1. Short Answer Problems

1. Mean-shift would be the most appropriate algorithm for the task. Mean shift is a simple iterative procedure that seeks modes or local maxima of density in the feature space. Hough transform is a voting technique. Points with the most votes in Hough space indicate line in image space. The mode in feature space for mean-shift algorithm is similar to the point with the most votes in Hough space.

Even though K-means performs a similar function, it is sensitive to outliers and initial centers. Moreover, mean-shift only needs to tune one parameter ---window size and not have to perform calculations for all votes placed in the vote space.

Graph-cut is suitable for binary image problem but not images where pixels can have more than two different labels.

2. The resulting shape of the clustering assignment would approximate an upper half of the circle and a bottom half of the circle. K-means converges by randomly initializing the center of clusters and minimizing the SSD among all the points to the center within each cluster. Therefore, k-means is unable to distinguish the circular shapes.

3. Since we already have foreground 'blobs,' we can assume that all blobs have pixels marked by '1'.

Variable:
width = 0
height = 0
area = 0

Because the blob might have irregular shape, we cannot assume it's a rectangle.

Count the number of '1' pixels in each cluster to get the area:

```
def area(blob):
  for each blob:
      for col  in columns in the blob:
            for row in rows  :
                  if blob[row][col] == 1:
                         area += 1
                         height += 1
            width += 1
    return area, height, width
```

```
def centroid:
     return the mean of the width and height

def  how_close_to_rectangle_ratio:
    rectangle = height* width
    return area / rectangle

def   how_close_to_circle_ratio:
    return sqrt( (height/2 - centroid )^2 + (width/2 - centroid )^2)
```

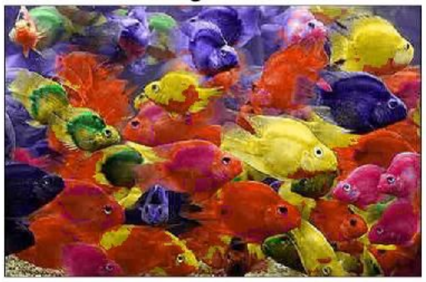
Algorithm:

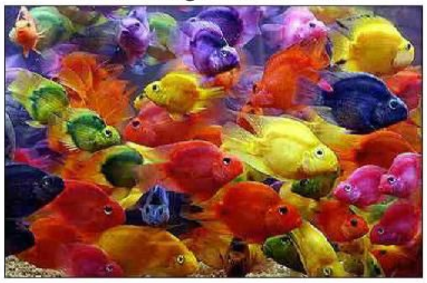
1. Find the area, height, width and centroid for each blob.
2. Find the **how_close_to_rectangle_ratio and how_close_to_circle_ratio** of each blob.
3. Cluster the blobs according to their rectangularity and circularity.

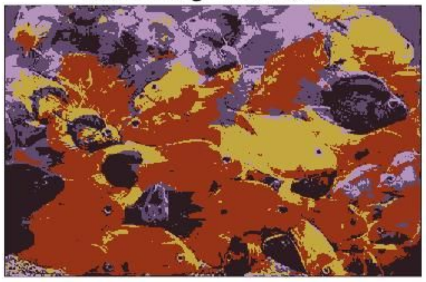
2. Programming

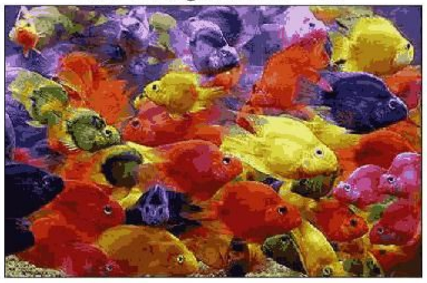
1. Color quantization withk-mean

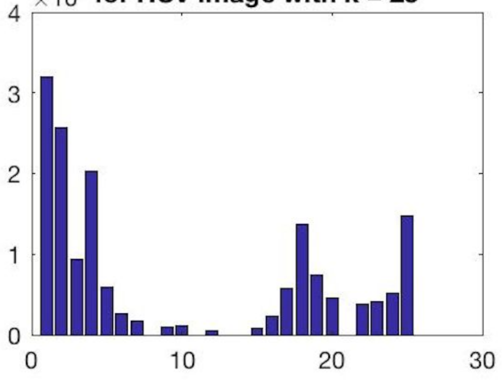
(e)

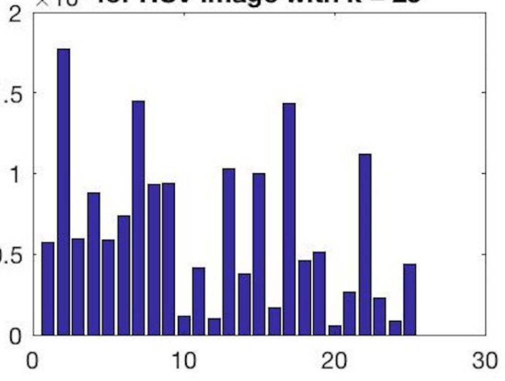

HSV image with k = 5

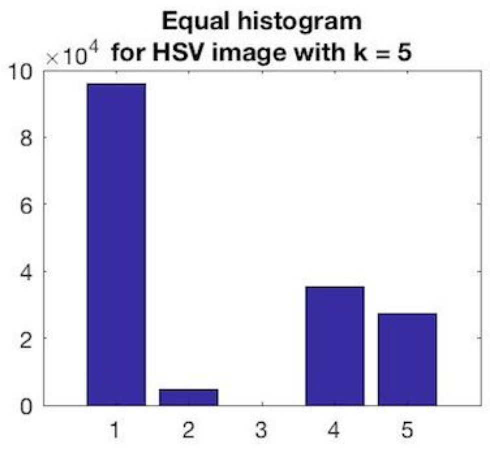

HSV image with k = 25

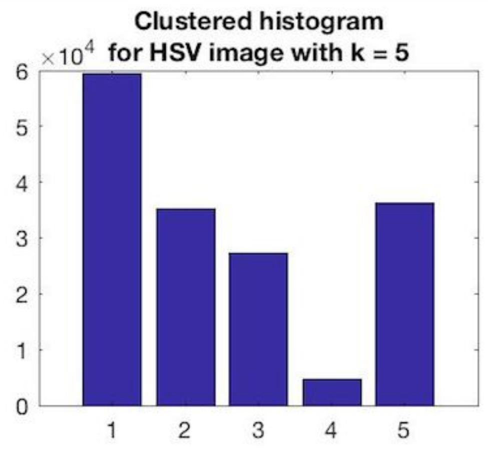

RGB image with k = 5


RGB image with k = 25


Equal histogram for HSV image with k = 25


Clustered histogram for HSV image with k = 25





The error of HSV image with k = 5 is 6.108185e+09
The error of HSV image with k = 25 is 6.108174e+09

The error of RGB image with k = 5 is 462329893
The error of RGB image with k = 25 is 134179450

f)
K-means quantization employs clustering to compress data. For images in the HSV color space, better clustering was achieved with higher k value. A k-value of 25 allows more fine-grained segmentation than a k-value of 5. The reason is simple: if each data point is its own cluster, then we would have zero-error rate because we preserve every bit of image information. Essentially, choosing a k-value is making a trade-off between image quality and memory/computation cost. Higher k-value corresponds to higher image quality but also higher memory cost.

For difference in color space, we see that HSV has higher image quality than RGB space with the same k-value. For k = 5, RGB clustering could not detect the object boundary at all. K-means algorithm is only used in the hue channel of the HSV space, compared to being used for all three channels in the RGB space. Another reason is that the color information is usually much more noisy than the HSV information.

However, RGB space has a lower error rate than HSV space because the image is represented in RGB space . In other words, computers treats color in RGB format, but we humans perceive color in HSV space,

2. Circle detection with the Hough Transform [40 points]

(a).

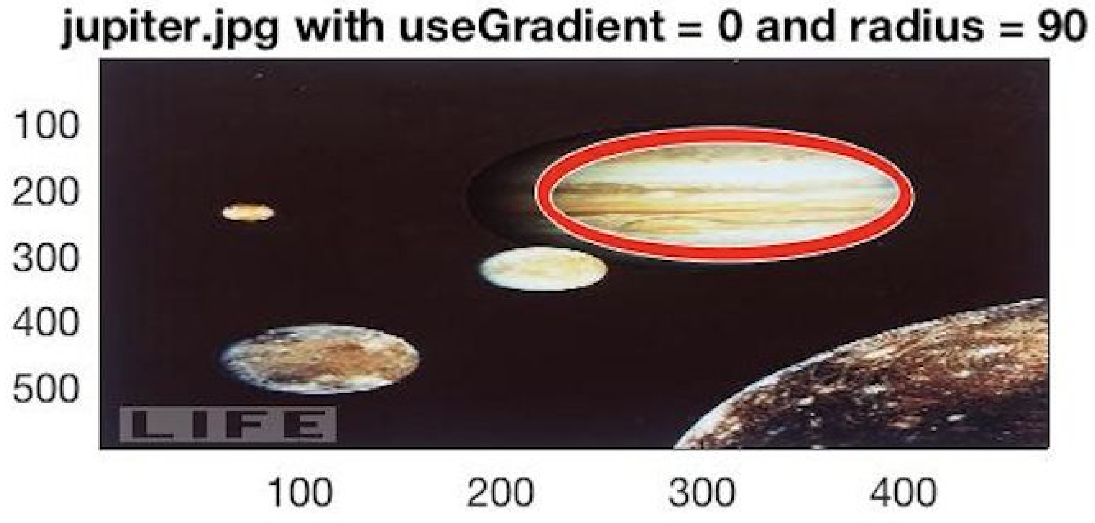
For preprocessing, I converted the image to grayscale in order to minimize irrelevant tokens. After that, I called Matlab’s Canny function to detect edges. I only take edge points with significant gradient magnitude. I create an accumulator matrix for edge points to cast voting.

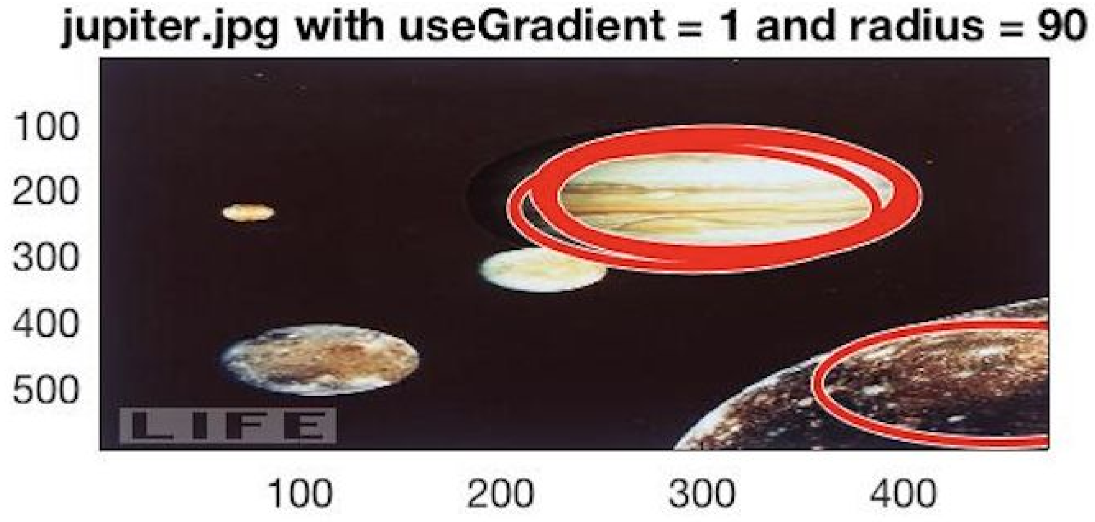
If gradient is not used, I would proceed to calculate the vector of cosine and sine in theta (0-360 degrees) range. Any given edge point gets to cast a vote for the center of every angle it forms with the radius.

If gradient is used, a and b are calculated with the gradient value acquired from gradient() and atan2 function. Using the gradient direction has the benefit of to reducing the number of votes by virtue of a narrower window of theta.

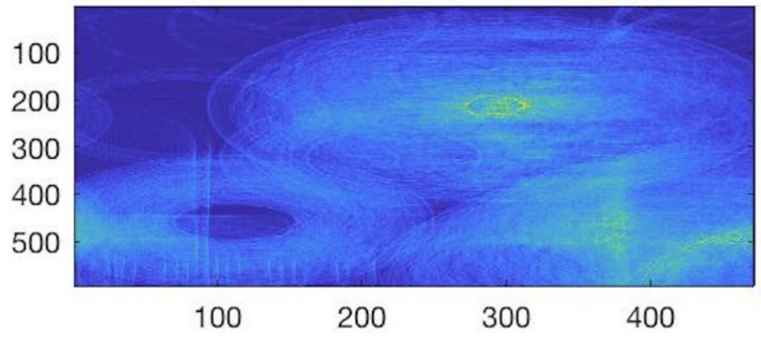
At the end, I only retain edge points that are above 90 percentile of the Houghs voting, which form the ‘centers’ array.

(b)

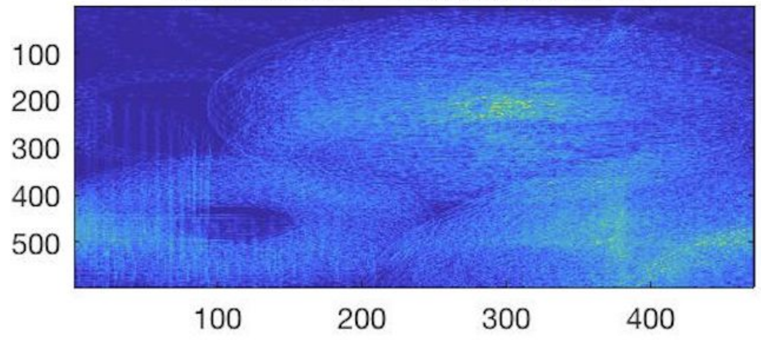

jupiter.jpg with useGradient = 0 and radius = 90
100
200
300
400
500
LIFE
100
200
300
400

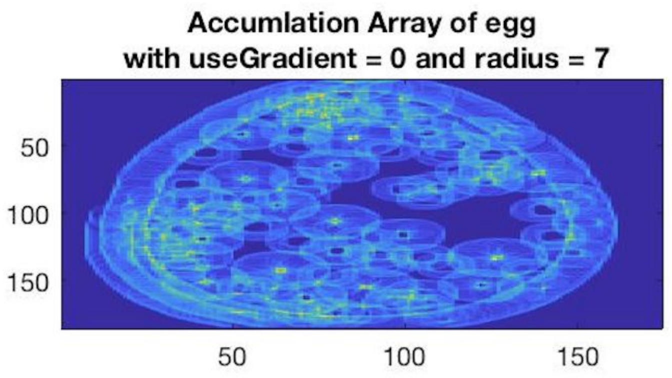

jupiter.jpg with useGradient = 1 and radius = 90
100
200
300
400
500
LIFE
100
200
300
400

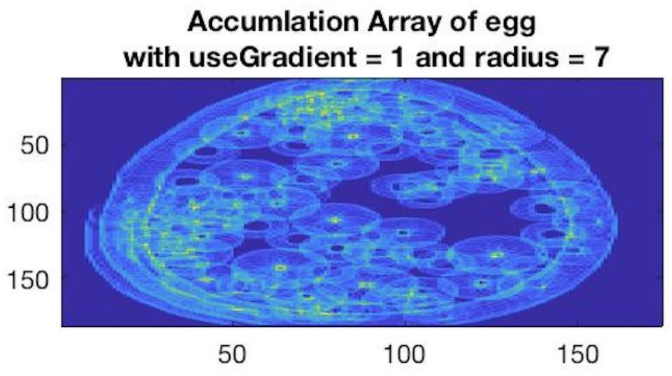
Accumlation Array of jupiter with useGradient = 0 and radius = 90

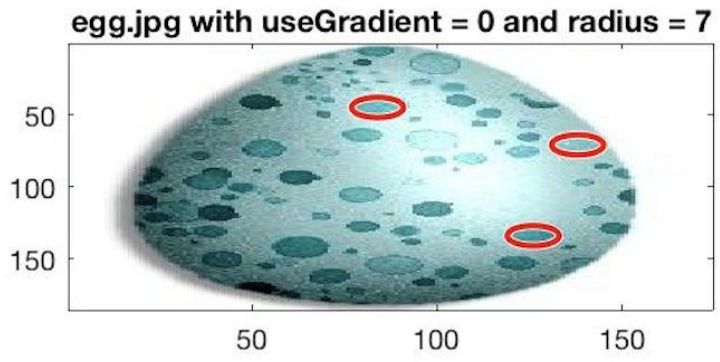


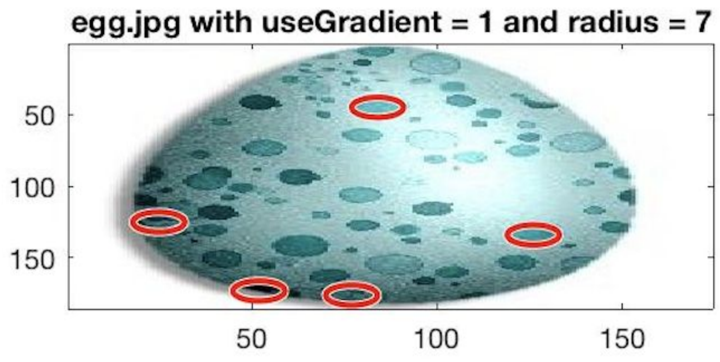
Accumlation Array of jupiter with useGradient = 1 and radius = 90




Accumlation Array of egg
with useGradient = 0 and radius = 7
50
100
150
50
100
150


Accumlation Array of egg
with useGradient = 1 and radius = 7
50
100
150
50
100
150


egg.jpg with useGradient = 0 and radius = 7
50
100
150
50
100
150


egg.jpg with useGradient = 1 and radius = 7
50
100
150
50
100
150

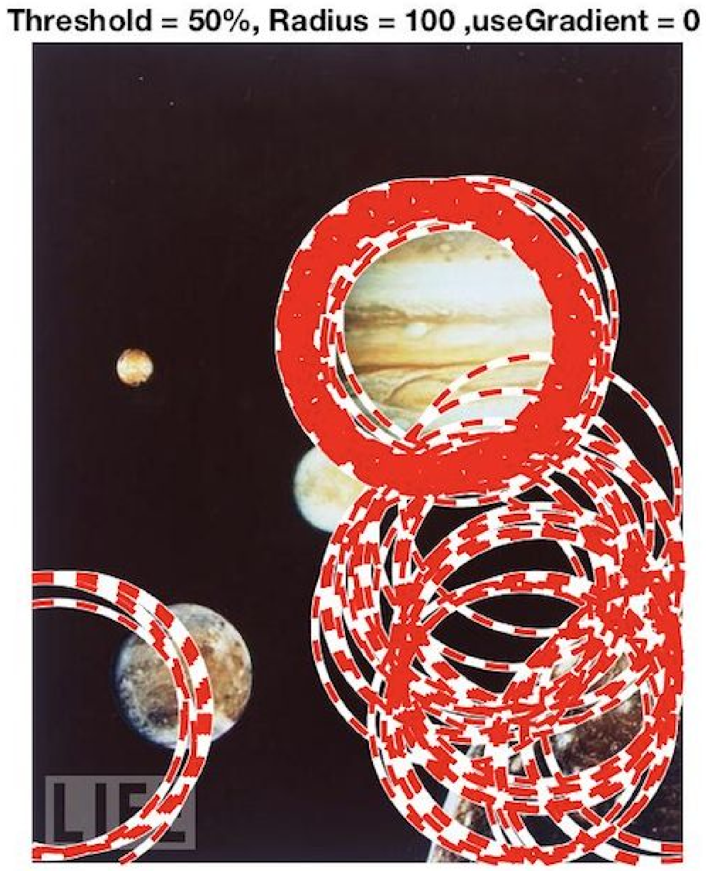
(c)See above. The egg's accumulator array image shows the inner working of the Hough transform.

We can see that many circular boundaries overlap even though they do not overlap in the image space. Almost all circular boundaries appear bigger than their counterparts in the image space.

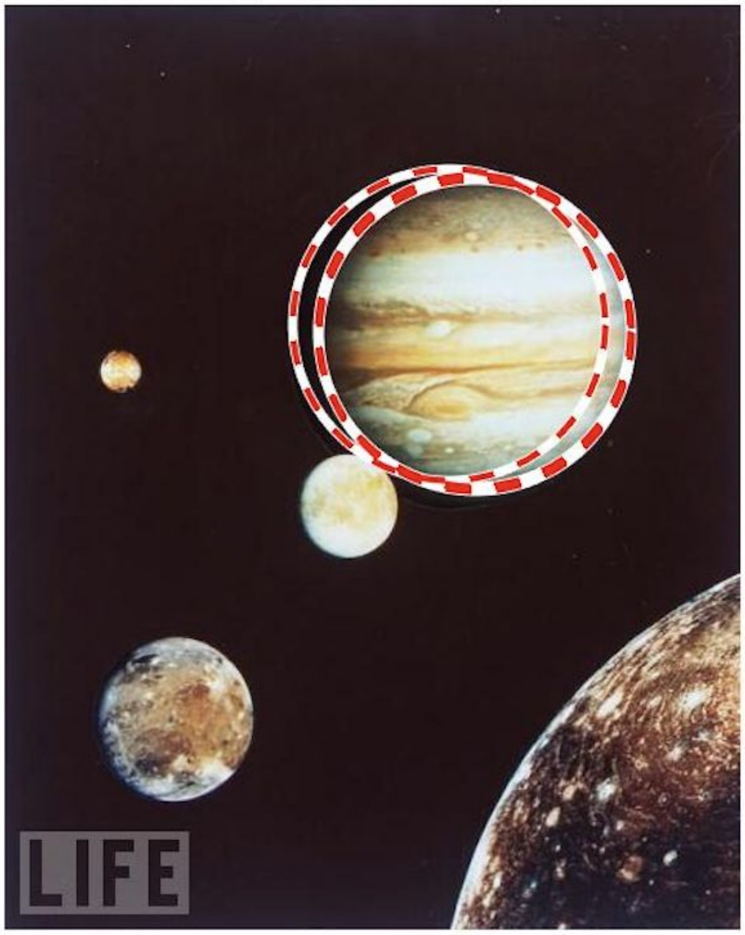
In the Hough space, bright value = high vote count.

(d) I experiment with the accumulator array by tuning the bin-size and radius. Because the radius is fixed, at any given time, I can only detect circle of the same radius.   It is hard to pick a good radius. I tried multiple quantization number before settling on a radius of 90 for Jupyter and 7 for egg. If I have the right radius, I can set the bin size to 95% and get an almost perfect boundary. If I set the bin-size to 65%, the algorithm fit many partial circles. Nevertheless, Hough transform is robust against outliers because of the voting system.

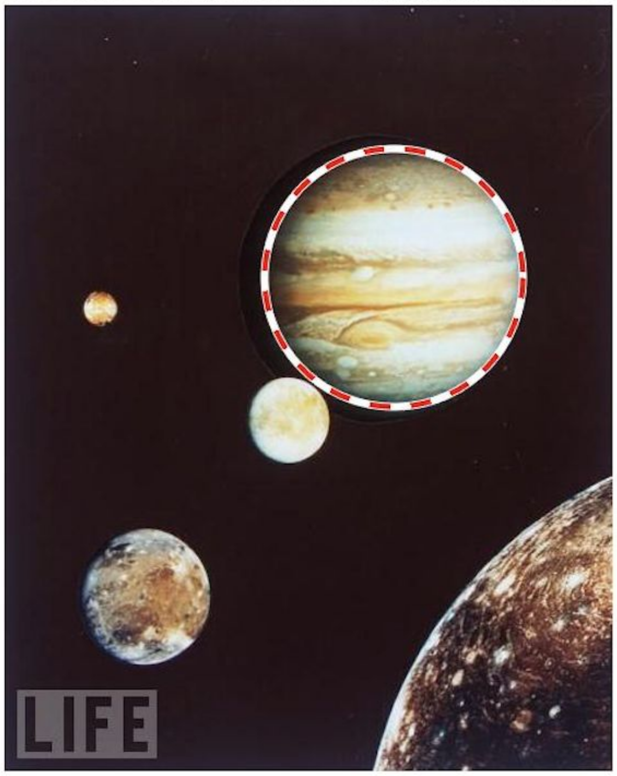
(e)



Threshold = 75%, Radius = 100 ,useGradient = 0

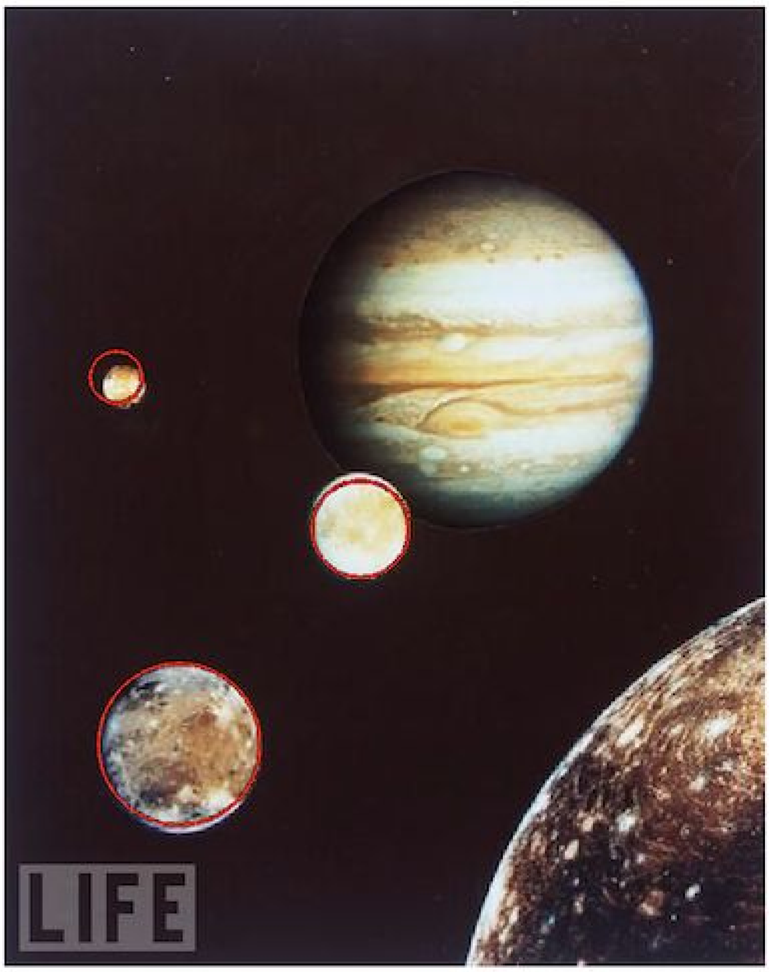


Threshold = 95%, Radius = 100 ,useGradient = 0



From the images with bin size ranging from 50% to 95%,we know that a small bin size gives rise to extremely noisy boundary detection. However, it is faster than a larger bin size. The bin size, which means minimum vote that it needs to be considered a circle, is critical in accurate boundary detection.

3. Optional:



“function[centers] = detect_radii(im,useGradient)”

I set the minimum radius to 0 and the maximum radius to max(height, width) / 2 , because a circle within an image cannot be bigger than the image.

Then I put the “for radius = minR : maxR” inside the nested for-loop where I increment the accumulator matrix. I have tmp array that stores every radius that satisfy “if(a > 0 && a < row && b < col && b > 0).”
When I extract centers and radi from the array, I make sure they correspond to each other.

Instruction:
Run main.m
My function is detect_radii.m

```
function[ centers] = detect_radii(im,useGradient)
%DETECT_RADII Summary of this function goes here
%   Detailed explanation goes here
    orig = im;
    im = rgb2gray(im);
    im = double(im);
    [row, col] = size(im);
    edges = edge(im, 'canny');
    edge_indices = find(edges);
    accumulator = zeros(row, col);
    centers = [];
    tmp = [];
    radi = [];
    minR = 0;
    maxR = max(row,col)/2;

    if useGradient == 0
        for i = 1: size(edge_indices,1)
            [x, y] = ind2sub([row, col], edge_indices(i));
            for radius  = minR : maxR
                for angle=1:360
                    a = abs(ceil( x - radius*cos(angle)));
                    b = abs(ceil(y + radius*sin(angle)));
                    if(a > 0 && a < row && b < col && b > 0)
                        accumulator(a,b) = accumulator(a,b) + 1;
                        tmp = [tmp ; radius];
                   end
                end
            end
        end
    end
    if useGradient == 1
        for i = 1: size(edge_indices,1)
```

```
            [x, y] = ind2sub([row, col], edge_indices(i));
        %   [dx, dy] = imgradientxy(edges);
            [dy,dx] = gradient(im);
            gradientDirections = atan2(-dy,dx);
            for angle =  (-gradientDirections-45):  (-gradientDirections+45)
                for radius  = minR : maxR
                    a = abs(ceil( x - radius*cos(angle)));
                    b = abs(ceil(y + radius*sin(angle)));
                    if(a > 0 && a < row && b < col && b > 0)
                        accumulator(a,b) = accumulator(a,b) + 1;
                        tmp = [tmp ; radius];
                    end
                end

            end

        end
    end



    accumulator = accumulator/max(accumulator(:));

    accumulator_inds = find(accumulator >0.85);

    for i = 1: size(accumulator_inds,1)
        [ax, ay] = ind2sub([row, col], accumulator_inds(i));
        radi = [radi; tmp(i)];
        centers = [centers; [ay,ax]];

    end
imshow(orig);
hold on
viscircles(centers,radi, 'LineStyle','--')
```