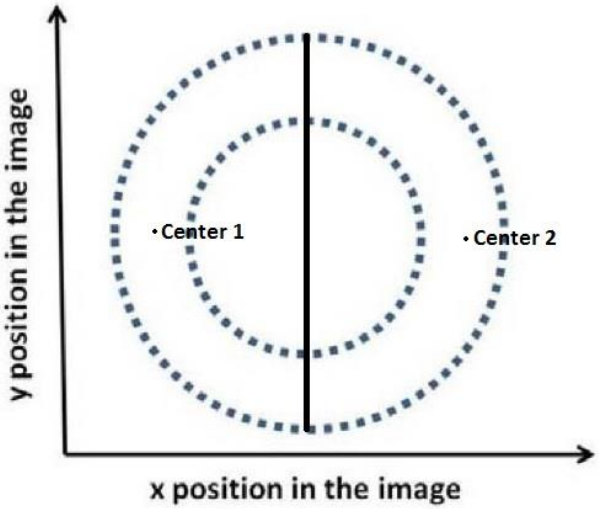
Solution document to PS2.
Prepared by Ahsan and Vivek

SHORT ANSWER PROBLEMS

1. It is sensitive to orientation. This can be explained by thinking about the final vectors we get after passing an image through the given series of texture-filters. If the image has predominantly horizontal edges, the first and seventh (from the left) responses will be higher than the others. Here, a high response from a filter would mean a higher value in the corresponding positions of the vector we get at the end. Similarly, the values at the fourth and the tenth positions in the vector will be higher than the others, in the case when the image has predominantly vertical edges.
2. K-means will divide the region into two halves, which is going to look something like the following. Note that this may be an incorrect result if we were looking to divide the points by the circle that they belong to.



3. Out of the three algorithms mentioned in the problem, mean-shift would be the best algorithm to quantize up the continuous Hough Space. This is because mean-shift will give us the mode of each cluster to give us the centre values, in centre-of-mass manner. Also, it will give us consistent results across different runs, while we don't have to worry about a lot of tuning parameters.
4. This grouping can be done by using any of the clustering algorithms we have covered in class so far, with each blob having two features (area and aspect ratio) which would be used for the grouping. Here we are using k-means to do the same.
   Pseudo-code:
   idx : the blob index
   blobs : blob containing the blob's index at row 'idx'
   blobs(:,1) : represent the blob area of all the blobs
   blobs(:,2) : represent the blob aspect ratio for all the blobs
   % representing k-means as a black-box
   % here k is a given constant which is the number of groups you want
   x = kmeans(blobs,k)
   % x gives us the cluster membership of each blob.

PROGRAMMING PART :

1.

a) quantizeRGB

```
function [outputImg, meanColors] = quantizeRGB(origImg, k)
%initializing the output image
outputImg=zeros(size(origImg));
%pixels is the matrix with each row containing the rgb values of a pixel
pixels=reshape(double(origImg),size(origImg,1)*size(origImg,2),size(origImg
,3));
%performing the kmeans
[idx,c]=kmeans(pixels,k);
%the matrix containing the membership of each pixel
indmat=reshape(idx,size(origImg,1),size(origImg,2));
%putting all the values back into the original image
for i = 1:size(origImg,1);
        for j = 1:size(origImg,2);
            memval = indmat(i, j);
            outputImg(i,j,1) = c(memval,1);
            outputImg(i,j,2) = c(memval,2);
            outputImg(i,j,3) = c(memval,3);
        end
end
outputImg=uint8(outputImg);
meanColors=c;
end
```

b) quantizeHSV

```
 function [outputImg, meanHues] = quantizeHSV(origImg, k)
%initializing the output image and the hsv output image
outputImg=zeros(size(origImg));
hsvout=zeros(size(origImg));
%converting to HSV
hsvImg=rgb2hsv(double(origImg));
%pixels is the matrix with each row containing the rgb values of a pixel
pixels=reshape(hsvImg,size(origImg,1)*size(origImg,2),size(origImg,3));
%performing the kmeans
[idx,c]=kmeans(pixels(:,1),k);
%the matrix containing the membership of each pixel
indmat=reshape(idx,size(origImg,1),size(origImg,2));
%putting all the values back into the original image
for i = 1:size(origImg,1);
        for j = 1:size(origImg,2);
            memval = indmat(i, j);
            hsvout(i,j,1) = c(memval);
        end
end
hsvout(:,:,2)=hsvImg(:,:,2);
hsvout(:,:,3)=hsvImg(:,:,3);
outputImg=hsv2rgb(double(hsvout));
outputImg=uint8(outputImg);
meanHues=c;
end
```

c) computeQuantizationerror

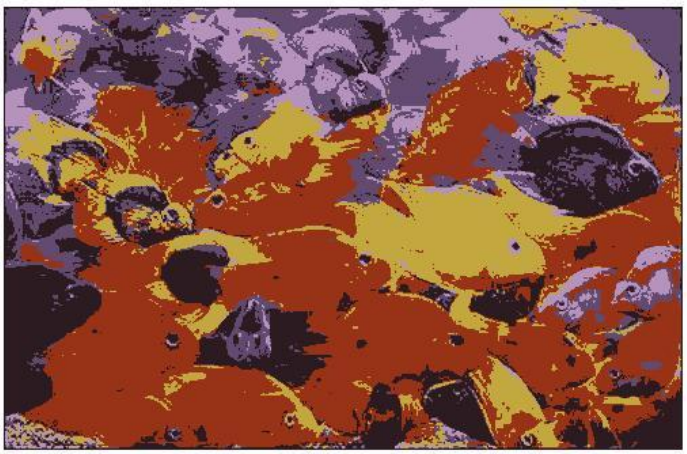
```
function [error] = computeQuantizationError(origImg,quantizedImg)
difference = (origImg - quantizedImg).^2;
%thrice because sum is a vector operation
error=sum(sum(sum(difference)));
end
```

d) getHueHist

```
function [histEqual, histClustered] = getHueHists(im, k)
%converting to hsv
hsvimg=rgb2hsv(double(im));
%pixels is the matrix with each row containing the rgb values of a pixel
pixels=reshape(double(hsvimg),size(hsvimg,1)*size(hsvimg,2),size(hsvimg,3))
;
%the equal histogram
histEqual = hist(pixels(:,1),k);
hist(pixels(:,1),k);
%now to get the clustered histograms, we first get the cluster centers
[~, centers] = quantizeHSV(im, k);
%new histogram with new cluster centers, sorted because the center values
%should be monotonically non-decreasing
histClustered=hist(pixels(:,1),sort(centers));
hist(pixels(:,1),sort(centers));
end
```

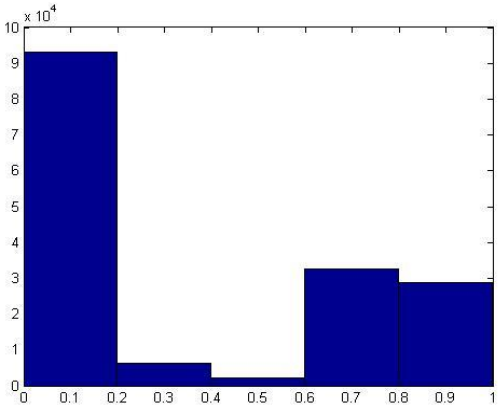
e) RESULTS

quantizergb

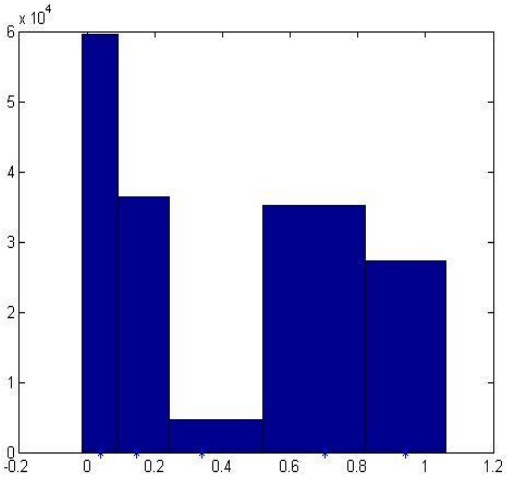


quantizehsv

histequal



histclustered



Error(with k=5, RGB space) = 43155377

Error (with k=25,RGB space) = 32997215

Error(with k=5,HSV space) = 12348656

Error(with k=25,HSV space) = 1797199

f) The two forms of histogram differ in the sense that histEqual has equally spaces bins, while the clustered histogram has uneven sized bins. This is because of the way kmeans clusters the data, which need not give us equal cluster sizes.
HSV color space has much lower SSD. This is because the HSV color space is a much better representation of the way we perceive colors. For the same pair of pixels, we may get way higher RGB difference compared to HSV difference, especially when we perceive those the two pixels to be of roughly equal color.
Increasing k, or the number of clusters, should decrease the SSD because we are moving more and more towards the finer representation of data. Had there been no clustering, we had 256 levels to represent the value in one color channel. Due to clustering this was cut down to 5(or 25). Thus, as we keep increasing the number of clusters, we should move towards lesser error.
K-means' result depends on the initialization, which in MATLAB's implementation, is not under our control. So, since we are using a black-boxed version of k means, we can expect to see different results across every run.

2.

a) We first convert the original image into a binary edge image (we are using Canny edge detection for it, but that's not the only method to do it . We then take the votes from each '1' pixel and build our accumulator array. We then choose the centres located with a certain number of votes greater than a chosen threshold (e.g., 90% of the max-value).To build the accumulator array we take all the angles around the white pixels as votes. We can take finer measurements with smaller theta sizes, which will mean more votes per pixel. We can also change the amount of quantization that we provide to the accumulator array.

b)

function detectCircles

```
function centers = detectCircles(im, radius, useGradient)

%% Maximum number of output circle radii
maxNumOuts = 1;
%% Quantization controlling variable
bin_size=1;
%% Binarizing image using Canny and finding gradients which might be needed
grayScale = rgb2gray(im);
grayScale = double(grayScale);
% grayScale = filter2(ones(7,7), grayScale); %Optional Filtering to improve
results in some cases
edgeIm = edge(grayScale, 'canny');
[gx, gy] = imgradientxy(edgeIm);
```

```
gy=-1*gy;
[r,c] = find(edgeIm == 1);

%% Initializing Accumulator Array with radius padding
H = zeros(size(im,1) + 2*radius, size(im,2) +  2*radius);

%% Hough Space Voting as done in Lecture Slides
for i=1:numel(r)
    if (useGradient == 1)
        dy = gy(r(i), c(i));
        dx = gx(r(i), c(i));
%         thetas(1) = atan2d(dy, dx);
        thetas(1) = radtodeg(atan2(dy,dx));
%         thetas(2) = -atan2d(dy, dx);
        thetas(2) = 180+radtodeg(atan2(dy,dx));
        thetas(3:46)=thetas(1)+1:thetas(1)+44;
        thetas(47:90)=thetas(1)-44:thetas(1)-1;
        thetas(91:134)=thetas(2)+1:thetas(2)+44;
        thetas(135:178)=thetas(2)-44:thetas(2)-1;
    else
        thetas = 1:360;
    end

    for theta = thetas
        a = radius + c(i) - radius * cosd(theta);
        b = radius + r(i) + radius * sind(theta);
        %quantizing to the nearest bin value
        a=round(a/bin_size)*bin_size;
        b=round(b/bin_size)*bin_size;
        H(b,a) = H(b,a) + 1;
    end
end
%% Visualizing Hough Space
figure, imagesc(H);

%% Processing on Accumulator Array to output most voted centers.
[~, sortedIndex] = sort(H(:), 'descend');
% in the next two statements we find the row and column indices for the
% vector index found above.
r = mod( sortedIndex, size(H, 1) );
c = floor( sortedIndex / size(H, 1) );

r = r(1 : min(maxNumOuts, numel(r)) );
c = c(1 : min(maxNumOuts, numel(c)) );

%% Ouput best fit circle centers
centers = [r-radius c-radius];

end
```

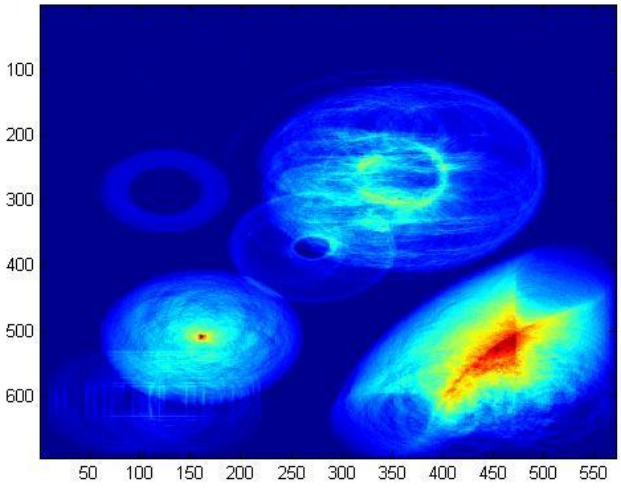
script main

```
clc;clear;close all;
im = imread('egg.jpg');

radius = 5;
useGradient = 0;
centers = detectCircles(im, radius, useGradient);
```

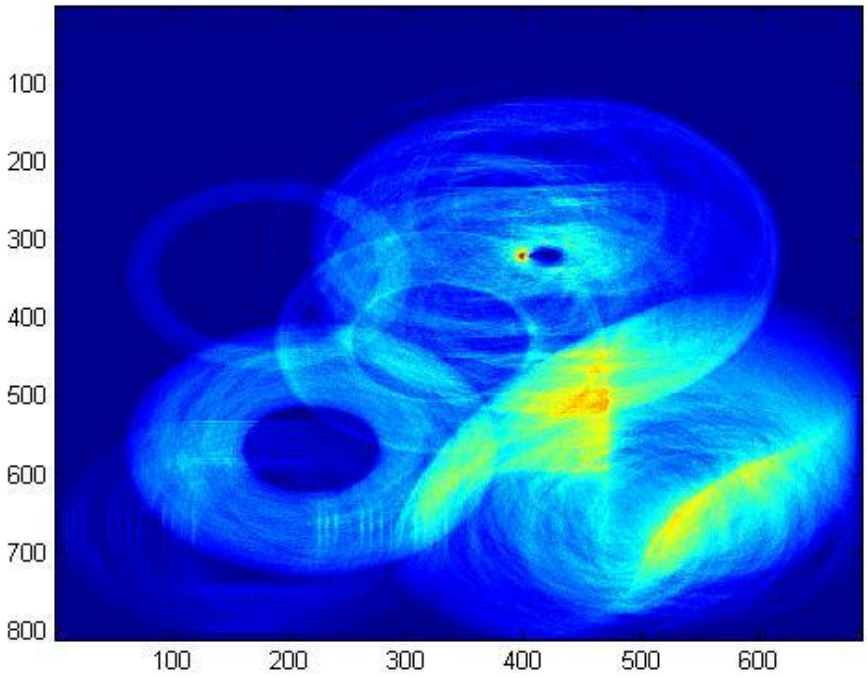
```
%% Visualizing Detected Cricles
figure, imshow(im)
hold on;
for i=1:size(centers,1)

    % Drawing Circles on figure using 'rectangle' command.
    rectangle('position', [centers(i,2)-radius centers(i,1)-radius 2*radius
2*radius], 'curvature',[1 1],...
        'edgecolor','r');
end
```

RESULTS :

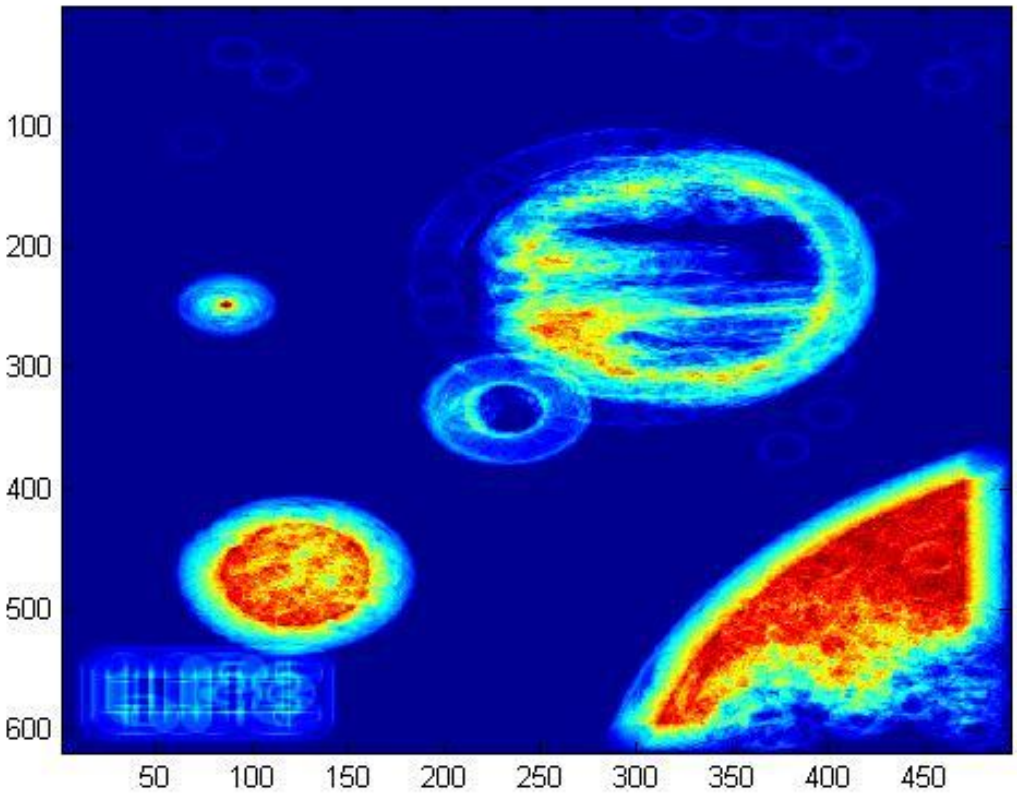
jupiter.jpg, for r=50 , useGradient = 0



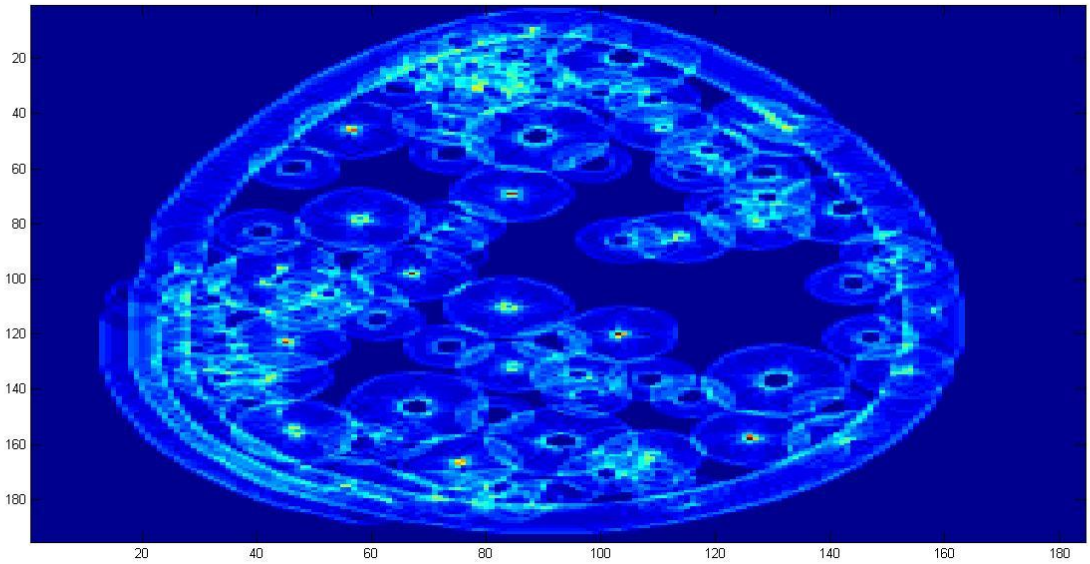
jupiter.jpg , for r=108 , useGradient = 0



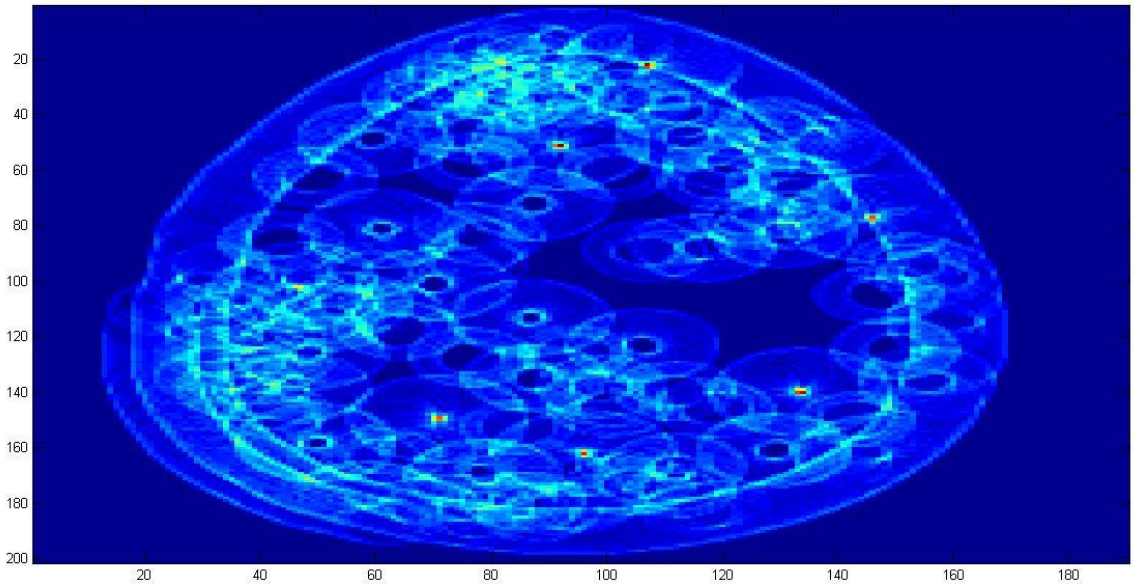
jupiter.jpg, for r=12, useGradient=0

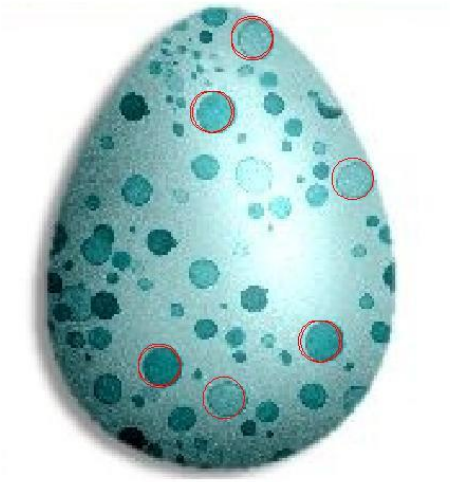


egg.jpg , for r=5, useGradient = 0

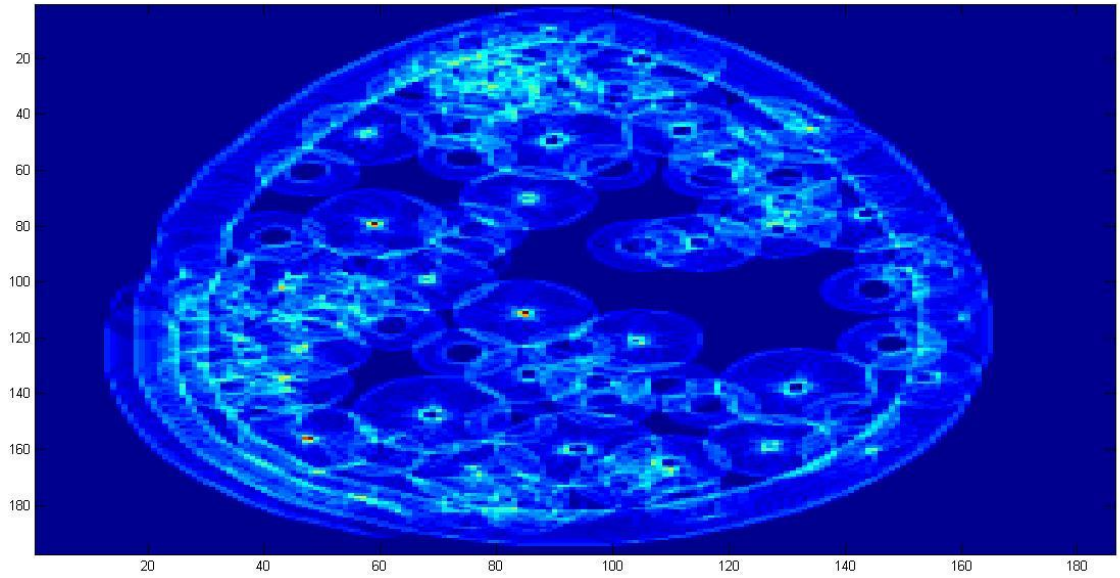


egg.jpg, for r=8 , ueGradient=0

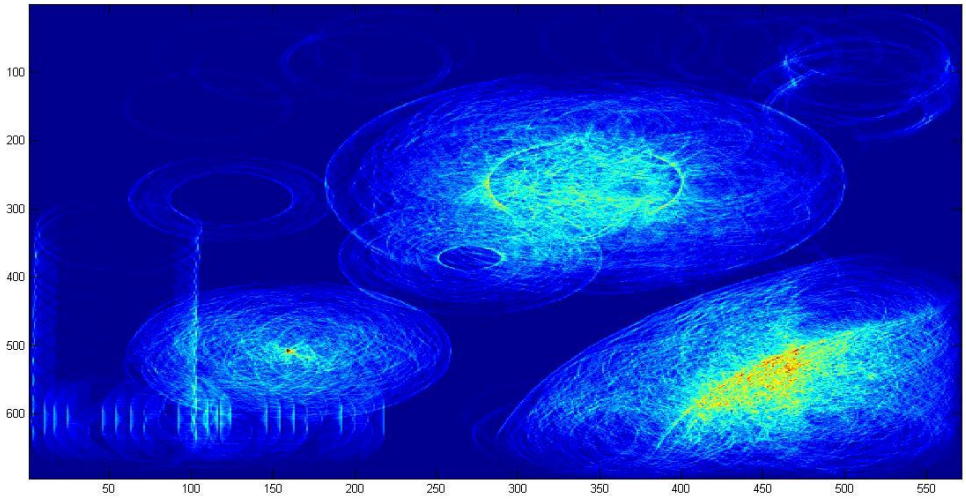




egg.jpg , for r=6 , useGradient = 0



jupiter.jpg , for r=50 , useGradient=1

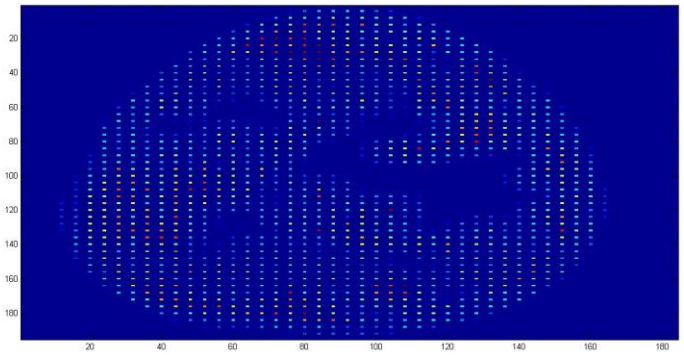


c) To visualize the Hough space, we used the ‘imagesc’ command on the accumulator array. Here, the blue areas mean lesser values and red mean higher. So we are essentially looking for the ‘red’ values in our accumulator array as they correspond to the highest votes. The appearance is circular because ever pixel corresponds to a circle in Hough space, and hence is casting the votes accordingly.

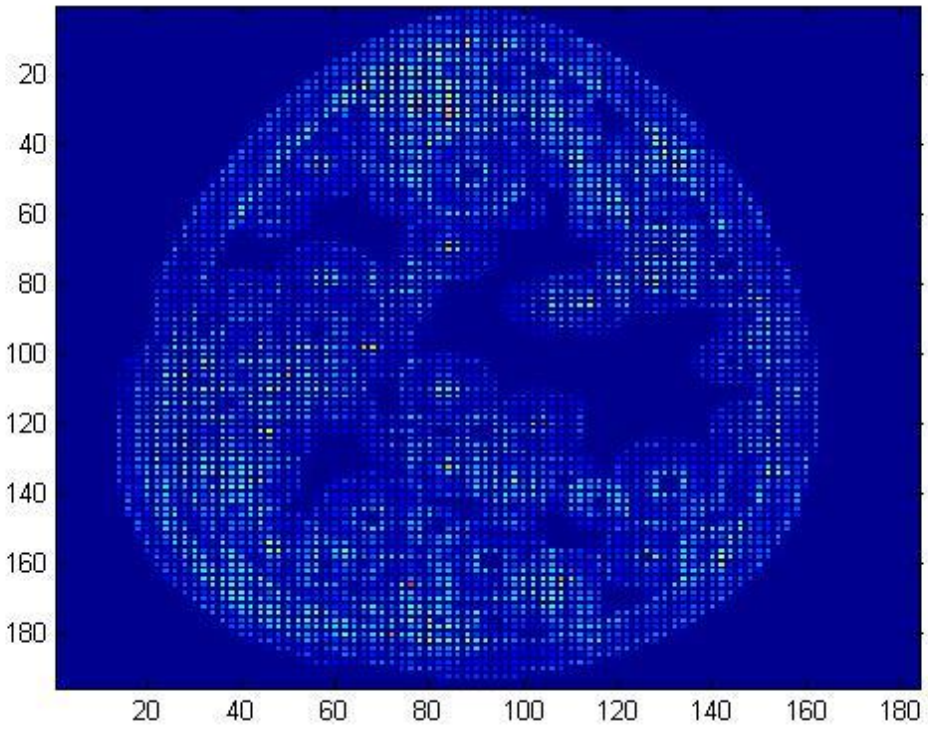
d) There are many possible ways in which one can post process the accumulator array to get the final number of circles. We can consider the top ‘n’ number of votes that are above a threshold. We can also consider the top few percentage of pixels with maximum votes. This method will take care of the cases when there are a lot of candidates will number of votes very close to each other. A threshold can be chosen by looking at the highest number of votes and the next highest and so on. Also, the threshold should be higher for a circle with bigger radius as the number of points voting for it would be higher. For the sake of simplicity, we have used the top ‘n’ circles as our post processing step. This is stored in our ‘maxNumOuts’ parameter in detectCircles.m.

e) Here are a few results (explanation after the results):

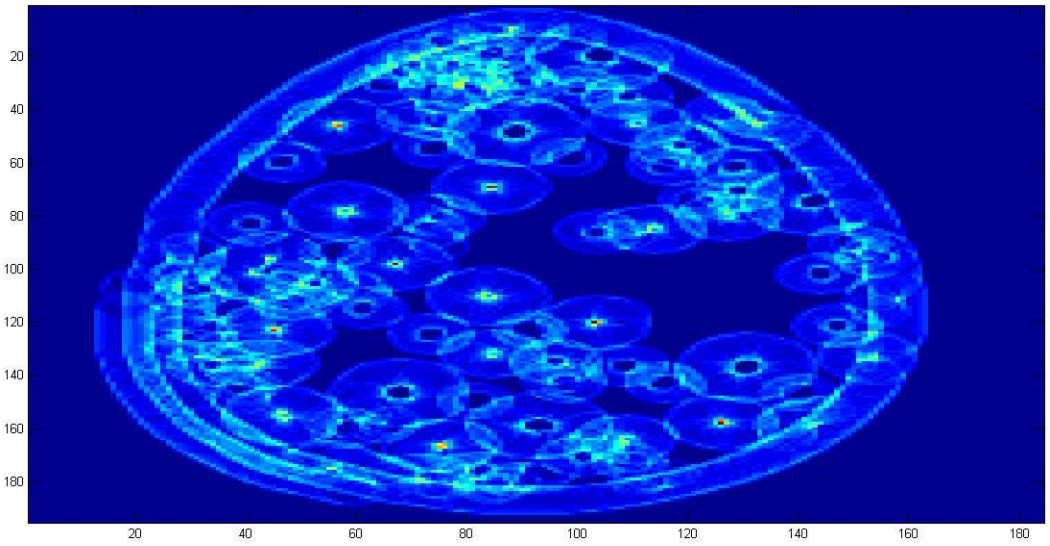
Following image is for binsize =4 , radius = 5 , useGradient = 0



Following images are for binsize = 2 , radius =5 , useGradient =0



Following results are for binsize = 1 , radius = 5 , useGradient = 0



Explanation :

We experimented with varying the binsize only, while keeping everything else constant. Here, ‘binsize’ is our variable that controls the ‘step-size’ of our centre values. That is to say, that if binsize becomes double of before, each accumulator vote bin will become double the size of before. So if we look at the results carefully, binsize = 1, gives us the perfect results. This is our default value. Now, if we increase the granularity and have binsize = 2, we can see that the centres are getting predicted ‘almost’ correctly. That’s because due to the step-size, it is not able to vote for the most accurate centre value. The results are thrown completely off when we increase the step-size to 4. Also, when we visualize the Hough Space, it appears more and more granular as we keep increasing the binsize