \Delta p, q + \Delta q) = (1 - \Delta p)(1 - \Delta q)F(p, q) + \Delta p(1 - \Delta q)F(p + 1, q) + (1 - \Delta p)\Delta qF(p, q + 1) + \Delta p\Delta qF(p + 1, q + 1)$$

Interpolation technique is slightly different in forward mapping situations because the nearest neighbor points of the pixel of interest may not lie on the corners of a square.

Other concerned operations include Index-to-Cartesian & Cartesian-to-Index transformation. This is not mandatory because we can modify $\emptyset$ to produce the same transformation. But such practice makes the process for intuitive because we are representing the pixel value at *Image[i][j]* in the Cartesian coordinates by the its center locations *Cart(i+0.5, j+0.5).* Thus, the overall pipeline of matching can be visualized as follows:

$$ScrImg[i][j] \leftrightarrow ScrImg\ Cartesian\ (i + \frac{1}{2}, j + \frac{1}{2}) \underset{\emptyset^{-1}}{\overset{\emptyset}{\leftrightarrow}} DestiImg\ Cartesian(i`, j`) \leftrightarrow DestiImg[i` - \frac{1}{2}][j` - \frac{1}{2}]$$

Affine Transformation:
Affine transformation is a special kind of linear geometric transformation, which represents three basic linear transformations – shifting, rotation, scaling and their combinations. $\emptyset$ of affine transformation are linear functions thus can be written in matrix form.

$$Shifting \begin{bmatrix} x` \\ y` \end{bmatrix} = \begin{bmatrix} t_x \\ t_y \end{bmatrix} + \begin{bmatrix} x \\ y \end{bmatrix} \qquad Scaling \begin{bmatrix} x` \\ y` \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \qquad Rotating \begin{bmatrix} x` \\ y` \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

[1] "Introduction to digital image processing", William K. Pratt.

Their combinations can also be written in matrix form:

$$\begin{bmatrix} x` \\ y` \\ 1 \end{bmatrix} = \prod_{i=N}^{1} H_i^{3x3} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} x` \\ y` \\ 1 \end{bmatrix} = H^{3x3} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The matrix presenting each basic transformation will collapse into 1 matrix by chain multiplication. And also, worth noticing, matrix multiplication is not commutative meaning the order of the 3 basic transformations cannot be swapped. Since all the H_i^{3x3} is invertible, H^{3x3} is invertible too.

Affine transformation is the easiest to define explicitly by humans. We could calculate H^{3x3} after we determined the overall pipeline of the basic operations without solving any linear equations. But also, we could proceed in a data-driven way by determining at least 3 control points and then the coefficient can be determined solving linear equations. Transition matrices of affine transformations always have their last row to be:

$$[0 \quad 0 \quad 1]$$

Thus, the minimum amount of 3 control points could support at least 6 linear equations, which is sufficient to solve for H^{3x3}. We could introduce more than 3 points and take it as a linear regression problem.

<u>The first part of the assignment 3 doesn't involve any form of affine transformation, but it's used to solve for Defect Detection in the second part.</u>

Homography & Projective transformation:
Affine transformation is the special case of projective transformation. Projective transformation is also classified as linear transformation thus also possesses a matrix form but has two more degrees of freedom compared to affine transformations. The increase in model capacity makes it capable of transforming rectangles to at least any convex-shaped quadrangles rather than just parallelograms, which is used here for image stitching where we need trapezoidal outputs. Situations for concave quadrangles still needs further investigation. It can no longer be easily specified by humans and is more suitable to be approached by data driven methods. Obviously, we need at least 4 control points to form sufficient number of equations to solve for all the degrees of freedom[2]:

$$s_i \begin{bmatrix} x_i` \\ y_i` \\ 1 \end{bmatrix} = \begin{bmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ H_{31} & H_{32} & H_{33} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

In the case where 4 control points are used, coefficients in H^{3x3} is analytically solvable. The overall procedure is already presented in the problem statement section thus is not repeated here. The direct equation sets to solve after rearrangement is as follows[3]:

$$\begin{bmatrix} x_0 & y_0 & 1 & 0 & 0 & 0 & -u_0x_0 & -u_0y_0 \\ 0 & 0 & 0 & x_0 & y_0 & 1 & -v_0x_0 & -v_0y_0 \\ x_1 & y_1 & 1 & 0 & 0 & 0 & -u_1x_1 & -u_1y_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -v_1x_1 & -v_1y_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -u_2x_2 & -u_2y_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -v_2x_2 & -v_2y_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -u_3x_3 & -u_3y_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -v_3x_3 & -v_3y_3 \end{bmatrix} \begin{bmatrix} H_{11} \\ H_{12} \\ H_{13} \\ H_{11} \\ H_{22} \\ H_{23} \\ H_{31} \\ H_{32} \end{bmatrix} = \begin{bmatrix} u_0 \\ v_0 \\ u_1 \\ v_1 \\ u_2 \\ v_2 \\ u_3 \\ v_3 \end{bmatrix}$$

In this problem, the control points are suggested by SURF/FLANN which are inevitably contaminated by noises and outliers. Both will decrease the performance of projective transformation because linear models have limited capacity. We could consider all the points given by upstream algorithm and also model it as a linear regression problem. But it's not analytically solvable anymore because it has a non-linear L2 loss function[4]:

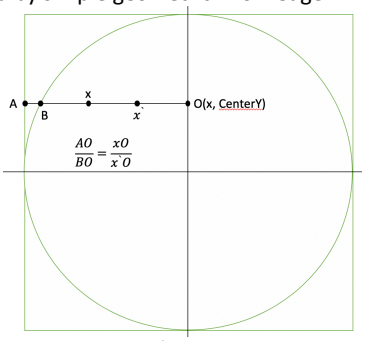
$$Loss = \sum_i (x_i` - \frac{H_{11}x_i + H_{12}y_i + H_{13}}{H_{31}x_i + H_{32}y_i + H_{33}})^2 + (y_i` - \frac{H_{21}x_i + H_{22}y_i + H_{23}}{H_{31}x_i + H_{32}y_i + H_{33}})^2$$

The API of OpenCV to conduct such calculation is *findHomography()* and is believed to approach this problem by gradient descent. It also kicks outliers from the dataset during iterations. The implementation of such process is obviously beyond the scope of this homework.

[2] https://docs.opencv.org/master/d9/d0c/group__calib3d.html#ga4abc2ece9fab9398f2e560d53c8c9780
[3] https://franklinta.com/2014/09/08/computing-css-matrix3d-transforms/
[4] https://docs.opencv.org/master/d9/d0c/group__calib3d.html#ga4abc2ece9fab9398f2e560d53c8c9780

Specific Solutions to the Problems:

In Q1(a), mapping function Ø can be explicitly calculated by simple geometric knowledge:



$$x` = x$$

$$\frac{y` - Width/2}{y - Width/2} = \frac{\sqrt{(Height/2)^2 - (Height/2 - x)^2}}{Width/2}$$

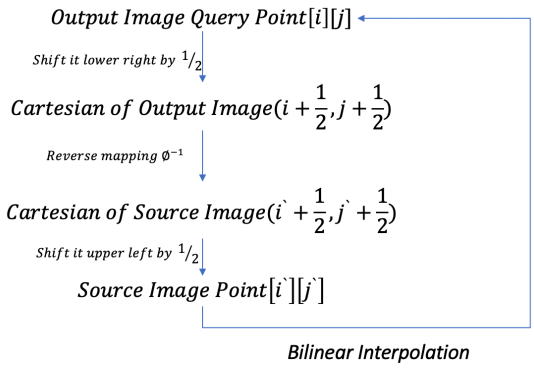
And its inverse for our reverse process is also obvious:

$$x` = x$$

$$\frac{y` - Width/2}{y - Width/2} = \left(\frac{\sqrt{(Height/2)^2 - (Height/2 - x)^2}}{Width/2}\right)^{-1}$$

This function addresses the mapping of the whole image because y & $y`$ are located on the same side of $Width/2$. The function is explicitly specified in the code.

In Q1(b), mapping function Ø adopts a matrix form as afore mentioned and is determined by solving linear equations. Due to the complexity of this problem, we only took 4 control points from the upstream feature generating (SURF) & feature matching (FLANN) makes it easily solvable via matrix inversion & multiplication helper functions. The rest fits into the reverse mapping pipeline mentioned above together with the application of bilinear interpolation – iterate though all pixels in the area of interest to get the whole output image.

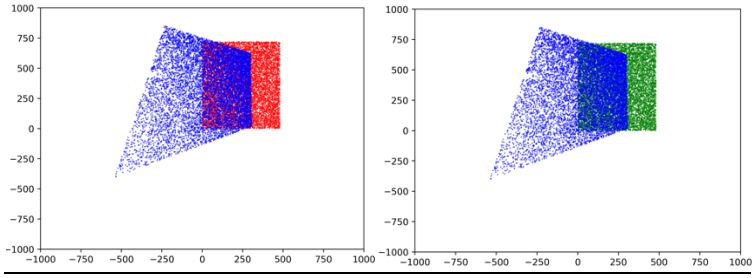


Some Implementation Details:

1. There are always 4 neighbor pixels in the bilinear interpolation stage. ***As long as any one of the 4 neighbors are located outside the image, that pixel is forced to be black***. This is the main source of the small black regions when we reverse the transformed image back.
2. All matrix operations are self-implemented in the ***Matrix_ToolBox*** class as static methods. ***Inverse_Cal()*** is a modified version of an online source code[5] to fit into the whole software system that runs on *pointers*. All of its interfaces run on *float* and there are multiple functions overloaded to address different situations.
3. ***The 8-by-8 matrix on the left-hand side of the equation we're solving is extremely biased and sparse***. Thus, ***Inverse_Cal()*** internally runs on *double* types though it has *float* interface. Type casting in implemented right before and after input & output.
4. In stitching, image is dynamically resized every time after any transforms for time & memory efficiency. An alternative way of addressing this problem will be enlarging the image using dummy black pixels to $\geq 9x$ larger size. Then the true pixels we care about will not likely to run outside the boarders. ***Such dynamic image resizing method is based on the linearity of projective transformation***. After we calculate the transition matrix (& its inverse), we first perform forward mapping on the four corner points to see their 4 corresponding points before reverse mapping the whole output image back. These 4 points provides enough information for ***the size of the transformed image & margin to shift the transformed image to the origin.*** Thus, it will minimize the time complexity by decreasing the number pixels of interest in the output image & memory to store them. No other points in the output image will step out of this boundary. "Any transforms" mentioned above simply means stitching the images together after trapezing is on earth shifting operation, thus is also a type of transformation. This method is applied in both cases.

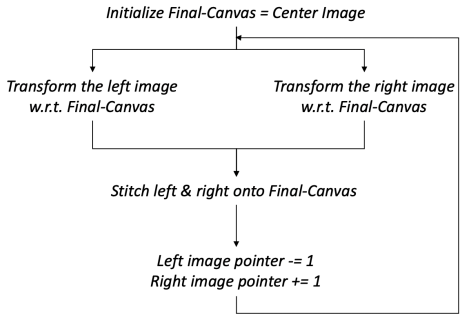
[5] https://www.geeksforgeeks.org/adjoint-inverse-matrix/

5. The control points coordinate that were fed into the matrix is only processed by ***INDEX_TO_CARTTESIAN().*** An example of the direct output of the forward & inverse mapping is visualized by a toy Python script[6] using Monte-Carlo sampling method:



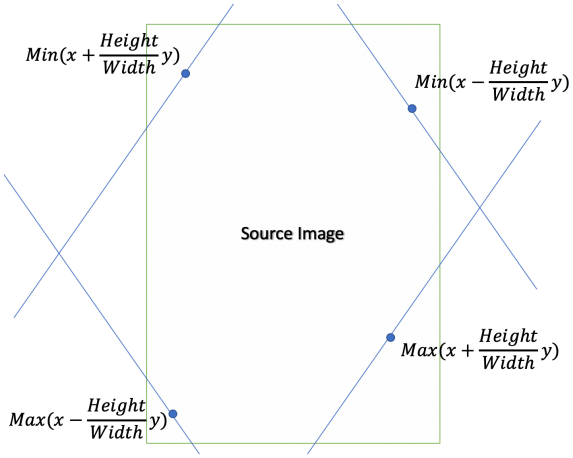
Clearly it also proves that projective transformation is invertible. The dynamic resizing method mentioned in point 4 will address the problem of some point reaching to negative indices during visualization.

6. The overall image stitching procedure is put under an iterative structure, but still lacks the last piece because we are manually choosing the control points. In such procedure, we specify the list of images to stitch as well as the center image.



Each side stops growing if there is no image left on that side, and the whole progress terminates after both sides stop. If we could automate the feature matching selection stage and so will the whole procedure. But sadly, though FLANN algorithm possesses some metric over the quality of every matching it provides by *CV::DMatch.distance*, such score is not the main consideration for choosing good matchings that serves good for projective transformation as discussed later.

7. FLANN & SURF[7] are from official OpenCV tutorial. The underlying mechanism is not the scope of this assignment, thus is not presented here. As can be seen later, the feature matching provided by the two algorithms are not ideal, which means non-noiseless & having outliers. This is the main issue keeping us from automating the feature selection stage. If they are guaranteed to be good, we could easily suggest the following matching selection method:



The basic idea is exactly the same when we choose points manually – we want the control points to locate as close to the corners as possible to alleviate the impact of noises and also make sure they are not outliers.

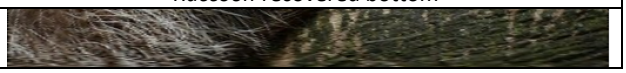
2. Results & Analysis:

The results are put in the appendix for a better organization.

In Q1(a), there are observable differences between original images and recovered image. From a high-level perspective, we lose information when we are performing spatial transformation because the number of effective pixels decreases. The most obvious consequence is the ***horizontal blurring*** of the top & bottom row:

[6] The transition matrix used here is not the same as the one used in final stitching

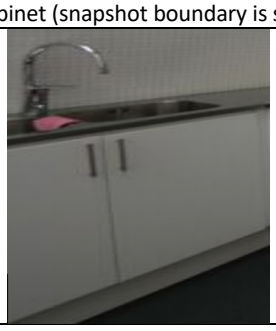
[7] https://docs.opencv.org/3.4/d5/d6f/tutorial_feature_flann_matcher.html

Q1(a) Artifacts	
Hedwig original bottom	*Raccoon* original bottom
	
Hedwig recovered bottom	*Raccoon* recovered bottom
	

For a particular row during recovering, we are trying to construct a whole row of 512 pixels using lesser pixel. The closer to the boundary, less original information (pixels) is given and more pixels in the recovered image come from the same set of neighbors though they each doesn't share the same averaging weights. Bilinear interpolation naturally provides smoothing by taking the mean but can only provide limited visual improvements and still cannot combat information insufficiency. The severe-ness of the blurring is actually linearly related to the number of source pixels given, thus such non-linear disk-shape transformation will restrict the number of heavily compressed rows. Most of the rows could still have a considerable good recovery.

And also, due to the cropping mentioned above in (1), there are some black noises on the horizontal two sides. Generally, the upper / bottom rows tend to have more such noises, but they are not strictly positively correlated. ***Reasons may be related to floating point rounding & discrete arithmetic multiplication / division.*** It also leads to the "spikes" for objects such as the base where the hedwig is standing.

For Q1(b) as afore mentioned in (7), the methodology for selecting control points in the stitching problem can be summarized as: ***(1) they are as much far away from each other; (2) they are not outliers, which means they're valid matchings. There's also an implicit requirement that any 3 of the 4 points should not form a line, which will make the 8-by-8 matrix rank deficient hence un-invertible***. From the ***all_matching*** presented, we could see that there is a considerable number of outliers and they have a relative high possibility to lie close to the boundary. Though the choosing method stated in (7) may be valid in this case, it will not be a universally applicable approach as long as we can't detect outliers. But to the best of our knowledge, outlier detection is quite straightforward if we approach this problem with linear regression mentioned above but not that easy if we are only allowed to inspect the transformed image from the 4 points subsampled.

Close but still is outlier	Obvious outlier
	
Non-vertical cabinet (snapshot boundary is strictly vertical)	Dis-alignment after stitching
	

If inspected carefully, we could notice the imperfectness of the stitching results. 2 examples are presented above. The dis-alignment is not resulted from incorrect shifting distance of the dynamic resizing because the transformed left image has a higher table surface and lower down cabinet. Noises should be responsible for such results, especially for those "visually further" control points, the left two points for the left-middle stitching and the right two points for the right-middle stitching. Because the expanding coefficients are larger on the two sides, and errors will reversely have more impact on the intersection regions create those pixel-level dis-alignment shown above. ***Averaging pixels where the image intersect may alleviate such impact. But here we are putting the middle image on top, covering the left & right because averaging will create blur and double image in the final result.***

Another important factor in image stitching is that cameras will often non-linearly distort the image, due to its physical structure or the optical characteristic of the lens. Thus, we could possibly improve the stitching performance with a quadratic model and of course needs more points.

Problem 2, Morphological Processing

1. Motivation & Approaches:

Overall description:

Morphological processing is widely used in extracting structures of objects or patterns from the images. Since most of the morphological process depends on logical operations like AND / OR, it's only compatible to binary images where the image only has BLACK(0x00) & WHITE(0xFF). Unlike other topics covered throughout the course, morphological processing often serves as a preprocessing step for the convenience of downstream image analysis, thus is more of a standard tool to this course and the theory guided by **Mathematical Morphology (MM)** is out of scope for result analysis & discussion[8].

Three of the basic morphological processing techniques are shrinking (S), thinning (T) and skeletonizing (K). They can be conducted by algorithms that come in difference forms. A two-stage algorithm adopted in this assignment was originally forwarded by William K. Pratt & Ihtisham Kabir[9] and is modified in the textbook[10] by William K. Pratt himself. The graph visualization of the pipeline is:

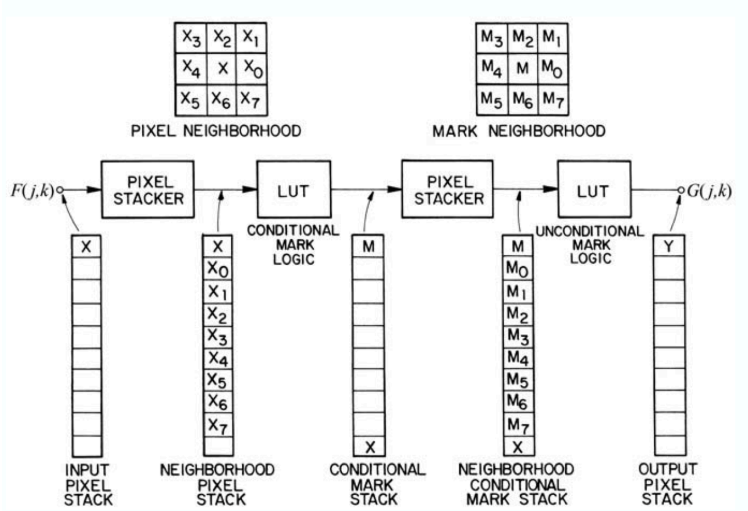


FIGURE 13.3-1. Look-up table flowchart for binary conditional mark operations.

$$G(j,k) = X_{jk} \cap [\overline{M_{jk}} \cup P(M, M_0, \ldots\ldots, M_7)]$$

$$NeighborByte = 0bA_0A_1A_2A_3A_4A_5A_6A_7; \quad \begin{matrix} A_3 & A_2 & A_1 \\ A_4 & X & A_0 \\ A_5 & A_6 & A_7 \end{matrix}$$

The three different operations share the same pipeline with their own conditional mask & unconditional mask pools. Detailed morphological procedure & the pattern table is clearly stated in the problem statement and also the graph above, thus is omitted here. It is done repeatedly before the image stops changing.

The majority of this section is implementation of SKT and its application to perform image analysis. Many standard algorithms like graph search methods are used here and we are not discussing the details.

Some Implementation Details:

1. Both mask matching stage is conducted by bitwise logical operations. It's both easier to interpret and have time efficiency advantages. Each pixel has exactly 8 neighbors thus *unsigned char* is just the right type to do such matching. Conditional masks only contain 0 & 1 elements thus after encapsulating the neighbors of *Img[i][j]* into one byte, we perform XOR operations. We adopted the neighbor encoding order suggested in the textbook:

   $$M_{ij} = True\ if\ \exists m_p \in \{m_q\}_{q=1}^{N}\ s.t.\ InputByte\ XOR\ M_p == 0x00\ /\ 0b00000000$$

   For each unconditional mask, we have three corresponding bytes. Take $0b01AD1C0B$ for example, we have:

   $$M = 0b01001000; M_{01} = 0b11001010;\ M_A = 0b00100101$$

[8] https://en.wikipedia.org/wiki/Mathematical_morphology

[9] "Morphological Binary Image Processing with a Local Neighborhood Pipeline Processor"

[10] "Introduction to digital image processing", William K. Pratt

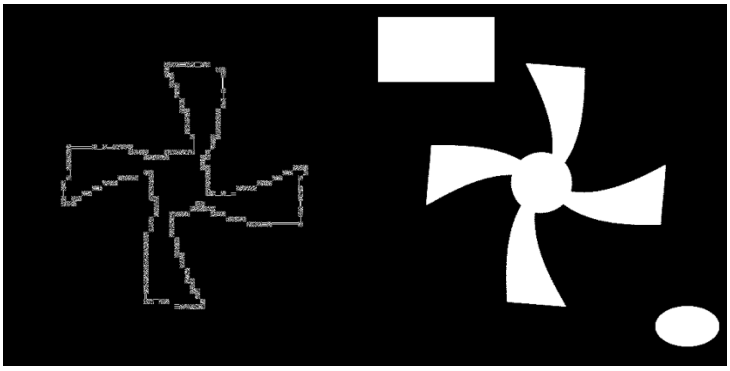
$$Y_{ij} = True\ if\ [(InputByte\ \&\ M_{01})\ XOR\ M == 0x00\ \&\&\ (InputByte\ \&\ M_A)! = 0x00]$$

And like conditional masks, we iterate through the pool of unconditional masks and return *True* if any one match. M serves to match the 0/1 in the *InputByte*, M_{01} serves to extract 0/1 in the unconditional masks and M_A are extracting *ABCs*. M is constructed by setting all except 1 to 0, M_{01} sets all except 0 / 1 to 0 and M_A sets all except *ABC(AB)* to 0. Situations where no *D* or no *ABC* is present:

$$MASK\_FOR\_NOD = 0xFF;\ MASK\_FOR\_NOABC = 0x00$$

In the latter case, the second term is always set to *true* when determining Y_{ij}. Setting $MASK\_FOR\_NOABC = 0xFF$ also seems valid, but $0x00$ is more consistent with its definition.

2. ***The standard stopping criterion of iterations is when the image stops changing. For this, we could easily do this by creating a buffer to store the image from the previous iteration and compare. But here since it's not required by the problems, we just explicitly set the number of iterations. Leaving interfaces to specify the number of iterations also makes life easier for the following problems.***
3. The original images given are not pre-binarized. We have to manually binarize the image before continuing on because the algorithm runs on binary objects. For example, here is a visualization of non 0x00 & 0xFF pixels:



In the first three images in Q2(a), we could naively set threshold to 128. But later in other images, especially *stars*, the choice of threshold will affect the final result. ***We will explicitly give all the threshold used***.

4. We added 1 oval & 1 rectangle to the *fan* image that has no overlapping to the original shape itself. ***It serves as a good reference to determine the correctness of the algorithm.***
5. Size of the stars are later changed to ***the number of pixels in the star***. To the best of our knowledge, there is not beautiful solutions other than brute force to approach this problem. The number of pixels in the star could be easily counted during *DFS*.
6. We implemented naïve attempts to address the problem of discontinuity in *fan* & *cup* based on analysis on the patterns during process. This is done by modifying the ***unconditional*** masks and will be discussed later. But we called it "naïve" because it leads to other problems and it's not that easy to fully correct the pattern tables. Based on our understanding of the original paper noted above, the pattern tables only serve as a tool to present the morphology algorithm clearly and intuitively. We need more **Mathematical Morphology (MM)** knowledge to find out the mistakes that lies in the masks and correct them, because all the masks (including conditional & unconditional) are found to be strongly coupled. It's not simply just a typo.

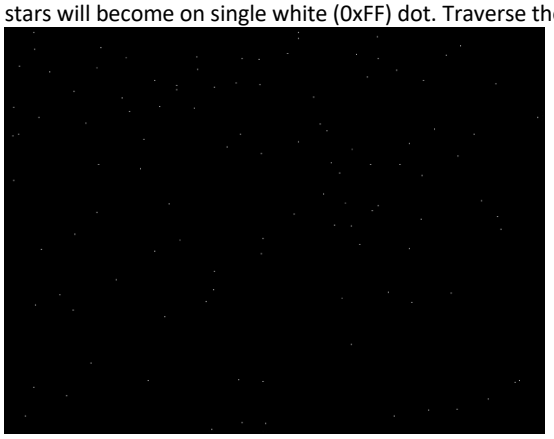
Specific Solutions to the Problems:

Q2(a):

Binarizing threshold = 128.
None. Direct implementation of the algorithm.

Q2(b):

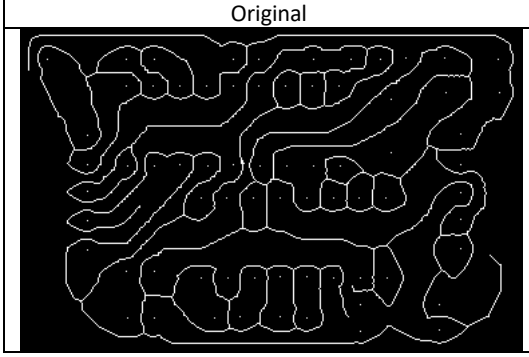
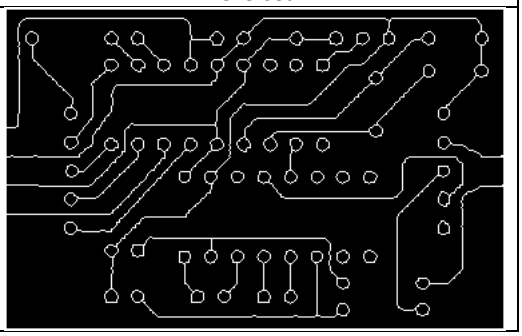
Binarizing threshold = 50.
(1) Perform excessive ***SHRINKING***, all the stars will become on single white (0xFF) dot. Traverse the image and count the number of dots.



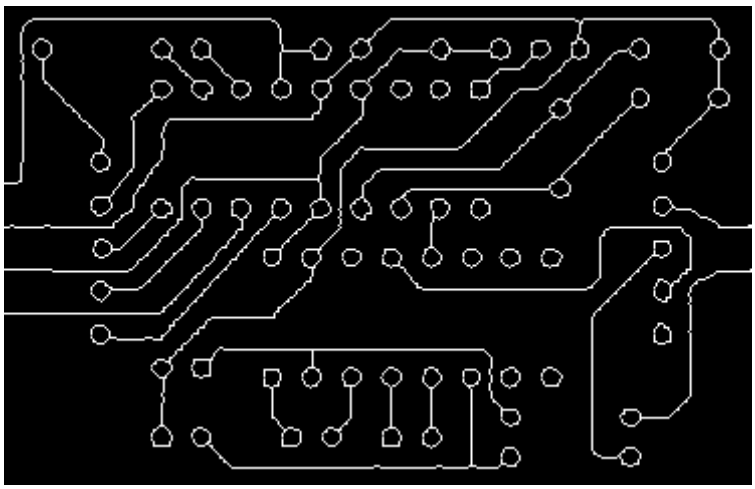
(3) Standard Depth First Search (***DFS***) using recursion. Mark current white 0xFF pixel black 0x00 and call recursive function on **its 8 neighbors**. Recursive function returns if it runs out of boarders or it's already black. The return values vary according to our goals.
(2) *DFS* with the recursive function returns **the sum of the white (0xFF) pixels encountered by its recursive call on its 8 neighbors plus itself.**

Q2(c):

Binarizing threshold = 56. It's exactly the value to disconnect all the adjacent black regions.
Perform ***SHRINKING*** with ***max_iter=50*** on both the original binarized image & its reverse image:

Original	Reversed
	

(1) Perform mask matching like what we done in conditional matching stage with 0x00 every time we meet white (0xFF) pixels when we are traversing the image. **Count the total number of successful matches and it's the total number of holes.**
(2) During (1), record all locations of holes. **Mark the first white pixel to the left of all holes as gray (0x80) in the reversed image.**
Perform *DFS* every time we meet white (0xFF) pixels when we are traversing the reversed image. Add 1 to the return value if it meets grey (0x80) pixels and ***the recursive function will return the number of holes on each wire so far***. If DFS reaches black (0x00) pixels, it returns 0; if DFS reaches the boarder, it returns 1. **If DFS eventually returns a number greater than 1, add one to the counter of wire numbers. Such design will not count the isolated holes that isn't connected to any other hole nor the boundary.**

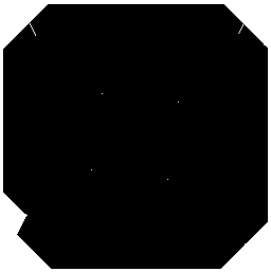


Zoom in to see the grey dots.

Q2(d):

Binarizing threshold = 128.

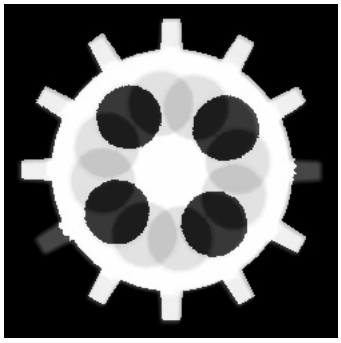
Perform ***SHRINKING*** with ***max_iter=50*** on the reverse image. The black circle holes will shrink to one single white dot.



Likewise in Q2(c), we could get the center of the gear by averaging the 4 centers. Then perform 12 rotations w.r.t the center with $step = 30^{\circ}$. The transition matrix could by explicitly specified as where (x_c, y_c) is the center:

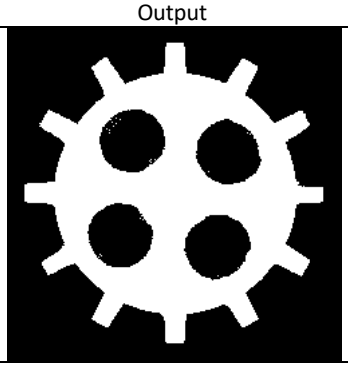
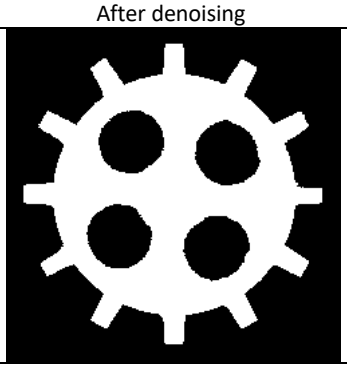
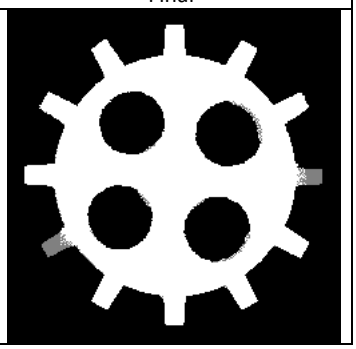
$$\begin{bmatrix} x` \\ y` \\ 1 \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta & x_c(1-cos\theta) + y_c sin\theta \\ sin\theta & cos\theta & -x_c sin\theta + (1-cos\theta)y_c \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

This transition matrix can be directly computed by plugging in θ or by matrix multiplication. We adopted the former because of time efficiency. We define a *int* array to store the sum of the 12 images and add the original image 24 times to the array afterwards. After dividing every pixel by sum of weights (12+24), output the image:



Binarizing threshold = 60. This threshold could be derived analytically by determining how many 0xFF are added to each region / pixel. **But here, we just determine it by trial & error**. Another denoising step is applied to the result to remove isolated dots using mask 0xAA – if a white 0xFF pixel has no 4-connections, we'll remove it:

$$Remove\ if\ InputByte \& 0xAA == 0x00$$

Output	After denoising	Final
		

We'll again read the original image, and for every pixel:

$$Img[i][j] = 0x80\ if\ Img[i][j] == 0x00\ \&\&\ Denoised[i][j] == 0xFF$$

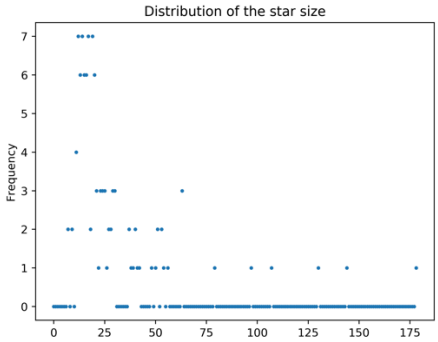
We'll get the final result. This method could be applied to any number of missing teeth except 0, with different number of missing teeth may need different thresholds and the weight of the original image added to the aggregated rotated image. **The *denoised* image could we taken as defect complement.**

2. Comments & Analysis:

It's a snapshot of the terminal output:

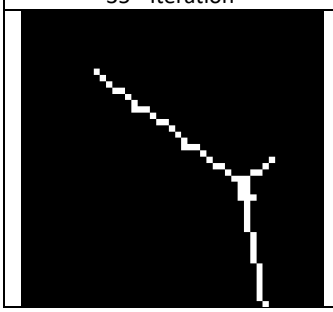
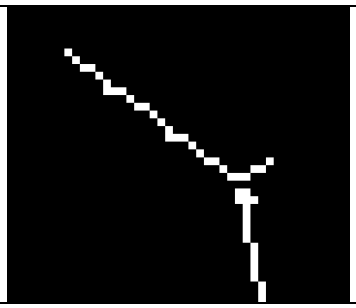
```
The total number of stars counted by shrinking is 111
The total number of stars counted by DFS is 111
The distribution of the star size is:
12 15 40 20 41 29 8 51 13 20
14 24 13 15 16 25 145 12 30 15
13 26 12 18 31 16 14 18 14 10
24 8 13 10 26 13 21 17 43 22
20 31 18 21 19 21 16 23 21 18
16 64 54 13 17 14 22 41 52 17
25 54 15 27 28 17 30 80 12 22
21 52 20 64 18 15 26 25 30 98
19 49 16 38 20 55 179 14 21 13
18 15 28 14 15 131 57 39 31 18
16 17 108 38 20 20 64 42 24 29
17
The number of holes is 71
The number of wires is 25
```

Visualization using simple Python script with ***matplotlib.pyplot***:



Since the morphological transformation algorithm is explicitly specified and this problem is an implementation problem, this part will be presented as comments on the results.

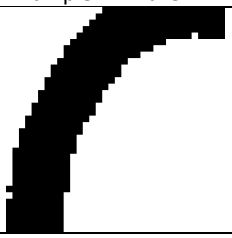
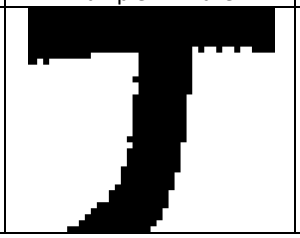
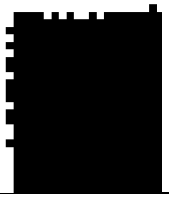
i. Disconnection happens when we are performing SHRINKING & THINNING on *fan* & *cup*, and it's the reason why we're getting two isolated white 0xFF dots after SHRINKING terminates. If inspected carefully, it can be found to be resulted from disconnections during the process in both SHRINKING & THINNING, but since THINNING will not remove the tip pixel at the end of the skeleton it's not that observable. Take *fan +SHRINKING* for example, the problem lies in the 35th & 36th iteration:

35th Iteration	36th Iteration	Conditional mask result in 36th
		$\begin{matrix} 0 & 0 & 0 & 0 \\ 0 & M & M & 0 \\ 0 & M & M & 0 \end{matrix}$

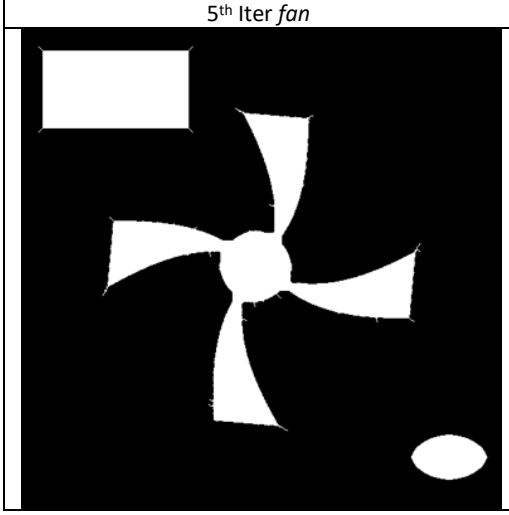
Corner Cluster S/T	*additional Corner Cluster in K*
$\begin{matrix} M & M & D \\ M & M & D \\ D & D & D \end{matrix}$	$\begin{matrix} D & D & D \\ D & M & M \\ D & M & M \end{matrix}$

Red *M* are the two pixel that got swiped away simultaneously that causes discontinuities because they do not match any unconditional masks. *SKELETONIZING* has both two masks, and we changed the unconditional "*corner cluster*" of S/T to the other one that *SKELETONIZING* possesses. This solves the problem as can be seen in the final results, but it introduces "spikes" to the *cup* results. The original "*corner cluster*" will create straight vertical line of the left-hand side of the residual ring, but our new mask will leave 4 stand-out dots. Again stated, this is only one attempt to fix the problem and still needs more work.

ii. We can observe numerous "branches" in the SKELETONIZING results of *fan* & *maze* but not *cup*. Reasons for this lies in the micro-surface of the original images:

Example 1 - maze	Example 2 - maze	Example 3 – fan	Example 4 - fan
			

You will not find those "step-out" black 0x00 or white 0xFF dots on the edges of the *cup* object but there are a lot in the other two images. SKELETONIZING will not touch those dots because they are already taken as skeletons and the algorithm will not shrink them a pixel and leave them as they are. And they may merge anywhere during the iteration process and will stay till the very end:

5th Iter *fan*	5th Iter *maze*
	

These patterns are not included in the conditional mask for SKELETONIZING because it seeks to protect skeletons. Thus, we could possibly address this problem by some preprocessing step to eliminate those “noises”.

iii. STK all works perfectly on the reference objects.

iv. The reason why SHRINKING in *maze* took so long is we are only removing 1 white 0xFF pixel on the tip, and that there is such a long way to go by following the path.

v. Approaches to count the star sizes only by morphological process & logical operations is still not found.

vi. There are 3 slightly different versions of *DFS* used in this problem. They share the same recursive structure and only differs in the return values. A version used in star size counting is as follows:

```
int ee569_hw3_sol::DFS(unsigned char ** scr_Img, int i, int j, int height, int width){
    if(i < 0 || i == height || j < 0 || j == width || scr_Img[i][j] == 0x00)
        return 0;
    scr_Img[i][j] = 0x00;
    return  1 + DFS(scr_Img, i + 1, j + 1, height, width) +
                DFS(scr_Img, i + 1, j - 1, height, width) +
                DFS(scr_Img, i + 1, j, height, width) +
                DFS(scr_Img, i, j - 1, height, width) +
                DFS(scr_Img, i, j + 1, height, width) +
                DFS(scr_Img, i - 1, j + 1, height, width) +
                DFS(scr_Img, i - 1, j - 1, height, width) +
                DFS(scr_Img, i - 1, j, height, width);
}
```

vii. The assumption made for ***defect detection*** is actually quite strong for real world cases, but they are already satisfied in the problem statement – even distribution of the teeth & strict circle shape of the gear. They made the approach implemented feasible. But we could still notice that completion of the missing teeth is slightly smaller than other teeth in size and the upper right black hole is not strictly round anymore. There are ways to address this problem but is not main focus of this problem.

Appendix:

Main function Index (public methods):

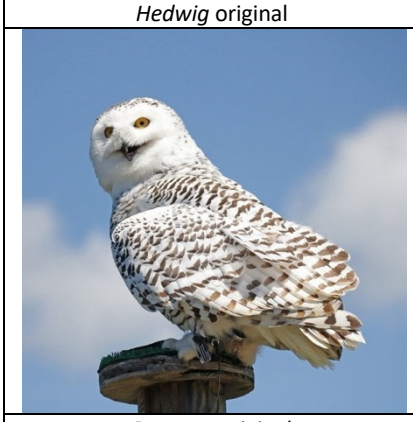
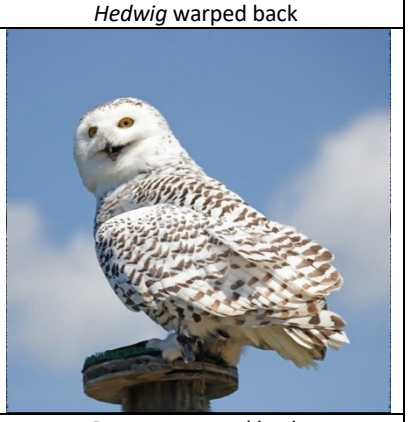
All referenced code is cited in the footnote in the previous sections.

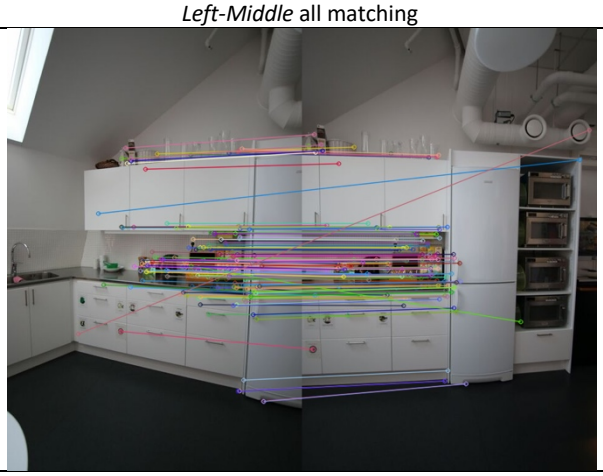
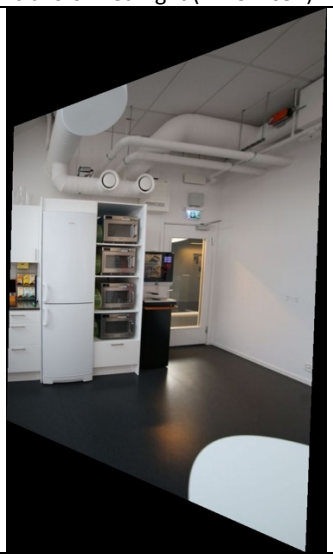
Matrix_ToolBox: Self-implemented class with static methods for matrix allocation & calculation, including matrix-matrix / vector-matrix / matrix-vector multiplication, inversion & transpose.

ee569_hw3_sol:

Geometric_Warping();	Highest level API for Q1(a)
FeatureMatching_FLANN();	Fetch specific number of feature matching
Image_Stitching();	Highest level API for iteratively stitching images.
MorphologicalProcess_Basic();	Highest level API for Q2(a)
MorphologicalProcess_CountStars();	Highest level API for Q2(b)
MorphologicalProcess_PCBanalysis();	Highest level API for Q2(c)
MorphologicalProcess_DefeatDetection();	Highest level API for Q2(d)

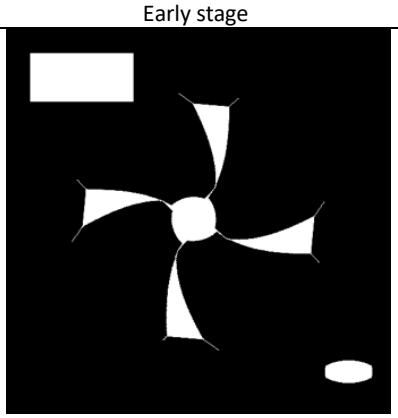
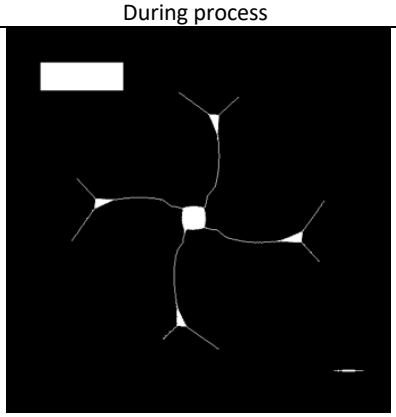
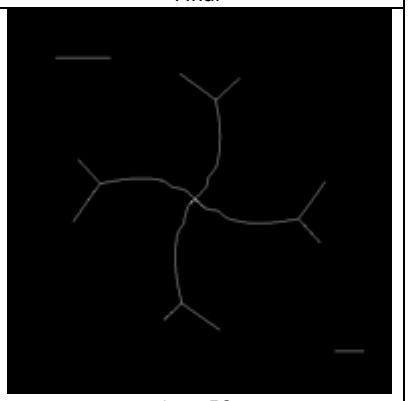
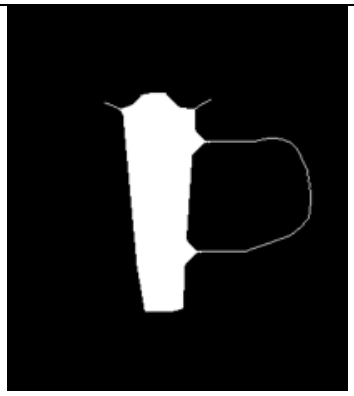
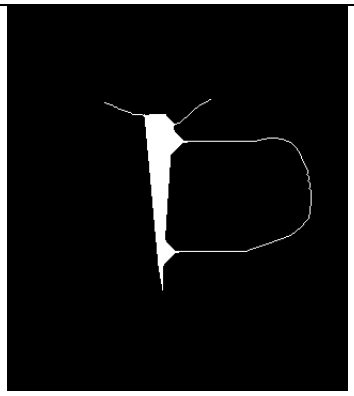
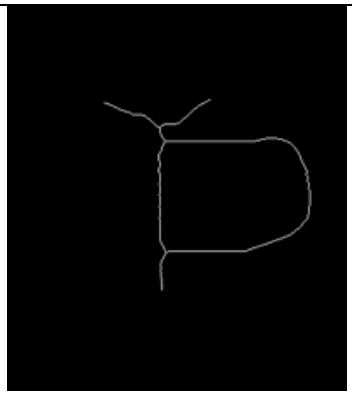
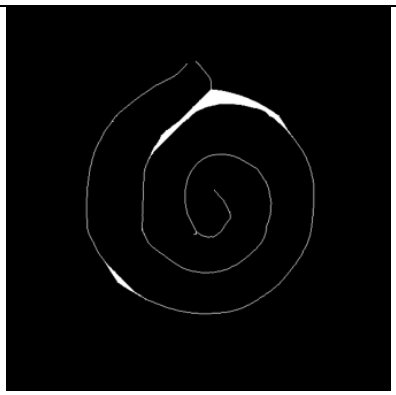
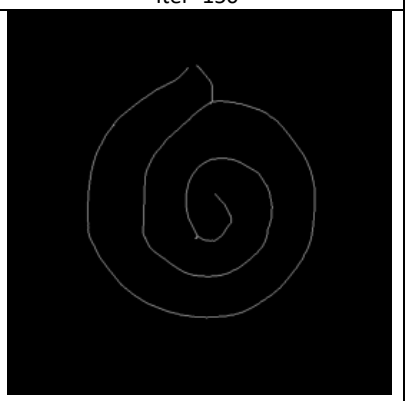
Results:

Q1(a) Geometric Warping & its inverse		
Hedwig original	*Hedwig* warped	*Hedwig* warped back
		
Raccoon original	*Raccoon* warped	*Raccoon* warped back
		
bb8 original	*bb8* warped	*bb8* warped back
		

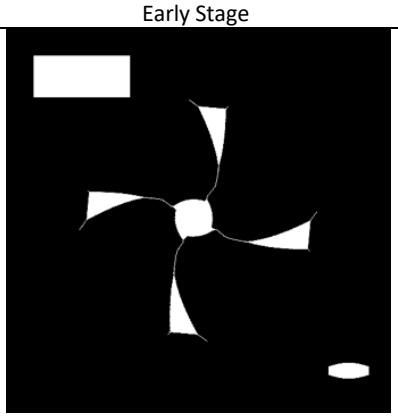
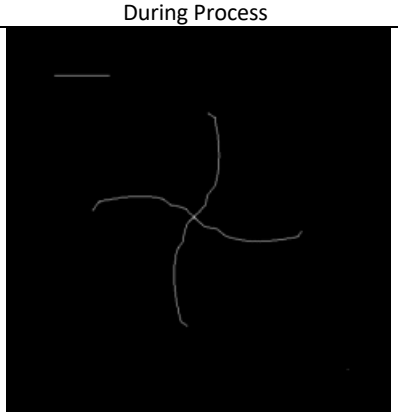
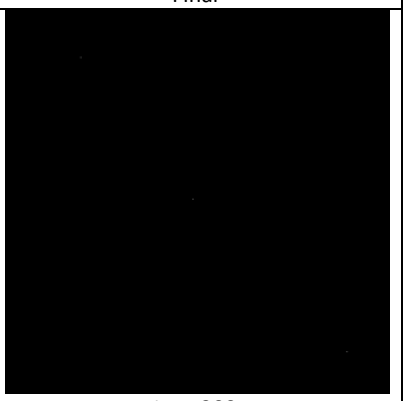
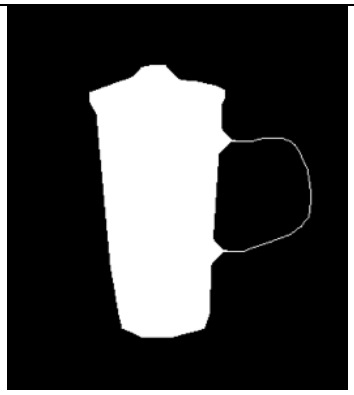
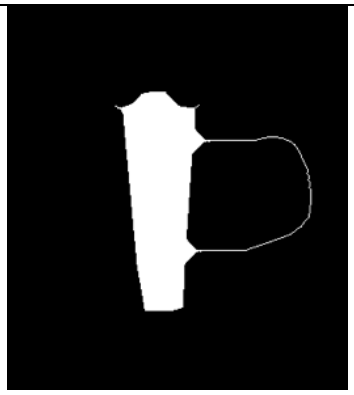
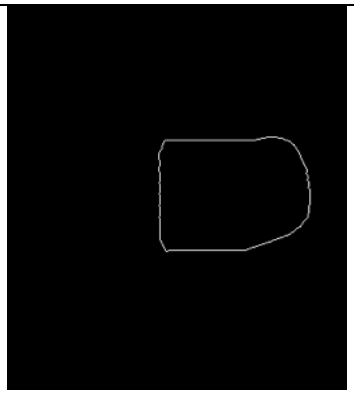
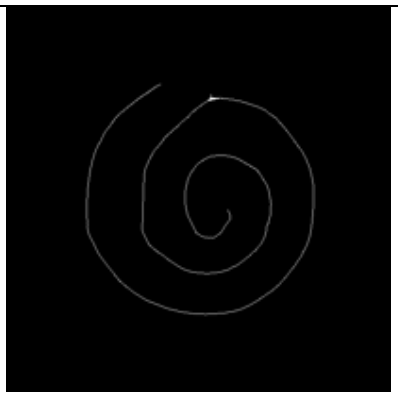
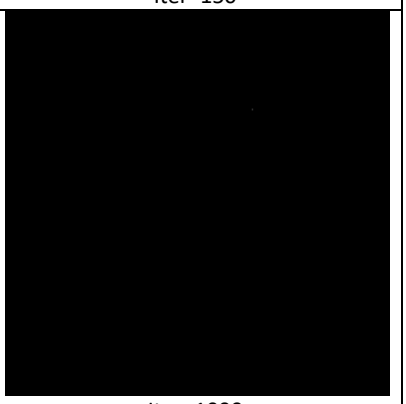
Q1(b) SURF & FLANN results	
Left-Middle all matching	*Left-middle* selected matching
	
Right-middle all matching	*Right-middle* selected matching
	
transformed left (1154×662)	transformed right (1115 x 657)
	

Q1(b) Final result (1160×1166)

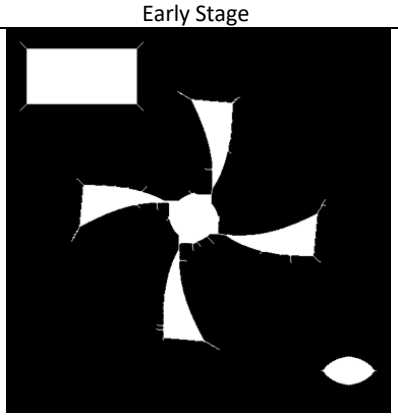
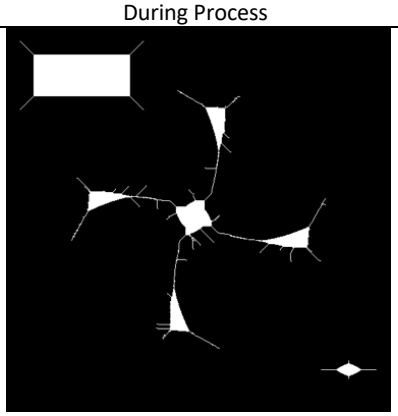
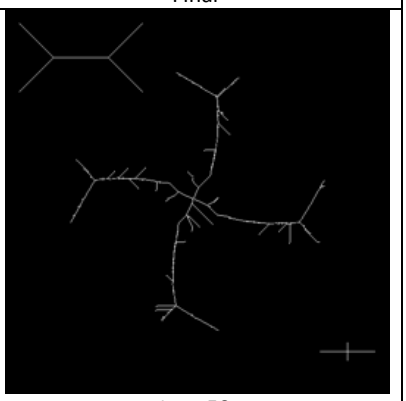
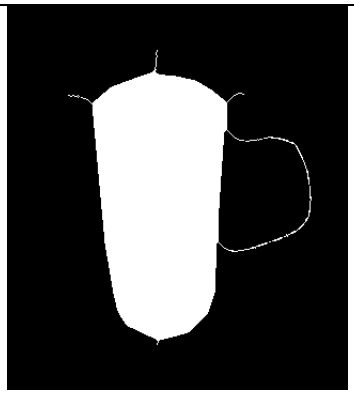
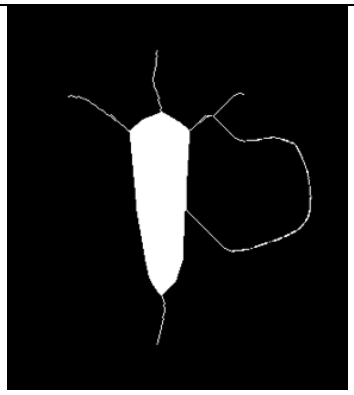
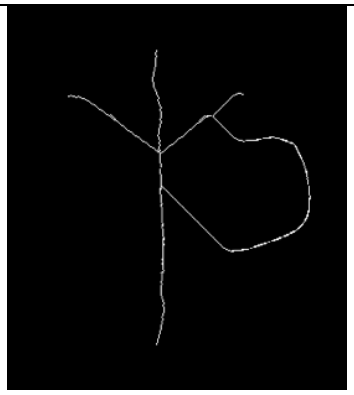
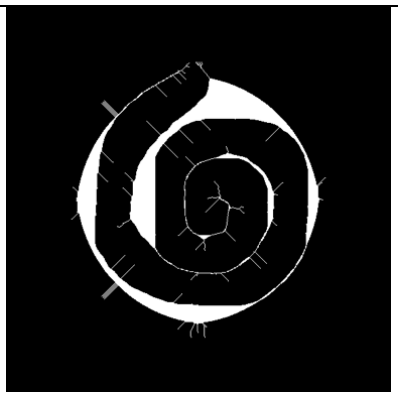
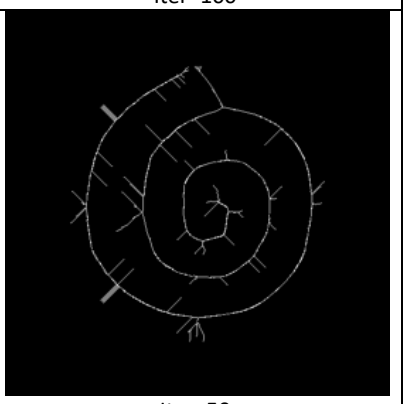
SKT operations of *fan*, *cup* & *maze*
Thinning (T)

	Early stage	During process	Final
fan	 Iter=15	 Iter=30	 Iter=50
cup	 Iter = 50	 Iter=70	 Iter=150
maze	 Iter=20	 Iter=40	 Iter=100

Shrinking (S)

	Early Stage	During Process	Final
fan	 Iter=20	 Iter=50	 Iter=300
cup	 Iter=25	 Iter=50	 Iter=150
maze	 Iter=20	 Iter=50	 Iter=1000

Skeletonizing (K)

	Early Stage	During Process	Final
fan	 Iter=10	 Iter=20	 Iter=50
cup	 Iter=20	 Iter=50	 Iter=100
maze	 Iter=10	 Iter=20	 Iter=50