Yusufu Feierdawusi
4929-7851-40
Feierdaw@usc.edu

EE569 Homework #3
March 10, 2022

Problem 1: Geometric Image Modification

a. Image Warping

1. Motivation

There are many purposes for modifying an image, as we have implemented many different programs to denoise, demosaic and many other operations to correct, clarify and analyze images for our benefit. Image warping is another image modification algorithm to counteract the effect of image distortion and to have an interesting and creative way of creating new artwork [1]. For this specific problem, we are to implement a warping method to modify a square image into a star-shaped image.

2. Approaches and Procedures

In this problem, we first create an empty canvas filled with all black pixels. We then split the original image into 4 different triangle regions with 6 control points for each region. These 6 control points are found by considering the three vertices of the triangle and the median of each of the three sides. Then these points are converted into the Cartesian coordinates using the formula provided in the lecture. In order to warp these coordinates, we must solve the unknown parameters in the relevant equation also provided in the discussion. After solving this equation we can get a mapping of the original (x, y) coordinates onto the new canvas with (u, v) coordinates. However, since the transformed coordinates are mostly in decimal places, we have to use inverse address mapping with bilinear interpolation to avoid empty black pixels compared to simply rounding up or down. Finally, we can use this mapping to produce the newly transformed image onto the empty canvas.

3. Experimental Results

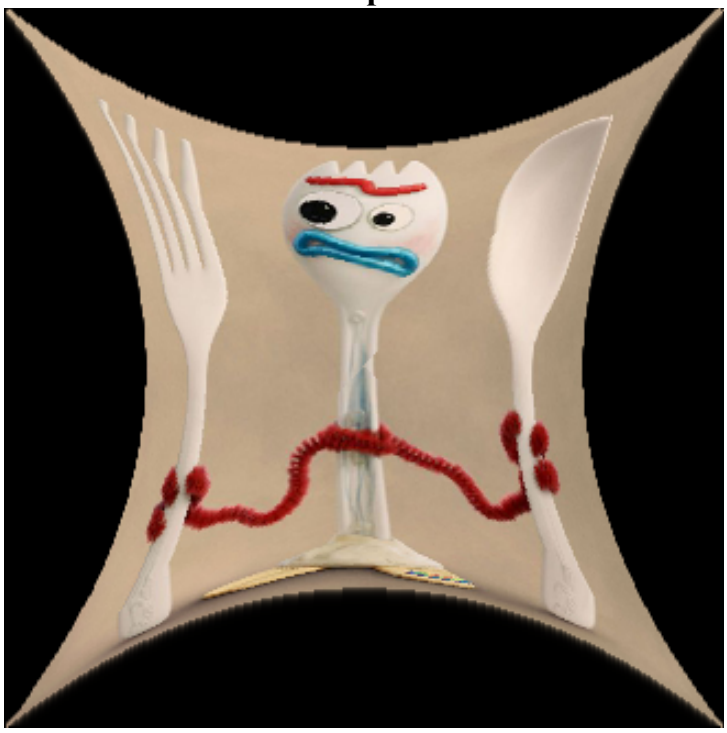


Figure 1: Forky Warped Image

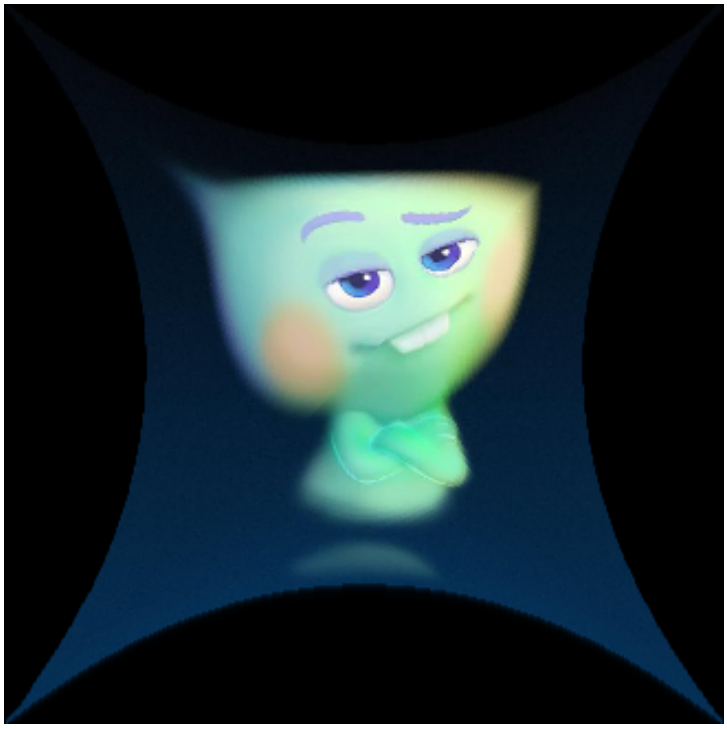


Figure 2: Forky Warped Image

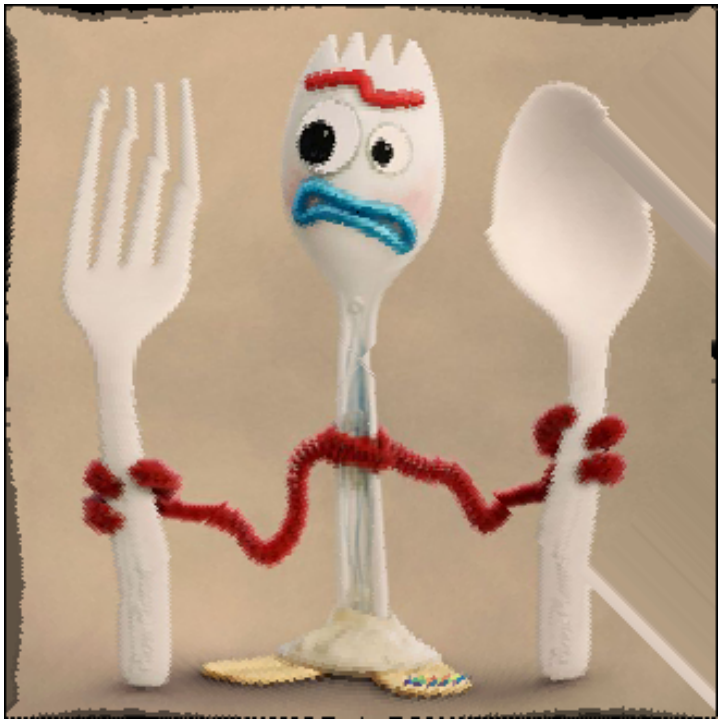


Figure 3: Forky Warped Image

Figure 4: Forky Warped Image

4. Discussion

1). The implementation details and steps of the warping process are described in the Approaches and Procedures section. The unwarping process is similar to warping. We take in the warped result from the previous part, we still choose the three vertices and the median of the legs as our control points, but this time we take the median of the four sides then raise 64 pixels of height to each corresponding coordinate. These would replace our old control points on the largest sides of the triangles. The reason why we choose 64 pixels is because the transformation matrix we solved in the previous step has the input of the curved control points which is 64 pixels closer towards the center. Next, we solve the transformation matrix again, now using the new curved control points as the input and the original control points as output. Then we loop through each triangle region and use the transformation matrix to find a new coordinate and use inverse address mapping to map the RGB value from the input image (which is the warped image) onto that new coordinate in the new image (which is the unwarped image). Lastly, during this process, we also apply bilinear interpolation to avoid new decimal coordinates and dark pixels.

3). There are many distinct and visible differences between the result image produced by reverse warping and the original square image. The first and most distinct distortion we can see in both the unwarped images is the small dark curved regions at the edges of the images (shown below). These regions are most likely produced from the coordinates calculated by the transformation matrix. As mentioned earlier, the resultant of these coordinates are in decimal places, thus the transformation cannot produce perfect results, especially in edge regions where the input image is dark. Another obvious distortion we can see in the unwarped forky image is only in the 4th triangle region, especially the top and bottom of the spoon. I believe there are two reasons for this distortion, one is that the x,y coordinates for this region were traversed badly as my implementation is to loop backwards from the right

edge of the image to the center. The second reason could be the bad result of the transformation matrix in this region, as the other regions are behaving as intended. Similar occurrence happens in the fourth triangle region in the unwarped 22 image.

Problem 2: Homographic Transformation and Stitching

1. Motivation

One of the more popular ways to obtain an image of a large scenery is to take a panorama. In order to obtain an image, we have to first understand what 3D projection is. The basic definition is that we are trying to project a three dimensional object onto a two dimensional plane [2]. We can conceptualize this idea by imagining a box where the walls contain the skewed projection of images, and we are trying to find similar points in order to find a homographic transformation that can stitch these images together on one 2D plane [3].

2. Approaches and Procedures

In this problem, we first create a large enough canvas that can contain all three images (left, middle, right), in this case I chose 2000 pixels by 2304 pixels. Then in order to detect SURF points and features we need to convert RGB into grayscale images. We achieve this by using the grayscale formula to change each channel separately. After calling the in-built detectSURFfeatures and matchFeatures matlab functions we can obtain matching points between left, middle and right images. These matching pairs represent the similar local features between those input images [5]. Next, we can either compare which of these points are more accurate in similarity between the two images or we could also randomly select matching points until we find a transformation that is satisfactory. We can then take 4 pairs of these matching points to calculate the left and right homographic transformation matrices provided in the discussion session. This would solve the 8 unknown parameters in the H matrix. After we obtain this matrix, we can loop through the corresponding left and right image indices and use this as the

input to find the output points that we are supposed to use for forward mapping and bilinear interpolation.

3. Experimental Results

Figure 1: Left Image Transformation

Figure 2: Right Image Transformation

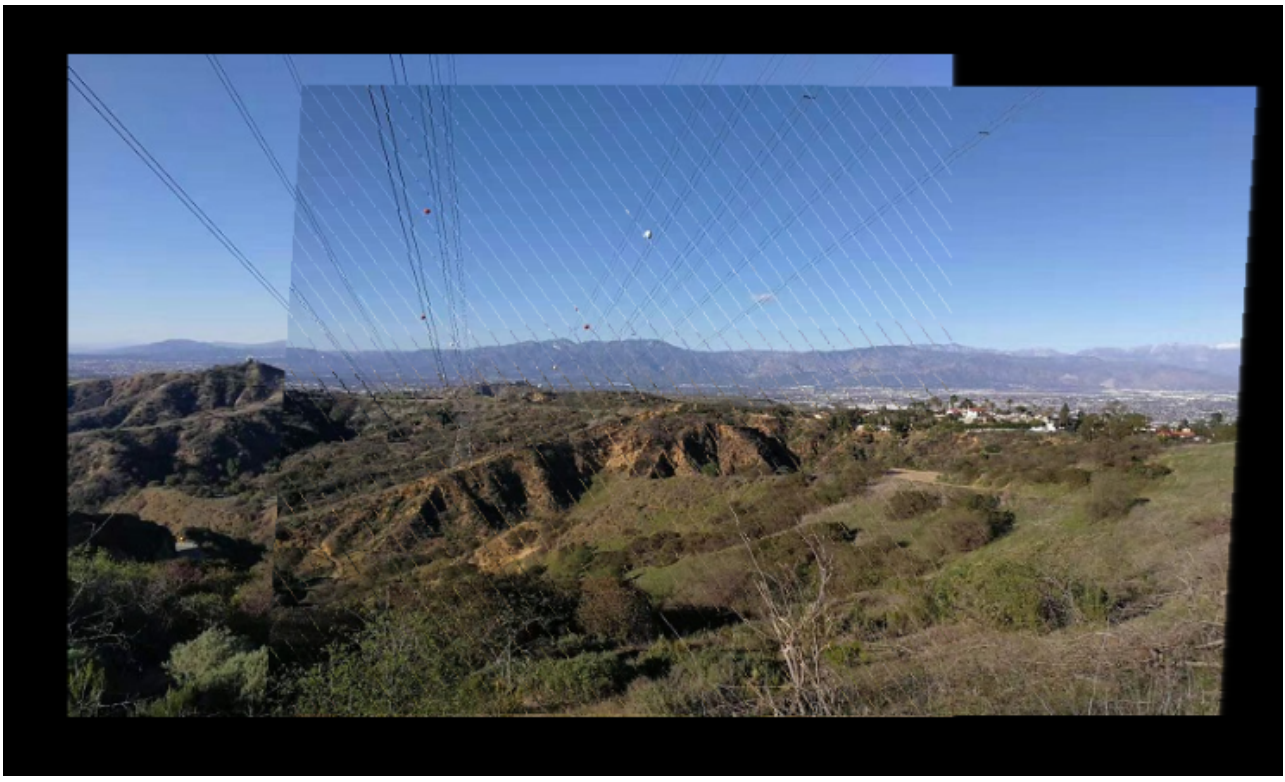
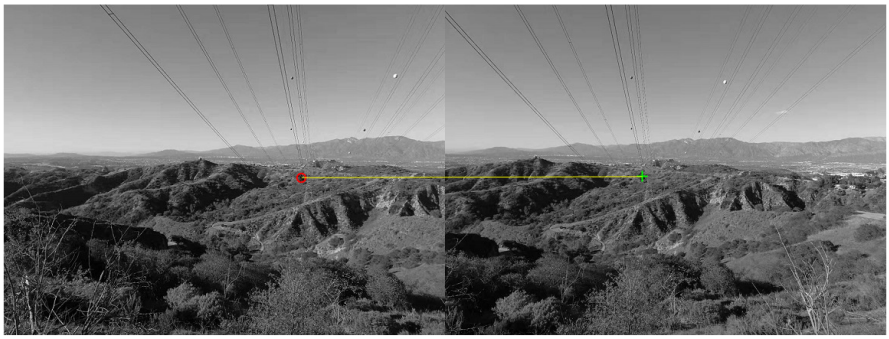


Figure 3: Combined Canvas

4. Discussion

1). 4 pairs of matching points were used for the left transformation and 4 pairs of matching points were used for the right transformation. In total there were 8 pairs of control points used.

Left-Middle Pairs:

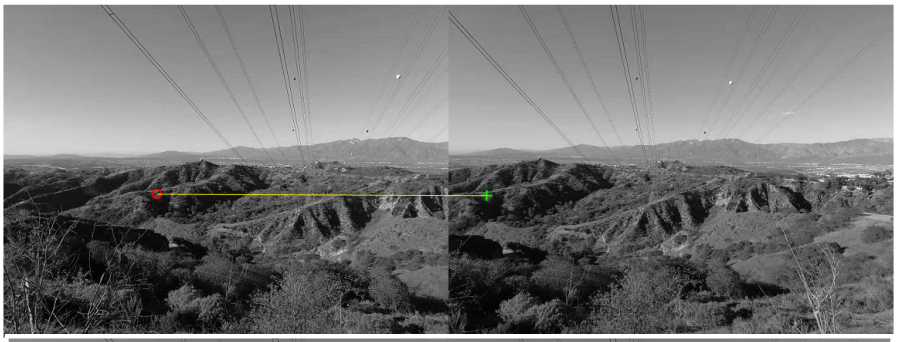


left_matched (5, :) = (389.338, 226.776)

middle_matched (5, :) = (258.17 225.41)

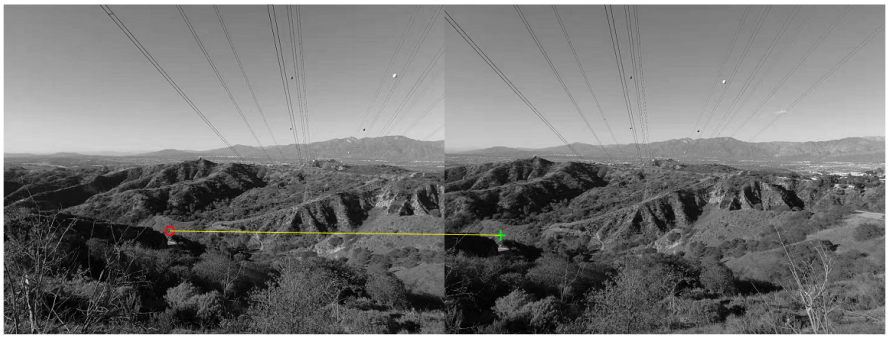
left_matched (10, :) = (531.49 406.78)

middle_matched (10, :) = (383.14 386.92)



left_matched (140, :) = (199.08 246.44)

middle_matched (140, :) = (50.46 248.35)



left_matched (145, :) = (217.75 297.26)

middle_matched (145, :) = (73.88 302.68)

Middle-Right Pairs:

middle_matched (5, :) = (326.5 264.6)

right_matched (5, :) = (176.93 268.83)

middle_matched (10, :) = (355.17 255.37)

right_matched (10, :) = (206.96 256.93)

middle_matched (85, :) = (474.74 267.19)

right_matched (85, :) = (318.16 263.20)

middle_matched (90, :) = (328.17 240.04)

right_matched (90, :) = (178.58 242.25)

2). The control points are randomly selected until I find the one that aligns and transforms relatively close with the middle image. How these matched features are used to create a panorama are explained in more detail in section 2.

3). Discussion on the results: The process of the algorithm seems correct, as seen with the left and right transformation lines up correctly for the most part and creates a half panorama. However, when combining these two transformations onto the middle image, they do not create the desired outcome as a complete panorama. I think that there are two main causes for this mismatch; since a random-pair choosing method is used, the control points for the left image could have been poorly chosen. And the second reason could be that the transformation matrix is poorly calculated, although from the results it seems like the transformation matrices work individually. The diagonal line distortions are the direct result of forward addressing mapping. Bilinear interpolation was applied to counteract this effect, but it only corrected lines outside the boundaries of the middle image.

Problem 3: Morphological processing

1. Motivation

Morphological processing manipulates the objects in an image using morphological operations like thinning, shrinking, eroding and many others. There are

many different uses for morphological processing like detecting defects within an image. One of the creative uses of this processing is to morph one object into another like faces or animals [6]. In this section, we mainly focus on thinning and shrinking operations to remove defects from an image and create segmentation masks to separate objects in an image.

2. Approaches and Procedures

a. Thinning

In this problem, I first hard coded each mask and all of the possible permutations of both pattern tables (stage 1 and 2) as 3 by 3 matrices. Then I binarized the input image and stored it into a new image matrix with zero padding. Next, I created an iteration loop, and within this loop, I first declared and initialized an M matrix filled with zeros that re-initializes itself for every iteration. Furthermore in this iteration loop, I create another loop that iterates through the binarized image with zero padding. For every index within this loop, I create a 3x3 matrix that contains all the neighbors of the current index. I also calculate the bond based on the values of these neighbors. Using that bond value, I compare this 3x3 patch with all of the masks corresponding to thinning and that specific bond. If a mask matches the pattern in this patch, I assign 1 to the same index in the M matrix, else it is left as 0. After this convolution, we should obtain a M matrix with ones and zeros. Then, we create another similar loop for stage 2 pattern matching, where we get a 3x3 neighboring patch of the M matrix using the current index. We then compare this patch with every single mask available in the thinning unconditional pattern table. If any mask pattern matches with this patch, we don't change the pixel value at this index of the image (continue to the next index in the loop). If none of the masks match with this patch, we assign 0 at this index in the image. We continue these two loops until the iterations end.

b. Defect Detection

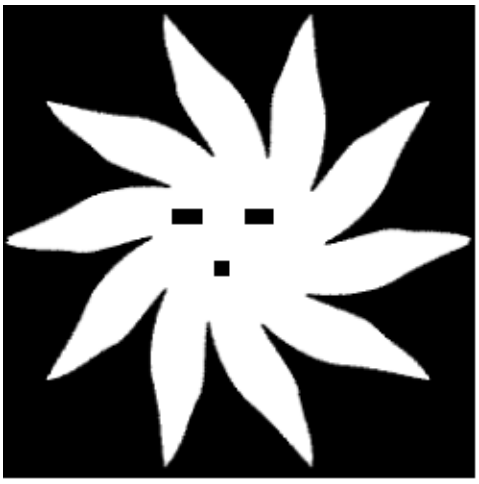
The details of implementation for each subproblem is described thoroughly in the results and discussion section.

c. Object Segmentation

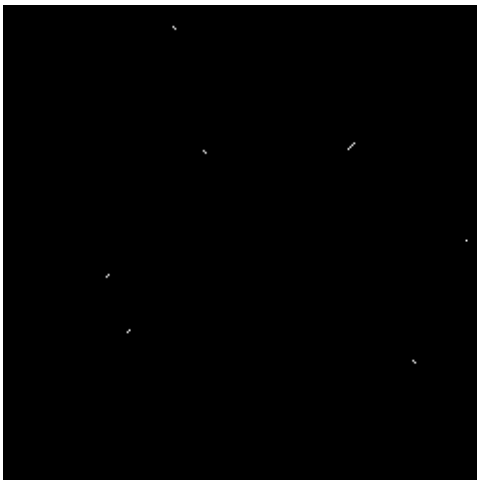
The details of implementation for each subproblem is described thoroughly in the results and discussion section.

3. Results

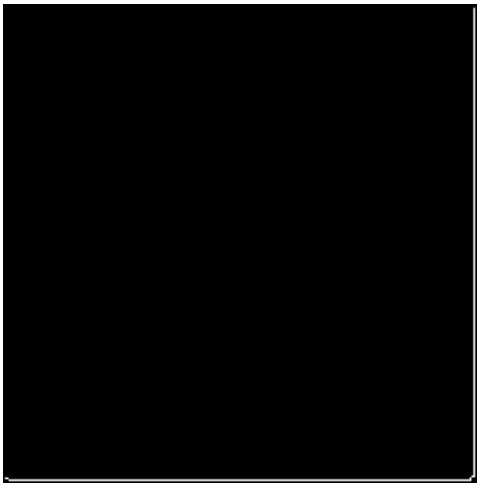
a. Thinning



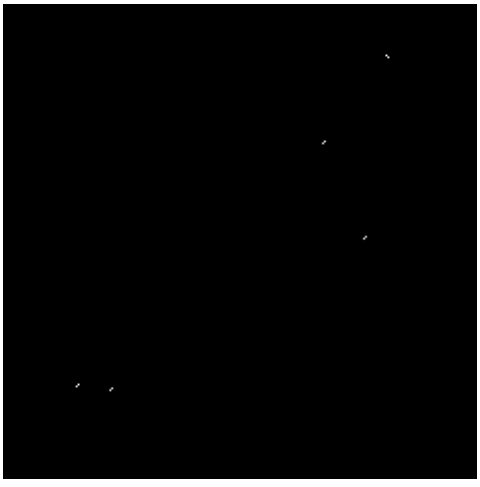
→



→



→



b. Defect detection

Figure 1: Original Image

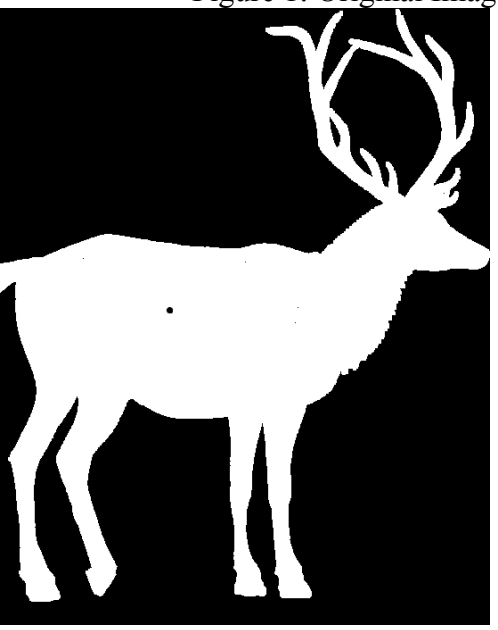


Figure 2: Inverted Image

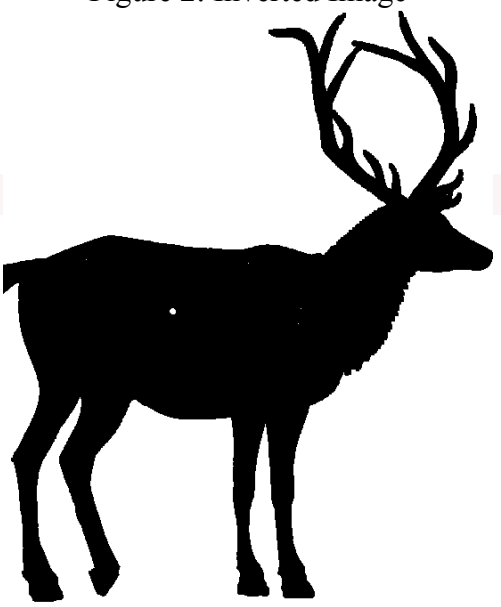


Figure 3: All Holes

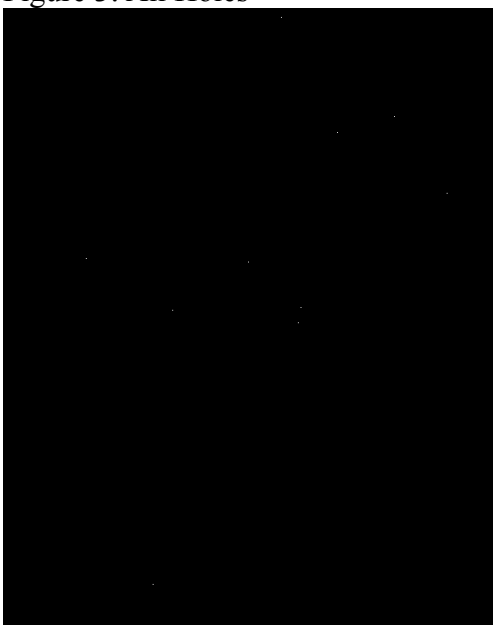


Figure 4: Defects < 50px

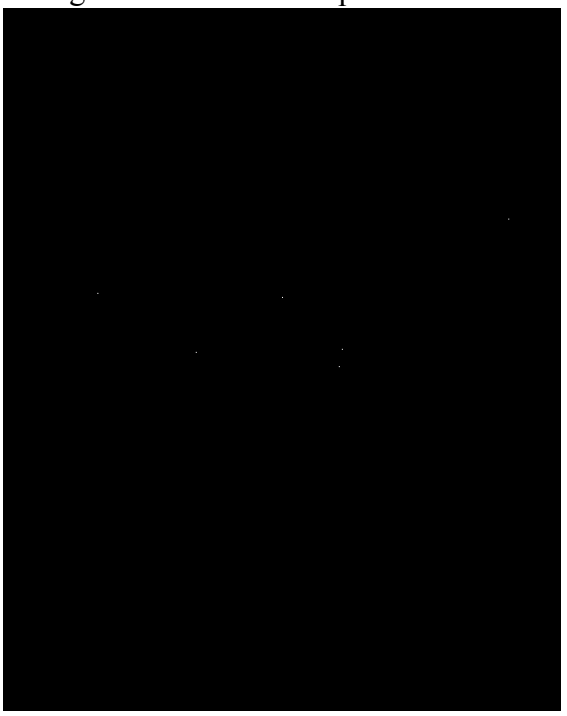
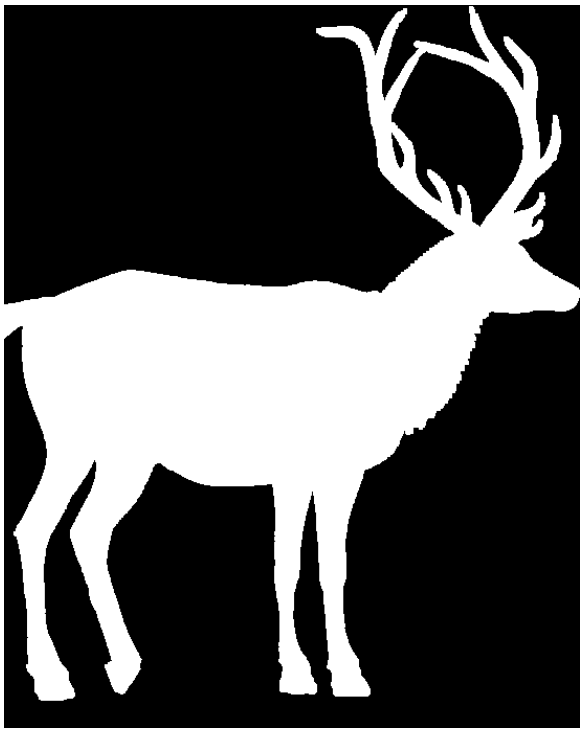


Figure 5: Final Image



c. Object Segmentation

Figure 1: Original Bean Image

↓

Figure 2: Grayscale Bean Image

↓

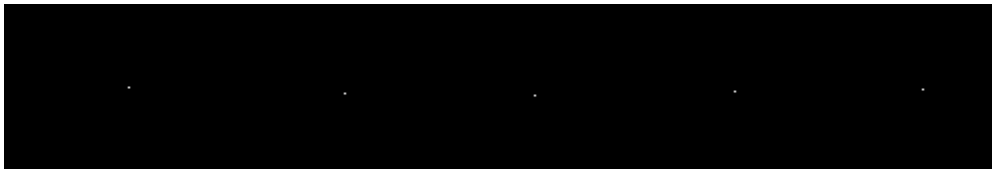
Figure 3: Binarized Bean Image

↓

Figure 4: Bean Segmentation Image

↓

Figure 5: Bean Segmentation Image



↓

Figure 6: Bean Count and Ranking Result

```
Total number of beans: 5
Ranking beans from small to large:
    "Bean: 5"    "Bean: 3"    "Bean: 1"    "Bean: 2"    "Bean: 4"
```

4. Discussion

a. Thinning

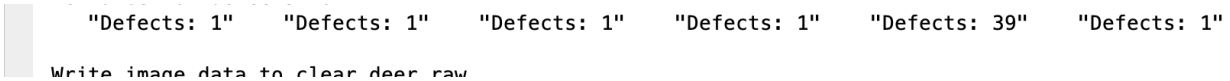
1). The results after the first iteration of the thinning algorithm is shown above. The reason why almost all pixels were removed from the image is most likely because it is not matching with any patterns in the Unconditional stage of the thinning process. The most probable explanation for these results I can find is that the rows and columns of the pattern table do not match with the patches that I take from the M matrix, as it is only matching with one specific mask and none of the other ones. However, it is hitting a decent amount of M patches, as the result of the M matrix looks correct.

b. Defect detection

1). `The number of defects: 6`

In order to implement a simpler way to count the defect, we first invert the binarized original image (background becomes white, object becomes black; **Figure 2**). The reason we do this is because if the defect holes are black pixels and the object is white, the result of shrinking will converge onto the hole as a white loop, and it would be difficult to detect such a loop. However, if we invert then perform shrinking, the holes will result as a single white pixel on the image (**Figure 3**). But as we can see in this figure, not all the holes presented in this image are actual defects in the original image. We know that the antlers of the deer on the right creates such a hole, but the shrinking algorithm determines it as a defect. To avoid this mistake, I assume that any hole that has a size larger than 50 pixels are not considered defects, thus anything below that threshold will be counted as a defect (**Figure 4**). Then, in order to find the size of these holes I use a method called connected islands. Whenever I encounter a white pixel in the shrinked image, I invoke this method and use the inverted image and the index of the white pixel as input. Within this function I create a stack structure and push this index onto the stack while increasing the “size” variable by one. Then I

have a while loop that stops when the stack is empty. In this loop, I get the current index by popping the top most element in my stack, then I find the eight neighbors of the same index in the input image. Furthermore if any of these neighbors are white (value = 1), which is the part of the defect, I push this index onto the stack and increase size by 1. The loop will end if all the surrounding pixels in that region are equal to 0. Lastly, I consider the hole as a defect if the final size returned is greater than 50, else it is not considered as a defect.

2). 

Each number returned in this "Defects" array represents the size of the defects. As we can analyze from this result, we have five defects with the size of 1 pixel and only one defect of size 39 pixels. The defect size calculation is explained in the question above.

3). The result of the clear deer image is shown in **Figure 5**. In order to correct the defect, we have to first obtain the locations of these defects. This is done when looping through the shrinked image and whenever a white pixel is detected, I save the x,y coordinates into an array. Then I find the same index in the original image (not inverted) and loop through that region depending on the size of that defect and change those pixel values to 1. In most cases, it is a single pixel value that needs to be changed to white.

c. Object Segmentation

1 & 2). In order to perform shrinking, we first convert the RGB into grayscale (**Figure 2**) and then binarize the image (**Figure 3**). The arbitrary assumption I made here is the quantization threshold. After testing a few times, I decided to set any pixel value above 221 as 1 and any value below 221 as 0. Then I invert the image to make finding the beans easier. However, in the second bean on this image there is a hole that we need to fill before we can shrink the image. To implement this pre-processing, I used a morphological processing function called close, which is the combination of dilation and erosion. The combination of these two operations results in closing gaps within an object, which is what we need to fill the hole in the second bean [4]. After this step, we can

actually obtain the segmentation mask for this image, which is basically identifying objects in an image with the background being black and the foreground objects as white (**Figure 4**). Now we can apply the shrinking operation (**Figure 5**) and count the total number of white pixels to obtain the number of beans, which is 5. Finally, to find the sizes of these five beans, we use the same "connected island" function that we used for part b (See b.1 for implementation details about this function). To rank them from the smallest to the largest, we sort the array and the results are shown in the **Figure 6**;

Bean #5 < Bean #3 < Bean #1 < Bean #2 < Bean #4

Image References

All images are either directly given in the homework description or are the results of my programs.

References

[1]: Image Warping, From Wikipedia, the free encyclopedia, https://en.wikipedia.org/wiki/Image_warping

[2]: 3D Projection, From Wikipedia, the free encyclopedia, https://en.wikipedia.org/wiki/3D_projection

[3]: Homography, From Wikipedia, the free encyclopedia, https://en.wikipedia.org/wiki/Homography

[4]: N.A. University of Colorado Boulder https://canvas.colorado.edu/courses/61439/pages/morphological-opening-and-closing#:~:text=Morphological%20closing%20is%20a%20dilation,close%20gaps%20in%20the%20image.

[5]: SURF Matching, From Wikipedia, the free encyclopedia, https://en.wikipedia.org/wiki/Speeded_up_robust_features#Matching

[6]: Mathematical morphology, From Wikipedia, the free encyclopedia, https://en.wikipedia.org/wiki/Mathematical_morphology