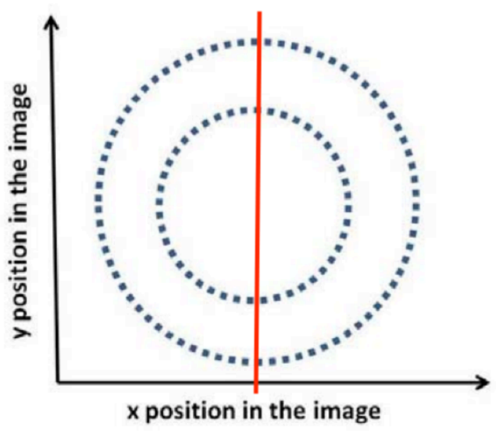
CS 6476

PS2

1.1 The resulting representation is sensitive to orientation. If the image has mostly vertical edges, then the response of the 4th and 10th image will have higher value than others in the vector result. Similarly, if the image has mostly horizontal edges, then the values at 1st and 7th positions will be the higher.

1.2 The result will look like below. Because the way K-means clustering works, we minimize the average distance between points to two centers, the algorithm will divide the image into two halves. However, clustering them into two concentric circles might be a more "correct" way to cluster them, then in this case, K-means clustering is not a good choice to implement the task.

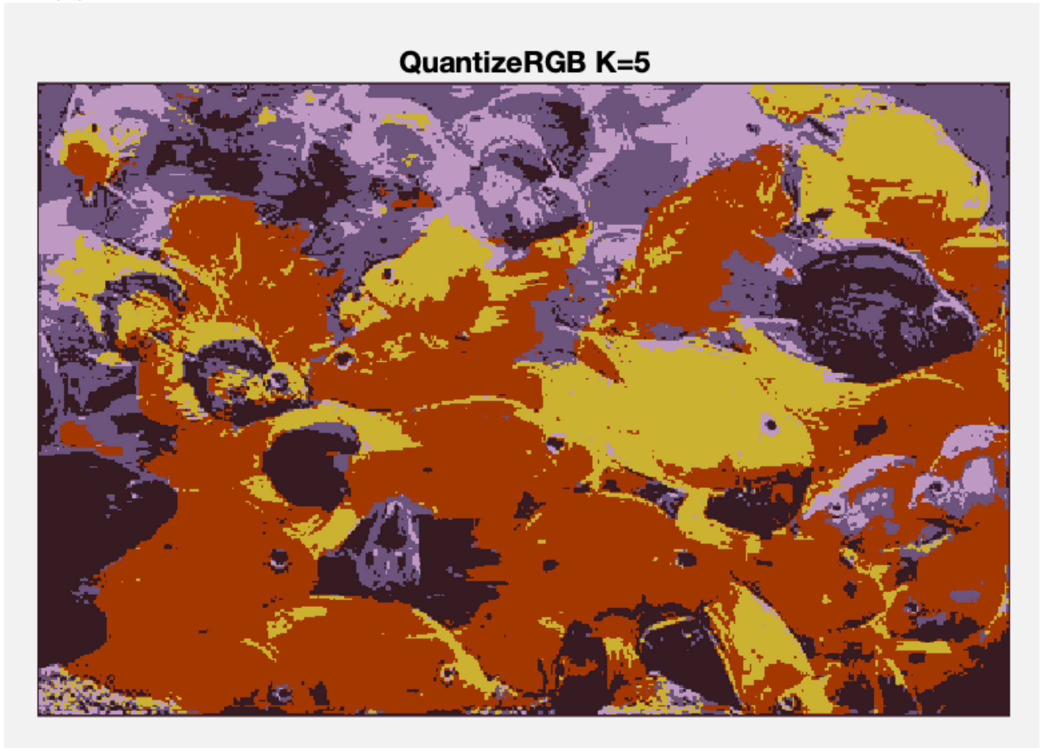


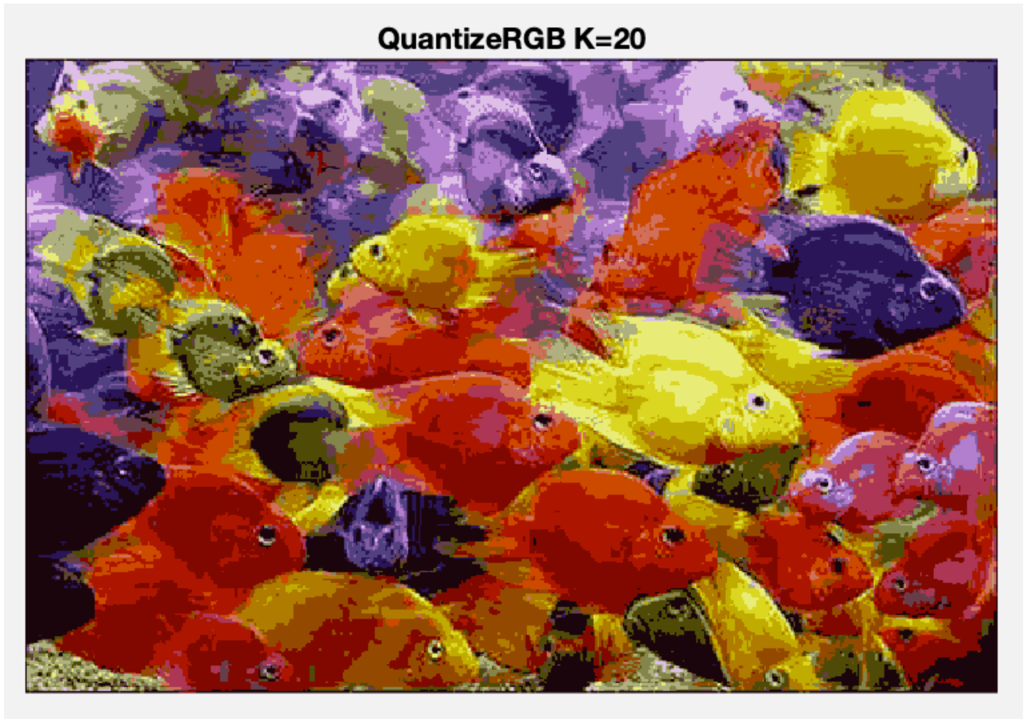
1.3 Mean-shift will be the best way of the three to recover the model parameter hypothesis from the continuous vote space. Mean-shift clustering aims to discover "blobs" in a smooth density of samples. It will give us candidates for centroids within a given region. Also, mean-shift is consistent throughout different runs despite the fact that we may have many parameters than discrete vote space. Therefore, we can use mean-shift for continuous vote space. Note that K-means returns labels for each data point and graph cut only cuts vertices.

1.4 We can use K-means clustering to solve this problem.

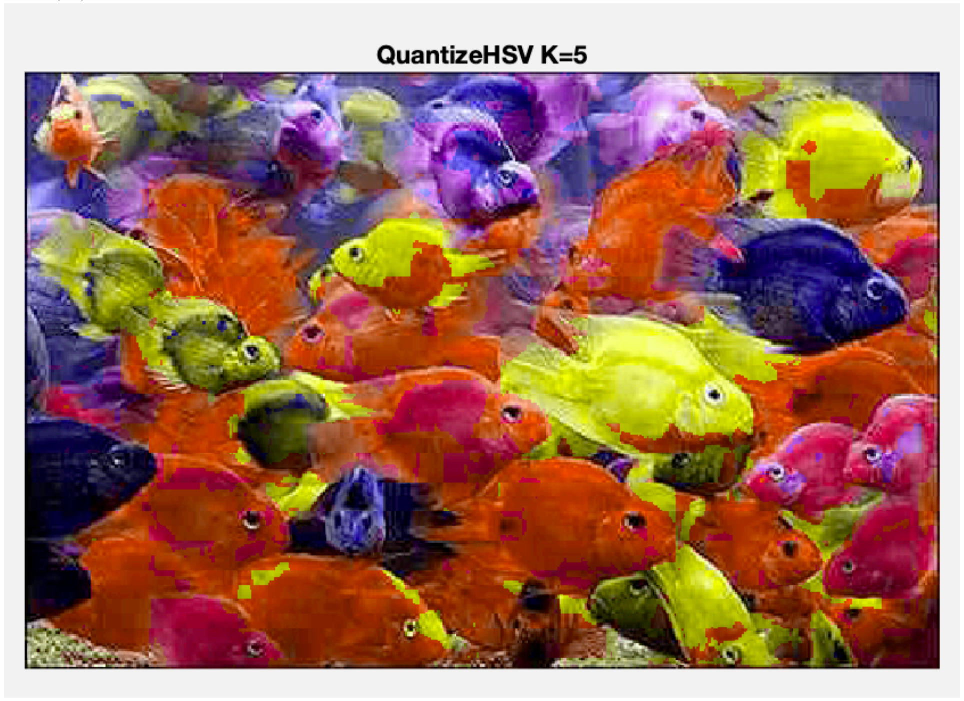
```
Step1: Find the center of mass for each blob
        For each pixel $p_i$ in blob $b_i$:
                Calculate sum of pixel positions sum_of_pixel += position of $p_i$
                Calculate center of mass: center_of_mass = sum_of_pixels / size($b_i$)

Step 2: Extract radius invariant circularity feature.
        Calculate mean_of_radius = size($b_i$)
        For $p_i$ in boundary($b_i$):
                sum_of_squared_distance += (center_of_mass – position of $p_i$)$^2$
                circularity = sum_of_squared_distance / size($b_i$)

Step 3: Cluster blobs according to their circularity feature.
        Kmeans(list_of_circularity_obtained_from_step2)
                Return result of label
```

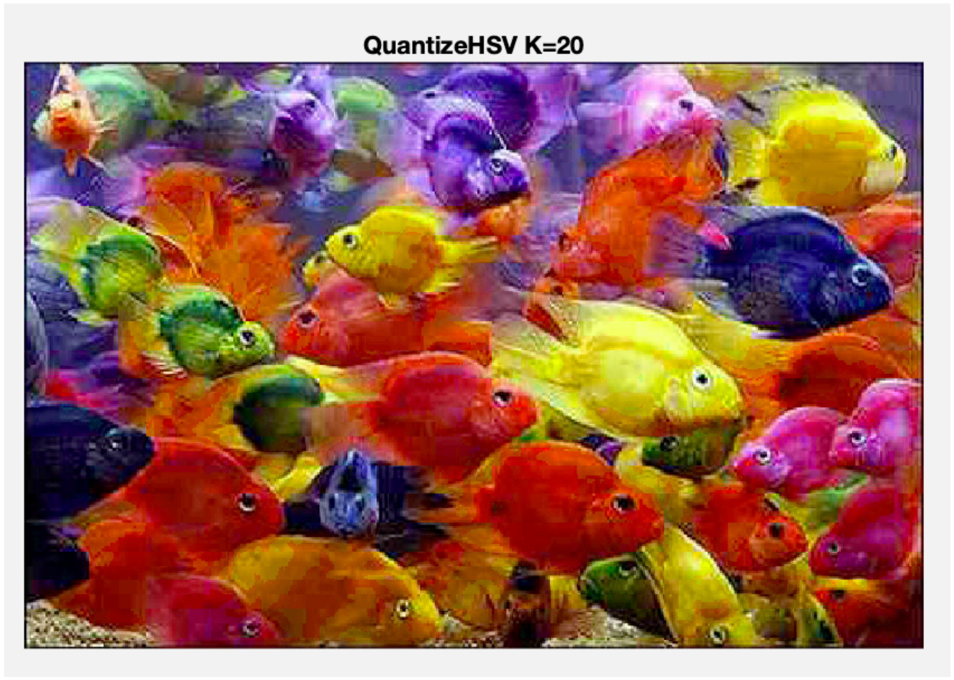
2.1 (a)





2.1 (b)



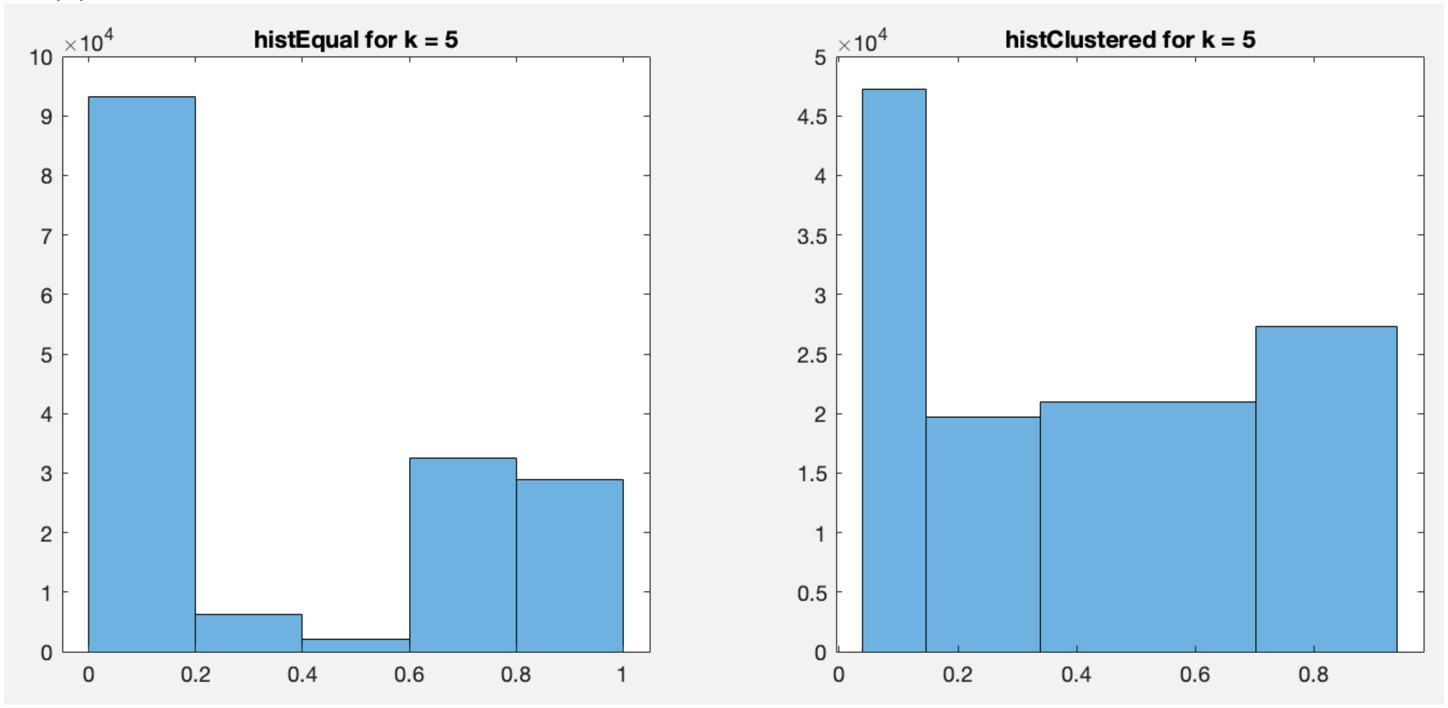


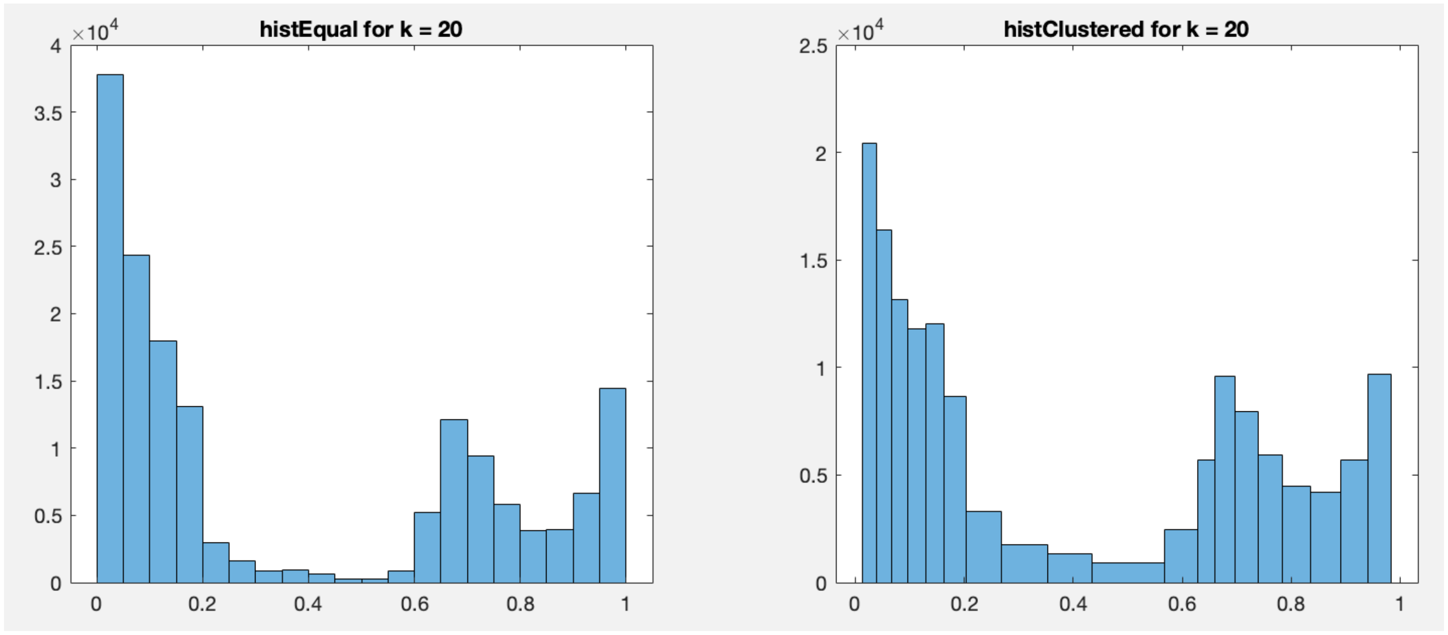
2.1 (c)

SSD error for RGB:
K = 5: 42695075
K = 20: 34549669

SSD error for HSV
K = 5: 10201663
K = 20: 2744370

2.1 (d)





2.1 (f)
The histograms look different because histEqual has equal-sized bins but histClustered does not. Because of the way K-means clustering works, the bin/cluster size does not need to be the same. The distribution of the colors also seems uneven because the image apparently does not have equal distribution of R,G,B colors throughout.

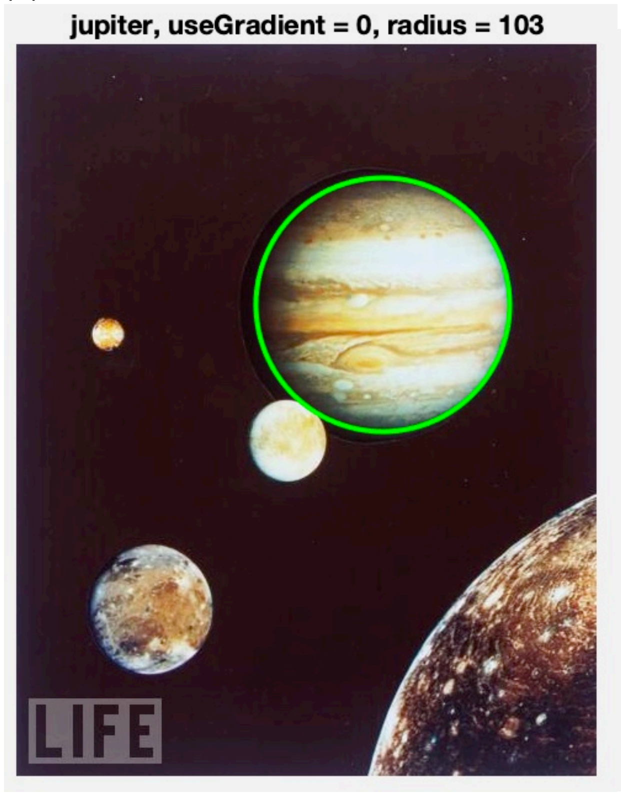
RGB quantization maps the image into only k colors, but HSV does not. We see that HSV color space has much lower SSD than RGB. We also preserve the saturation and value channels, so the result of HSV quantization looks more vivid and closer to the original one than RGB quantization results.

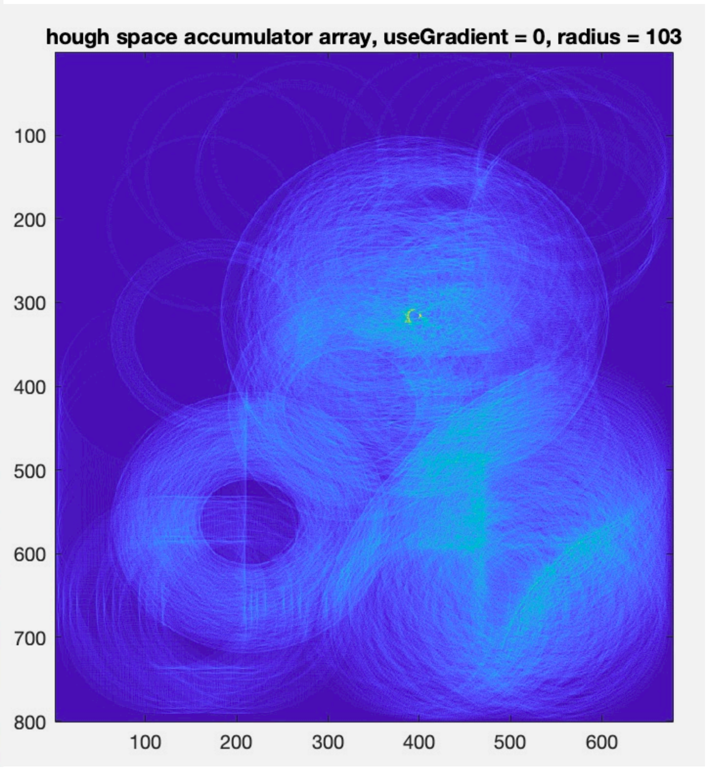
We observed that If k is larger, then SSD is smaller. This makes sense because by increasing cluster numbers we increase the colors levels, more colors are represented and thus smaller SSD.

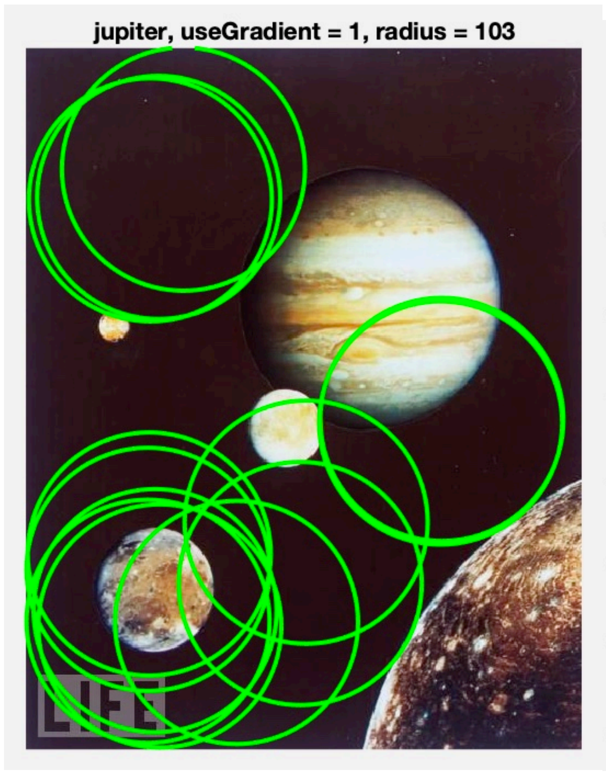
Across different runs, I observe different results for output quantization values. This is because the result depends on initialization, but it seems MATLAB randomly picks one in each run, so the result should be different.

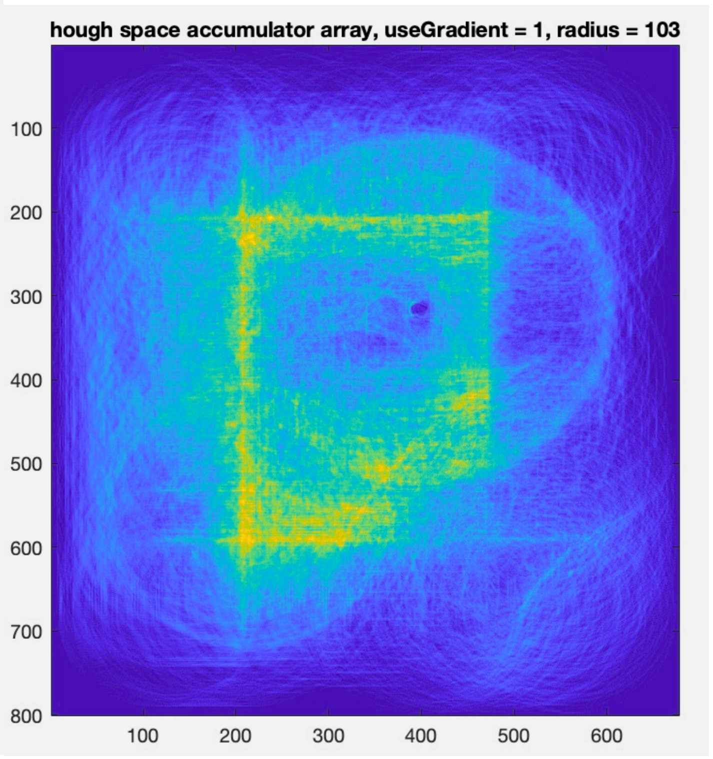
2.2 (a) First we use Canny edge detector to convert the original image into binary edges. We then calculate gradients, count the votes from each pixel and build the accumulator array. After obtaining the array, choose the centers of the circles where the votes are larger than some threshold (can be changed manually, usually it is 0.9 * max_value_of_all_bin_votes). For the candidates of centers of circles, transform the coordinates back to original image and display the circles with radius.

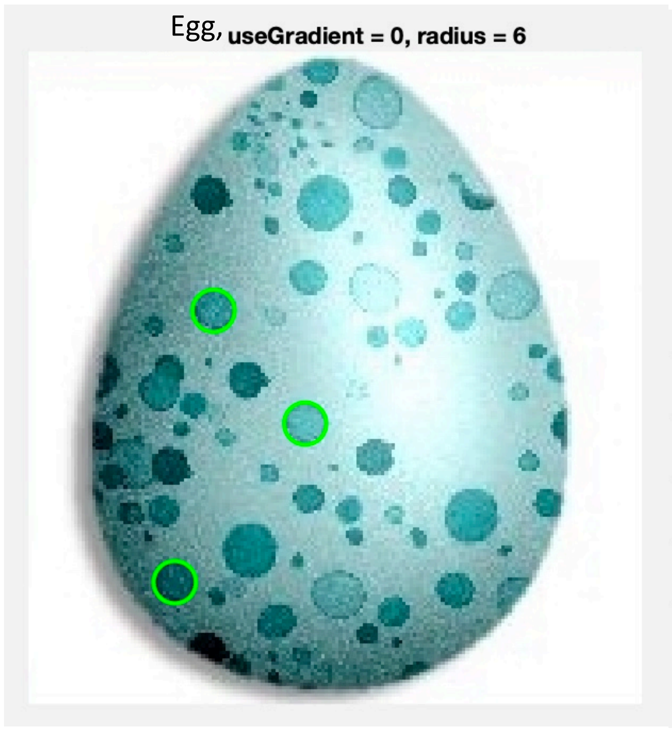
(b)

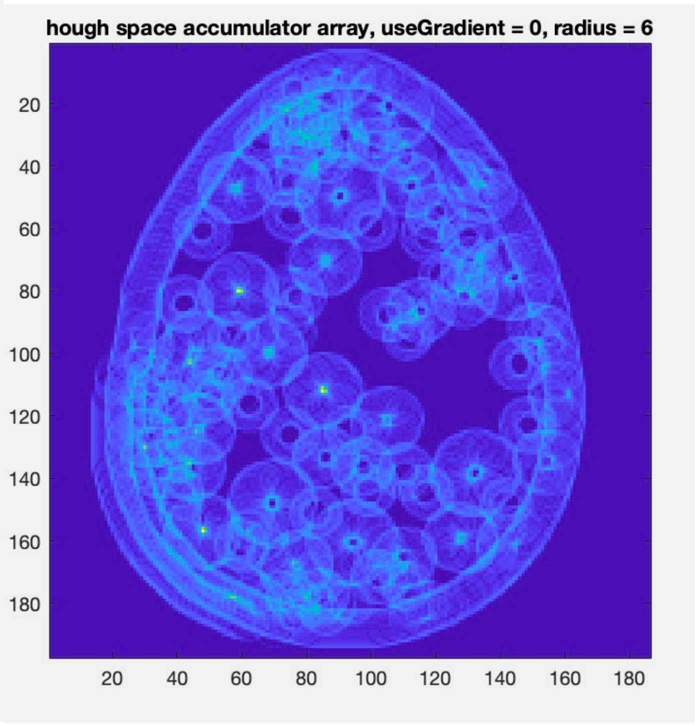


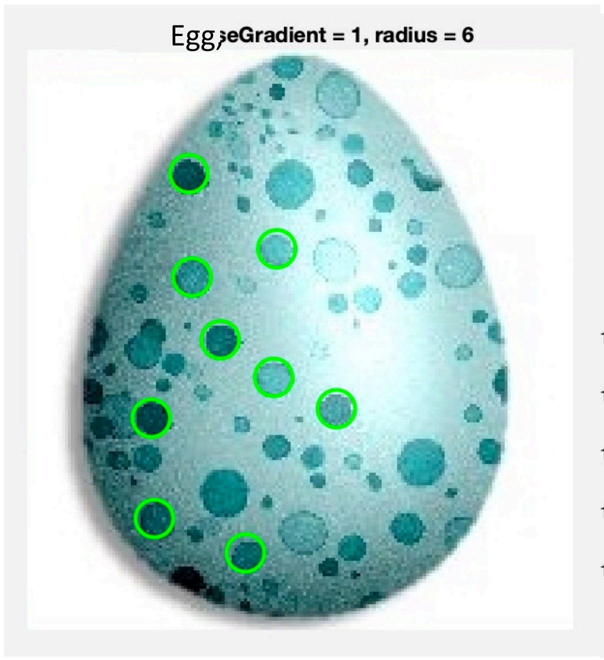


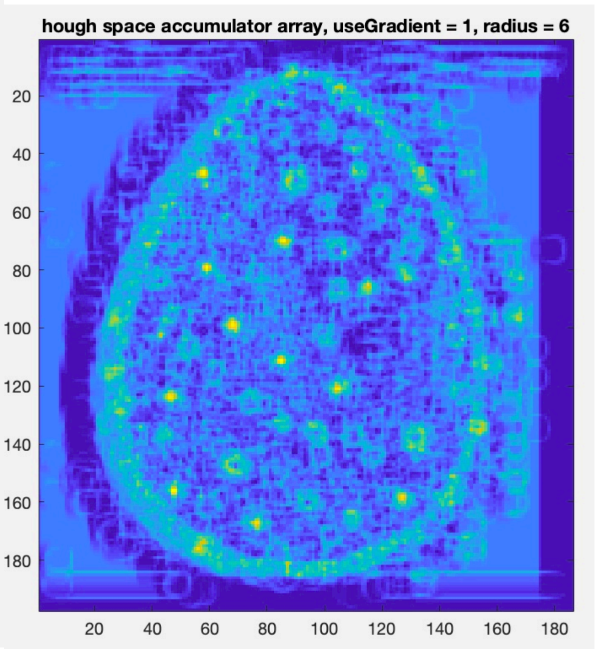

jupiter, useGradient = 1, radius = 103

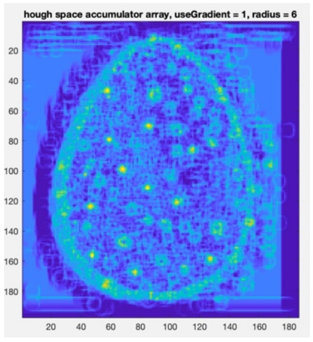

hough space accumulator array, useGradient = 1, radius = 103
100
200
300
400
500
600
700
800
100
200
300
400
500
600


Egg, useGradient = 0, radius = 6


hough space accumulator array, useGradient = 0, radius = 6
20
40
60
80
100
120
140
160
180
20
40
60
80
100
120
140
160
180

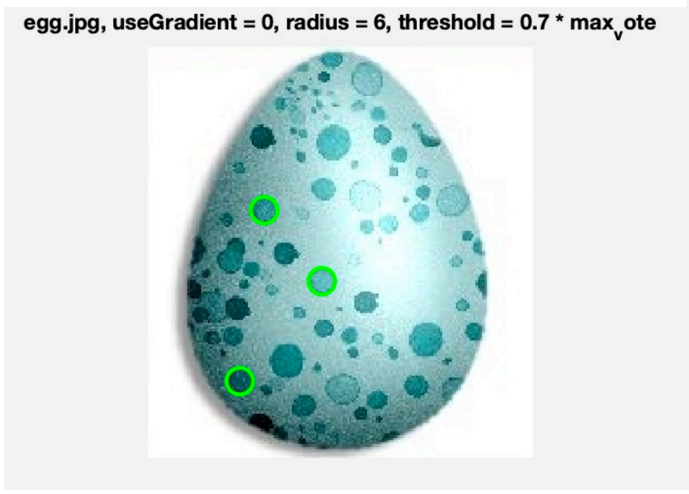


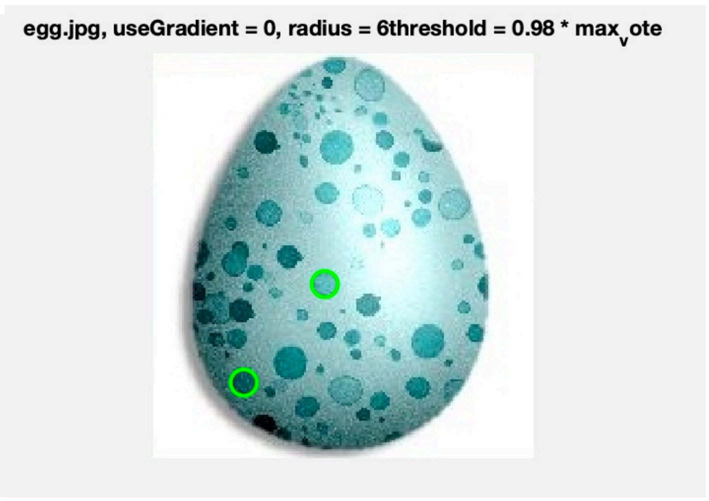


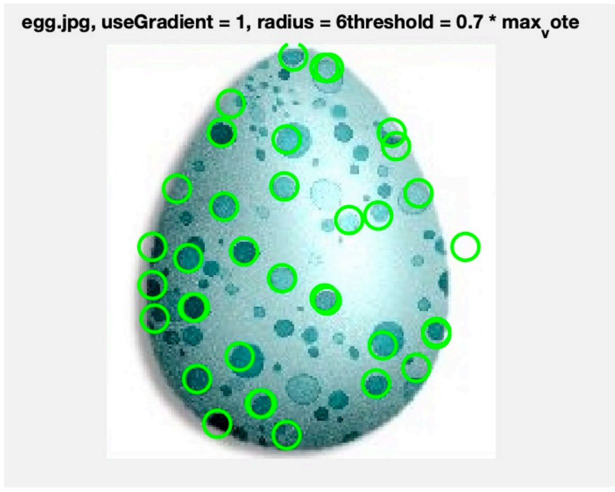


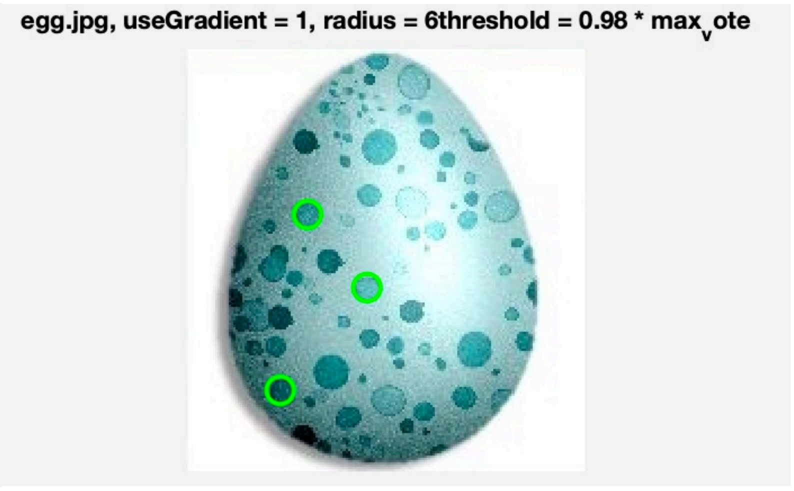
2.2(c) We can see in the images above on the right (useGradient = 1, radius=6 for egg.jpg), there are 8,9 -ish bright dots in the blue hough space image. Those are the bins that get higher votes. The brighter/intense they are, the more likely they are circles with radius=6. We mark those bright dots in the original image and we can see that those are the centers of circles with radius = 6. Depending on the threshold value, the number of circles may vary. There are other “circles” in the hough space image, but they didn’t get enough vote for the algorithm to consider the area as “center of a circle”.

2.2(d) The easy way to change the number of circles presented is to change threshold value. In the algorithm I use 0.9* max_vote. However, if we change the threshold value to 0.7 and 0.98 * max_vote, the number of circles increases and decreases, respectively. Note that for useGradient=1, threshold=0.7*max_vote, the result includes a lot of non-circles, because those bins with not enough votes also get filtered as centers. (Be aware if you have trypophobia.)

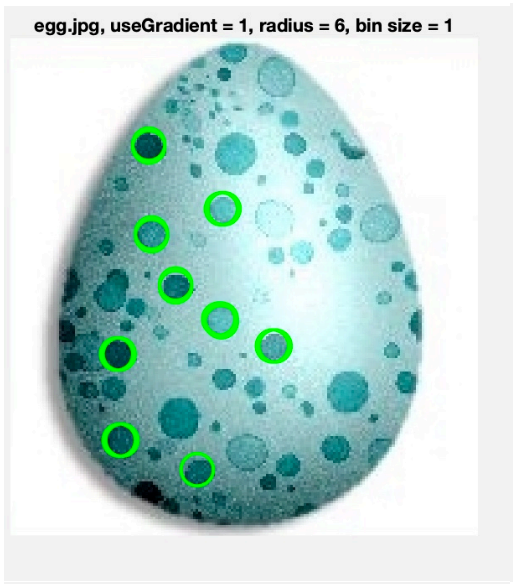

egg.jpg, useGradient = 0, radius = 6, threshold = 0.7 * max_vote

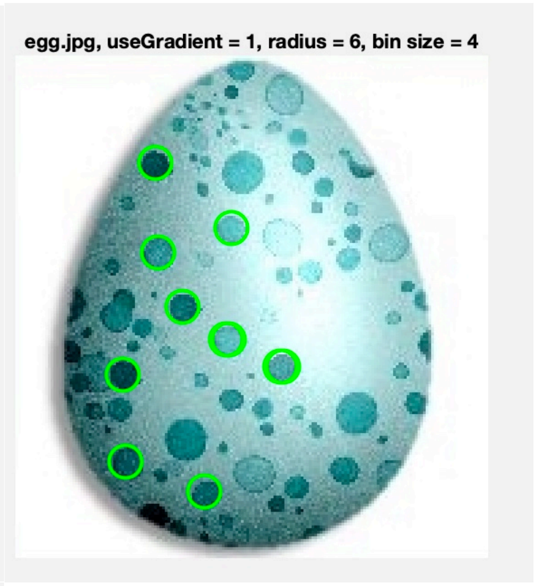

egg.jpg, useGradient = 0, radius = 6threshold = 0.98 * max_vote

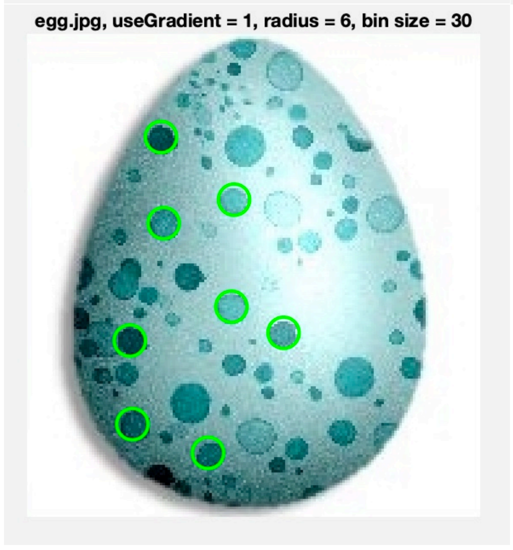

egg.jpg, useGradient = 1, radius = 6threshold = 0.7 * max_vote

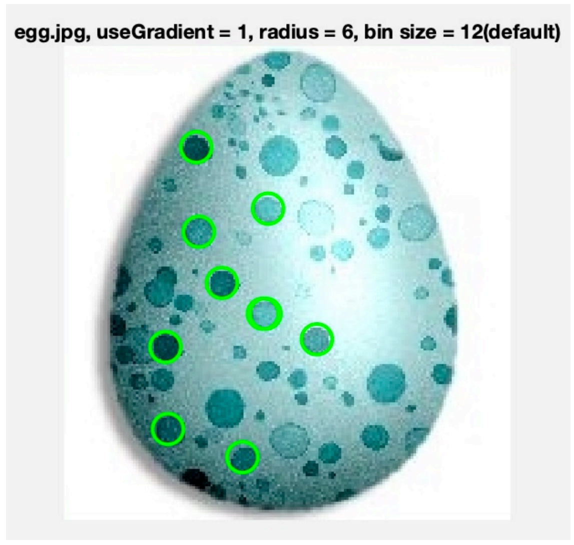

egg.jpg, useGradient = 1, radius = 6threshold = 0.98 * max_vote

2.2(e) For useGradient = 1, keeping other parameters same, I change the bin size to 1, 4 and 30 in comparison to the default value 12. The smaller the bin size, the system seems to pick up more circles, most of them overlap. The larger the bin size, the less circles. This makes sense because if we increase the bin size, the vote bin will be larger and therefore combining many possible centers as one. For useGradient = 0, there is no obvious changes of the result of Jupiter and egg image, so I didn't include results here.








egg.jpg, useGradient = 1, radius = 6, bin size = 12(default)

3.
Method: Instead of using 2D hough space, we increase the dimension to 3D space. The rest of the algorithm is similar: we use Canny edge detector to convert the original image into binary edges. For each possible radius smaller than the max_radius(manually set), create a 3D hough space accumulator array, calculate gradients, and count the votes from each pixel. Find the max value of each bin, and the max value of all bins. Choose the centers of the circles where the votes are larger than some threshold. For the candidates of centers of circles, transform the coordinates back to original image and display the circles with radius.

(I didn't succeed but include the incomplete code named 'detectCirclesAnyR.m')