Short answer problems

1. **Suppose we form a texture description using textons built from a filter bank of multiple anisotropic derivative of Gaussian filters at two scales and six orientations (as displayed below in Figure 1). Is the resulting representation sensitive to orientation, or is it invariant to orientation? Explain why.**

Yes, the resulting representation is sensitive to orientation because if two input images are only different in their orientations, the representation vector would respond differently to each. Some parts of the vector would respond with a higher value, and some other parts would respond with a lower value depending on the edges' orientations in the images. If an image is oriented in a way such that it has many horizontal edges, the filter that corresponds to horizontal edges would respond with a higher value. Same reasoning can be applied to any other form of orientation. Thus, this resulting representation is sensitive to orientation.

2. **Consider Figure 2 below. Each small square denotes an edge point extracted from an image. Say we are going to use k-means to cluster these points' positions into k=2 groups. That is, we will run k-means where the feature inputs are the (x,y) coordinates of all the small square points. What is a likely clustering assignment that would result? Briefly explain your answer.**

A likely clustering assignment would be a division line directly through the center of the circles, diving the circles and the space to half. This space is clustered based on the x and y coordinate values. When doing a k-clustering, it is almost like the centers of the clusters are pushing against each other, and trying to stay as far as they can from other centers in the problem space. In this case, there are only 2 centers, and the division line across the center of the circles can effectively allow the two cluster centers to establish a balanced state while staying far away from each other.

3. **When using the Hough Transform, we often discretize the parameter space to collect votes in an accumulator array. Alternatively, suppose we maintain a continuous vote space. Which grouping algorithm (among k-means, mean-shift, or graph-cuts) would be appropriate to recover the model parameter hypotheses from the continuous vote space? Briefly describe and explain.**

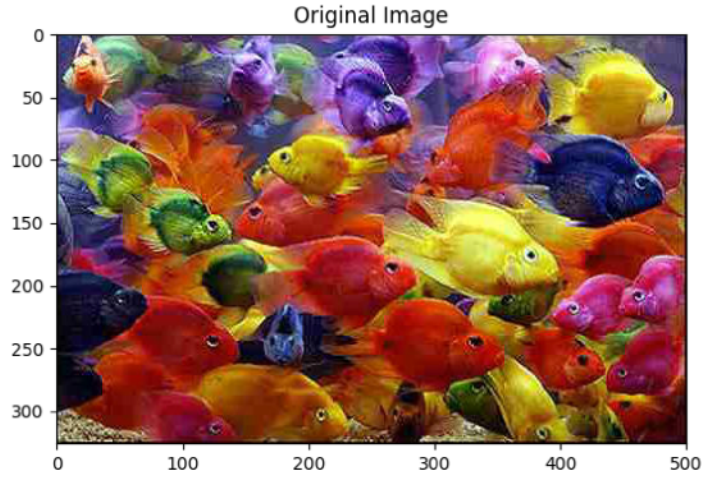
Mean-shift would be an appropriate algorithm to use in this case. Mean-shift basically allows a window to move toward a local maxima of density in the problem space. Mean-shift is also known to finding features (colors, gradients, etc.), which highly aligns with what the accumulator arrays is trying to do. accumulator array uses the grid to group the votes that fall into the same cell, and output the cells with the largest groups. These cells can also be seen as the local maximum of voting density. The window used in mean-shift can basically be understood as a bin in accumulator arrays. Thus, mean-shift would be absolutely appropriate to recover the model parameter hypotheses from the continuous vote space.

4. **Suppose we have run the connected components algorithm on a binary image, and now have access to the multiple foreground 'blobs' within it. Write pseudocode showing how to group the blobs according to the similarity of their outer boundary shape, into some specified number of groups. Define clearly any variables you introduce.**

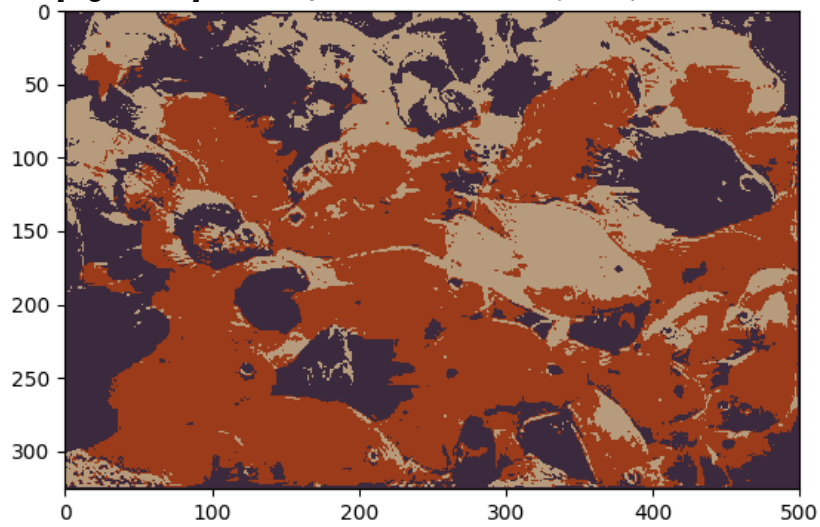
```
numBlobs = total number of blobs
blobData = a numBlobs x 6 matrix
for each row data in blobData:
// each row represents a blob
        data[0] = circumference of the blob
        data[1] = max width of the blob
        data[2] = min width of the blob
        data[3] = max height of the blob
        data[4] = min height of the blob
        data[5] = surface area of the blob
run k-means algorithm, k is a specified integer, on this newly constructed space, blobData matrix
the resulting clusters should contain blobs with similar boundary shape.
```

Programming

1. Color quantization with k-means

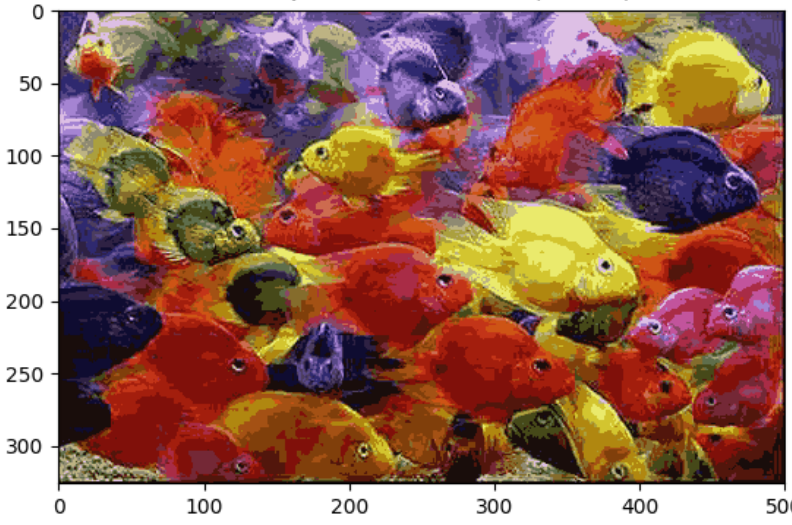


[Figure 1.1] Color Quantization in RGB (k = 3)



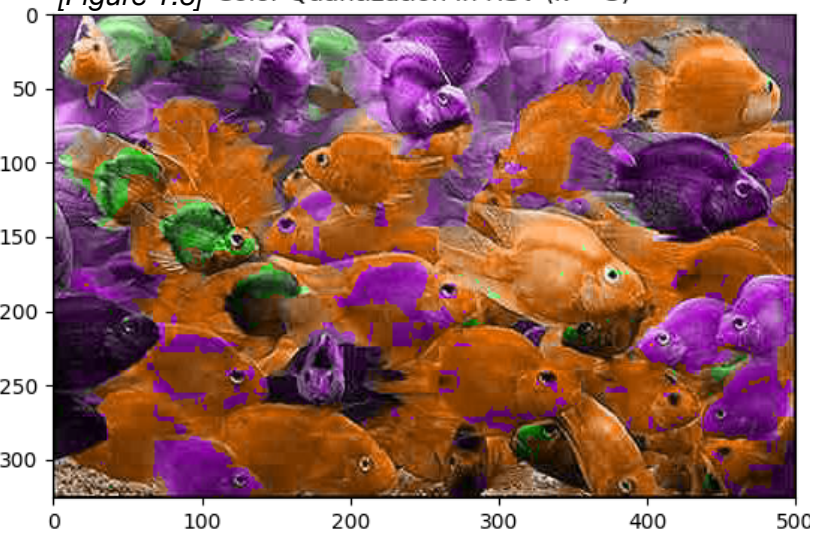
Error in RGB Space (k = 3): 50594062

[Figure 1.2] Color Quantization in RGB (k = 30)



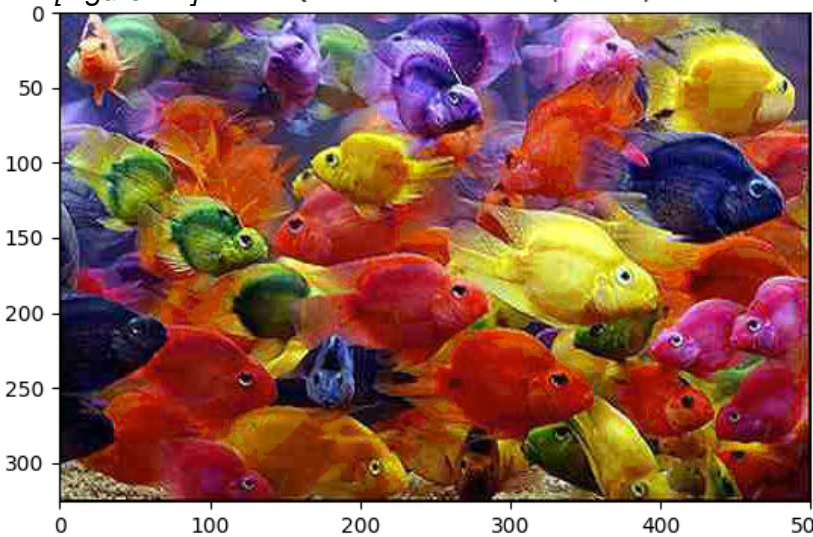
Error in RGB Space (k = 30): 39850794

[Figure 1.3] Color Quantization in HSV (k = 3)



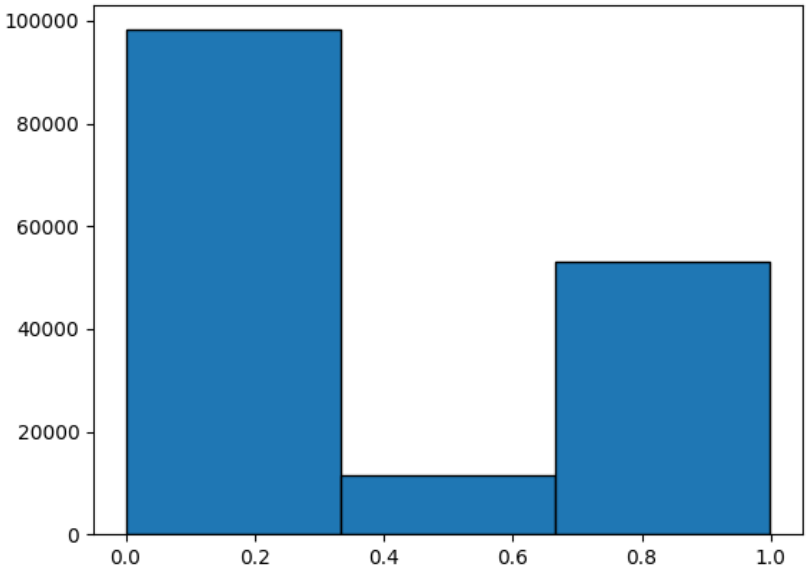
Error in HSV Space (k = 3): 20635123

[Figure 1.4] Color Quantization in HSV (k = 30)

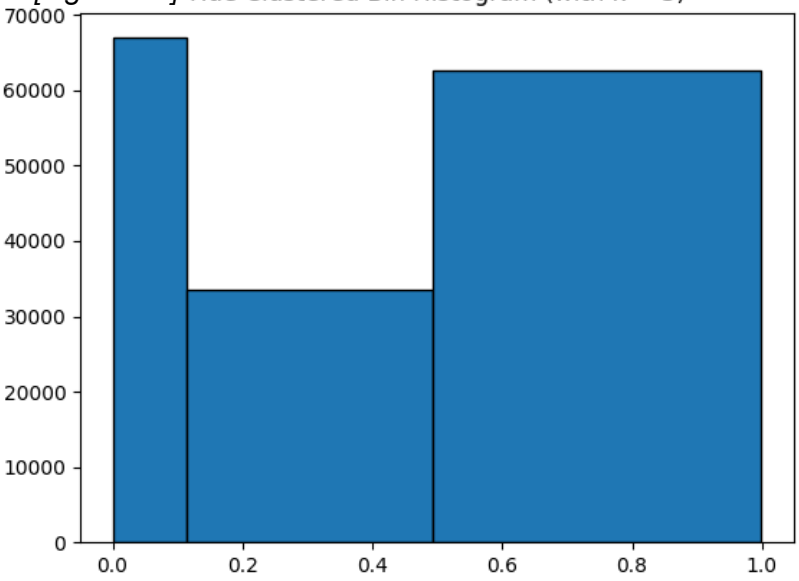


Error in HSV Space (k = 30): 7204866

[Figure 1.5] Hue Equal Bin Histogram (with k = 3)

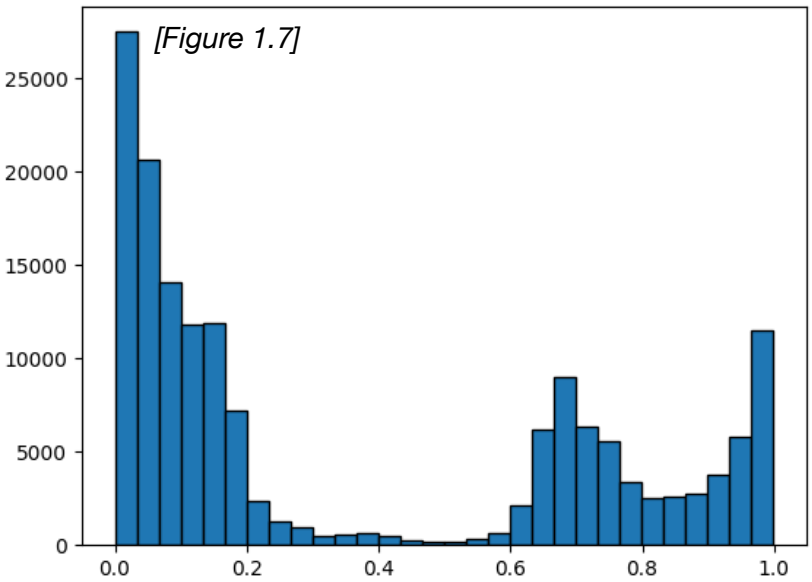


[Figure 1.6] Hue Clustered Bin Histogram (with k = 3)



Hue Equal Bin Histogram (with k = 30)

[Figure 1.7]



Hue Clustered Bin Histogram (with k = 30)

[Figure 1.8]

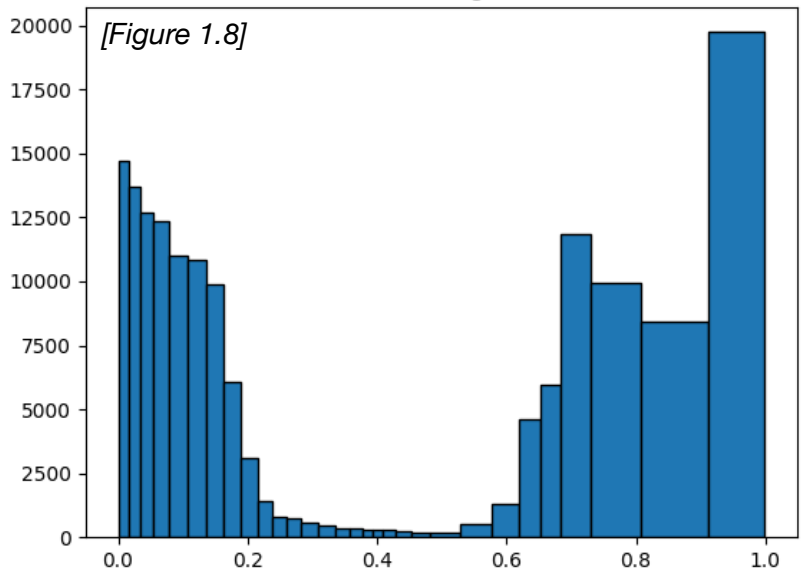


Figure 1.1 is a color quantization in the RGB color space with k value at 3, which means there are 3 clusters. From this image it is obvious to see that the three centers from the clusters represent colors along the lines of purple, red and khaki. Each center resulted from the average of its corresponding cluster. Figure 1.2 is the same quantization in RGB color space with k value at 30, which means there are 30 clusters in this case. This image's colors are very close with the original image because with the large number of clusters, each cluster would contain color pixels of a smaller range, which would result in an average that is closer to all the values in the cluster. 30 clusters also allow for more color variations, 30 to be specific, whereas 3 clusters only allowed 3 colors in the output image. Given these reasons, **with higher k values, the quantized image better resembles the original image**.

Figure 1.3 is a color quantization in the HSV color space using only the 1-dimensional H(hue) space with k value at 3. After the hue space is clustered, the centers of the clusters are plugged back to the HSV color space with the S(saturation) and V(value) values unchanged. The output image from this color space seems way more natural than the one from the RGB color space because with the saturation and value of the pixel colors unchanged, it still maintains the saturation and value from the original image while the hue values are limited to the centers of the clusters. This difference between RGB and HSV color space in this problem can be better illustrated using Figure 1.4. Figure 1.4 is a result from the same procedure as Figure 1.3, but with k value at 30. This one almost seems identical with the original image.

Same reasons as those explained for Figure 1.2, **as k value gets greater, each center closer represent the original colors, and more colors are allowed in the image**. Different from Figure 1.2, in Figure 1.4 **HSV color space allows smoother changes in colors as well because these are embedded in its saturation and value components, which remained untouched**. For example, on the big yellow fish in the center, a bright yellow transitions into a dimmer yellow smoothly in Figure 1.4; however, the bright yellow and the dimmer yellow are two different colors in the space of RGB with different RGB values, therefore, there is an obvious boundary line between the bright yellow region and the dimmer yellow region on the fish in Figure 1.2. **The quantized image from HSV color space has way smaller error than the one from RGB color space** because the error in HSV only comes from 1 dimension of the space which is the hue, whereas the error in RGB comes from all 3 dimensions of the space. **The quantized image with higher k values also has way smaller error than the one with lower k values** because higher k values allow more clusterings and less amount of data in each cluster. This would make the average in each cluster closer to all the values in this cluster, thus resulting in smaller overall error.

Figure 1.5 and Figure 1.6 are the histograms of the hue space for an image, where the horizontal axis represents the values of the elements in the bin, and the vertical axis represents the number of elements in the bin. Figure 1.5 uses equally-spaced bins with k at 3, and Figure 1.6 uses clusters as bins. The main difference of the two histograms is that the first histogram (Figure 1.5) has equal width for each bin, whereas the second histogram has different width for each bin depending on what values are clustered into the same clusters. The clustered bin histogram, Figure 1.6, is also a better representation of the quantized image. Both histograms show that there are three main hues in the quantized image, but the clustered histogram expresses more information. In Figure 1.6, the left most bar is thin and tall, which represents that the corresponding hues in this bin don't differ by so much in values, and there are many of these hues. From the quantized image, this can be deducted to be the orange color because orange takes a large region of this quantized image, and the differences among them are rather small comparing to the purple ones, some of which are very dark purple and others are very bright purple. With the analysis, the right most bin in Figure 1.6 has a large chance representing purple as it is thick and tall (large variation and large quantity). **Although the two histograms might have similar shapes and outlines, the histogram with equally spaced bins merely blindly divide up the hue space by their values evenly, which might be helpful to understand the overall distribution of values in the hue space, whereas the histogram with clustered bins has underlying knowledge about the hue space, so each bin represents some important information about the hues, which can be helpful to understand the hue distribution in the image.**

Figure 1.7 and Figure 1.8 are similar histograms of the hue space as Figure 1.5 and Figure 1.6 except with k value at 30. Like Figure 1.5 and Figure 1.6, similar comparison and contrast can be illustrated between Figure 1.7 and Figure 1.8. However, the difference between the histograms resulted from different k values is also interesting. Figure 1.7 actually has a similar shape as Figure 1.5, but Figure 1.7 seems to go in more details in each bin from Figure 1.5. Same analysis can be applied to Figure 1.8 and Figure 1.6. **With greater k values, each bin represents a more detailed hue distribution in the quantized image**.

While running these clustering, depending on where the centroids are initialized first, the results can be very different. Especially when k has a smaller value, there were times when different runs of the algorithm produced completely different colors in the quantized images due to different clusters. However, when the k value is greater, results from different runs don't seem to be different by too much because of the properties about greater k values discussed previously. As the output images can be significantly different at different runs, the histograms also differ by a lot in each run. When the k gets greater, the k-means algorithm actually kept producing some empty clusters, which is totally reasonable because on a 1 dimensional space, it is almost inevitable to produce some empty clusters in the result. For the results I produced here, many iterations of k-means had to be executed to avoid empty clusters.

2. **Circle detection with the Hough Transform**

a) Implementation explanation:

Before the process starts, the accumulatorArrays is initialized to have an extra space of the radius length appended to all sides of the image to account for any possible centers near the edge. AccumulatorArrays is then filled in two possible ways: without gradient or with gradient.

For the process without gradient, we assume that all detected edges are possible candidates to be part the circles we are trying to find. So with this assumption, for each pixel on a detected edge we are looking at, any pixel that is radius length away from the current pixel has a possibility to be the center that forms this circle. With this knowledge, we iterate through all the detected edges in the image, and each pixel would vote on the centers that it can possibly have in all angles. In the end the accumulatorArrays contains all the votes in each pixel as a score to be how many edge pixels believe this is the center. AccumulatorArrays is thus filled.

For the process with gradient, we also assume that all detected edges are possible candidates to be part the circles we are trying to find. However, for each pixel on a detected edge we are looking at, we don't just blindly vote for all the pixels that are radius length away from the current pixel; instead, we take the gradient of current pixel with respect to its neighbors, and will result in a dy (gradient along the y-axis) and dx (gradient along the x-axis). According to basic trigonometry, taking the tangent inverse of dy/dx will output an angle, at which direction the center might be if the current pixel is the on the target circle. However, depending on if the circle has a higher or lower intensity level comparing to its neighbors in x or y axis, the output angle might land in different quadrants, thus having 4 possible votes with one in each quadrant. With this more in depth knowledge, we just directly vote for the pixel that is radius length away from the current pixel in the calculated angle direction in each quadrant. With this voting procedure for all pixels on detected edges, the accumulatorArrays is filled.

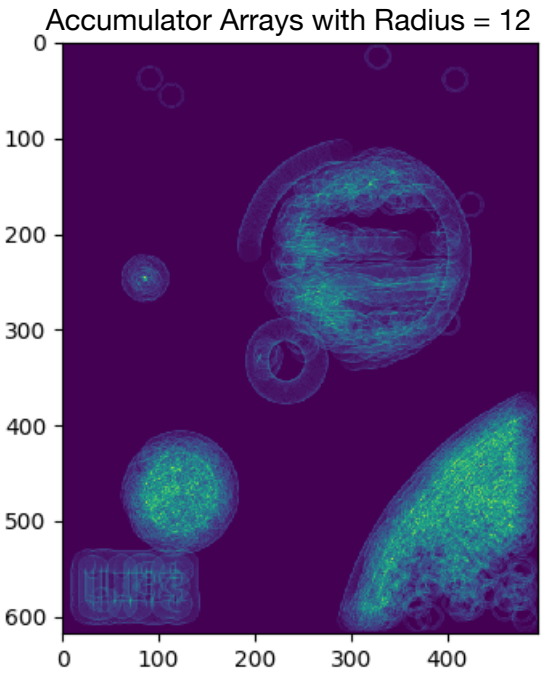
For the final voting process, there are two ways that I implemented:

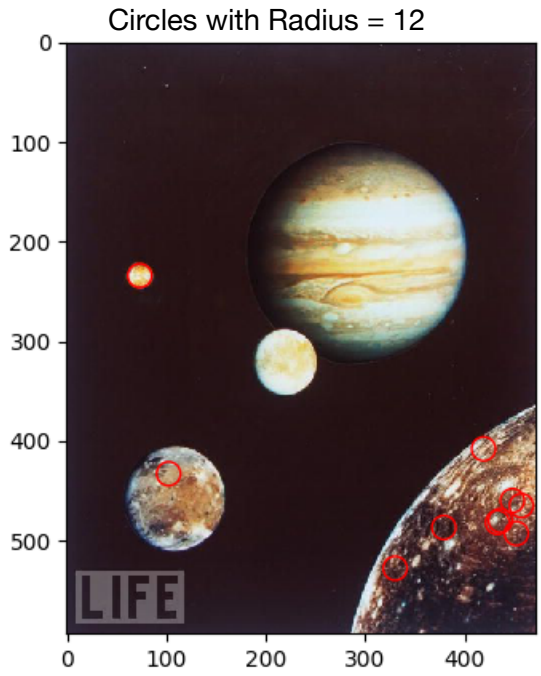
1. With bins of any size, the accumulatorArrays is dissected into these segments of bins. We simply pick the pixels that are the maximum within their corresponding bins, which should also be higher than or equal to the threshold. Each bin is limited to have at most 1 selected center, which should be the maximum within its own bin. Currently the threshold is set to be 90% of the highest pixel intensity value. These picked pixels thus form the official centers that are outputted. This method is extremely similar to non-maximum suppression.
2. With bins of any size, the accumulatorArrays is also dissected into these segments of bins. Each bin then sums up all the votes within the bin to construct a total score for the corresponding bin. Any bin with a score of higher than or equal to the threshold, which is set to be 90% of the maximum score, is selected. Finally the centers of the selected bins are outputted as the official centers. (commented out portion in the bottom of detectCircles.py)

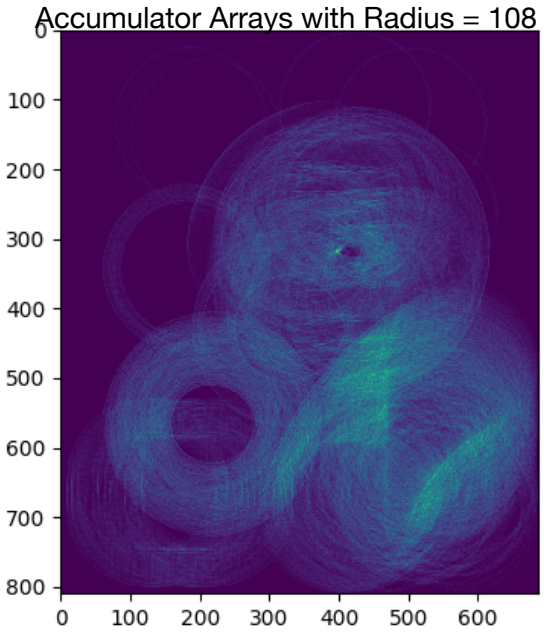
The first method aims to get rid of noise centers because sometimes there are many centers very close to each other, which are obviously representing the center for the same circle. In this case, the final circles detected are outputted in a cleaner manner. The second method aims to account for noisy edges. Sometimes, pixels from the edge of the same target circle might vote for different center pixels due to noise in the image. Thus providing a bin can allow these pixels to still vote for the same center.

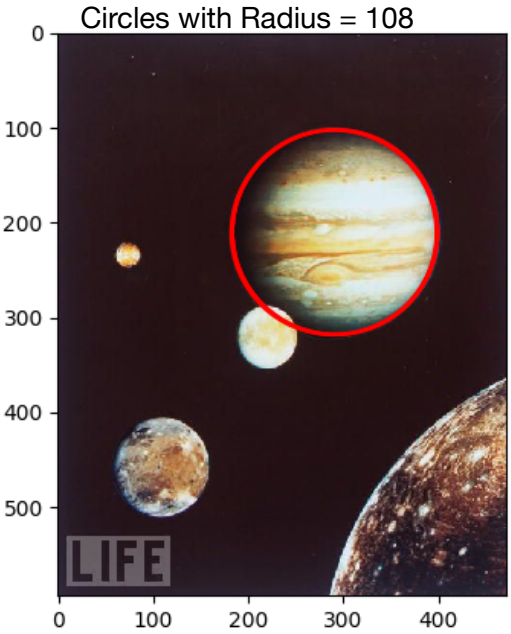
b) outputs on jupiter.jpg and egg.jpg

jupiter.jpg without gradient:



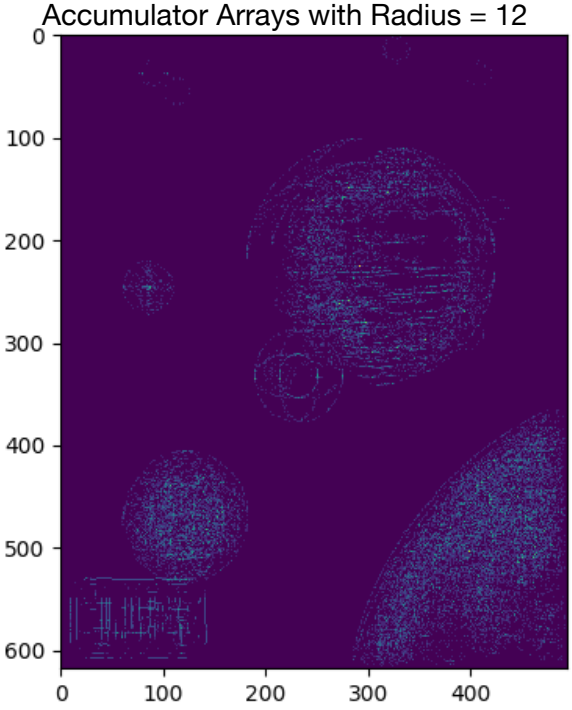




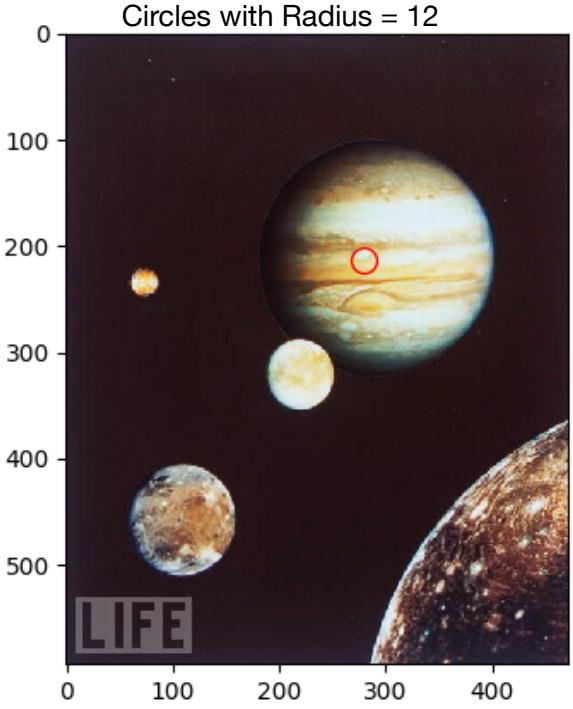


jupiter.jpg with gradient:

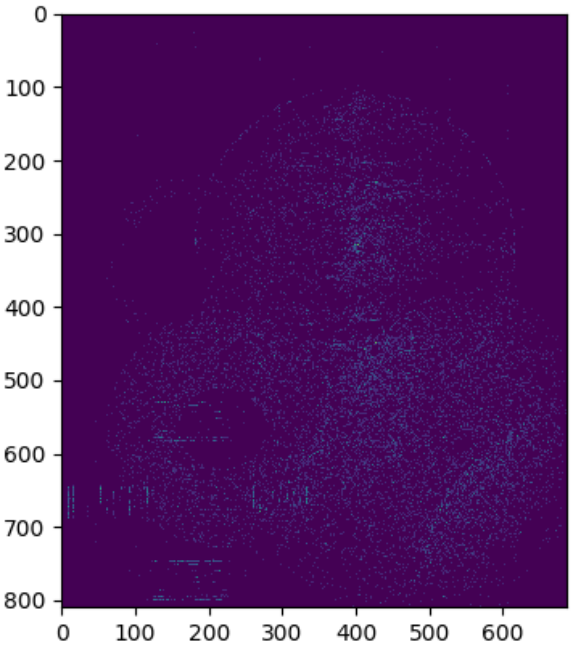
Accumulator Arrays with Radius = 12



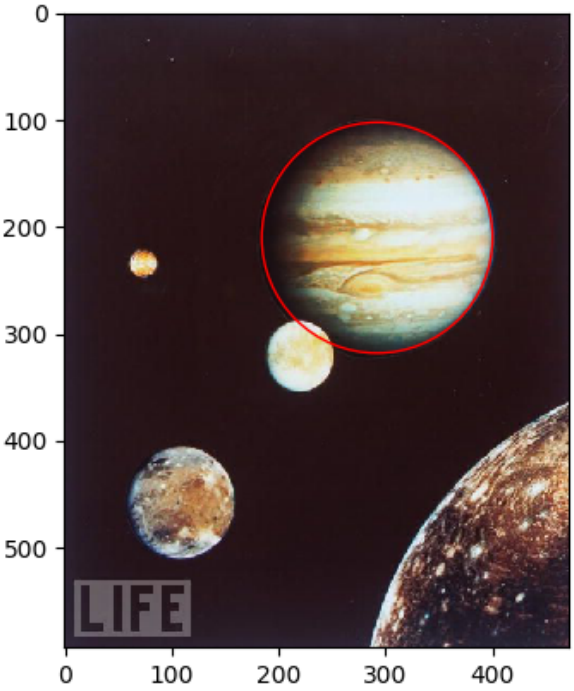
Circles with Radius = 12



Accumulator Arrays with Radius = 108

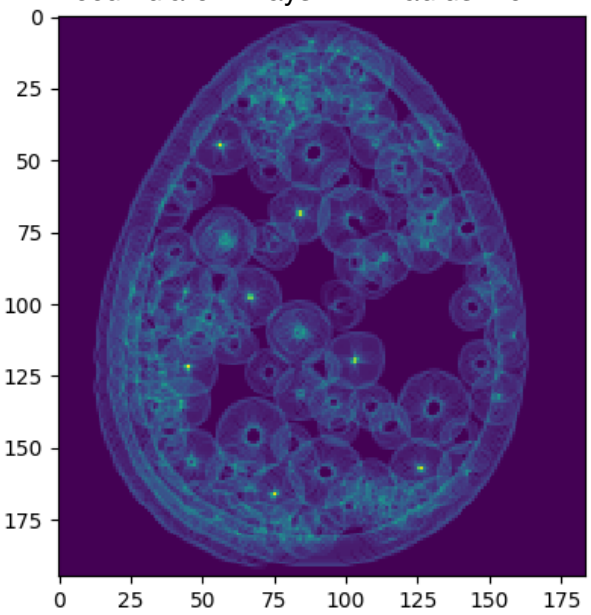


Circles with Radius = 108

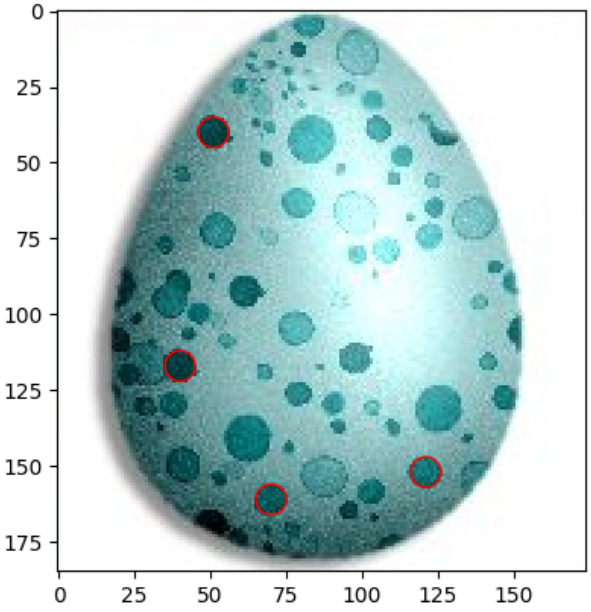


egg.jpg without gradient:

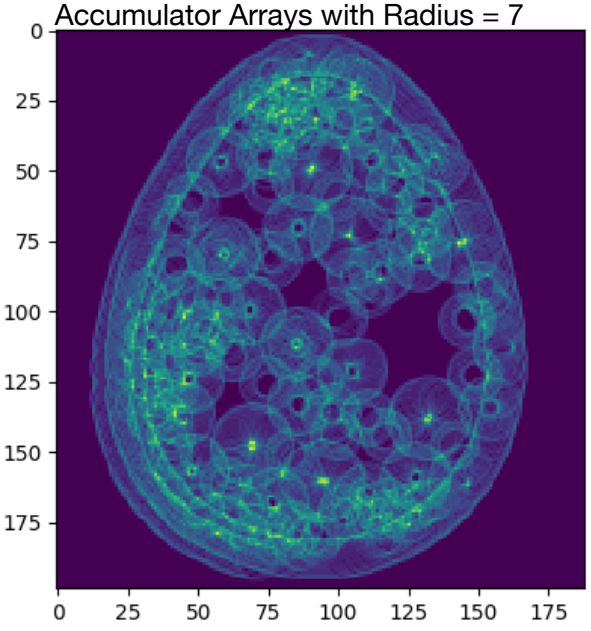
Accumulator Arrays with Radius = 5



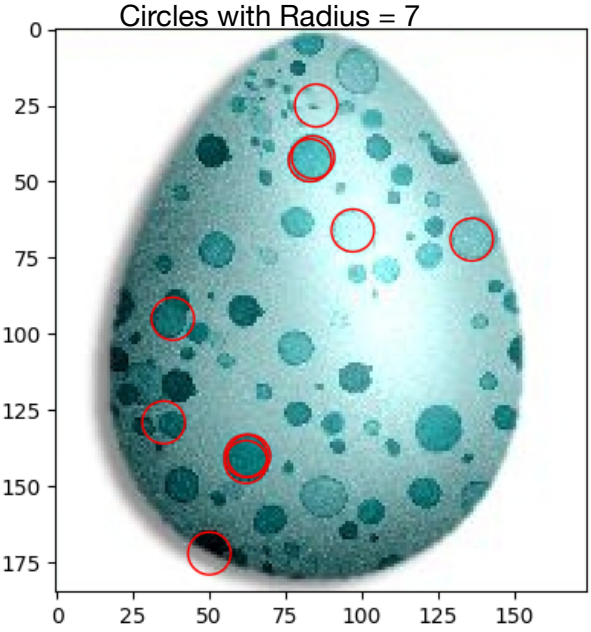
Circles with Radius = 5



Accumulator Arrays with Radius = 7

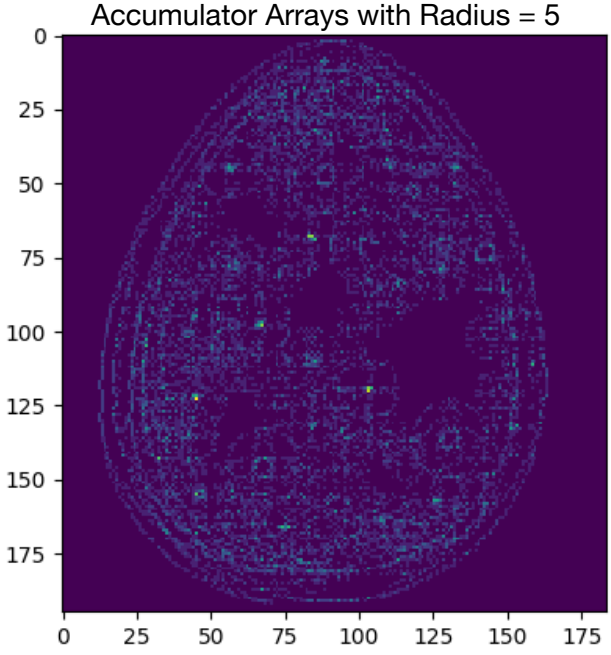


Circles with Radius = 7

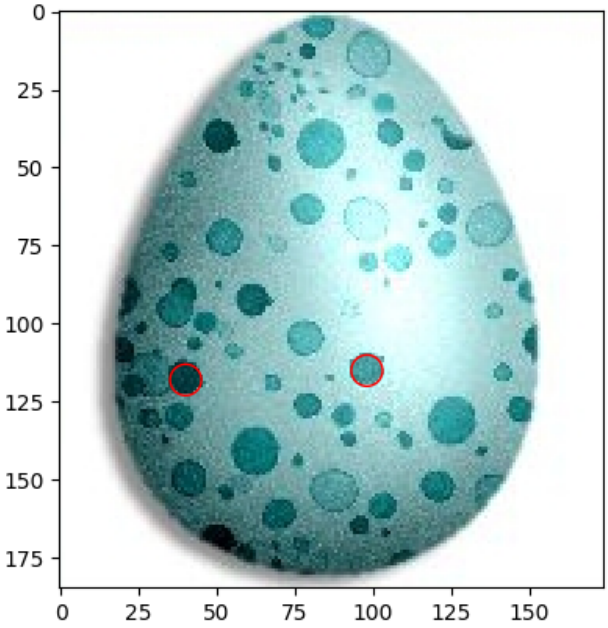


egg.jpg with gradient:

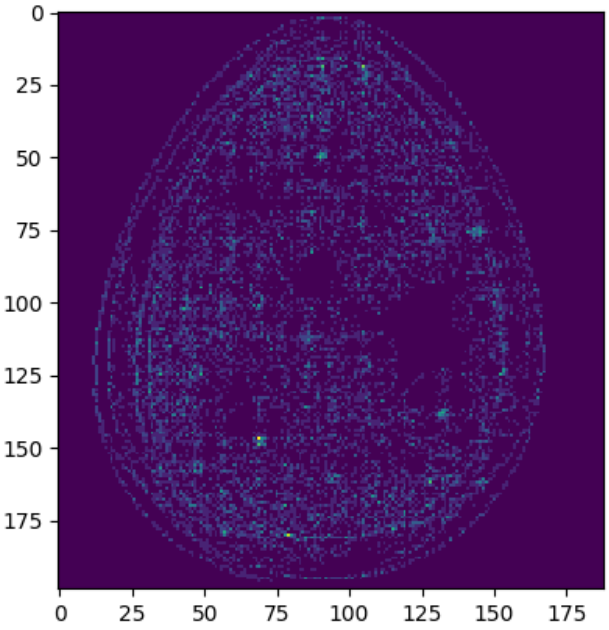
Accumulator Arrays with Radius = 5



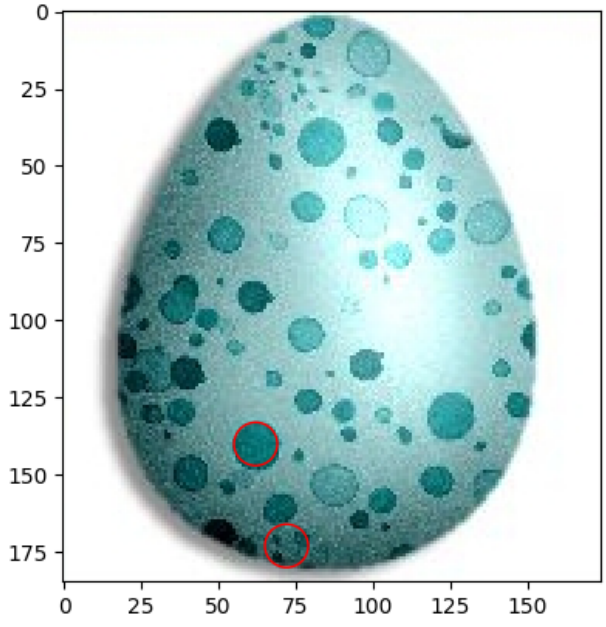
Circles with Radius = 5



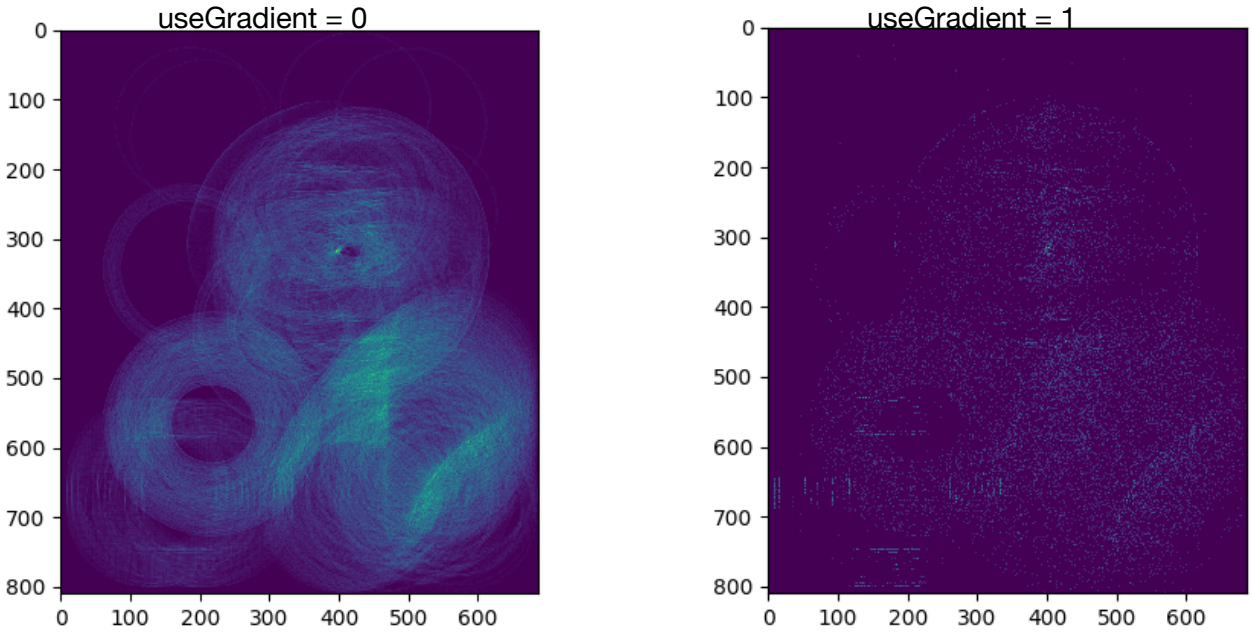
Accumulator Arrays with Radius = 7



Circles with Radius = 7

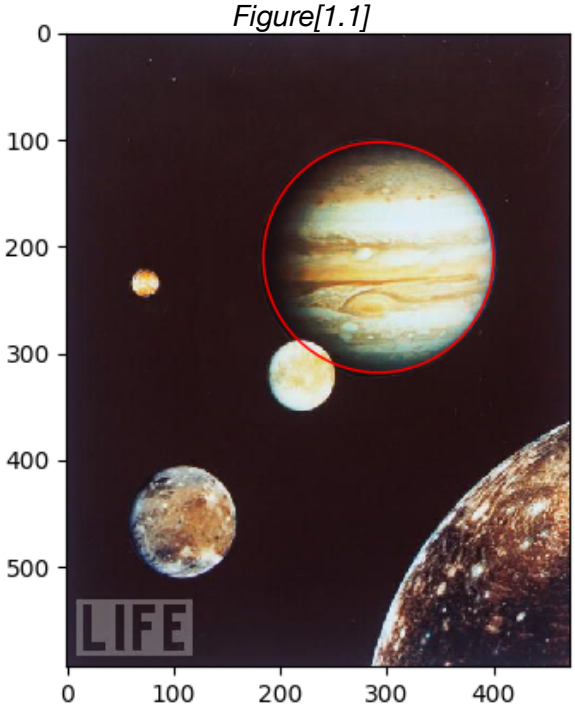


c) jupiter.jpg at radius = 108

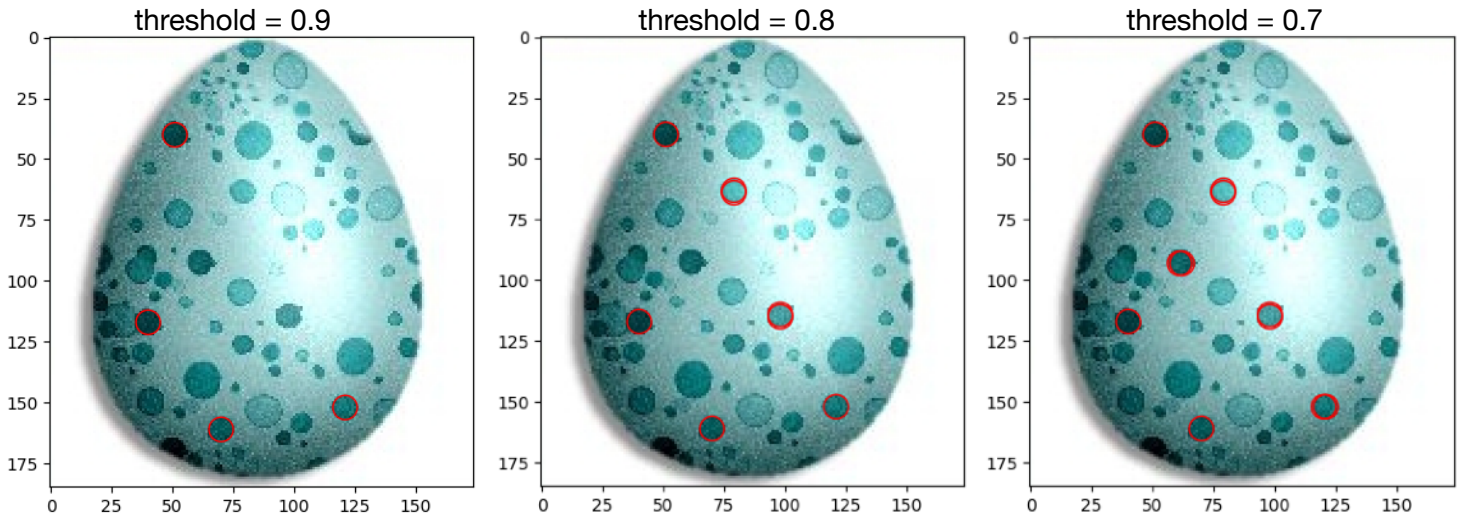


In the accumulator arrays displayed above, the left one doesn't use gradient, and the right one uses gradient. In the accumulator arrays, the more intense a pixel is in the image, the more votes it has. Apparently, the right one has way less total votes than the left one has, which makes sense because on the left one, each pixel on the detected edge votes for all pixels that are radius distance away from it, whereas, on the right one, each pixel on the detected edge only votes for 4 pixels. For this reason, although they come from the same image, they have very different presentations in the accumulator arrays. On the left one, it seems like that many circles with high intensity are displayed on the image because for each pixel on an detected edge, if it votes for all pixels radius distance away, it basically forms a circle centered on itself. For this reason, the left one, without gradient, reasonably consists of many circles of votes. On the right one, the votes seem more scattered around without forming any apparent patterns because of the small amount of votes from each pixel on the detected edges. There are just not enough votes to form any apparent pattern on this image. The one without gradient seems a little more intuitive and more understandable than the one with gradient because the patterns in the one without gradient seem to relate to the patterns in the original image a lot. Each vote can be almost traced back to where it came from. Although these two accumulator arrays are different in many ways, they actually result in exactly the same output. It is actually visible in the two accumulator array images that the most intense pixels in both images are at around the same coordinates. The image below (Figure1.1) is also the output image from both of the accumulator array.

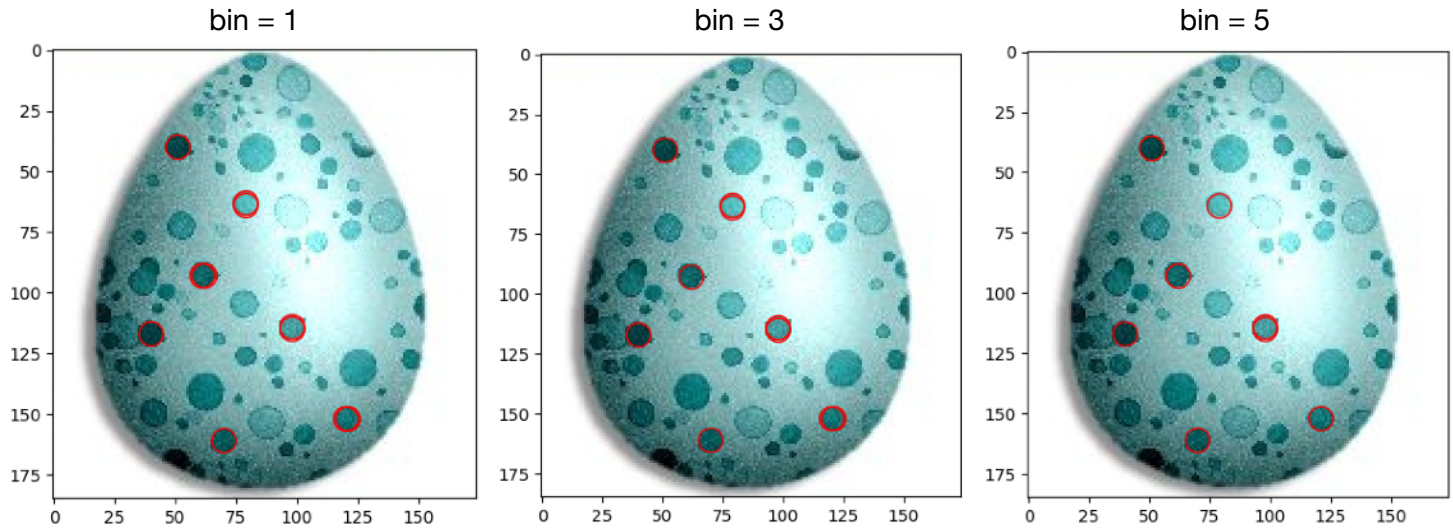
Figure[1.1]



d) post-processing the accumulator array for the number of circles



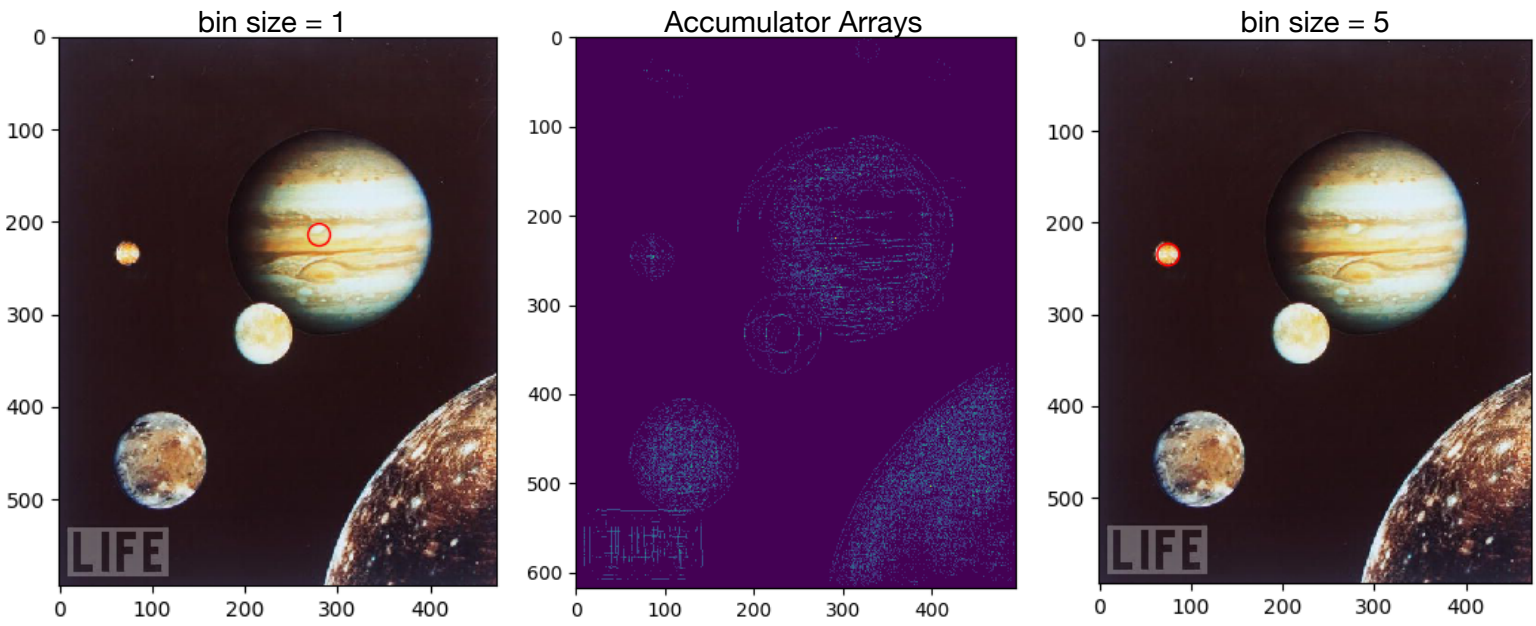
The threshold level directly indicates how many circles will be detected and displayed. If a threshold is at 0.9, that means that any pixel in the accumulator arrays with the vote count of 90% of the maximum vote count or above will be selected. A smaller threshold means more centers will be selected. However, it is also obvious to see that, as the threshold goes lower, there are more overlapping circles displayed: 2 sets of overlapping circles at threshold of 0.8 and 4 sets of overlapping circles at threshold of 0.7 (might need to zoom into the images to see). To make the image cleaner, a concept of bins is introduced. Not to confuse with the other bins concept, this one is the first bins method I mentioned in the implementation explanation, where each bin is limited to have at most one vote, which sounds a lot like non-maximum suppression.



The bin size here effectively removed the overlapping circles, which made the detection a lot cleaner and smarter. When the bin size is at 1, there are 4 sets of overlapping circles. When the bin size is at 3, there are 3 sets of overlapping circles. When the bin size got to 5, there is only one set of overlapping circles.

Lower threshold effectively selects more circles from the image, and higher bin size effectively removes overlapping circles, which are caused mainly by noise. These two methods effectively determine how many circles are presented and detected. With a set of well balanced values of threshold and bin size, the circle detection can be very powerful, thorough, and intelligent.

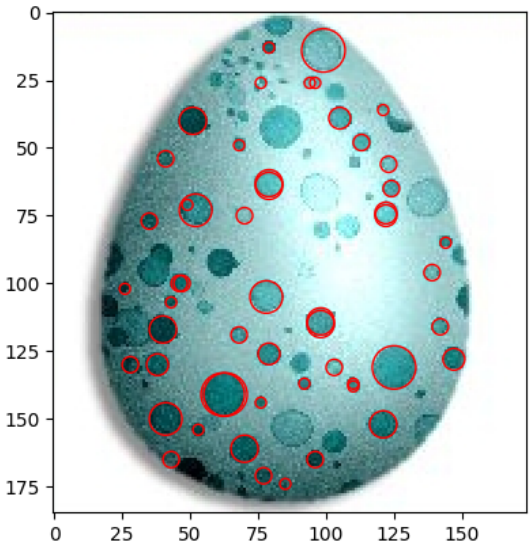
e) vote space quantization



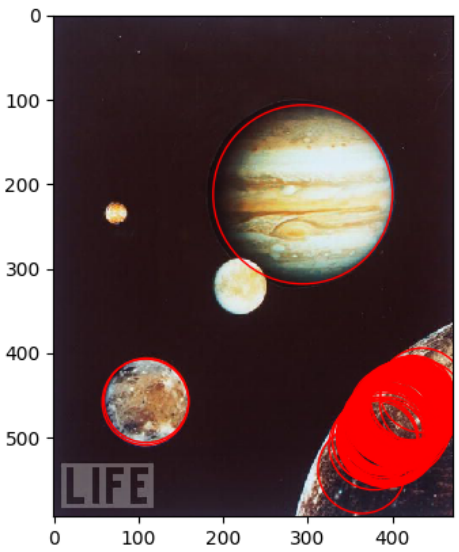
In this bins method, each bin is used to cluster the votes. All votes fall into a bin will count toward the bin, and a total score is calculated as the score of this bin. The bin with high scores wins, and the center of the bin will be returned as the elected center even though the highest voted pixel might be some other pixels within the bin. This method really helps with the noise in the image to allow better circle detection. In the left image, when bin size equals to 1, the pixel with the highest intensity/vote is selected. However, due to the high level of noise in the image, a wrong circle is detected and returned. In the circle that is returned, it does have a center with high intensity, but it was due to the noise around it. In this case, the noise actually over-powers the actually circle I am trying to find. In the accumulator arrays, it can be seen that although the highest intensity pixel is not in the target circle, the target circle does have a center with high intensity, but just not strong enough. In more details, the center in the target circle is not strong enough because the votes are not concentrated in one single pixel, instead votes all fall into adjacent pixels due to noise on the edge. Increasing the bin size can fix this issue because, with a larger bin, even though the votes are not falling into the same pixel, we make sure that they fall into the same bin, so that the sum still over-powers the noise. The image on the right illustrates the circle detection after a bin size of 5 is applied. After increasing the bin size, the correct circle is outputted. Notice how the accumulator arrays are unchanged, but the right bin size is able to fix the issue. Generally, from my observation, accumulator arrays using gradient tends to benefit more from greater bin size because less vote means that each vote counts more. This means that if the votes are under the influence of noise, the results will be more noisy, and such noise is easily fixed by greater bin size.

Extra Credit:
To find circles of any radius, just a simple modification is needed in the current system. To better illustrate this extension, gradient and bins won't be used. The extension will be based on a basic circle detection without gradient and a bin size of 1. The 2d accumulator array needs to be changed to a 3d accumulator array, with a new dimension added to map different values of radius. The main procedure mostly stays that same, which is that for each pixel on an edge, iterate through the angle space which is from 1 to 360. For each angle, iterate through all possible radius value. This is a newly added step because we had radius provided to us before, so we didn't need to worry about finding the radius. For all these possible combinations of x, y, theta and radius, calculate the a and b values using the same formula we have been using. After a and b are calculated, voting can be done using the indices a,b and radius. The main difference between this problem and the previous one with the radius provided is that one more dimension needs to be added to accumulator array. Instead of voting in the 2d accumulator space, in the new one we vote in the 3d accumulator space. The final output centers consists of the center coordinates and also the radius that corresponds to this center coordinate. Normally, this algorithm would run for a very long time, so I added in some constraints for r, minimum radius and maximum radius, to define a smaller range of possible radii of detection. For the egg image, the threshold is kept at 0.7, and the result is pretty nice. For the jupiter image, the image seems very noisy, so I had to use two thresholds (lower threshold of 0.7, higher threshold of 0.8), and the centers are chosen when they are lower than the upper threshold and higher than the lower threshold to cap out some the noise points. (the bottom commented out portion of the detectAllCircles.py)

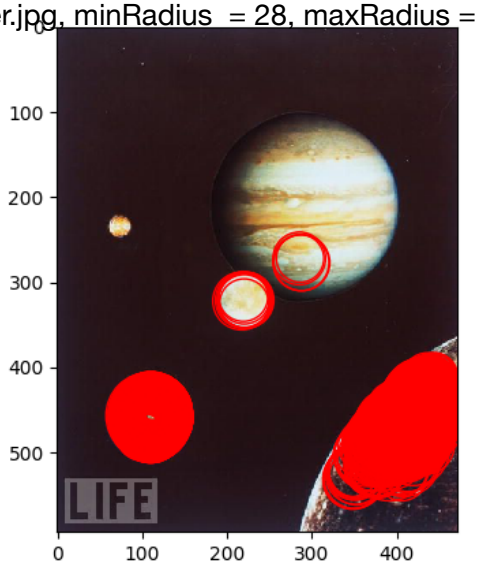
egg.jpg, minRadius = 2, maxRadius = 10



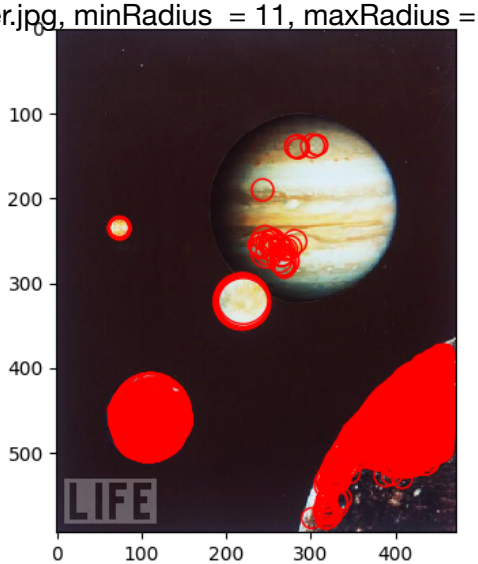
jupiter.jpg, minRadius = 50, maxRadius = 110



jupiter.jpg, minRadius = 28, maxRadius = 50



jupiter.jpg, minRadius = 11, maxRadius = 40



Other Files:

main.py
- run 'python2 main.py' to recreate the part of the answer sheet including the circle detection and the extra credit. (Before running, choose the radius value and the image) (The extra credit might run for a long time)

detectAllCircles.py
- main.py can be used to run this file. This is the extra credit portion of this assignment

progressBar.py (all usage commented out)
- helps to print a progress bar in command line as the warping process is pretty slow, so the user knows how the process is going
- very buggy when running on Mac small-size terminal, but seems fine when running in PyCharm terminal or maximized screen Mac terminal

more images
all images outputted in this process are in the folder with meta data as their titles