CS 6476: Computer Vision, Fall 2019

PS2

Submitted by: Disha Das

GTID: 903542819

Date: 09/25/2019

1 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.

Ans. Since we have textons oriented in 6 different directions, we get a representation **insensitive to orientation**. Anytime a texture shows up that has patterns running along any of these 6 texton directions, the texton corresponding to that particular direction shows the highest response. Since the textons are oriented along directions that are approximately 30 degrees apart, they cover almost all directions. Textures detected along any of these directions will be detected using this filter bank, making the representation insensitive 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.

Ans. K-Means clustering is sensitive to initial centers. So the clustering would depend on the randomly chosen initial centers. After every iteration, every data point finds the center closest to itself and each center finds the centroid of the points it owns. Then the center jumps to the centroid. That being said, the algorithm doesn't recognise the grouping of the points in the inner circle and the outer circle. Since it computes clusters according to the shortest distance between datapoints and their centers, the partition between the formed clusters is likely to cut along the diameter of the larger circle, dividing both the inner and outer circles into **two concentric semi-circles**.

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.

Ans. Mean-Shift algorithm takes datapoints as input and clusters them according to their location in the attraction basins of various modes. Since the equivalent of maximum votes in Mean-Shift algorithm is the mode itself, we can be sure that the mode exists as a pixel somewhere among the input datapoints. Hence the result of segmentation won't be affected by the fact that the vote space is continuous since we know that the mode exists (and whose values can be used to 'colour' the rest of the datapoints in their respective clusters.)

K-means algorithm takes datapoints as input and needs a discretized space to collect votes. A continuous vote space may lead to a space between pixels to have the maximum votes. Therefore K-means clustering isn't suitable.

Graph-cut algorithm works on the image and returns an affinity matrix. Affinity matrix is always discretized, never continuous.

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.

Ans. Given- labelled blobs

For each blob among the labelled blobs:

    Get centre of mass of the blob;
    Compute circularity from the centre of mass;

```
        Store circularity in array C;
End For loop
[Clusters, Labels] = kmeans(C[], k);

%k – number of clusters
%Labels are the various Circularity values assigned to each k value
%Clusters are the kmeans values assigned to each blob.

Function getCentreOfMass(blob):
        For each pixel (x,y) in blob
                sumX += x;
                sumY += y;
        end For loop
        xc = sumX/length(blob);
        yc = sumY/length(blob);
        return xc,yc

Function getCircularity(blob):
        Xc,yc = getCentreOfMass(blob);
        K = set of boundary pixels of blob;
        For each value in K
                Sum += distance between K(i) and (xc,yc);
        End for loop
        Mean Radial distance uR = Sum/length(K);
        For each value in K
                SumV += ((distance between K(i) and (xc,yc)) – uR)^2;
        End for loop
        Variance of radial distance V = SumV/length(K);
        Circularity C = uR/sqrt(V);
        Return C;
```

2Programming

1.

Original image fish.jpg

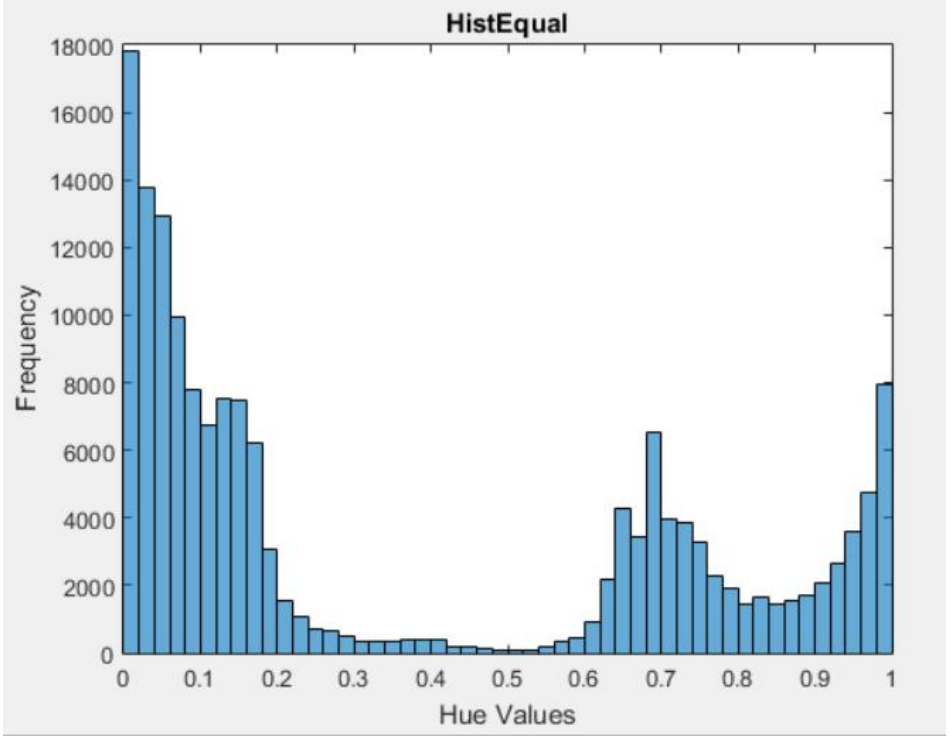
Image after RGB Quantization with k=50

Image after Hue value Quantization with k=50

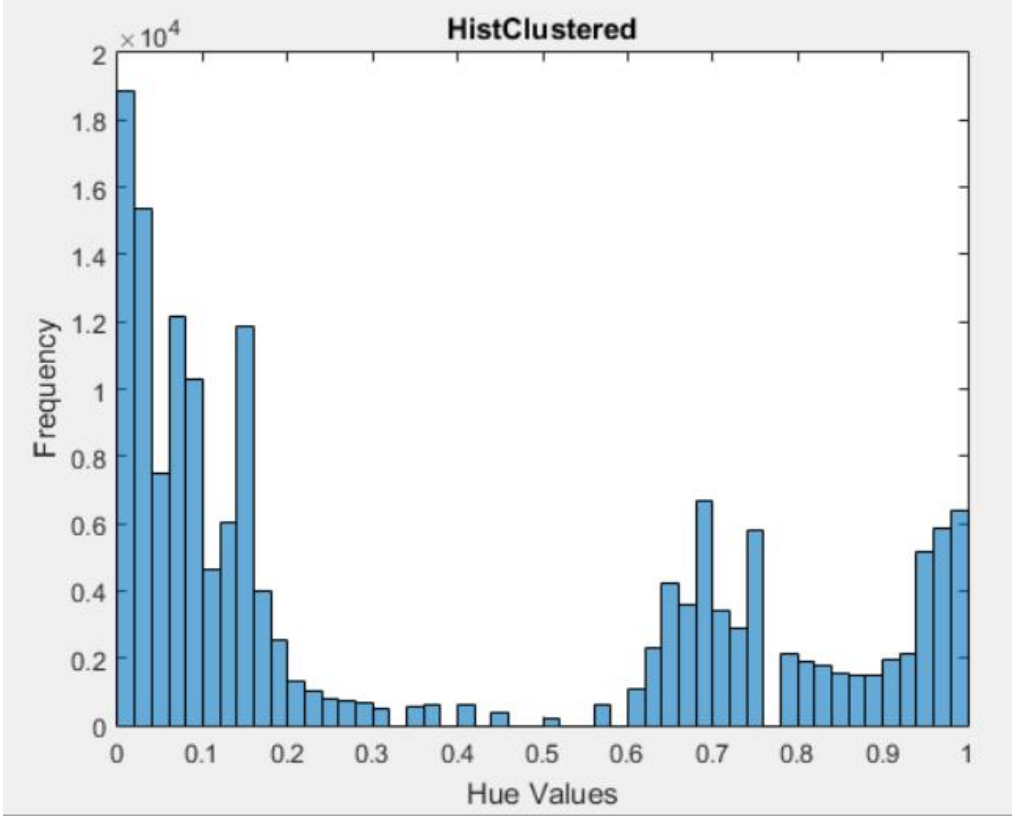
SSD Error of original Image against Quantized RGB image: 83,162,243

SSD Error of original Image against Quantized Hue of HSV image: 997,290

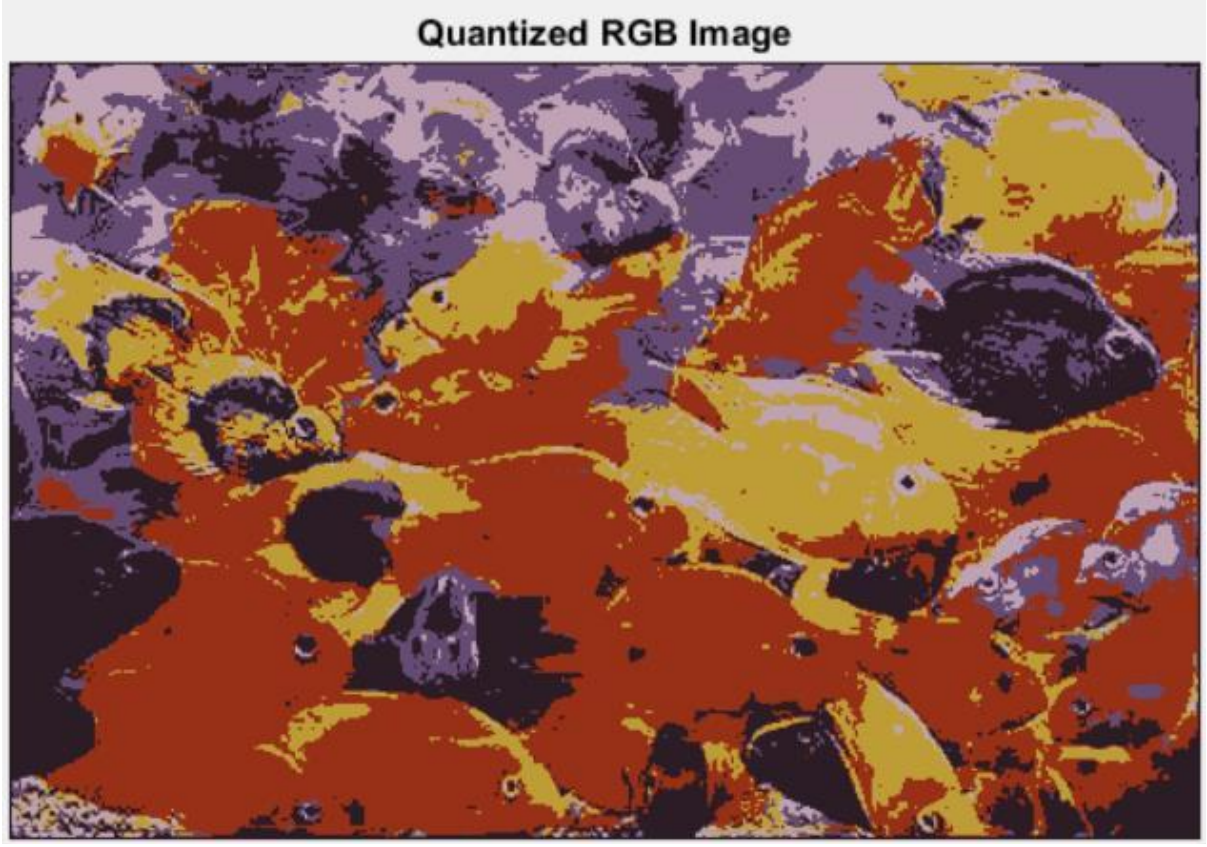
HistEqual:



HistClustered:



Lower value of k (k=5)



Higher value of k (k= 50)

(f): Quantized RGB image: For lower values of K, only a small number of colours are assigned to the entire image. This results in the patchy look of the output because pixels having

colours lying in the nearby spectrum of K colours get allocated the K colour value. Computing this over the 3 R, G, B channels, we get the resulting output image. Whereas for a larger value of K, the patchy look is reduced because a larger number of colours are being allocated to the pixels of the input image. For the fish image, we find that most of the image consists of the reddish hues of certain fishes. Therefore, when K is low, the majority of the pixels get allocated the reddish hue, as is obvious in the output image.

HSV quantized: Hue value is represented as an angle value that points to a particular hue in the colour spectrum. In the above program, hue values are in the range [0,1]. The values of angles chosen for K are in the neighborhood of the most common hue values in the input image. Hence, for fish.jpg, for K=3, we find K values to be 0.9380, 0.6919 and 0.0877, with maximum pixels allotted to the 3rd cluster, implying that most of the pixels in the HSV image lie in the neighborhood of this value. This value gives a reddish-orangish hue, resulting in the majority of the image pixels allotted this hue. For large values of K, the output image tends to look similar to the input image because the K values are selected in order of the most common Hue values of the input image. For large K, most of the common hue values in the input image are captured in the K-means palette and assigned to neighborhood pixels. Therefore, there's not much of a difference between the input image and the output image.

Error: The error values decrease with an increase of K value. This is because when number of clusters are limited, only a few pixels exist that have a value in the neighborhood of K value. The rest of the pixels are assigned K values with large errors due to limited choice of K values. As K value increases, more and more pixels are assigned K values in their own neighborhood, reducing the error. This applies to HSV images as well.

Histograms: histEqual histogram shows the distribution of the pixels along Hue values of the image. Upon taking a large number of bins irrespective of the K value, we find that there are certain values of Hue where local maxima occur. Upon plotting the histClustered histogram, it becomes clear why. The values of Hue where the local maxima occur are exactly the values that the K centers take. No wonder, K centers have values where local maxima occur in a normal histogram. As the K value increases, we see that the maxima is broken down to several peaks around the previous peak. These are the values that the subsequent K centers take up. Also, the frequency changes for each peak as pixels from bins (from histEqual) are redistributed over the K centers that are nearest to them. Thus the frequency values of histEqual may differ from histClustered, especially when number of bins is equal to K. This is because bins show the frequency of a range of Hue values. Pixels in this range may get distributed over different K centers. Thus, frequency varies. But upon taking a large number of bins, the shape of histEqual approaches histClustered.

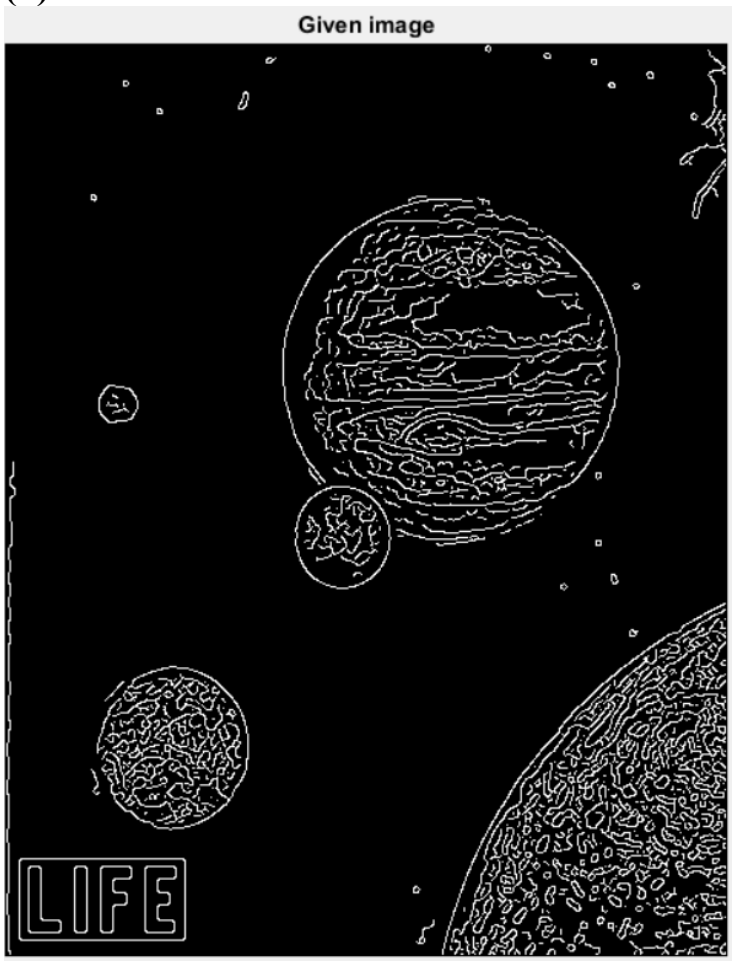
2.2 Circle detection with the Hough Transform

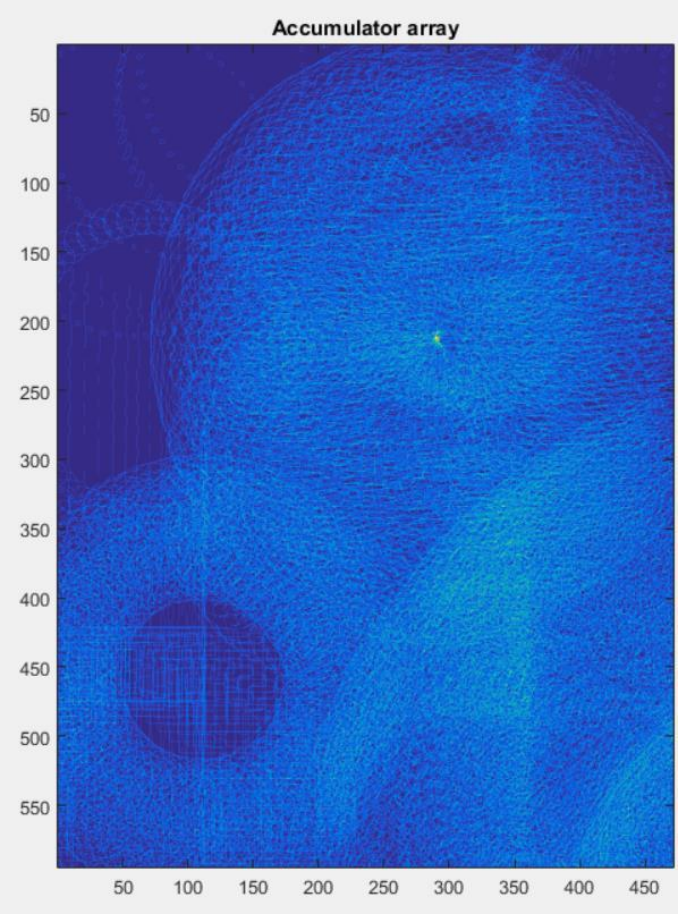
(a). The function detectCircles.m has the input image, radius of circle to be detected and gradient flag as input parameters. First, the image is converted to gray so that edges can be easily detected using a canny edge detector. This results in a binary image (bw) consisting of detected edges. Next, we construct a two-dimensional Hough space matrix (value 0) having the same size as the binary image bw. Next, we iterate through every pixel in bw that is non-zero. If gradient flag is set to 1, the gradient angle at the pixel is calculated using the formula given in the code. Otherwise, if gradient flag is set to 0, every angle is calculated first from –

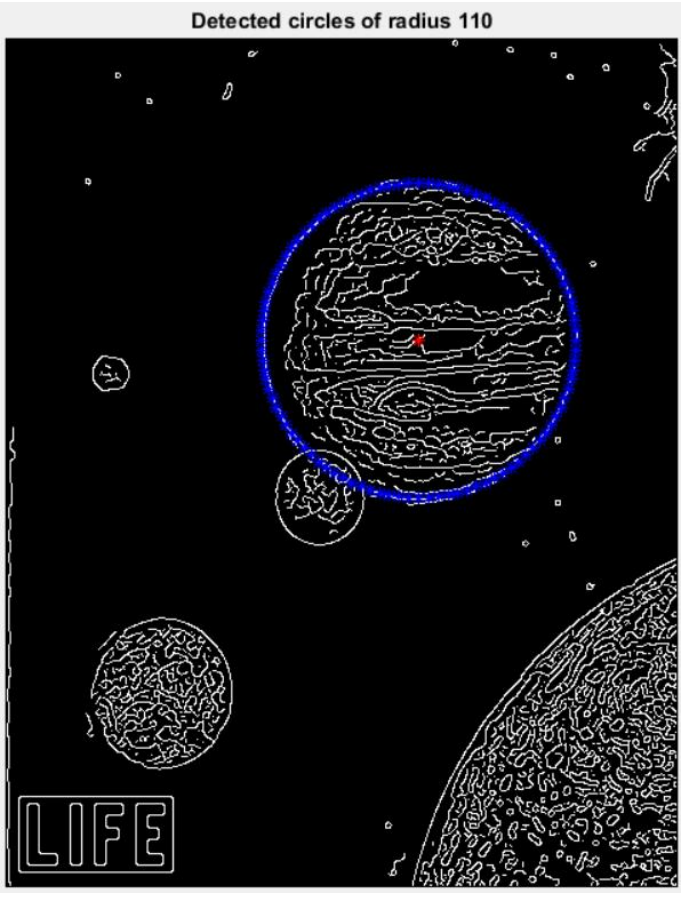
pi to pi. Then using these angles, the x and y coordinates of the edges of the circle are calculated , whose center is the given pixel. This gives the pixels that could be the possible centers of a circle on which the said pixel lies. For every such point, the corresponding point in the Hough space is updated. We also keep track of the maximum value in Hough space and its location. Thus, after every iteration of a pixel, the Hough Space gets updated. Now the pixel having a high value in the Hough space could be the center of the said circle whose radius is given.

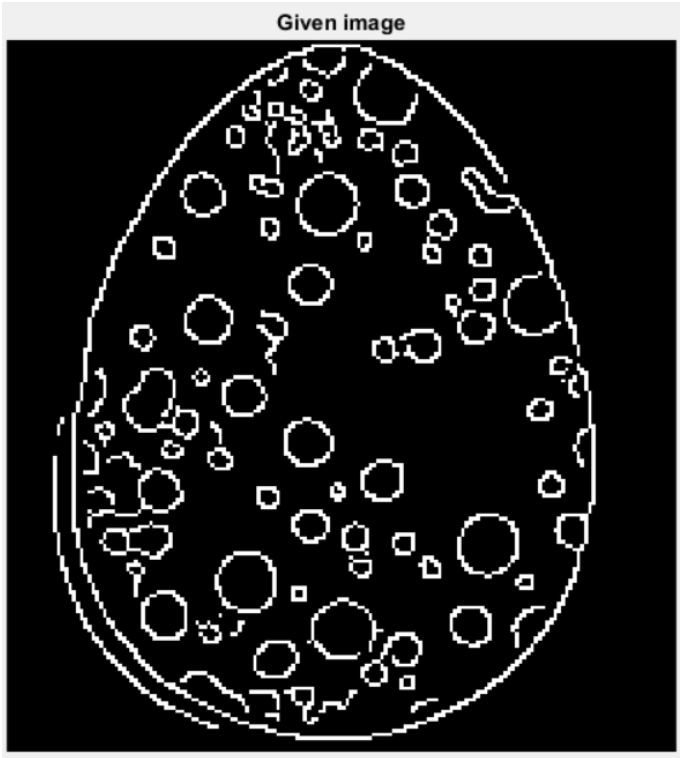
But not every maximum in the Hough space necessarily points to a circle. Hence, to make sure the detected maximum (max) is indeed the center of a circle, we take the 100^{th} and 99^{th} percentiles of the values of the Hough matrix. Dividing the 99^{th} perc value by 100^{th} perc value gives a scalar which would determine whether a circle is being detected. By trial and error, it is found that if the value(perc) is below 0.41, then a circle of the given radius indeed exists. Hence, if perc > 0.41, we return an empty matrix of centers. If perc <0.41, then we detect the top 10% of maximum values in the accumulator matrix. Iterating over them, we plot the centers and the resulting circle of given radius. Finally, an Nx2 matrix consisting of detected circle centers is returned.

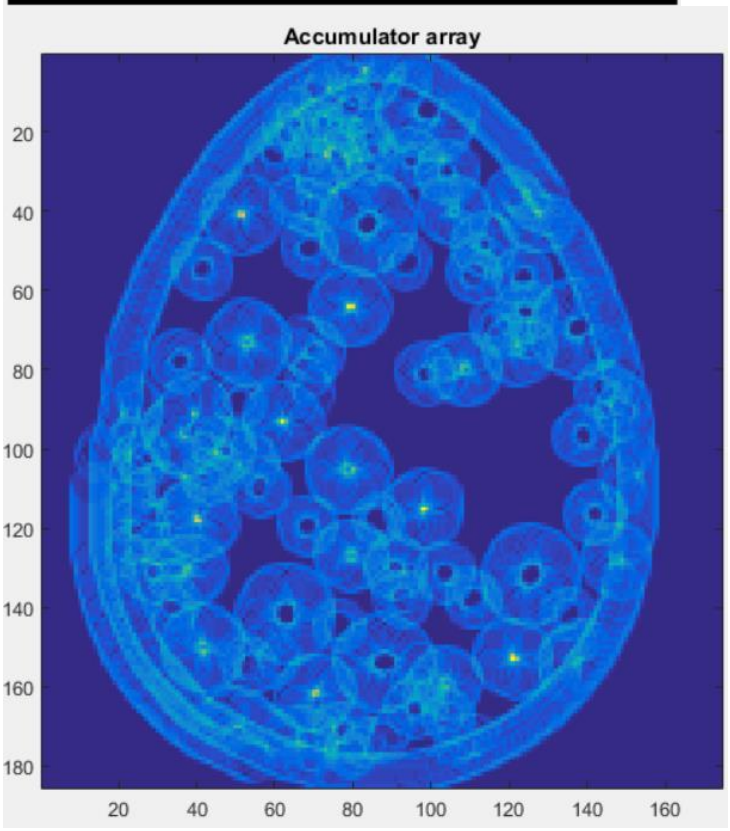
(b).

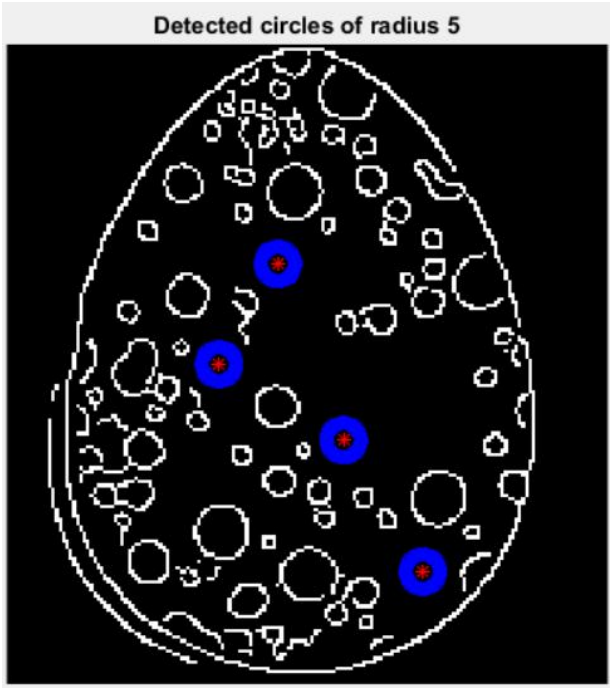



Accumulator array
50
100
150
200
250
300
350
400
450
500
550
50
100
150
200
250
300
350
400
450

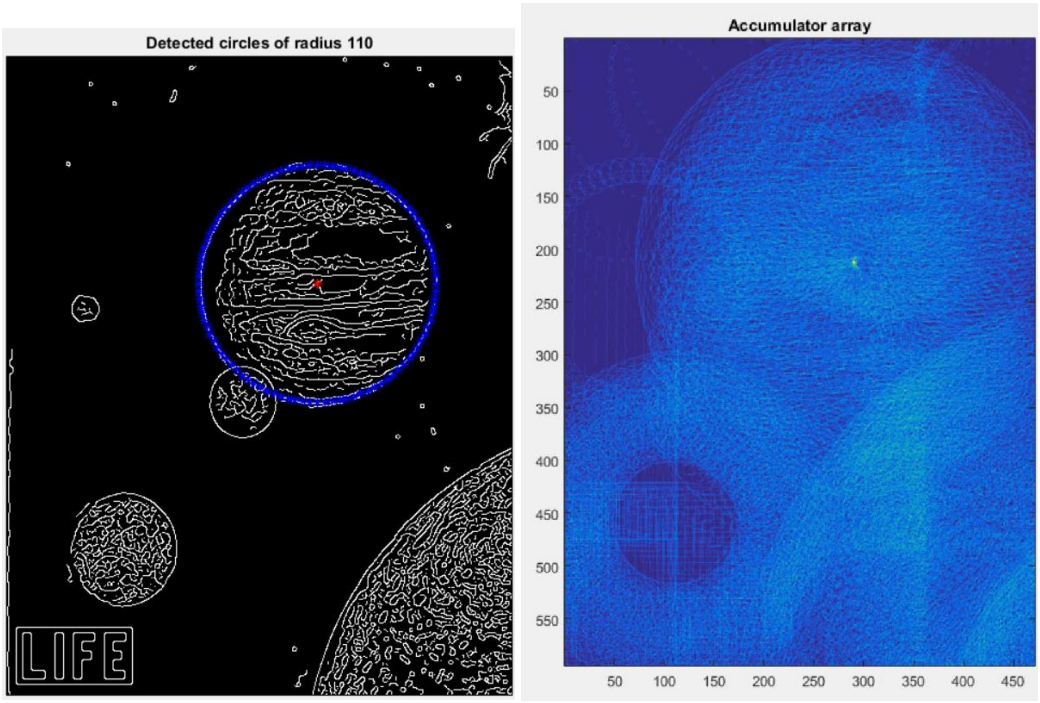

Detected circles of radius 110
LIFE


Given image

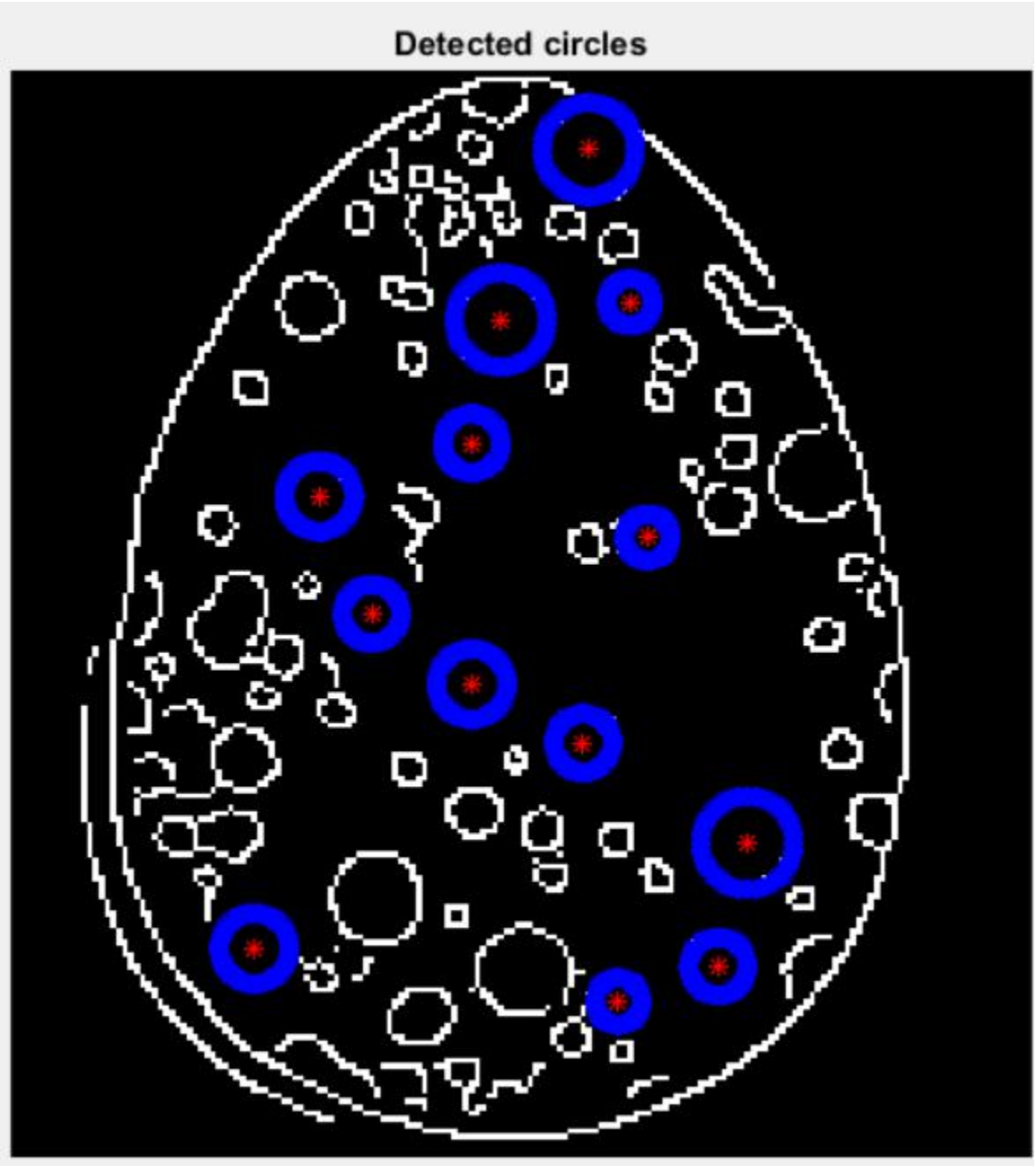

Accumulator array
20
40
60
80
100
120
140
160
180
20
40
60
80
100
120
140
160



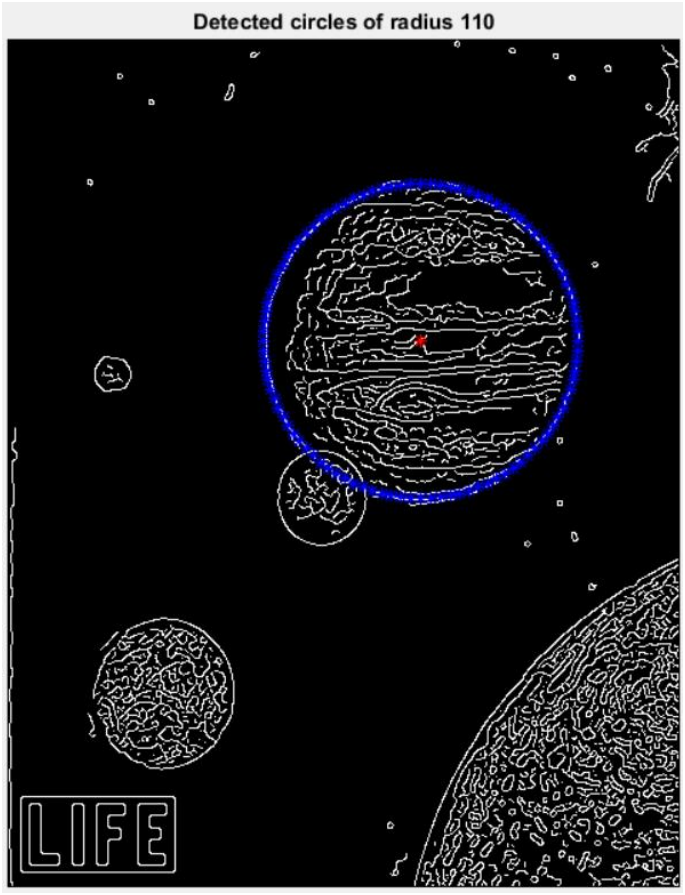
(c). The accumulator array for jupiter.jpg. The Hough space accumulator array consists of circles drawn at a radius of 110 from every detected pixel in the binary image that has a non-zero value. The above image shows that. For the pixels on the edge of a circle of radius 110, the circles drawn from each of these pixels converge at a point. This point has the highest votes, and is seen as a bright spot in the image above. This point is extracted and hence, deemed the center of a circle of radius 110.



(d). By iterating over every possible radius, we find the accumulator arrays that correspond to the particular radius. By superimposing the detected circles over all the accumulator arrays on the binary image, we can detect all the circles in the given image. We can keep a count of every circle detected in an accumulator array that has a valid detected circle. In this way, we can acquire the number of circles in an image by post-processing the accumulator array. In the image below, the entire code was iterated over every radius ranging from 1 to 20.

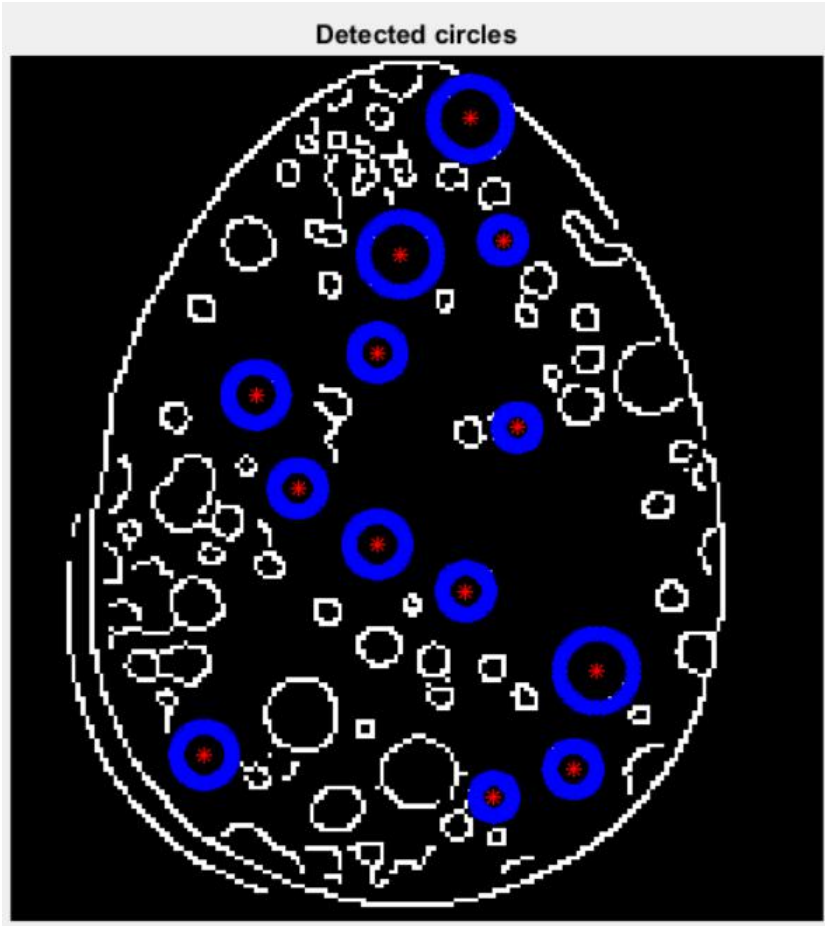


(e). Suppose the bin size of of vote space is increased. This means that if we calculate a value of ‘a’ and ‘b’ for a given pixel, the resulting vote goes to the bin that contains within its range the indices ‘a’ and ‘b’. Therefore, instead of getting various values for accumulator array for stray pixels within a circle, we get a single value for a region of hough accumulator array. Thus is effective in exclusively detecting a circle whose area fits into the bin area.



3 Extend your Hough circle detector implementation to detect circles of any radius. Demonstrate the method applied to the test images.

To detect circles of any radius size, we need to iterate the entire code over a range of possible radius values. For each radius value, we get an accumulator matrix containing detected circles. Now, we need only superimpose the detected circle edges of such accumulator arrays on the binary image or original image to detect the circles. In the image below, the entire code was iterated over every radius ranging from 1 to 20.


Detected circles