MSML640
Assignment -3 Report

1. The orientation of the filter shows the direction of the texture or intensity changes in the image that will be detected. With that being said, the provided filter bank consists of all angles of orientation and so will be efficient in detecting changes in every orientation. Hence a filter bank made from 12 different oriented filters will be insensitive to orientation.

2. K-means will not produce a relevant result for the edge points because K-means is a distance based clustering algorithm which creates new centroids every iteration and assigns data points based on minimum distance from the centroids. This method will split the edge points into different circular clusters which will not help in this case. Most probably, with k=2, kmeans will classify all points on one side into a cluster and all points on another side into the other cluster. Hence using K-means will not be useful and we have to use a better suited algorithm like DBSCAN.

3. Mean shift algorithm would be the most efficient option to use for Hough transform because the end result is the center point around which most density of the data points is seen, which is similar to getting the maximum vote area for Hough transform. K-means divides data points into clusters around allocated centroids based on distance. But for Hough transform , we expect the output to one location with maximum votes. Graph-cut algorithm cuts out the links with lowest weight between pixels. Hence K-means and Graph cut algorithms are not useful in this case.

4. *Pseudo code*

```
func get_contours(blob):
  {
      Contour_blob = contours(blob)
      return contour_blob
  }

func get_boundingbox(countour,blob):
      {
      bb_dim=[ ]
      for c in contour_blob:
            x,y = boundingrect(c)
            bb_dim.append(x,y)
      return bb_dim
     }

func get_dimensions(blob):
```

```
                {
                    dim= [ ]
                    for b in blob:
                              contour_blob=get_contours(b)
                              bb_dim=get_boundingbox(contour_blob,b)
                              dim.append(bb_dim)
                              dim=[bb_dim,b]
          return dim
          }
```

Func get_shape(dim):

```
    {
      Properties  []
          Height = x2 - x1
          Width = y2 - y1
          area  = height*width
          Orientation = angle(width, x-axis)
          properties.append([height, width, area,orientation])
   }
```

func k_means(properties,K)

```
                    return k_means
```

Variables:
blob = iterator for all the blobs
Contour_blob = object variable storing contour of the blob
Bb_dim = variable storing x,y coordinates for the bounding box for the blob
dim = the combined list of all x,y coordinates for all blobs and append the blob to the list
Properties = Array with all properties we are interested in. These can be height, area, width, diagonal, rotation, percentage of bounding box filled etc. These become input to kmeans.
K = the number of clusters variable for the K-means

Algorithm:
We loop through the blobs to get the contours of the blob
Loop through the contours to calculate the bounding rectangles
Store the bounding rectangle coordinates as dimension data points for the blobs
Use the coordinates to come up with different properties for blobs that we are interested in.
Pass these properties as input to the K-means clustering algorithm
The K-means algorithm would group the blobs based on the bounding box properties
This will cluster blobs based on their shape’s properties.

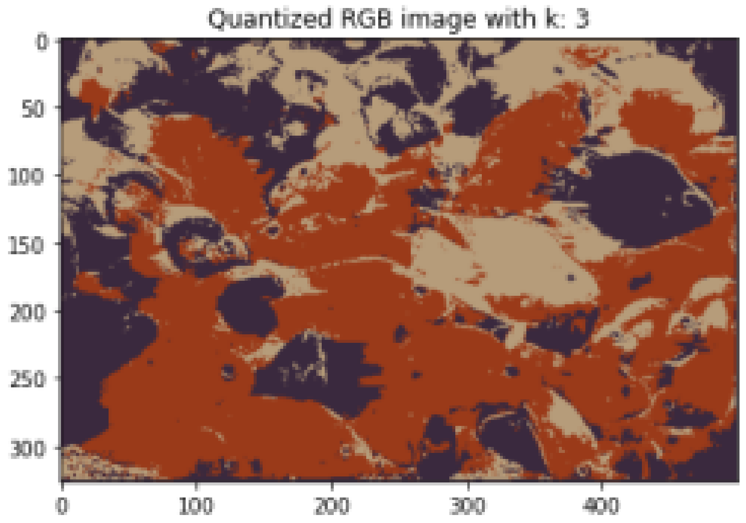
Programming

1e.

RGB Images

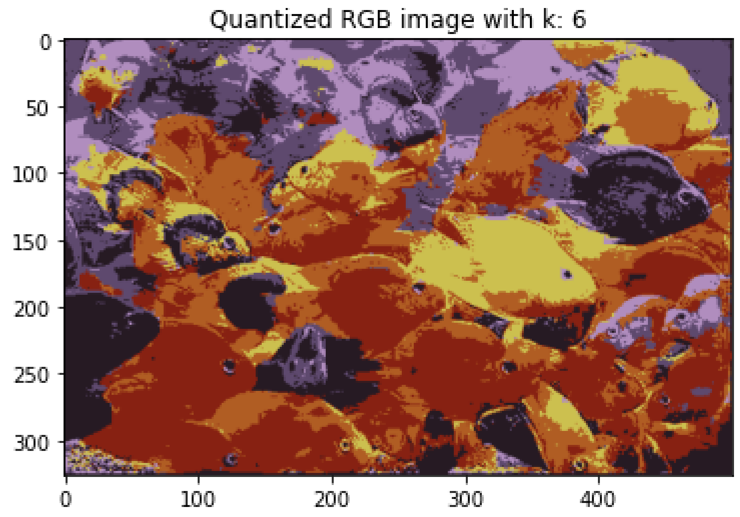
K=3

SSD = 721359300.0



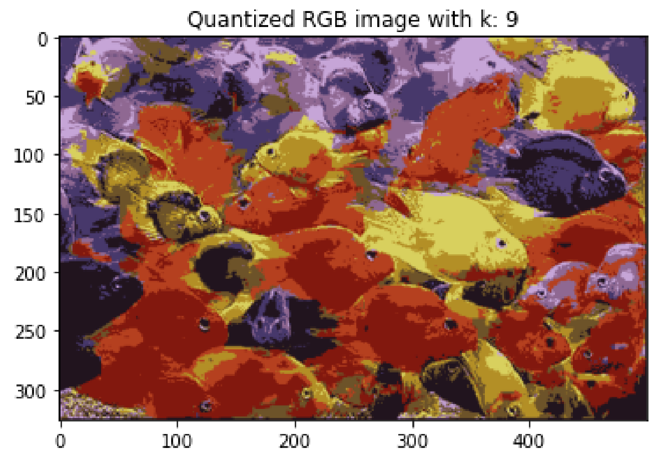
K=6

SSD = 385425300.0



K=9

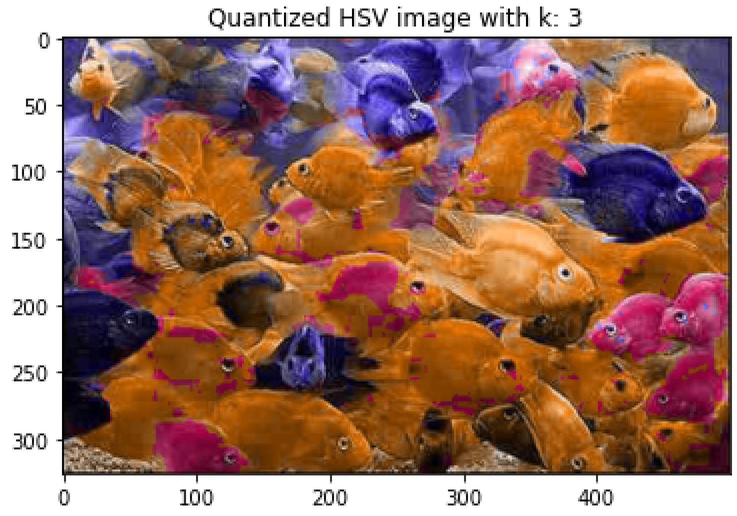
SSD= 283841950.0



HSV Images

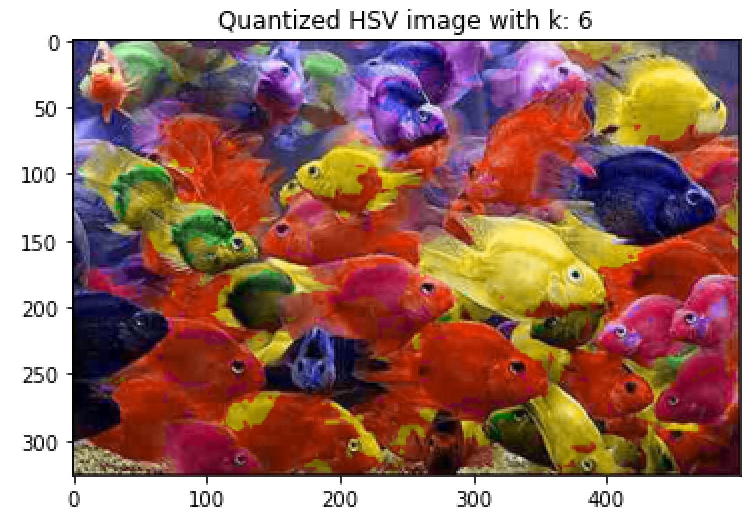
K=3

SSD= 215624220.0



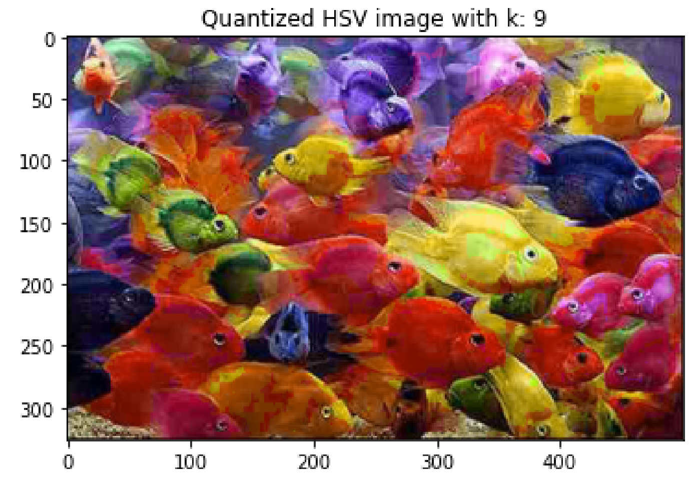
K=6

SSD =59976350.0



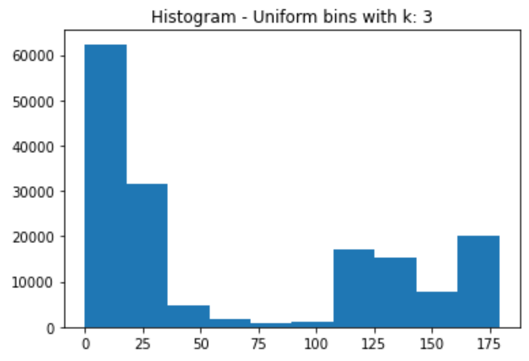
K=9

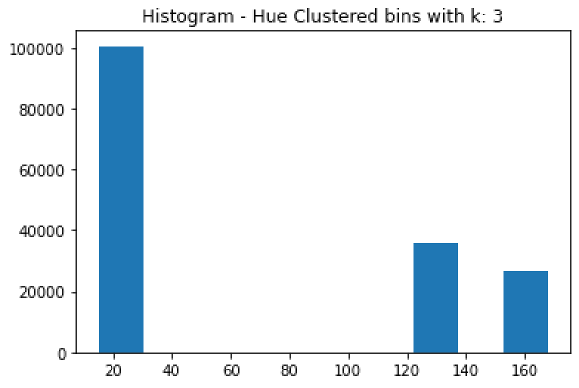
SSD = 29995730.0



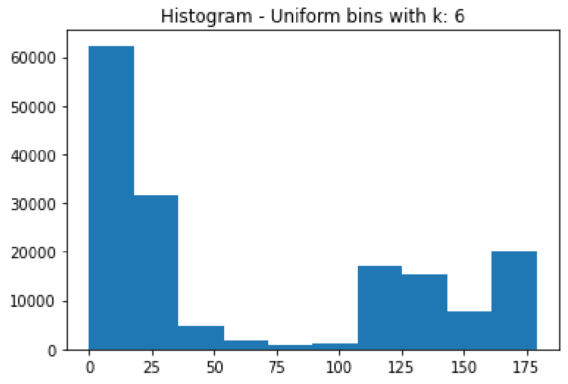
1f. Histograms with Uniform Bins and Hue Clustered Bins

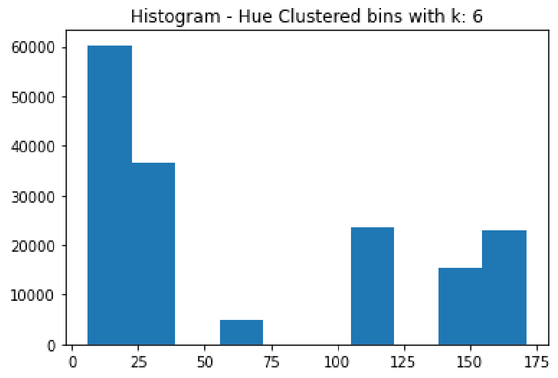
K= 3



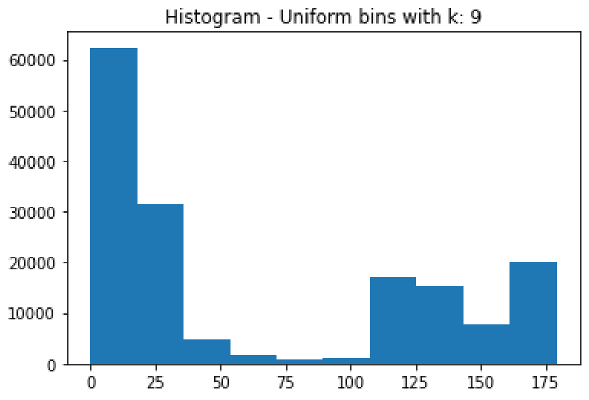


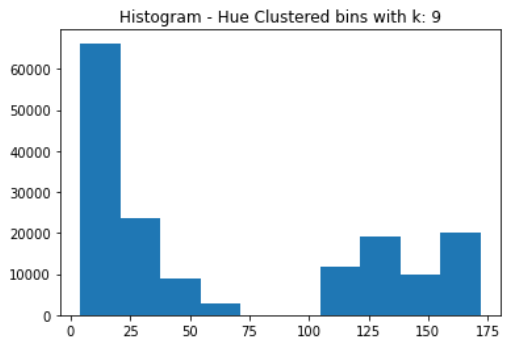
K=6





K=9





Histogram comparison:

The figures above show histograms for 2 cases:

1. Normal histogram of the hue column with default bins
2. K means clustered histogram of column k

Normal histogram shows the frequencies of hue values of pixels in equidistant segments. This is useful in identifying which ranges of Hue generally have more values.
With clustering, the pixels within the same cluster fall into the same bin in the histogram. Since KMeans creates circle-like clusters, it is expected that all the points grouped together form a range. This histogram is useful in identifying which clusters have more frequencies and which have less frequencies.

Plot comparison:
Looking at the plots, it's easy to identify that there is more information loss for RGB clustering compared to HSV clustering. With the same k, the HSV clustering is much closer to the real image than the RGB image. However, most of this could be attributed to the fact that we applied clustering to all the channels of RGB while we only applied clustering to the Hue channel for HSV. Because there's more clustering one to the RGB image, it does a better job at segmenting the different parts of the image into separate clusters based on color.

Error Comparison:
As expected, the error for the RGB clustering is always more than HSV clustering. Error for a specific type of clustering decreases as k increases. This basically shows that as we increase K, there's less and less information loss from img1 to img2. When K reaches the actual number of RGB pairs/ Hues; the error will drop to zero. As one can imagine, this K value will be higher for RGB clustering as there are simply more unique RGB pairs than unique Hue values.

2a. Hough Circles Implementation:

The function takes an image of shape (M,N,3) , a fixed radius and a setGradient boolean flag which decides if we want to use the gradients of the image.
The ndarray is assumed to be an rgb image and is converted to a gray image. This retains the edge but reduces computing. This gray image has edges but also too many noise values. Medium Blur is used because it is particularly effective against noise. After smoothing, Canny edge detection was used. This reduces the gray image to an edge image with 1 pixel width edges. We extract all the edges **before** starting the hough loop because this greatly reduces computing.

Hough Space:

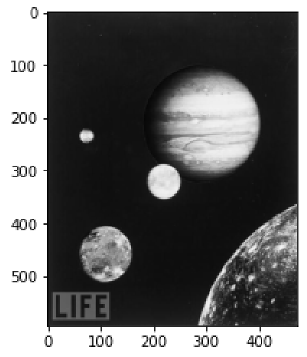
Here, we have to initialize the hough space which is the same as the input image size. However, for programming purposes, we don't need to do this right away as it makes more sense programmatically to just store the vote values in a dictionary and recreate the accumulator image after the complex processing is done.

Voting: (with and without gradient information**)**
We use the parametric equation of the circle to start voting in hough space. Since we already have a fixed radius, the only indices required are x,y values. ($x = xc + r\cos\theta$, $y = yc + r\sin\theta$)
Here, we use the "useGradient" flag. If this is set to false, we can just use θ as range(0,360,step_size). If this is set to true, we can compute the gradient of the image with sobel, and get 2 points based on arctan function of the angle of the gradient. So, the computation is greatly reduced (a couple of seconds for one image) but since each point only votes for 2 points, there's more probability of error. However, it is observed that the most voted circle is always right in either case.
Once we have x,y values; we can update the dictionary by using (x,y) as a key and incrementing the value by 1 (we can use whatever integer we want, results won't change)
Once all pixels that are a part of the edge have finished voting, we will have a large dictionary with votes. This can be used to build the accumulator array with x,y values.
Top N values in the dictionary can be used to find circles. Focusing on the most voted center for a fixed radius always gives dependable results.

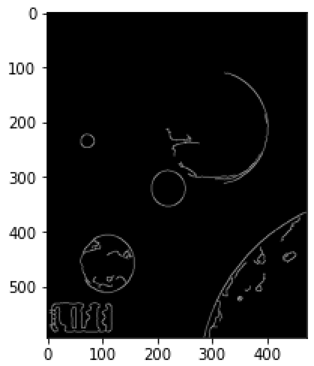
2b. <u>Detect Circles Results</u>

JUPITER:

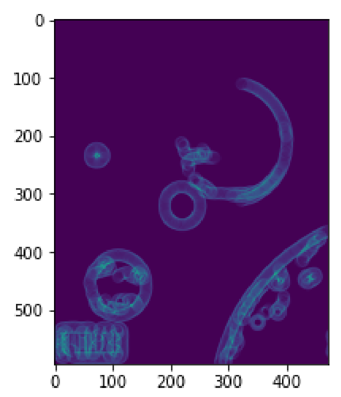
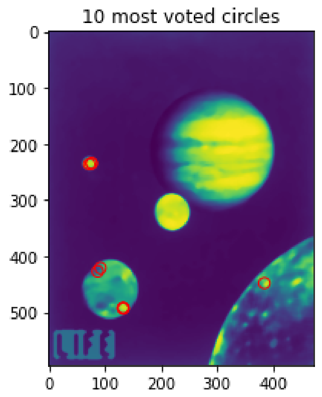
Input Image



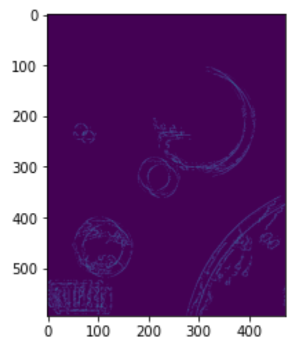
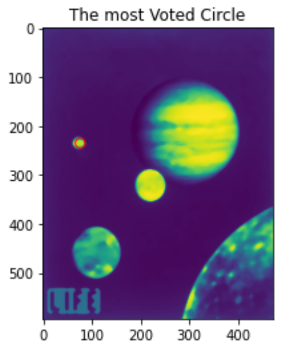
Edge Map

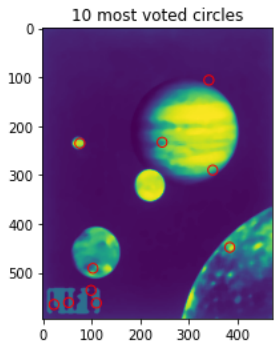


- UseGradient = 0 , **Radius = 10**:

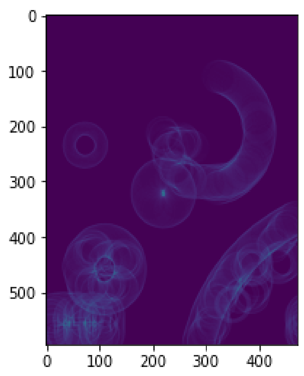
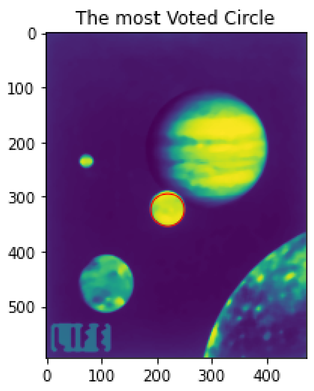



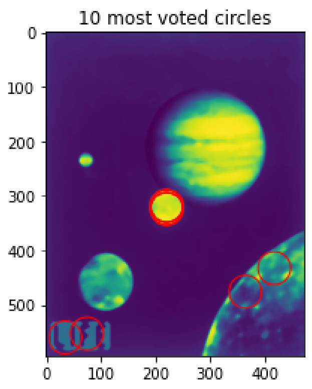
- UseGradient = 1 **Radius = 10:**

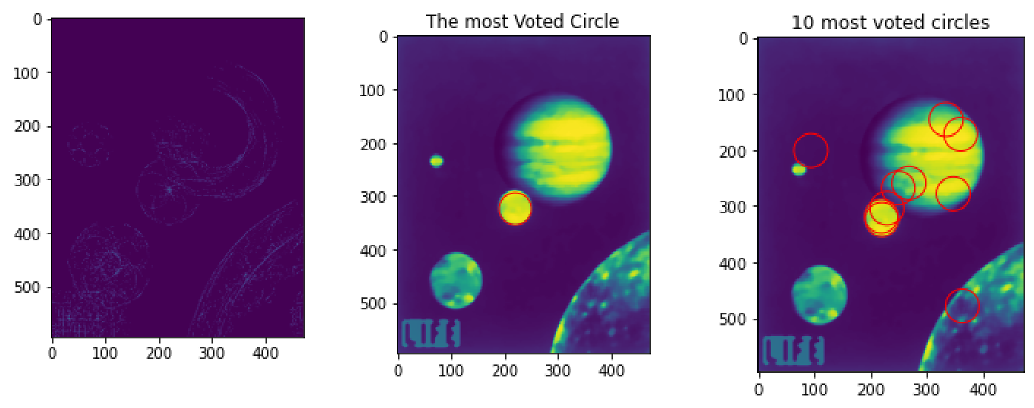


- UseGradient = 0 **Radius = 30**

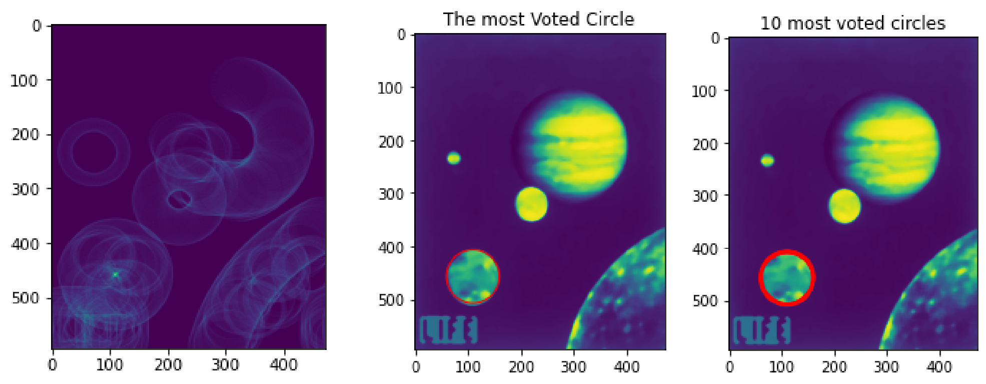





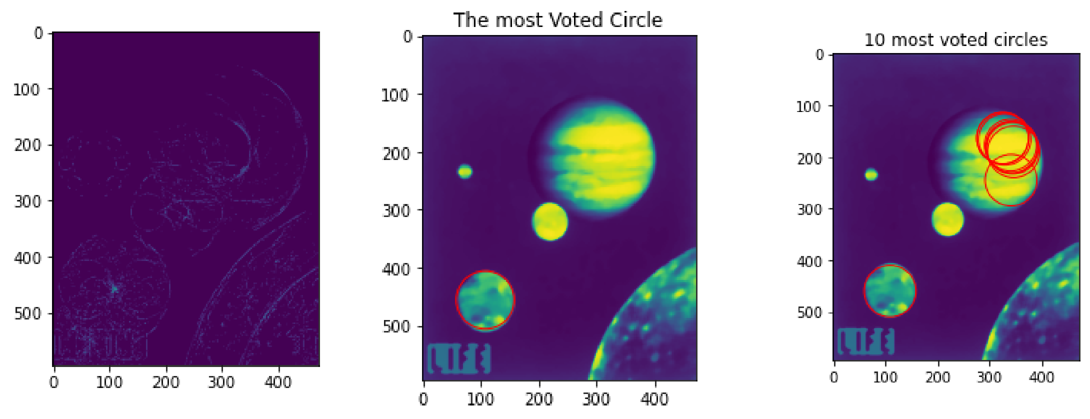
- UseGradient = 1 **Radius = 30**



- UseGradient = 0 **Radius = 50**

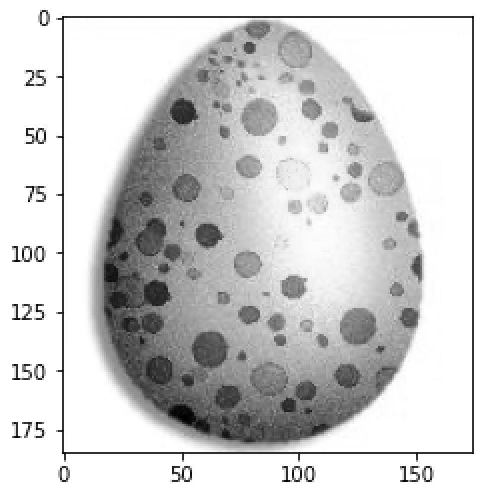


- UseGradient = 1 **Radius = 50**

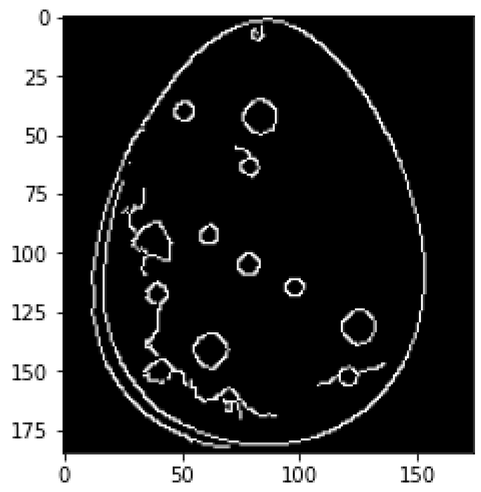


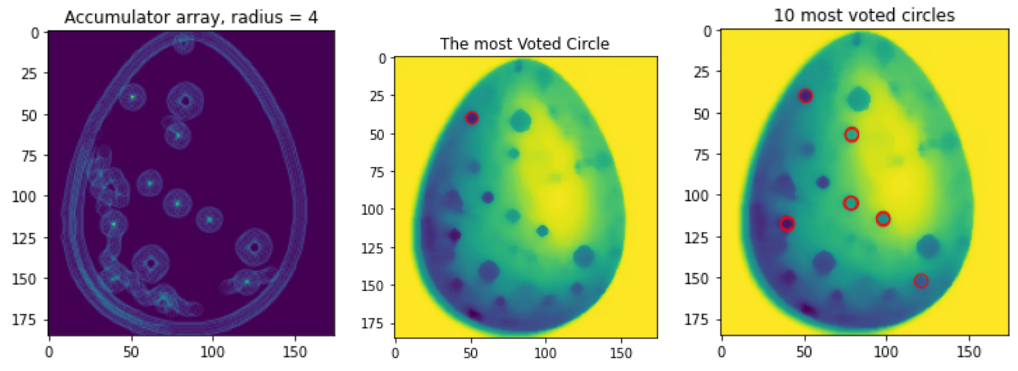
EGG

Grey Image:

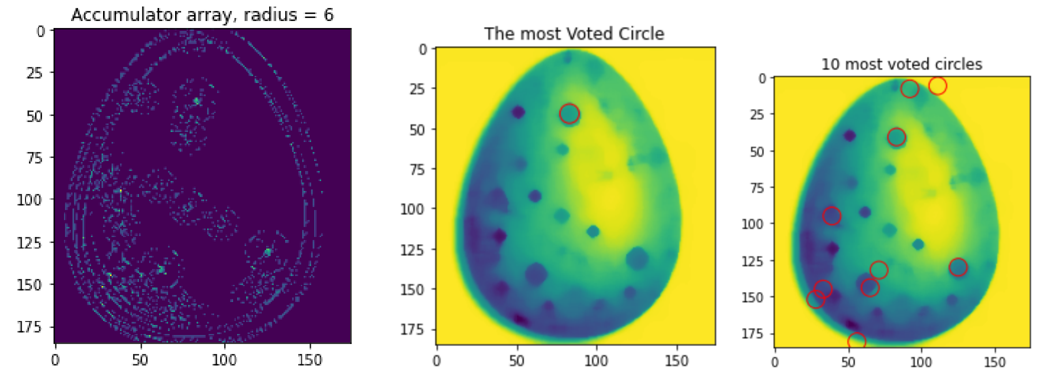


Edge Map:

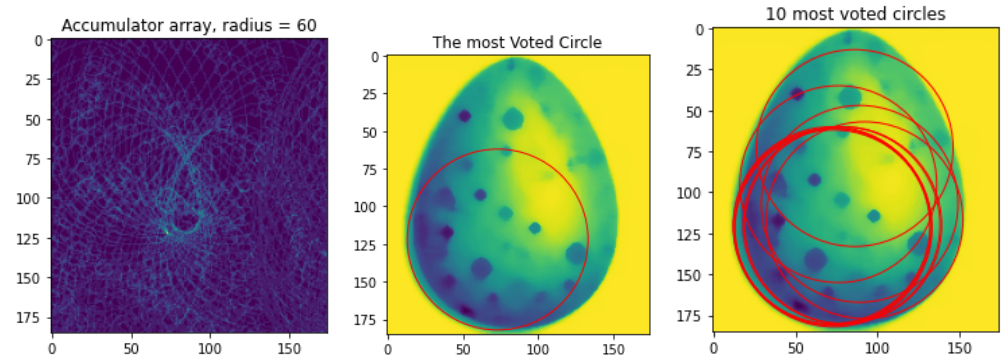




(Gradient is False)

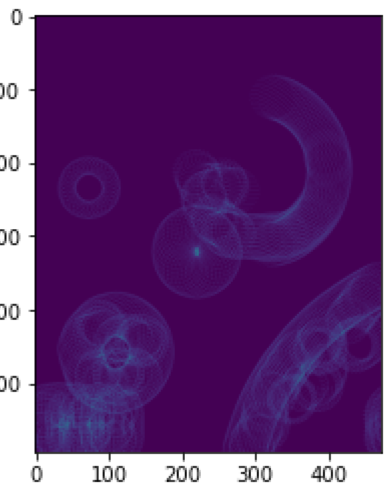


(Gradient is True)



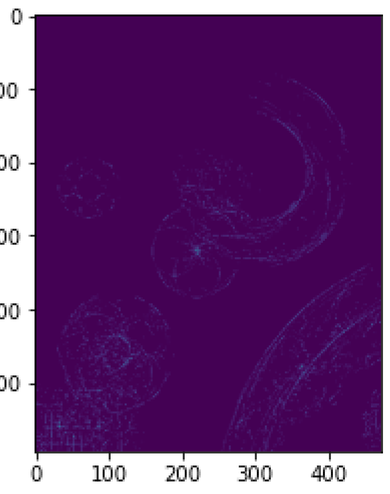
(Gradient is False)

2c . <u>Accumulator Array for R =30 and UseGradient = False</u>



The hough space is the ndarray which stores the voting of all the edge pixels in the edge image. All the edge pixels that correspond to 5 different circles and “life” stamp have contributed in voting for circles around them with a fixed radius r, which in this case is 30. However, we see that the brightest point in the image is corresponding to the sphere in the center, whose radius is the closest to 30. There are other bright points too, because of noise, but they are outvoted by the actual circle center.

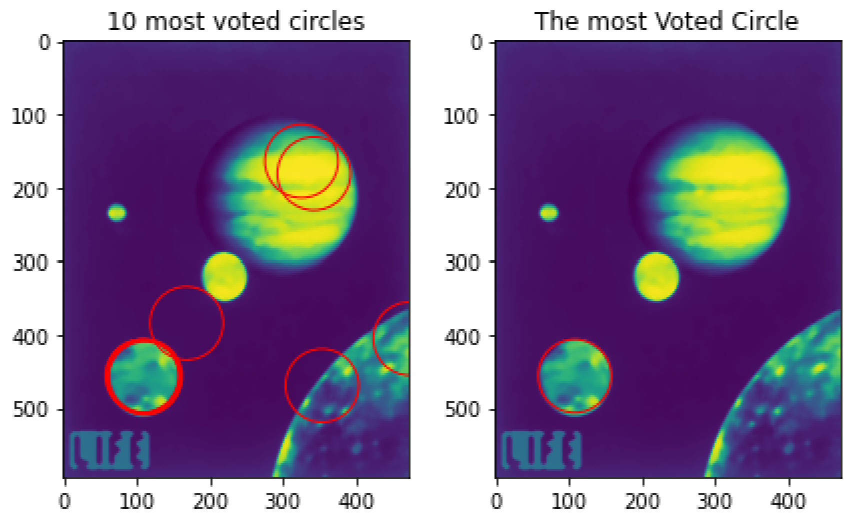
Accumulator Array for R =30 and UseGradient = True:



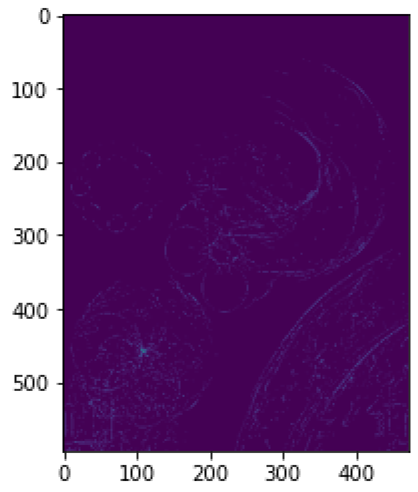
In the accumulator array created by using the gradient, there are far lesser votes compared to the previous array. However, the most votes are still bagged by the true circle in the center.

2d.

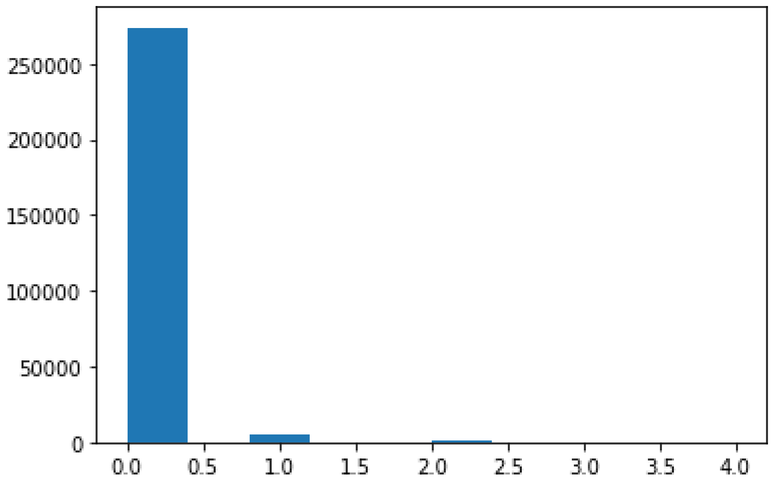
Plots when useGradient=True for radius 50

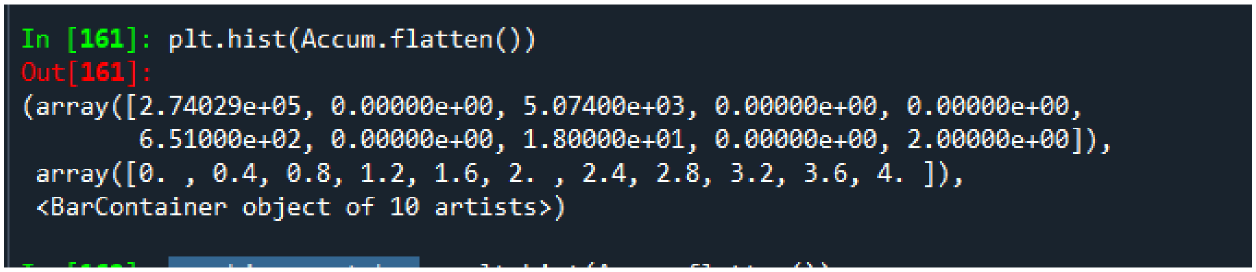


As we can observe from the image, which the true circle got the most votes, it is hard to identify where we need to stop considering the circles. The values of the accumulator array are heavily dependent on edge detection, smoothing choices and hyperparameters and the theta steps and bin sizes. Hence, it is not always known apriori what 'N' is supposed to be in "N most voted points"
One way to deal with this is to post-process the accumulator array. Consider the accumulator array for the above example:



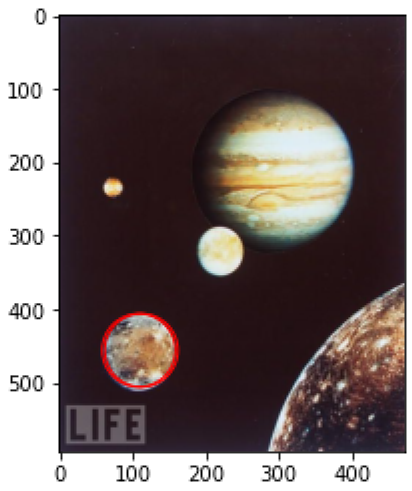
It is clear for a human that there's only one true center. In order to recognize this from post-processing, consider the histograms for the accumulator array:





```
In [161]: plt.hist(Accum.flatten())
Out[161]:
(array([2.74029e+05, 0.00000e+00, 5.07400e+03, 0.00000e+00, 0.00000e+00,
        6.51000e+02, 0.00000e+00, 1.80000e+01, 0.00000e+00, 2.00000e+00]),
 array([0. , 0.4, 0.8, 1.2, 1.6, 2. , 2.4, 2.8, 3.2, 3.6, 4. ]),
 <BarContainer object of 10 artists>)
```

Things become incredibly clear that the values are forming natural clusters. It is not visible in the histogram, but the bin with the highest value (4) has 2 points in it.This is the bin we are interested in. When we plot the circles for only this bin, we get:



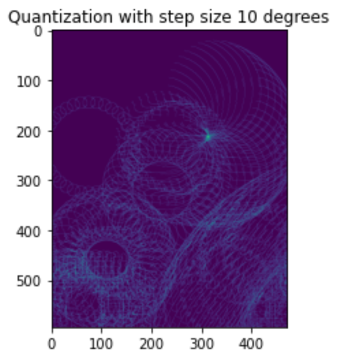
This is how post-processing the accumulator array can help us find circles.

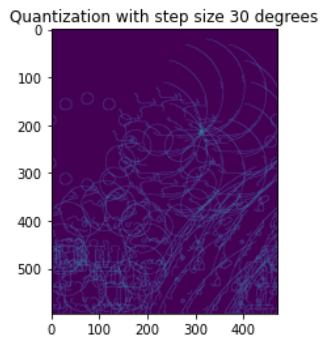
2e.

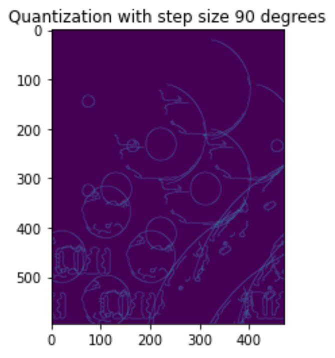
<u>Vote space quantization:</u>

The vote space of the accumulator array can be quantized into bins. There are several ways of doing this. Increase theta with step sizes, rounding off the Hough space parameters to a nearest integer, reducing the hough space, or even using special kernels that sum up local votes.

Quantization is helpful in speeding up the process when the computation is too much - however, this comes with a cost at efficiency. Observe the hough spaces as the quantization keeps increasing:


Quantization with step size 10 degrees


Quantization with step size 30 degrees


Quantization with step size 90 degrees