MSML640 - Assignment 3 Report

Short Answer Questions

1. According to the given image, the filter banks cover orientations along various angles. This will enable detecting of the texture changes in any direction,thus the resulting filterbank will be robust against any orientation i.e insensitive to orientation.

2. K-means is a clustering algorithm which uses distance metrics like Euclidean distance to find new centroids and assign the data points to these centroids. This process can create random centroids and divide the points based on minimum distances from different these centroids. Further, K-means is also sensitive to outliers and initial centers. This can divide all the points up into random clusters (even for the points originally belonging to the same circle) which is not an accurate answer.

3. K-means clustering algorithm is used to aggregate data points into clusters distinctively while for Hough transform , we require the output to be the maximum vote area. Graph-cut algorithm uses the feature space, eliminates the relations with low similarity between features and results in only 2 features. Mean shift algorithm seeks local maxima density of feature space i.e uses each data point in every iteration each to move closer to where majority of the points are situated. The central point of this group is the cluster center. This cluster center can be the area with maximum votes in hough transfer and hence mean shift algorithm is most useful.

4. **Algorithm**:
    - Assuming that we have run the components connectedness algorithm and have foreground blobs.
    - We can use the circularity feature of these blobs to group them by the closeness of their boundary shape to a circle,this will group the blobs with equal closeness to a circle under the same shape category.
    - Find the centroid of the blob
    - Calculate the circularity for the blobs

- Use clustering algorithm to group the blobs using the circularity values.

Pseudo Code:

```
def  findCOM(blob):
        for pixel in blob
        Pos += position(pixel)
        COM = Pos/size(blob)
 return COM

def circularity(blob):
        radius = size(blob)
        for each pixel in boundary(blob):
                DistSqsum += (COM - position(pixel))**2
                Circularity = DistSqsum / size(boundary(blob))
return circularity

def  k_means(circularity,K):
         return kmeans
```

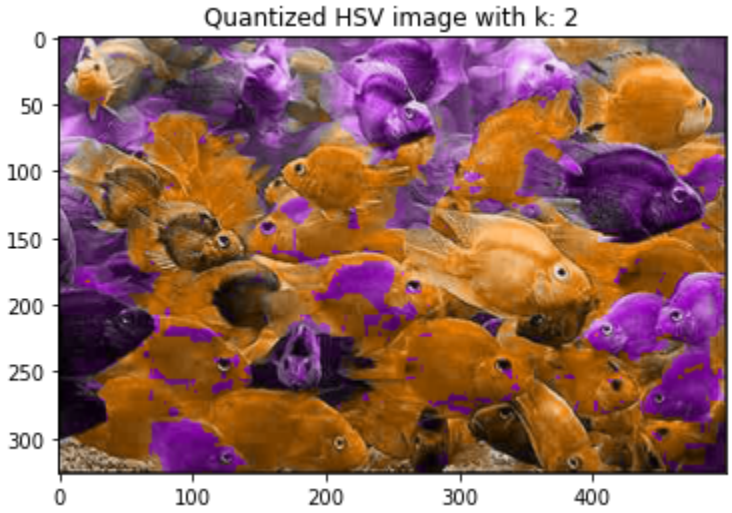
Variables:

- Pos: sum of position of pixel.
- COM: Center of mass point of the blob.
- DistSqSum: sum of squared difference between : Position(pixel)−Position(COM)
- circulatity: We normalize the sum of the squared distance is with the total number of the pixels bounding the blob to get an insensitive feature
- K : number of clusters for the K-means Clustering algorithm

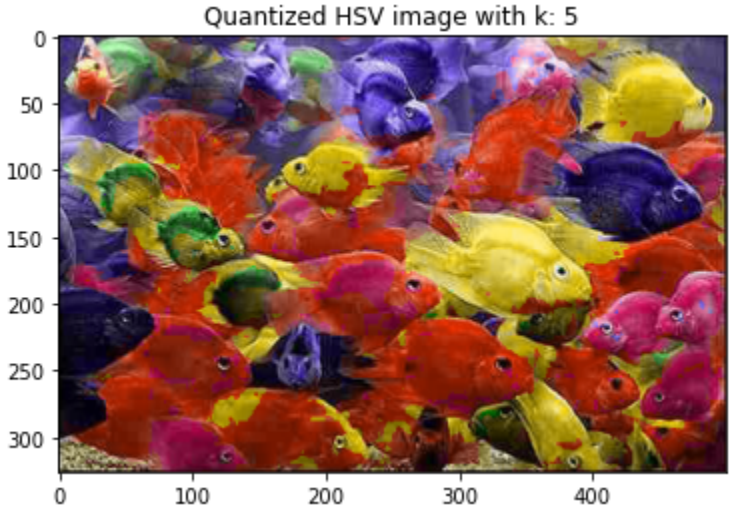
Programming

1e. **SSD values for HSV images**

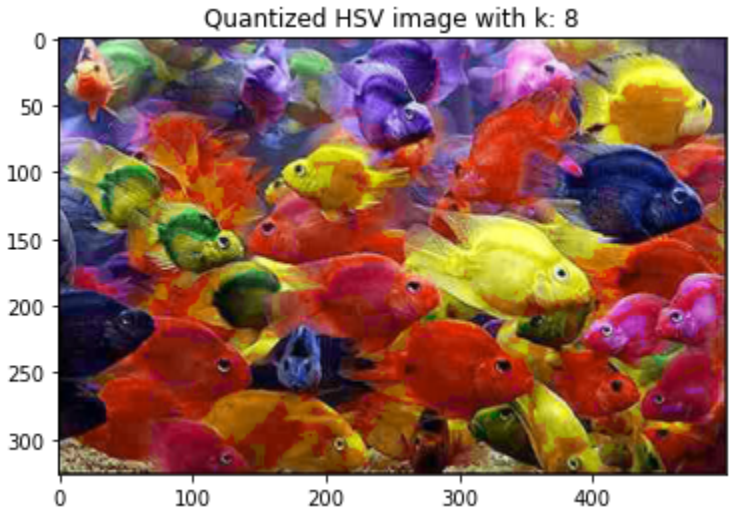
SSD for HSV image quantized with K =2: 385388540.0



SSD for HSV image quantized with K =5: 73019530.0

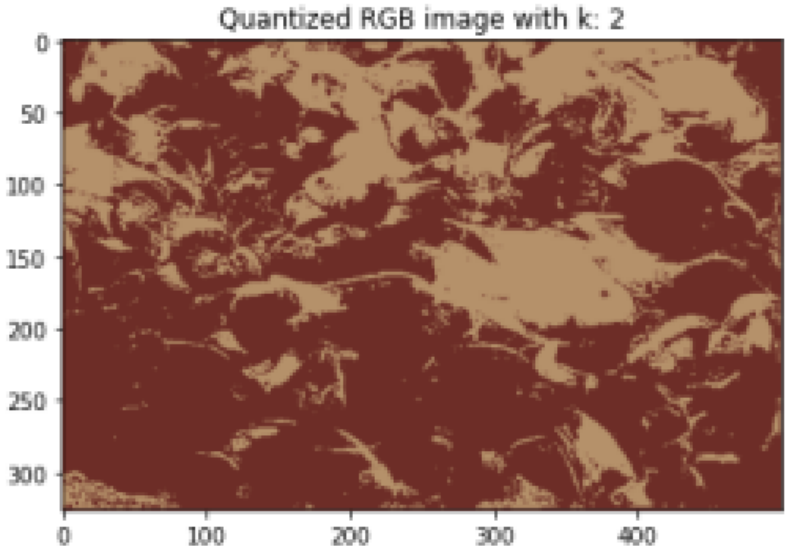


SSD for HSV image quantized with K =8: 35123304.0

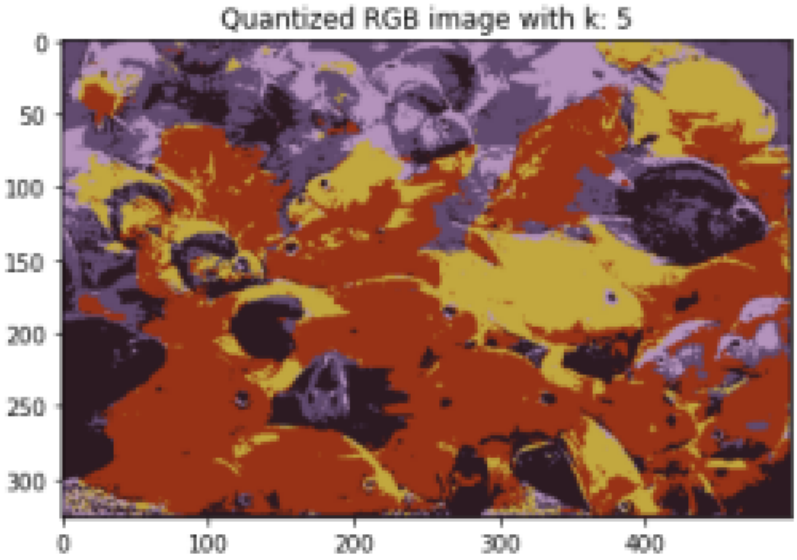


SSD Values for RGB Images:

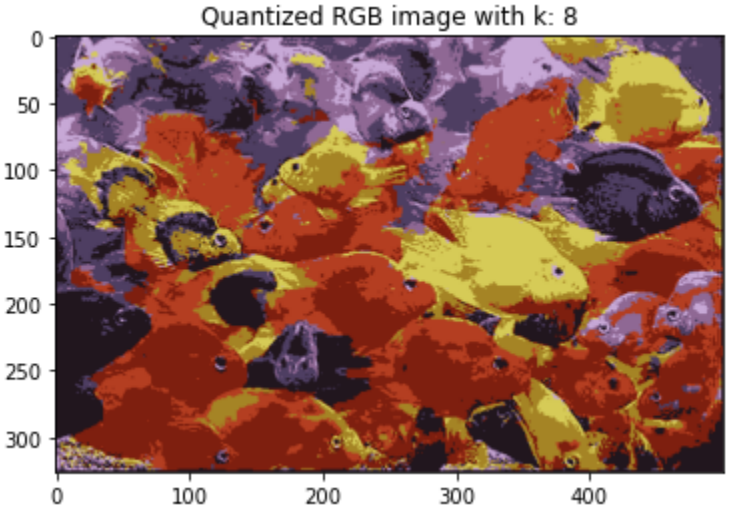
SSD for RGB image quantized with K =2: 1018948200.0



SSD for RGB image quantized with K =5: 462478300.0



SSD for RGB image quantized with K =8: 310571650.0



1f.

Color Quantization on RGB , HSV spaces, histograms and SSD functions were applied to fish.jpg

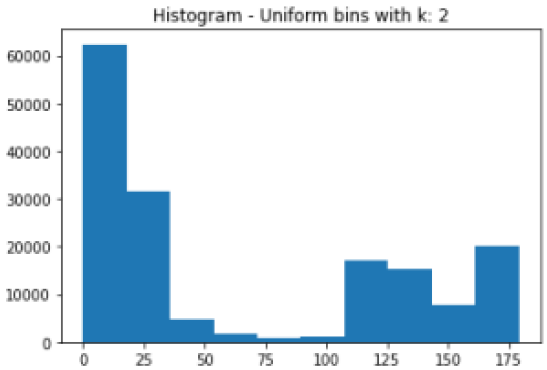
The SSD errors tend to decrease as the k value increases , as expected.

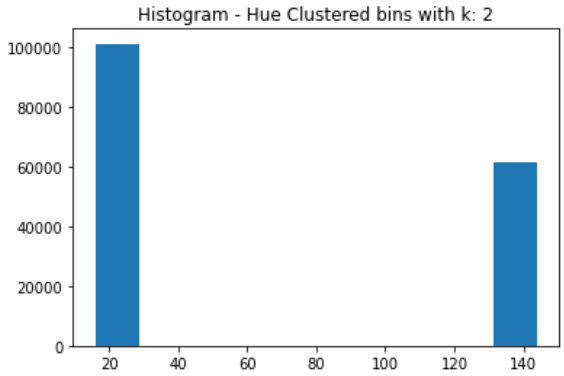
The SSD value drops from 1018948200 to 310571650 for rgb when k is increased from 2 to 8

The SSD value drops from 385388540 to 35123304 for rgb when k is increased from 2 to 8

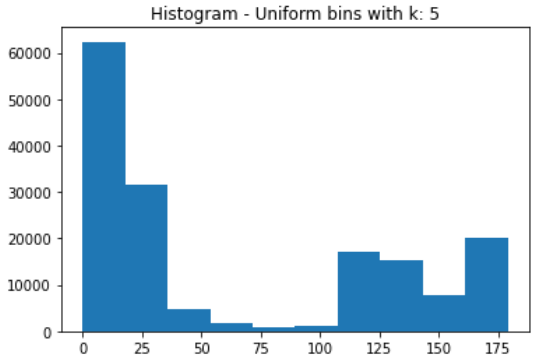
The SSD values also vary greatly between rgb and hsv clustered images. ALthough the process is mostly the same, the SSD value is universally lower for HSV clustering.

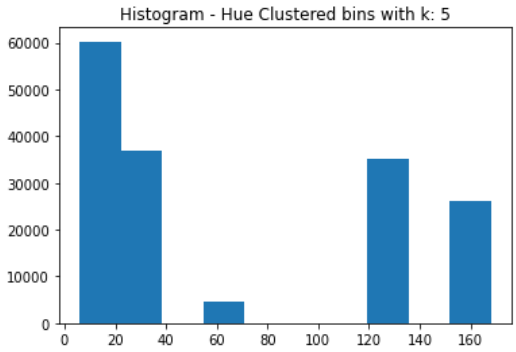
Histograms for K=2



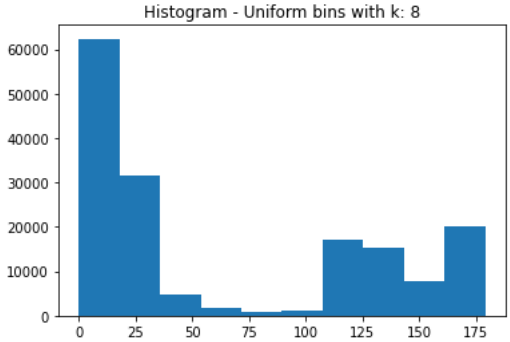


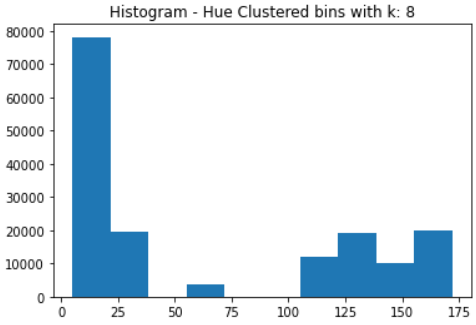
Histograms for K=5:





Histograms for K =8:





The normal histogram (with uniform bins) just displays the frequencies of the values as grouped naturally. These don't change for every k value. However, the Hue clustered bins group all the pixels belonging to a cluster under one group. These histograms tell the frequency of items in each cluster.
The values are very constant across multiple runs. This tells that the code is constantly finding the same clusters, which talk about the reliability of the algorithms.

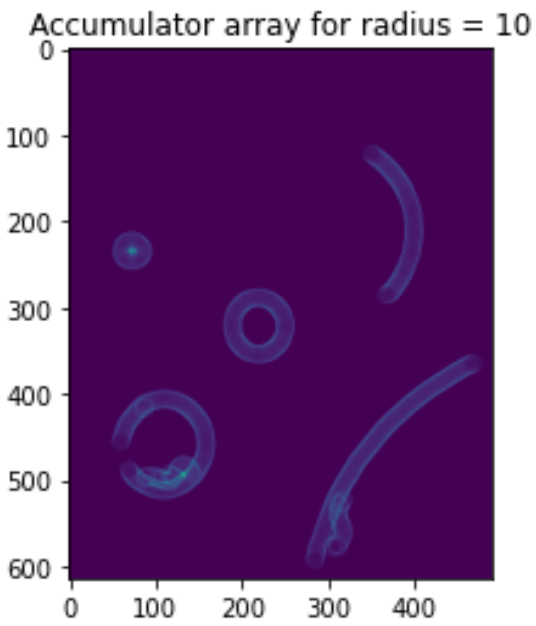
2a. Implementation of Hough Circle detection:

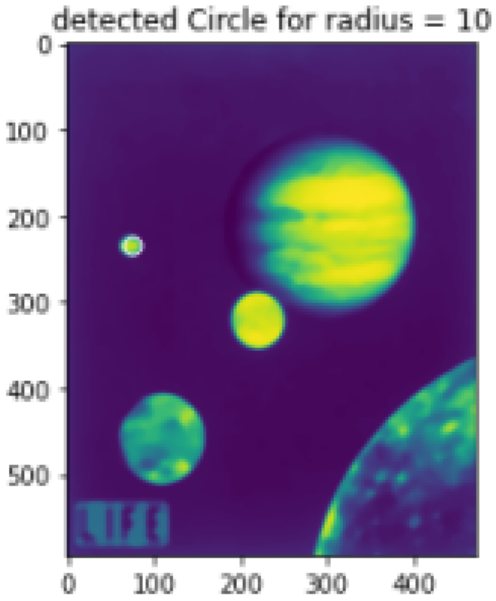
- The function "detectCircles" takes an image, a fixed value of r and fits circles with that value of r using hough circle detection.
- Firstly, an RGB image is given as an input to the function. This is then converted to a grayscale image. One of the important pre-processing steps is using smoothing/blur for edge detection, as this gives better results. Then, canny edge detection is applied to the image. This gives an edge image that becomes input to our hough code.
- To implement the Hough Transform, we first need the accumulator array. This array needs to be a little bigger than the actual image, as the border points can 'vote' for centers beyond the image borders. This array is initialized to all zeros.
- Next, we find out all the indexes of edges detected from the canny edge detector. We use the parametric equation x+r*cos(theta) , y+r*sin(theta) to find a new point in the hough space. If we do this for all theta (0,360); we get a circle in hough space. We 'vote' for all these points by incrementing the values with one in the accumulator array. If gradient is set to true, we can just use the gradient information to vote for only 2 possible points - this greatly reduces the computing time with similar results.
- This process is repeated for all edges detected. Now, our accumulator array has all the vote information.
- We can now set a threshold and only retrieve these values, as there are thousands of values listed. With these centers, circles can be plotted over the input image to see the results.

2b. Accumulator Array:

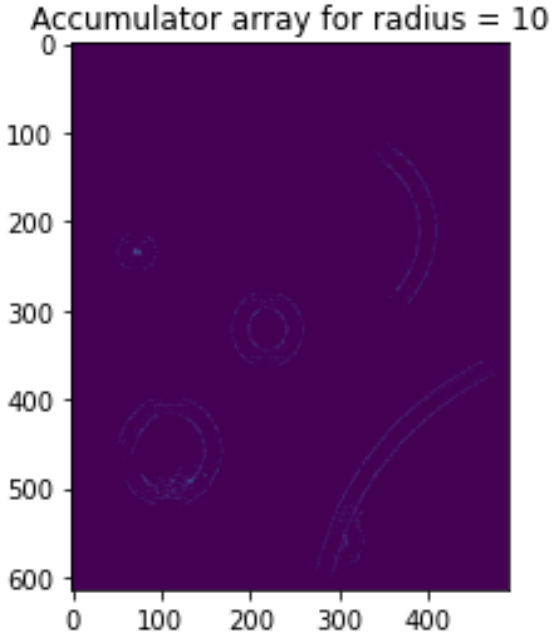
Radius = 10

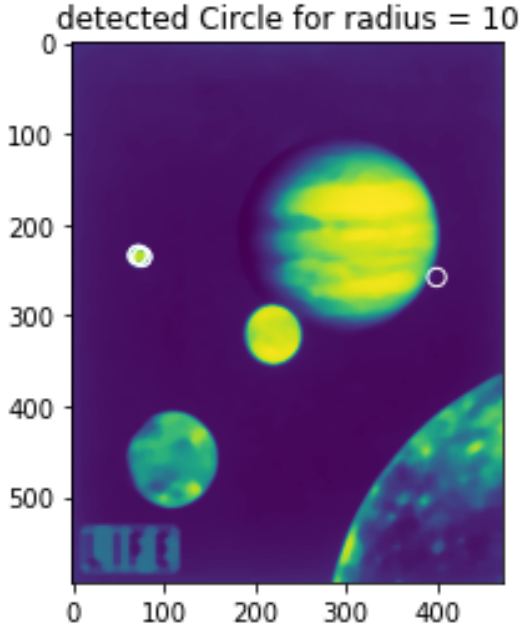
UseGradient = 0





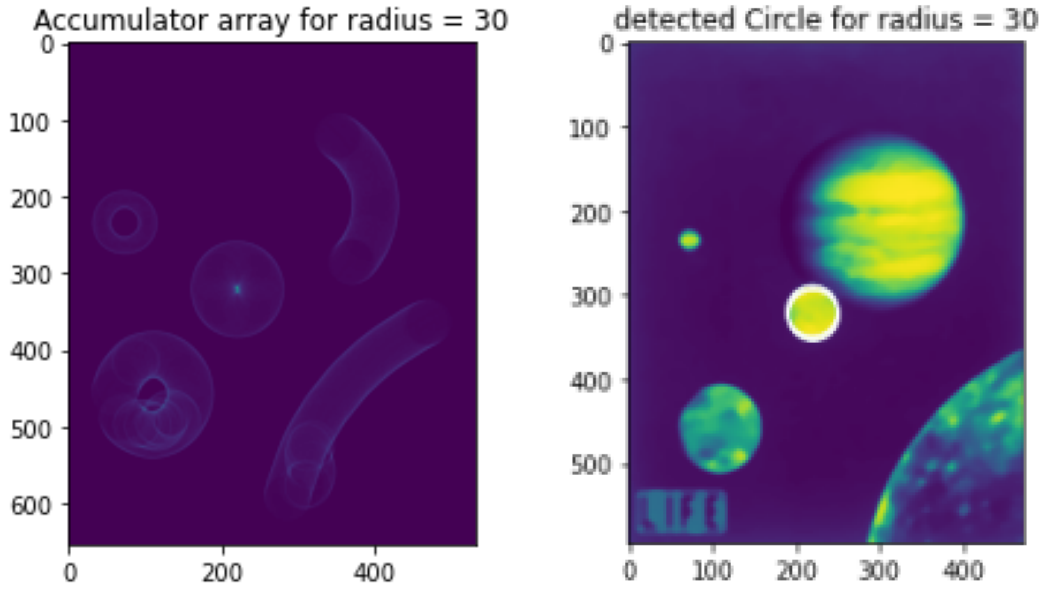
UseGradient = 1



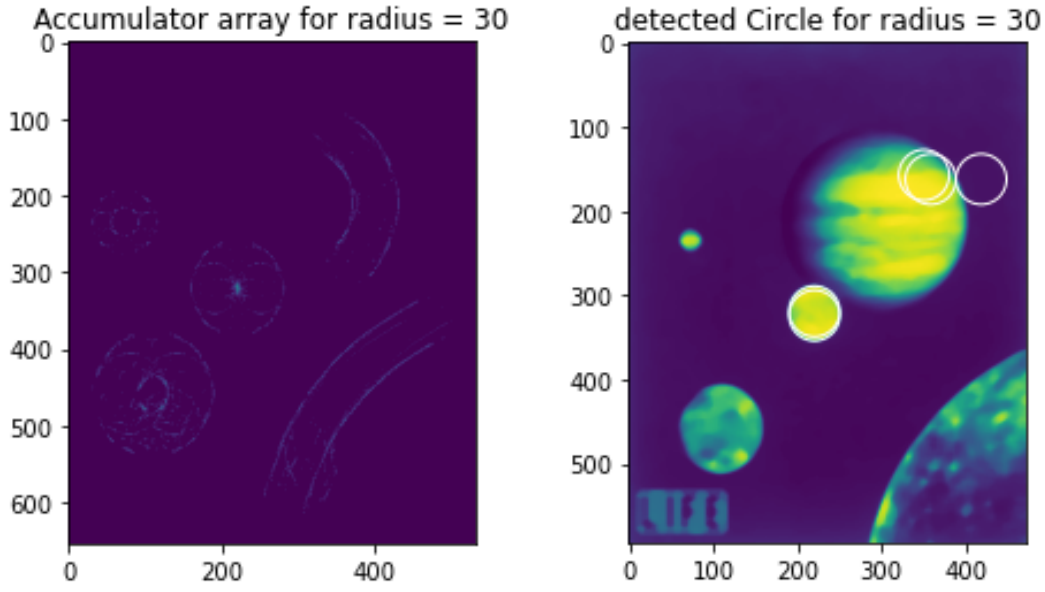


Radius = 30

Use Gradient = 0

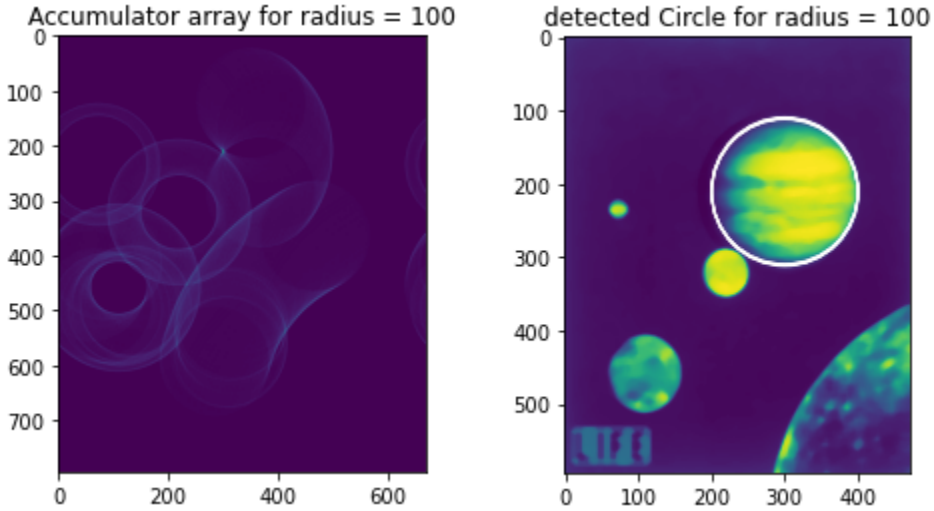


Use Gradient = 1

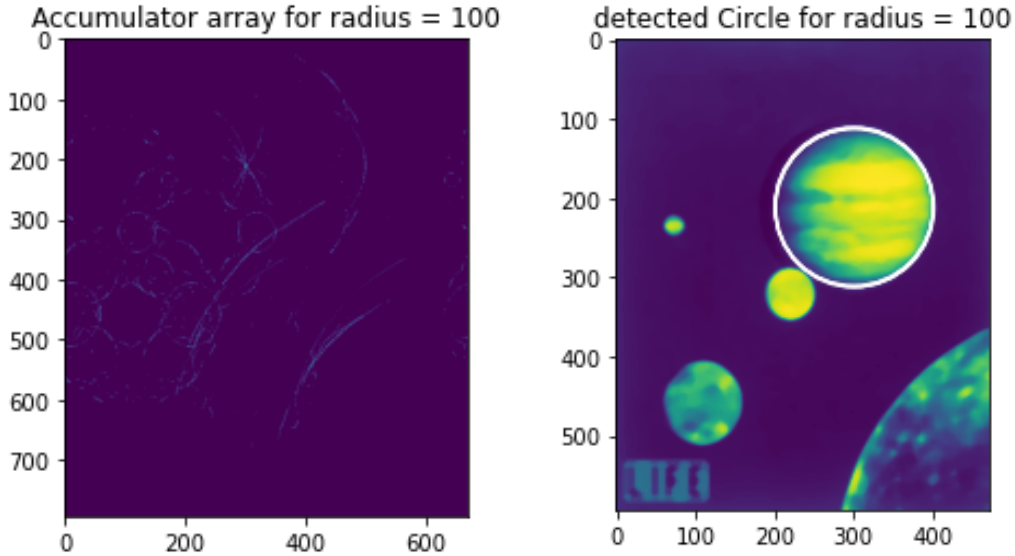


Radius = 100

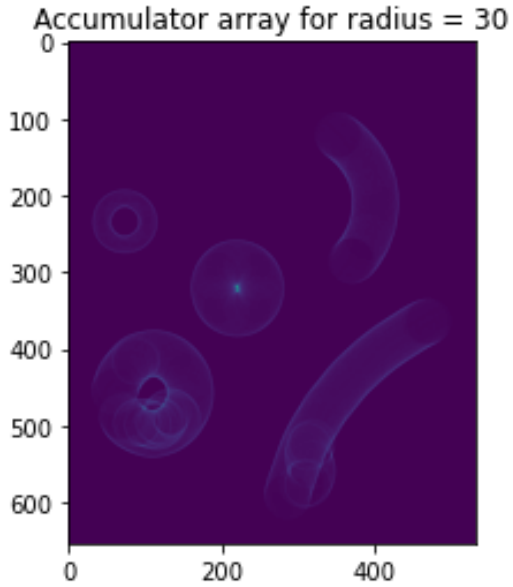
UseGradient = 0



UseGradient = 1



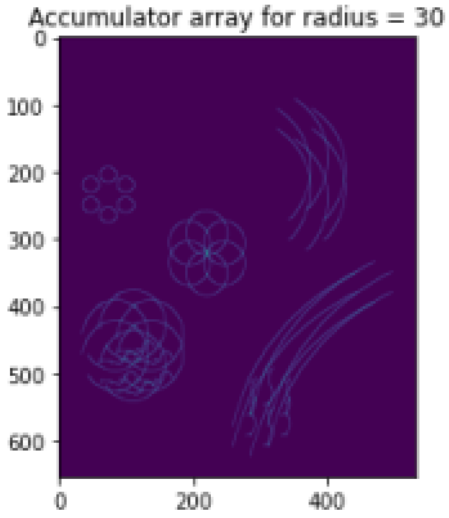
2c . Accumulator Array for R =30 and UseGradient = False



The hough space accumulator array stores the votes for potential circle centers with fixed radius R. Each detected edge votes for a total of 360 points that form a circle around it with that radius. So, the accumulator array is filled with overlapping circles. If a circle of that radius truly exists , all the points of that circle in normal space will create circles in the hough space that all intersect at the true center, hence this point will light up. In the observed image, this bright spot occurs in the center circle with radius 30.

2d. Since the accumulator array stores the number of votes for every point, we can create a top n list of centers with highest votes. We can leave the n value as a hyperparameter or we can try to automate the process of finding the number of circles we can plot the vote values in descending order and look for an elbow point. All the centers before this point can be denoted as having a circle.

2e.



.

One of the ways we can achieve the vote space quantization is to increase the theta values stepwise with a fixed step. The above is a figure for step size of 60. Since, this has lesser computations , the algorithm is faster but less reliable. However, the true center is detected in the hough space nonetheless.