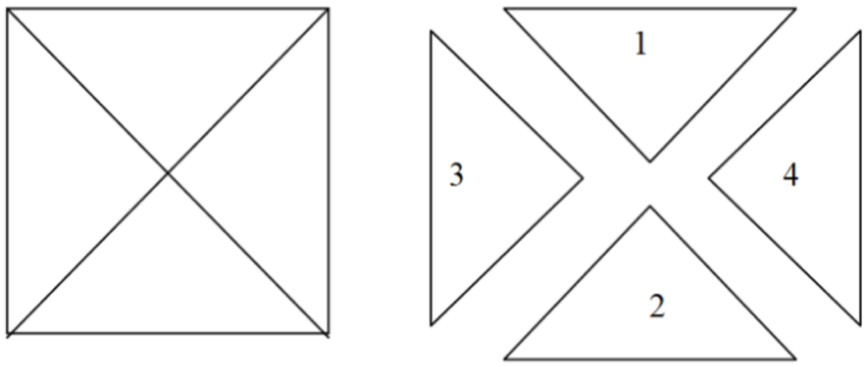
Problem 1: Geometric Image Modification

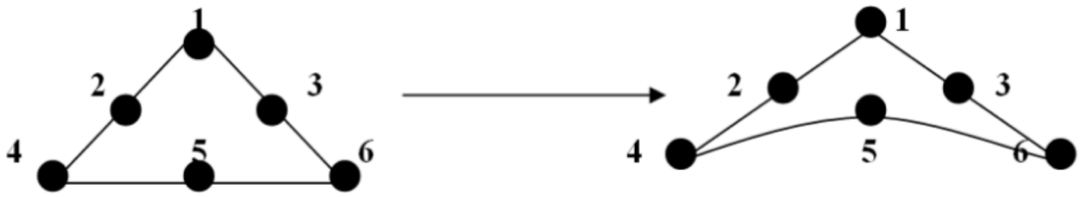
Motivation

As the title suggests, geometric image modification creates a mapping function between the source image coordinates u, v and the destination image coordinates x, y, and fill pixels in the target image with their corresponding source image pixels to create the effect of warping. Common affine transformations such as scaling, shearing, translation, and rotation have readily-defined matrices that can be multiplied with the coordinates, cascaded to get compound transformations, and inversed for inverse address mapping as we'll explain soon. An easy concept at first glance, but there are a few things to take note here.
First, row and column numbers in an image are discrete, but transformation functions generally have domain and range that encompass non-integers. Besides, in some cases having the top-left corner of the image as origin, as the row/col coordinates do, might not be intuitive. Therefore it's common to first convert from row/col coordinates to Cartesian coordinates before the transformation. For some curvy transformations, it may even be preferable to convert to polar coordinates.
Second, although intuitively, we would expect the transformation to go in the forward direction, i.e. $f : u, v \to x, y$, this will create inconveniences such as when the resulting x, y are decimals, where it can be tricky to decide how to round the x, y without leaving blank spaces or overlapping with another already-filled pixel. For this reason, we use inverse address mapping, i.e. $f^{-1} : x, y \to u, v$; for each pixel in the destination image with coordinates x, y, we run $f^{-1}(x, y)$ to get its corresponding coordinates u, v in the source image, and if u, v are decimals, we can interpolate with neighbors to get the pixel value at u, v.
Third, for a transformation given source and destination images but unknown transformation function, a common strategy is to select pairs of control points consisted of pixel coordinates we wish to map from and to, and solve for a_{0-5}, b_{0-5} the system:

$$\begin{bmatrix} u_0 & \cdots \\ v_0 & \cdots \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 & a_4 & a_5 \\ b_0 & b_1 & b_2 & b_3 & b_4 & b_5 \end{bmatrix} \begin{bmatrix} 1 & \cdots \\ x_0 & \cdots \\ y_0 & \cdots \\ x_0^2 & \cdots \\ x_0 y_0 & \cdots \\ y_0^2 & \cdots \end{bmatrix}$$

The non-linear terms are added to allow the system to approximate curvy transformations if needed. At least 6 pairs of control points need to be selected in order for this system to be solvable. Some complex transformations might require splitting the image into smaller regions, and selecting control points and solving system for each region, as shown:





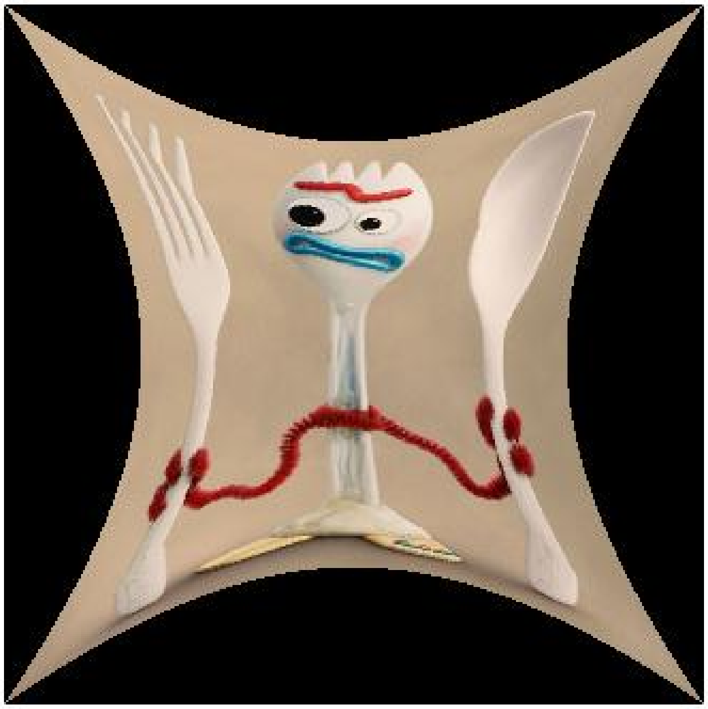
Methods

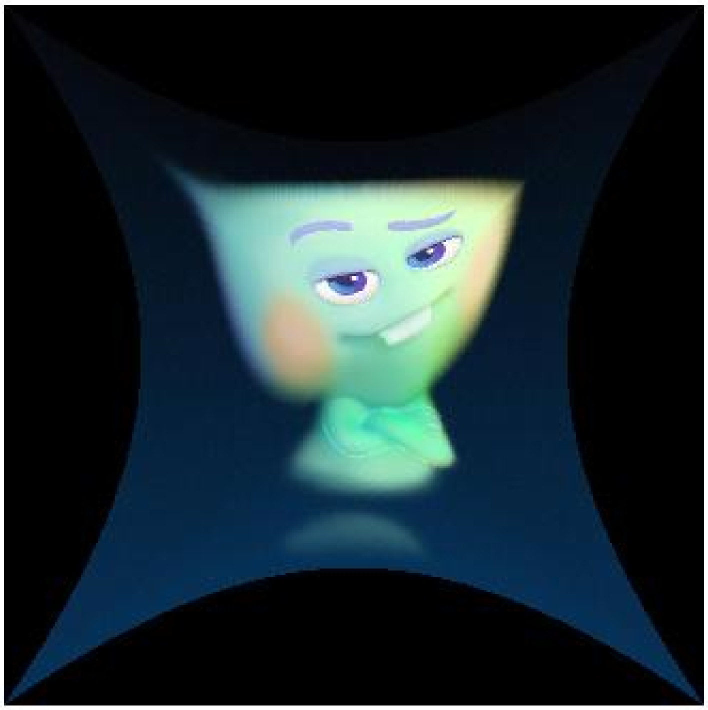
The transformation required here is an example of the "complex transformation" for which we need to select control points and solve system. The scheme for splitting the image and choosing control points for each split is the same as shown in the illustration above. More formally, the image is split along the two diagonals where $r == c$ or $r == (nrows - c)$, and for each region, control points are selected as the three vertices and three midpoints in the source domain, and almost same in the target domain except control point 5 is "curved in" by 64 pixels.
As a self-aware noob, I also simplify the problem in three ways. First, I do not convert to Cartesian coordinates, and instead entirely use row/col numbers, rounding to the nearest integers whenever decimal row/col numbers arise. Second, when doing inverse address mapping, I do not use interplolation for decimal coordinates but, again, round to the nearest integers. Third, for the gaps in the destination image created by the curved-in parts, I assume that those pixels will be mapped to source pixels that extend out of the bottom of the triangle, and thus simply skip destination image pixels whose corresponding source coordinates go out of bounds of the source image, leaving them as zero. These are ad-hoc assumptions and I cannot really justify all of them, but I think the results look alright. For the reverse direction, I simply solved system in the reverse direction, i.e.

$$\begin{bmatrix} x_0 & \cdots \\ y_0 & \cdots \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 & a_4 & a_5 \\ b_0 & b_1 & b_2 & b_3 & b_4 & b_5 \end{bmatrix} \begin{bmatrix} 1 & \cdots \\ u_0 & \cdots \\ v_0 & \cdots \\ u_0^2 & \cdots \\ u_0 v_0 & \cdots \\ v_0^2 & \cdots \end{bmatrix}$$

Results and Discussion

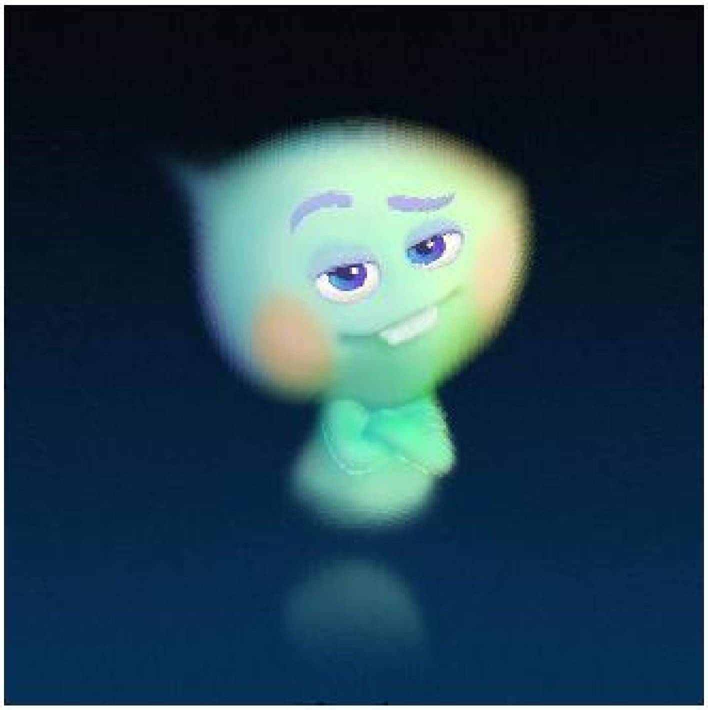
(1)





Please see Methods for approach.

(2)



(3) There are three notable differences.
First, the restored images have some black pixels along the borders roughly at positions that ”curved in” the most in the warped image. These correspond to pixels that got mapped to gap regions in the warped image.
Second, the restored image is more blurred than the original. This and the first difference are likely due to the errors caused by rounding during the coordinate mapping, which may, for example, use the same source image pixel to fill two adjacent target image pixels or shift pixel locations. I think using interpolation instead of rounding would alleviate this problem, but there will likely still be visible blur for cruder interpolation methods such as linear.
Third, the restored image still somewhat pushes the pixels that used to be on the curve towards the middle while mostly restoring the diagonal pixels correctly. I think this is because the warping process ”compressed” pixels towards center by shortening their distances to the center, and this was done more aggressively towards the midpoints of each border where the borders curved in the most. During the ”compression”, pixels that were adjacent in the original image got merged into the same pixel. This was amplified again during the restoration by a similar process except in the reverse direction, where multiple adjacent pixels in the restored image got their sources from the same warped image pixel. It should also be noted that fundamentally, getting transformation function by solving non-linear system for some control points is merely an approximation of the ”true” transformation function, and in general, straight lines are easier to approximate that curves.

Problem 2: Homographic Transformation and Image Stitching

Motivation

Two images can be stitched together by finding coordinates in each image that refer to similar pixels, and then, using the coordinates from the two images, solve for a transformation that essentially maps the first image onto the second image.
This problem is different from problem 1 mainly in two ways. First, since one image will end up on the left of the other, with only partial overlaps, the transformation needs to be solved under the canvas coordinates, where canvas is a container where image2 is put right after image1 on the right, and padding is added around the juxtaposed image1 and image2. This means image1 coordinates need to be offset by padding, and image2 coordinates need to be offset by padding and image1 columns. Second, it is assumed that image1 can be mapped to image2 by an affine transformation, so no non-linear terms need to be created for the coordinates; instead, another scaling term w is added to the coordinates to handle cases such as translation where parameters other than x and y are needed. The resulting coordinate system is called homogeneous coordinates, and its corresponding transformation matrix takes the form of a 3-by-3 matrix.

$$\begin{bmatrix} x'_2 & \cdots \\ y'_2 & \cdots \\ w'_2 & \cdots \end{bmatrix} = \lambda * \begin{bmatrix} x_2 & \cdots \\ y_2 & \cdots \\ 1 & \cdots \end{bmatrix} = \begin{bmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ H_{31} & H_{32} & H_{33} \end{bmatrix} \begin{bmatrix} x_1 & \cdots \\ y_1 & \cdots \\ 1 & \cdots \end{bmatrix}$$

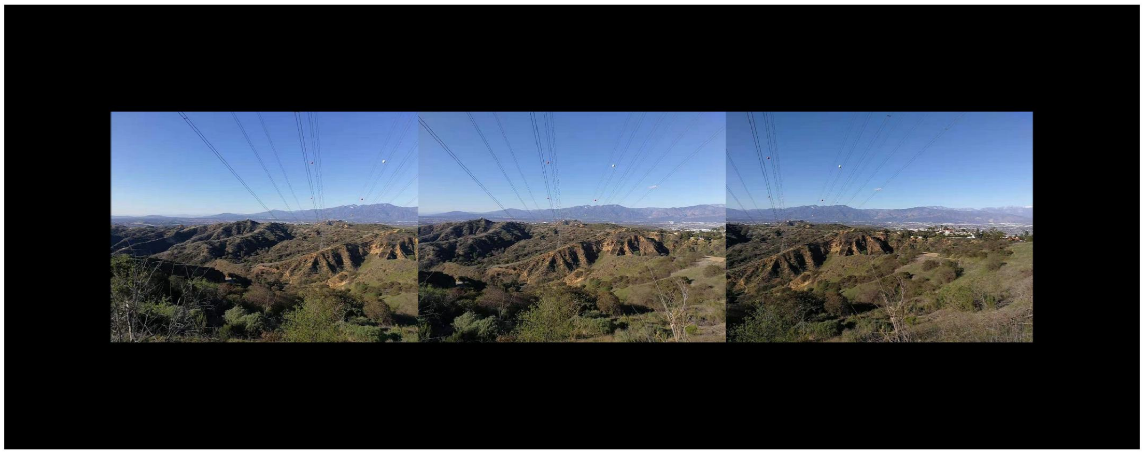
$$H_{11} * x_1 + H_{12} * y_1 + H_{13} = (H_{31} * x_1 + H_{32} * y_1 + 1) * x_2$$

$$H_{21} * x_1 + H_{22} * y_1 + H_{23} = (H_{31} * x_1 + H_{32} * y_1 + 1) * y_2$$

Fixing H_{33}, there are 8 coefficients to solve. By the equations above, each control point provides two equations, so at least 4 **different** pairs of control points need to be chosen in order to solve the transformation.

Methods

First, the images are read in and converted from RGB to YUV, where the luminance matrix is used for SURF.
Second, procedures showcased in **Matlab panorama example** are used to detect, extract, and match SURF features. This results in two pairs of coordinate sets, where each pair contains coordinates that refer to matching SURF features.
Third, a canvas is created by juxtaposing left, middle, and right images, and padding the juxtaposed image block by 200 black pixels on all sides.



The SURF feature mapping procedure may sometimes return false matchings, which need to be excluded from the control points or else the solved transformation will be wrong. In order to select 4 pairs of control points, I first tried a simple RANSAC, i.e. randomly choose 4 non-repeating pairs of control points from the SURF matching result, calculate a transformation using these 4 pairs, and apply the transformation to all source control points to get their pixel values from the aforementioned grey canvas, and calculate the MSE between source and post-transform pixel values to get the error of the homography and control point selection. I tried running this for 100 iterations and choosing the lowest error pairs. **However**, the control points, and by extension homography, obtained in this way were not ideal, possibly because the error was measured by simple pixel value difference; maybe using SURF similarity would give better results but I didn't have much time to look into it.

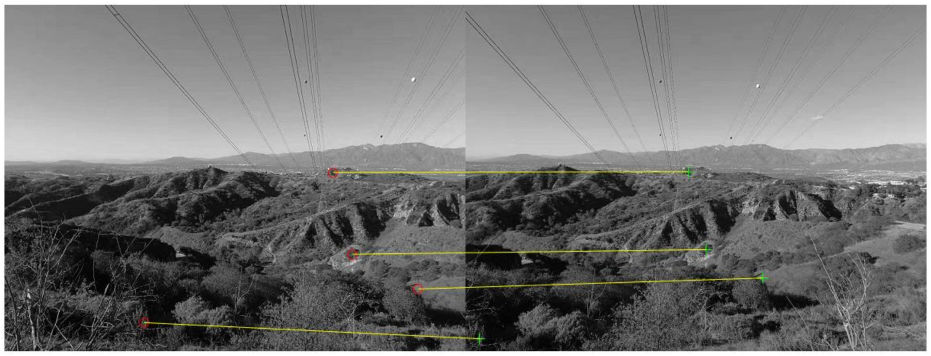
Instead, I simply eyeballed for SURF point pairs that truly match, and to be sure, I mostly chose salient points such as hilltops, grasslands etc. In addition, I leveraged the intuition that control points far from each other were more stable and informative pivots for the transformation solution, and therefore tried to select control points that were far from each other.

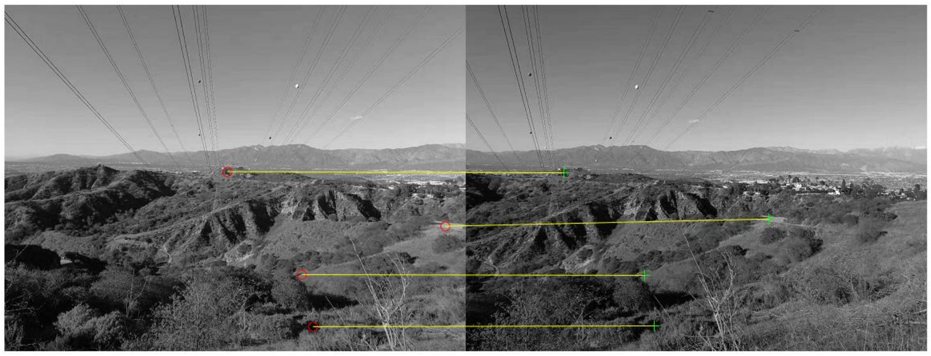
After selecting 4 pairs of control points between left and middle images, and 4 more for middle and right, I calculated the left-middle and middle-right homographies using said control points and the equations in Motivation.

After getting the left-middle and middle-right homographies, I iterate through the canvas and do inverse address mapping, again rounding coordinates to nearest integers whenever encountering decimals. Actually, rounding to nearest integers for row and column numbers is compatible with the way MATLAB implements image cartesian coordinates by making the "center" of each pixel integer. I chose two ad hoc column cutoff values $cutoff_1 = 964$ and $cutoff_2 = 1064$; for all columns before $cutoff_1$, I used left-middle homography; for all columns after $cutoff_2$, I used middle-right homography; for all columns in between, I simply copied pixels from the same location.

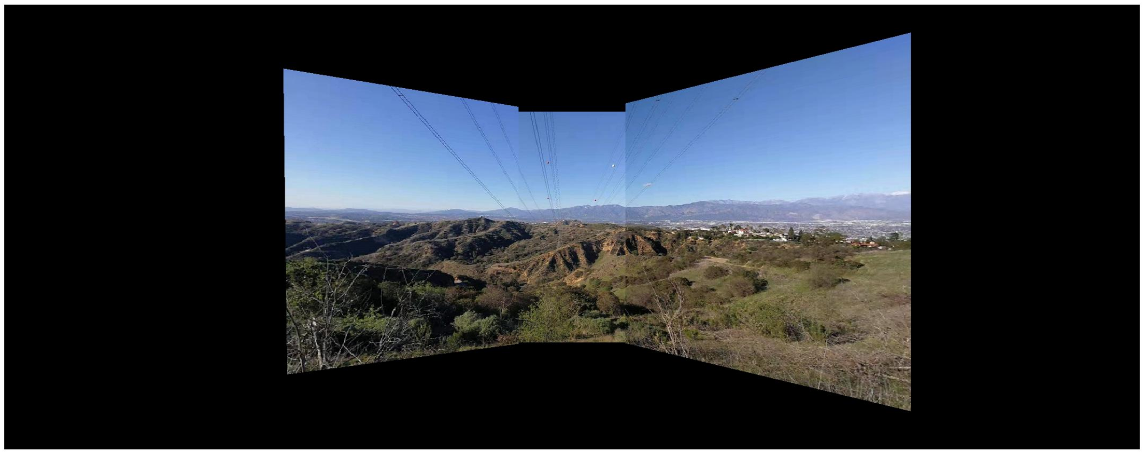
Results and Discussion

(1) As explained in Methods, I manually inspect SURF match pairs and select pairs that are salient and truly match while trying to keep control points far from each other. Shown below are my selection of control points.





(2) Please see Methods. Shown below is the stitched image.



Problem 3: Morphological processing

Motivation

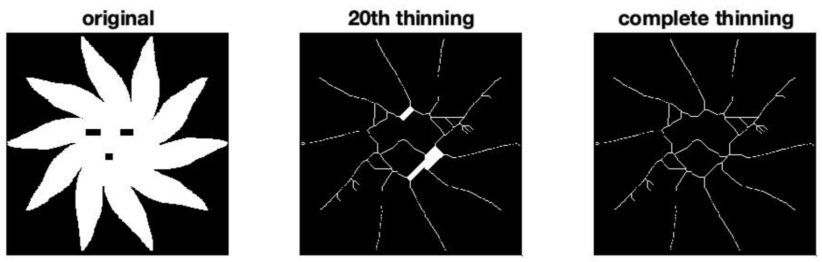
Morphological processing refers to a general category of operations on binary images that sweeps a structuring element over the image and either remove, keep, or turn on each pixel depending on whether its neighborhood matches the mask.
Simpler operations of this category such as erosion and dilation uses a single structuring element; however, for more complex operations which need more information on the local neighborhood, two structuring elements are needed where the first one marks pixels for removal and the second one checks on the marked pixels and decides whether to actually remove. This more complex form of morphological processing is also called hit-and-miss transformation, and include operations such as **shrinking**, **thinning**, and **skeletonization**. It's tricky to explain the difference, but generally speaking, shrinking shrinks a connected solid blob to a single pixel or forms circles around holes if there's any; thinning thins the blob down to a line that approximately connects from one end of the blob to another; skeletonization is similar to thinning except its result also contains branches that lead to corners of the blob.
In this assignment, morphological processing is mostly used to simplify images for analysis.

Methods

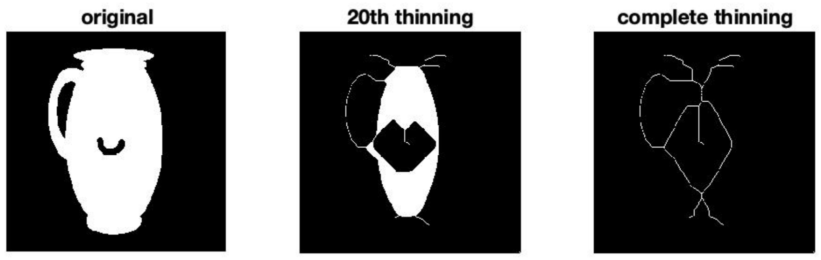
Images are first binarized. For the first 4 images which contain only 0 or 255 pixels, 255 pixels are set to 1 while 0 pixels remain 0. For rgb images, binarization is done as given in the instructions, i.e. take the luminance and set all pixels with luminance greater than half of maximum to 1.
I use MATLAB *bwmorph* function for shrinking and skeletonization.
For thinning, I implement the two-stage check approach described in class. The first stage checks against the unconditional pattern tables for thinning, i.e. rows in the *patterntable.pdf* that contain "T". This fills a matrix M in which pixels whose 3by3 neighborhoods match the unconditional structuring element are set to 1 and all else to 0. The tables are encoded by flattening in column-major order, and then matching is done by a simple strcmp. To speed up the lookup, I first check the bond for a pixel where each 1 on the cross neighbors add 2 to bond and each 1 on the diagonal neighbors add 1, and then directly go to the set of tables corresponding to the calculated bond. The second stage checks the matrix M against the conditional pattern tables for thinning and skrinking, again given in *patterntable.pdf*. The encoding for unconditional tables here is a bit more complex because of terms such as $A \vee B \vee C = 1$ and D, so I use regex for matching. This stage produces the resulting image, where if hit or if the current pixel is 0 in M, its value is copied from the original image, and if the current pixel is 1 in M and not hit, its value is set to 0. Since structuring elements (tables) are 3by3, the source image and M is zero-padded by 1 pixel.

Results and Discussion

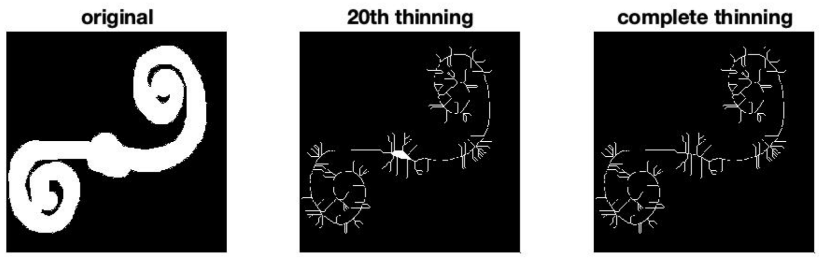
(a)



The original flower image contains three holes, so the resulting shrinked image has lines inside that circle that circumvent what used to be these holes. Also, pixels that are relatively from all 0 pixels, which can be either holes or background, are removed relatively late into the shrinking process, as can be seen in the middle image.



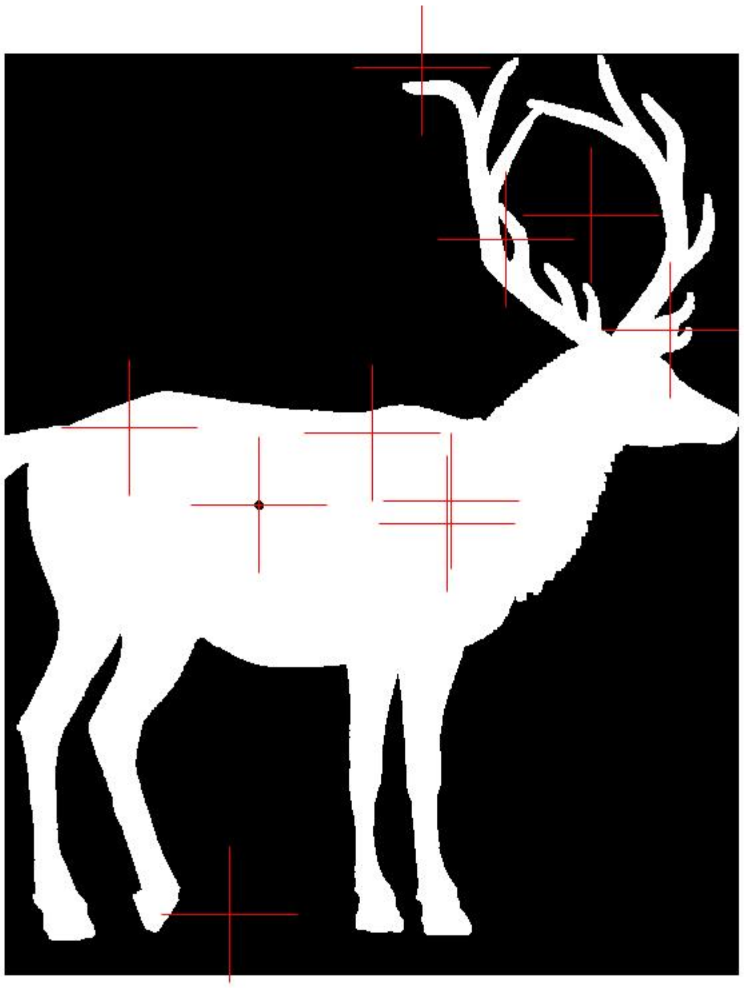
Again, the source jar image contains a curve of 0 pixels in the middle and a handle which forms another cluster of 0 pixels, and the shrinked lines circumvent these. Also similar to flower, 1 pixels far from 0 pixels are removed later. It's also interesting to see a line that leads to what used to be the hold in the middle; this is consistent with the way thinning keeps a line that runs from "one end to the other", if we consider the hole to be an end.



The source spring image is connected in many places, so there are many circle. Also, we see again lines that lead to what used to be holes. In this case I think the narrow gaps are similar to holes so there are many branches.

(b)

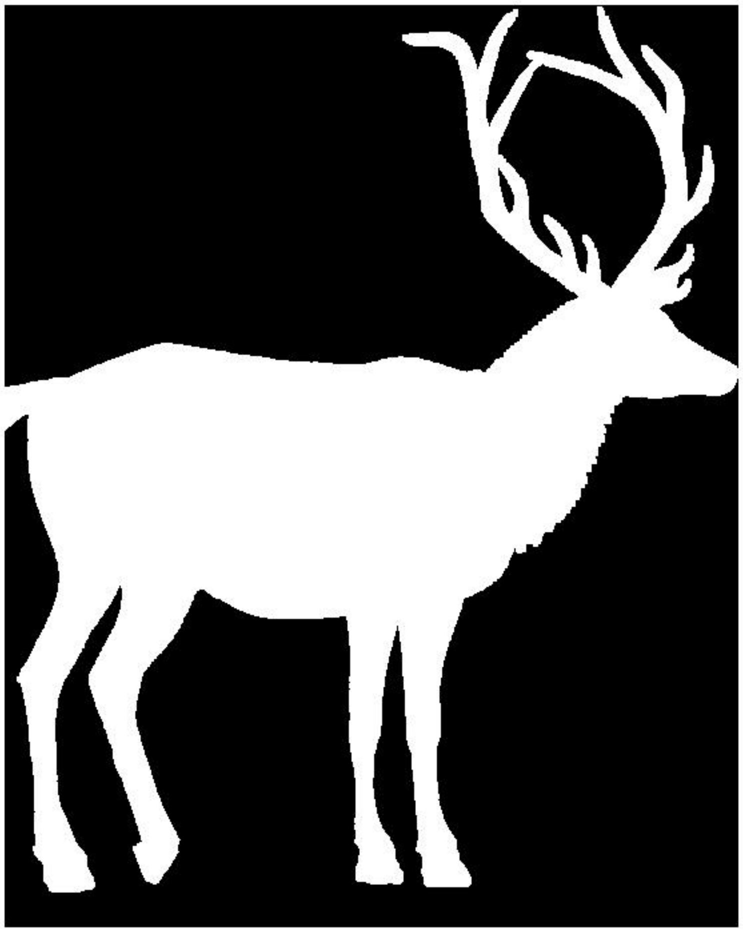
(1) There are 6 defects in the image. Since defects correspond to holes inside the deer, defects can be found by first binarizing the deer image and then running shrinking on the complement of the binarized deer image. This gives 10 white pixels, whose locations are marked in the image below:



It is evident some locations, such as the one at the antler, are not defects but merely natural holes. These false positives are suppressed by thresholding defect size to be under 50 pixels. Since defects are connected regions of black pixels, I find the defect sizes by doing DFS on each of the aforementioned shrinked coordinates, while setting the current pixel to 1 and incrementing a counter to track the defect sizes. If counter exceeds 50, the pixels set to 1 so far are reset to 0.

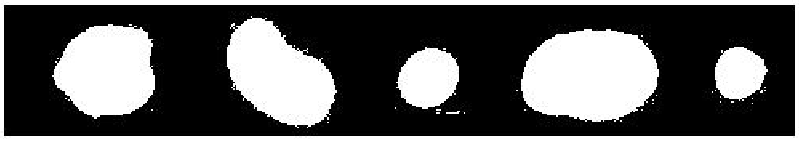
(2) Among the 6 defects, there are only 2 sizes – 5 defects have size 1, and 1 has size 39. My solution also shows several defects that have size 51, but those correspond to false positives. Please see (1) for how I checked sizes.

(3) Please see (1) for how I did it. The resulting image is shown below:



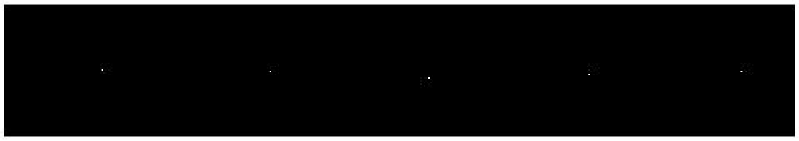
(c)

(1) First, the image is binarized by first converting to yuv and then using a luminance threshold of 230, i.e. any pixel with luminance below 230 are set to 1 and any above are set to 0 (since this image has white background). The resulting image is shown below:

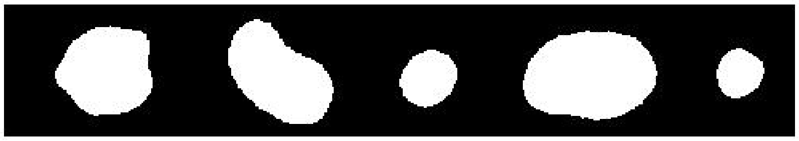


This is a relatively loose threshold that leaves many sporadic dots around the beans. Applying shrinking directly to this image will lead to messy result.

We learned in class that erosion erodes away smaller clusters, which makes it perfect for cleaning the binarized bean image. I use MATLAB *imerode* function with 3by3 square structuring element. The resulting image looks nice and can be seen as a segmentation mask of the beans.
Erosion leaves only large, connected clusters which are surely beans, so shrinking can now be applied to the eroded image to get 5 white pixels, i.e. 5 beans.



(2)



As explained in (1), this segmentation mask is obtained by eroding the binarized image with a 3by3 square structuring element. The beans sizes can be obtained in a similar way as in (2), i.e. doing DFS on bean coordinates and keeping a counter of pixels. The bean image is inverted before the DFS. Through this way, the 5 beans, from left to right, have sizes 2604, 2706, 957, 3648, and 690.