CS 4476: Computer Vision, Fall 2020

PS2

Name: Fei Ding

Due: Thursday, September 24th 2020

Problem 1

(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.

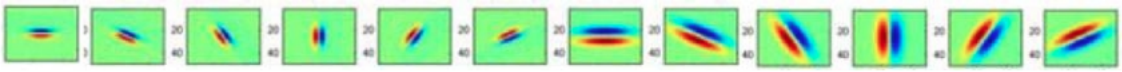


Figure 1: Filter bank

The resulting representation is invariant to orientation. The filter bank includes six rotated versions of the same filter, so now it has the power to generate a feature vector and transform that into a orientation-invariant texture-feature space. In fact, the purpose of rotation is to make this texture detection method orientation-invariant. Similarly, we can also make it more scale-invariant had we included even more filters of various sizes.

(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.

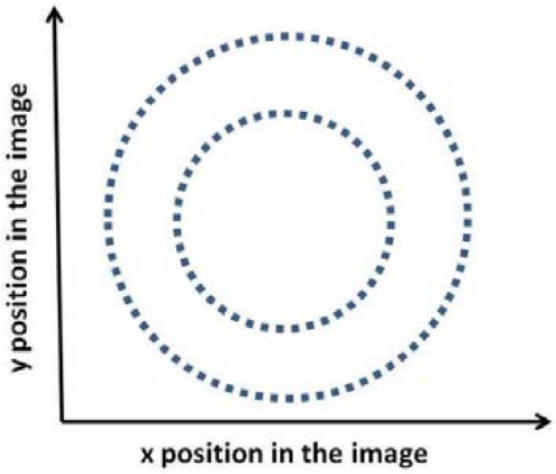
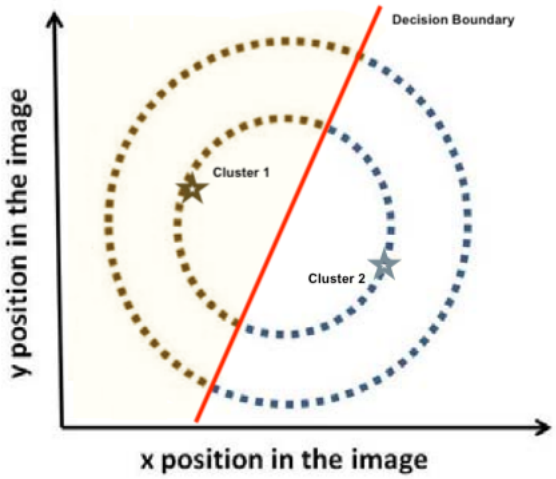


Figure 2: Edge points

We will likely have two clusters that split the two circles into equal halves. The perceived decision "boundary" would be a straight line passing through the common center point of the two circles. The two cluster centroids would lie somewhere between the outer circle and the common center. Collectively, they will also be symmetric to the common center. A possible assignment is shown below.



(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.

Mean-shift would be the most appropriate to recover the model parameters in the continuous Hough space. The target feature in the original image will result in highly dense clusters of vote points in the Hough space, guiding the mean-shift search window to gradually move toward the center of most votes. These final converging points correspond to the original feature parameters.

K-means is not appropriate because we don't know how many patterns there are in the original image, and thus choosing an appropriate K is difficult. The spherical cluster assumption is also unsound. Graph-cuts is not mode finding but graph based, and it assumes each pixel as a graph node, so it is not relevant to this problem setting of a continuous feature space.

(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.

There are many good features to measure shape similarities: aspect ratio, circularity, compactness, waviness, etc. Each one will add one dimension to the feature space and produce more satisfactory result. Below we consider only the aspect ratio and circularity and apply K-means clustering.

Algorithm 1 Blob Grouping

samples = [ ]
sampleToBlob = {}
for *blob* in *blobs* **do**
    # find edge points
    edgePoints = [ ]
    for *pixel* in *blob* **do**
        if NOT all eight-neighboring pixels of *pixel* exist in *blob* **then**
            add *pixel* to *edgePoints*
        end if
    end for

    # find aspect ratio
    $maxX$, $minX \leftarrow$ max(column indices in *edgePoints*), min(column indices in *edgePoints*)
    $maxY$, $minY \leftarrow$ max(row indices in *edgePoints*), min(row indices in *edgePoints*)
    aspectRatio $\leftarrow$ max(($maxX$ - $minX$) / ($maxY$ - $minY$), ($maxY$ - $minY$) / ($maxX$ - $minX$))

    # find circularity
    centroid $(\overline{r}, \overline{c}) \leftarrow$ FindCentroid(*edgePoints*)
    $K \leftarrow$ length of *edgePoints*
    $\mu_R \leftarrow \frac{1}{K} \sum_{k=0}^{K-1} ||(r_k, c_k) - (\overline{r}, \overline{c})||$
    $\sigma_R \leftarrow \sqrt{\frac{1}{K} \sum_{k=0}^{K-1} [||(r_k, c_k) - (\overline{r}, \overline{c})|| - \mu_R]^2}$
    $circularity \leftarrow \mu_R / \sigma_R$

    add *sample* $\leftarrow (aspectRatio, circularity)$ to *samples*
    mark *smapleToBlob*[*sample*] $\leftarrow$ *blob*
end for

clusters = KMeans(*samples*, K = specified number of groups)
for *cluster* in *clusters* **do**
    for *sample* in *cluster* **do** replace *sample* with *blob* by looking up *sampleToBlob*
    end for
end for

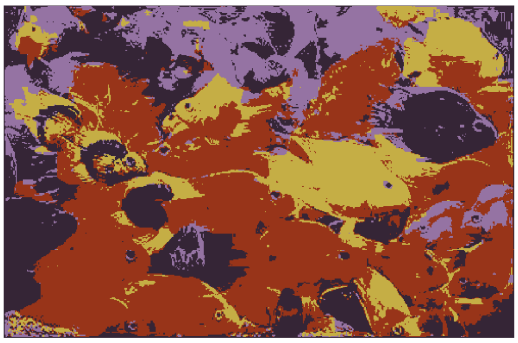
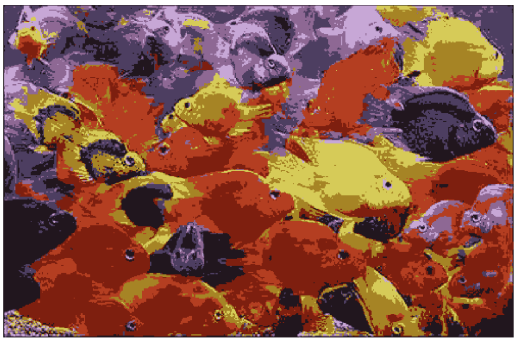
Now the *clusters* consist of grouping of *blobs*.

Problem 2

2.1 Color quantization with k-means

(1) Use the functions defined above to quantize the fish.jpg image included in the zip file from part A. On you answer sheet, display the results of quantizing the image in (a) rgb space and (b) hsv space (as described above), print the SSD error between the original RGB image and the results of the (c-d) two quantization methods, and display the (e-f) two different Hue space histograms. Illustrate all of the results with two values of k, a lower value and higher value. Label all plots clearly with titles.

(a) RGB Space Quantization
Left: K = 4; **Right:** K = 8

(b) HSV Space Quantization
Left: K = 4; **Right:** K = 8


(c) SSD Error for RGB Quantization

RGB-Quantization SSD Error, K=4: 563873940
RGB-Quantization SSD Error, K=8: 310583501

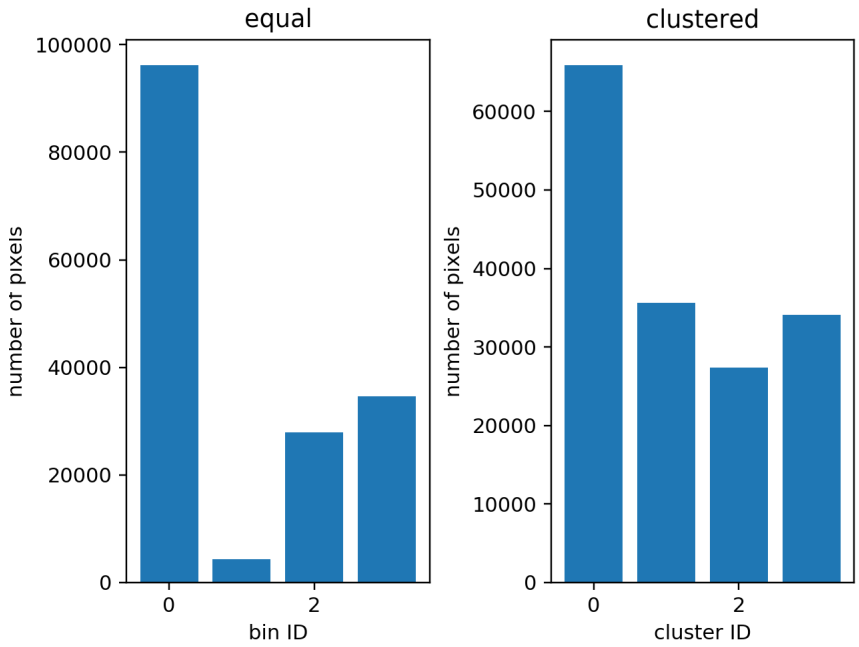
(d) SSD Error for HSV Quantization

HSV-Quantization SSD Error, K=4: 89881465
HSV-Quantization SSD Error, K=8: 34670048

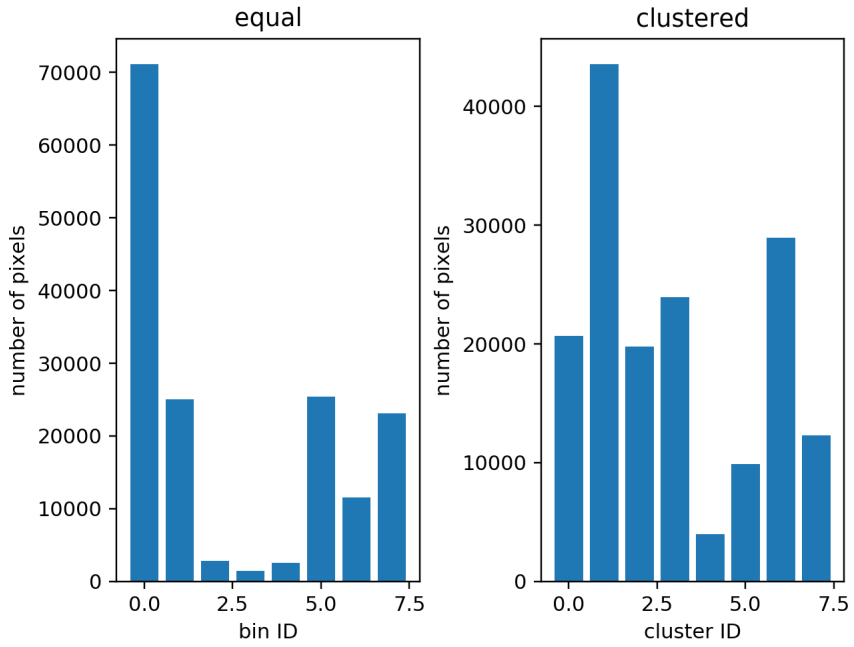
(e) Hue Space Histogram (Equally Sized Bins)

(f) Hue Space Histogram (Clusters)

K = 4



K = 8



(2) On your answer sheet, explain all of the results. How do the two forms of histogram differ? How and why do results vary depending on the color space? The value of k? Across different runs?

How do the two forms of histogram differ?

The two forms of histograms differ in terms of the sizes/ranges of bins. Both histograms display results of the hue space, but **hist_equal** divides the hue space into k equally-sized bins, whereas the **hist_clustered** creates bins based on the clustering results with k clusters. The later has bins of that are not necessarily equally spaced, meaning some bins may have ranges larger than other bins.

How and why do results vary depending on the color space?

The results on images differ because the functions `quantize_rgb` and `quantize_hsv` operate differently. `quantize_rgb` quantizes the images to have only k RGB colors. On the contrary, `quantize_hsv` only quantizes the hue channel into k values but leaves the brightness and saturation unaffected, resulting in more possible RGB values when converted to RGB images. In other words, for `quantize_rgb`, k corresponds to the number of colors, but for `quantize_hsv`, k only corresponds to number of hues.

The value of k?

As we increase the value of k, the SSD between the original image and the quantized ones dropped for both `quantize_rgb` and `quantized_hsv`. This is expected because adding more cluster centers would result in some data points become closer to one of the cluster centers. In the original image, the effect would be having pixels values more similar to that of the original image, thus lowering SSD.

Increasing the value of k would also make both images to look more similar to the original input image since we are allowed to have more representative colors. Meanwhile, the histograms also tend display more details: we can now observe some ranges of hues are much more frequent in the image than some others, because the distributions become more uneven.

Across different runs?

The results are consistent each time with a fixed random seed, and changing the value of k does not affect the trends we have observed. In particular, we can see the SSD for `quantize_hsv` is almost always lower than `quantize_rgb` because the former generally allows more RGB values than the latter.

2.2 Circle detection with the Hough Transform

(1) Explain your implementation in concise steps (English, not code/pseudocode).

I implemented a circle detection algorithm through Hough Transform. Detailed steps are shown below as a sequence.

1. Take the input RGB image and convert it to gray scale. The particular function I used was `skimage.color.rgb2gray`.

1b. If `use_gradient` is set to `True`, apply the first derivative filters in the x and y direction to create a gradient estimate matrix for all pixel points. Otherwise, skip this step.

2. Use the Canny Edge Detector provided by `skimage.feature.canny` to find the edges in the image. The output is a binary image. I also tuned the parameters `sigma` for optimal edge detection result: 6 for `jupiter.jpg` and 3 for `egg.jpg`.
3. Create the Hough Space accumulator array for voting. I used a 2-d `numpy` array the same size as the input image. *The voting space quantization was later implemented by scaling up/down the array.*

4. For every edge pixel in the binary image, when `use_gradient` is not set, iterate `theta`(θ) from 0 to 2π by some fixed stride (0.01), calculate the coordinate $(x + r\cos\theta, y + r\sin\theta)$ and cast one vote to the nearest bin in the accumulator array, where (x, y) is an edge pixel and r is the radius of circle to detect. If gradient is used, refer back to the gradient value matrix to find the gradient pertinent to the current edge point, use that to estimate the two directions (opposite) for `theta` and only cast two votes in the accumulator array.

5. Set a hard threshold based on the maximum vote present in the accumulator array and find all coordinates exceeding that threshold. These will represent the centers of detected circles.

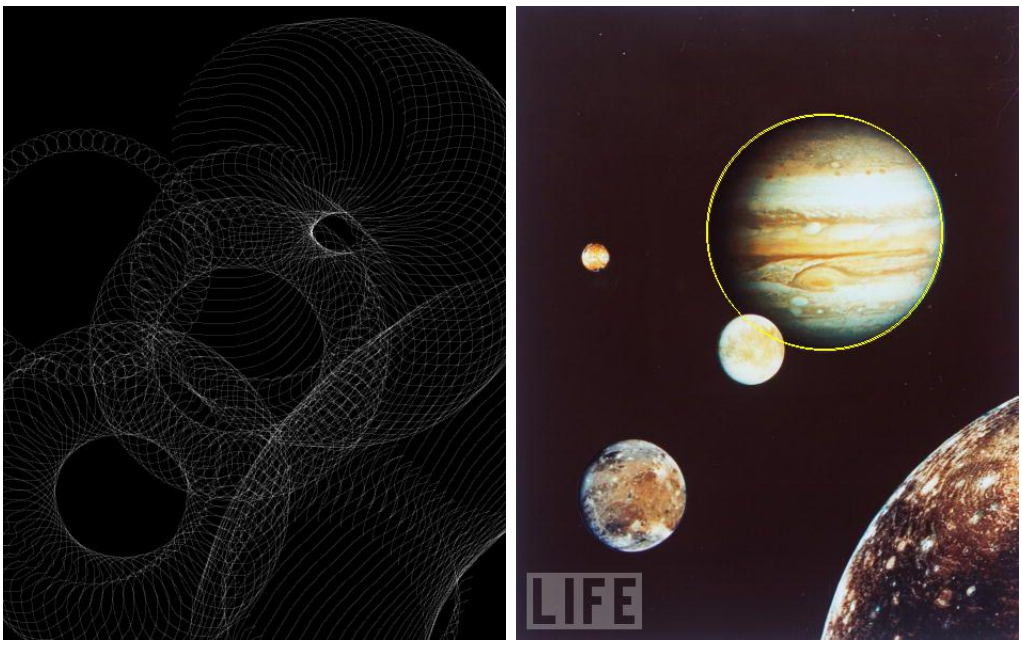
6. Transpose the `centers` if necessary to make its dimension $N \times 2$. Return it as the function output.

How to Use:
1. You can try `from submissionDetectCircles import detect_circles` in your script. This function has signature exactly as defined in the requirement PDF.
2. You can call `python submissionDetectCircles.py` on your command line, you will then be prompted to input an option. **You can call this way to reproduce the results shown in the following sections.**

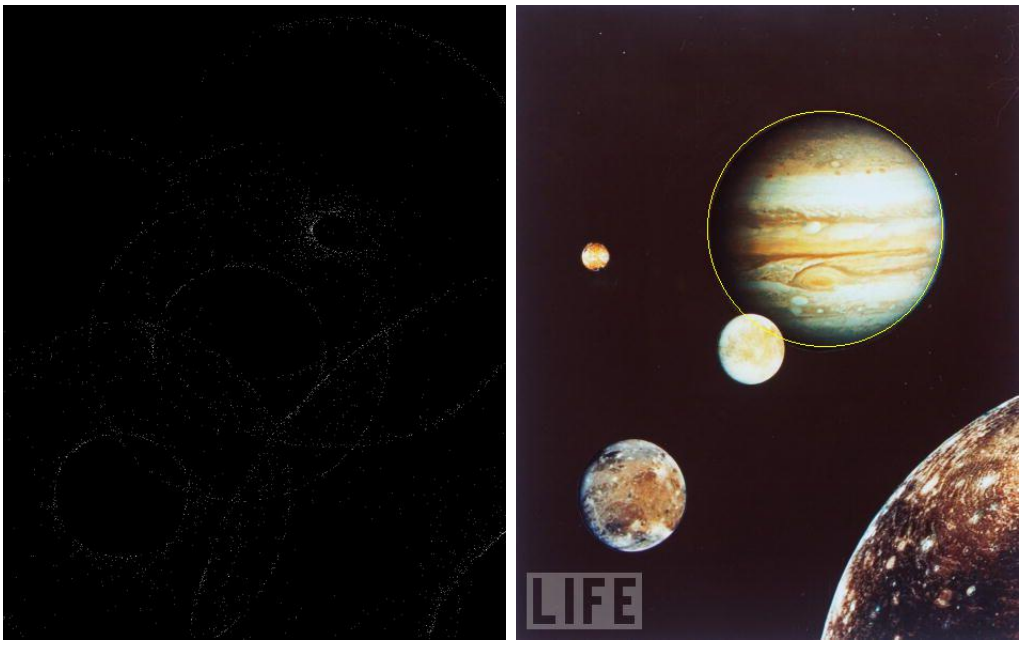
(2) Demonstrate the function applied to the provided images jupiter.jpg and egg.jpg. Display the accumulator arrays obtained by setting `use_gradient` to `True` and `False`. In each case, display the images with detected circle(s), labeling the figure with the radius.

Code Option 0, 1

1. Results for `jupiter.jpg` (`use_gradient` = `False`)
Radius: 110px. **Left**: Accumulator Array; **Right**: Detected Circles.

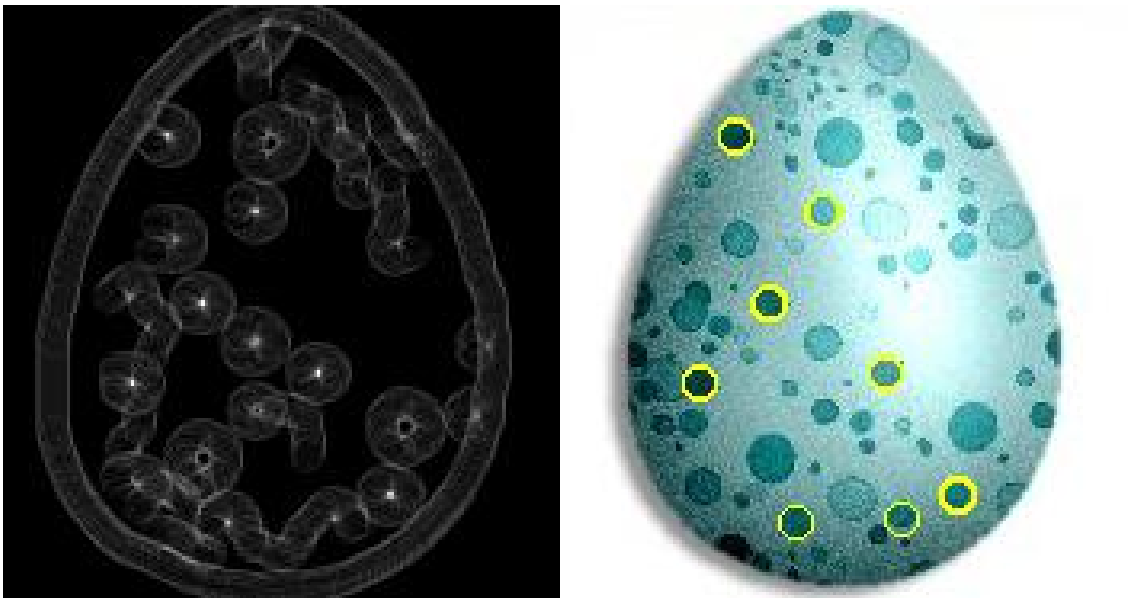


2. Results for `jupiter.jpg` (`use_gradient` = `True`)
Radius: 110px. **Left**: Accumulator Array; **Right**: Detected Circles.

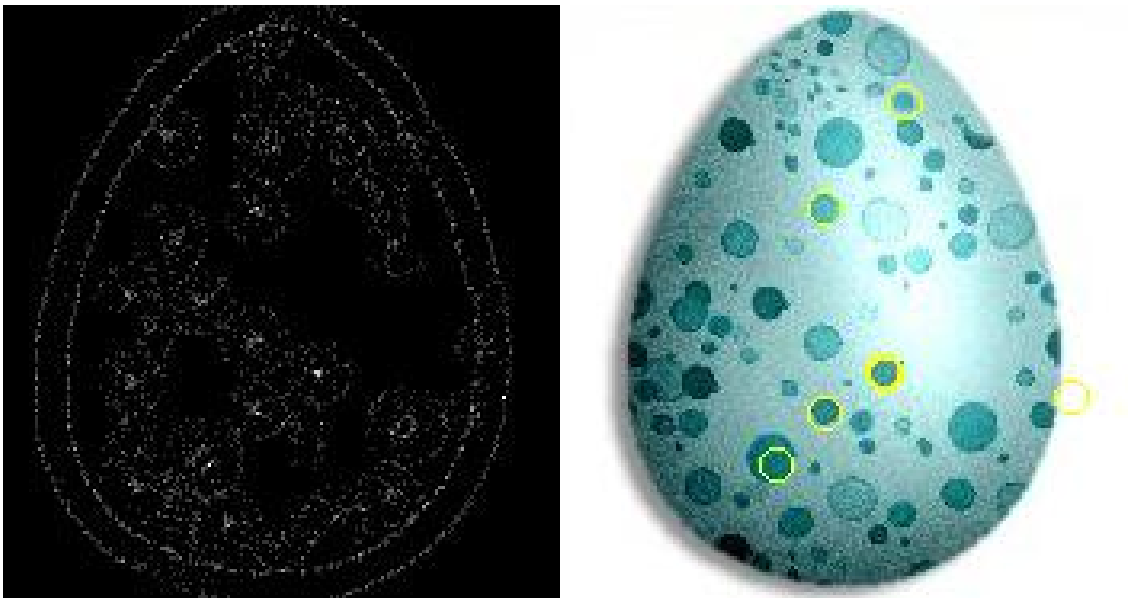


Code Option 2, 3

3. Results for `egg.jpg` (`use_gradient = False`)
Radius: 5px. **Left**: Accumulator Array; **Right**: Detected Circles.

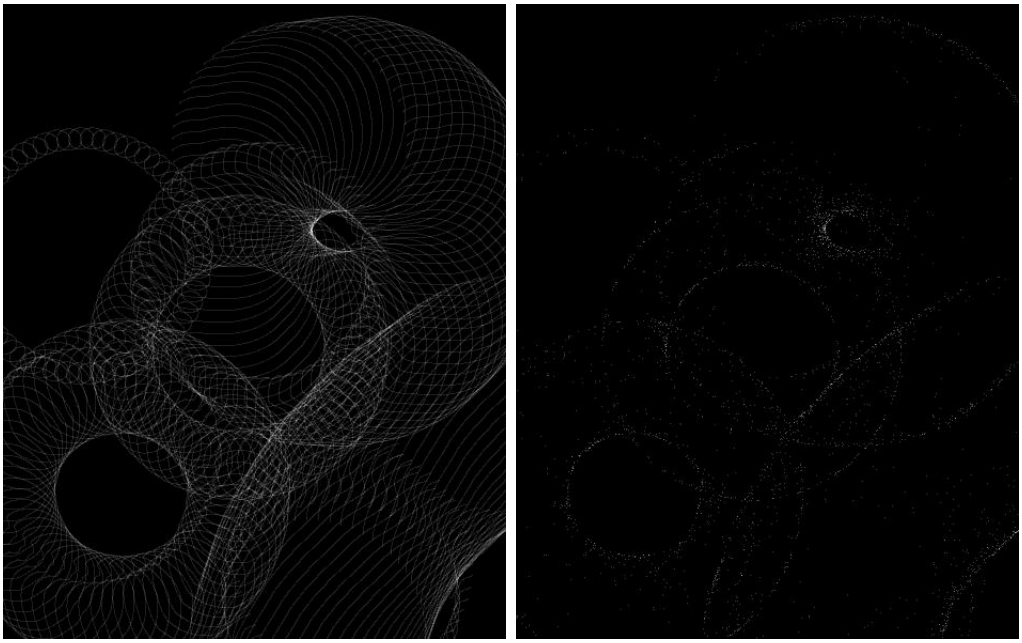


4. Results for `egg.jpg` (`use_gradient = True`)
Radius: 5px. **Left**: Accumulator Array; **Right**: Detected Circles.



(3) For one of the images, display and briefly comment on the Hough space accumulator array.

Code Option 0, 1
Image: `jupiter.jpg`; **Left**: Gradient Used; **Right**: Gradient Not used



When gradient is not used for circle detection, we can clearly see the overlapping circular votes cast by each edge pixel, and the accumulator array itself has many more votes than the one with gradient used.

When gradient is used, we can merely observe some clusters and streaks around the center region. This is because each edge point is only allowed to cast two votes based on its estimated gradient direction. Nevertheless, we can still observe the brightest parts in both accumulators correspond to the Jupiter center detected.

(4) Experiment with ways to determine how many circles are present by post-processing the accumulator array.

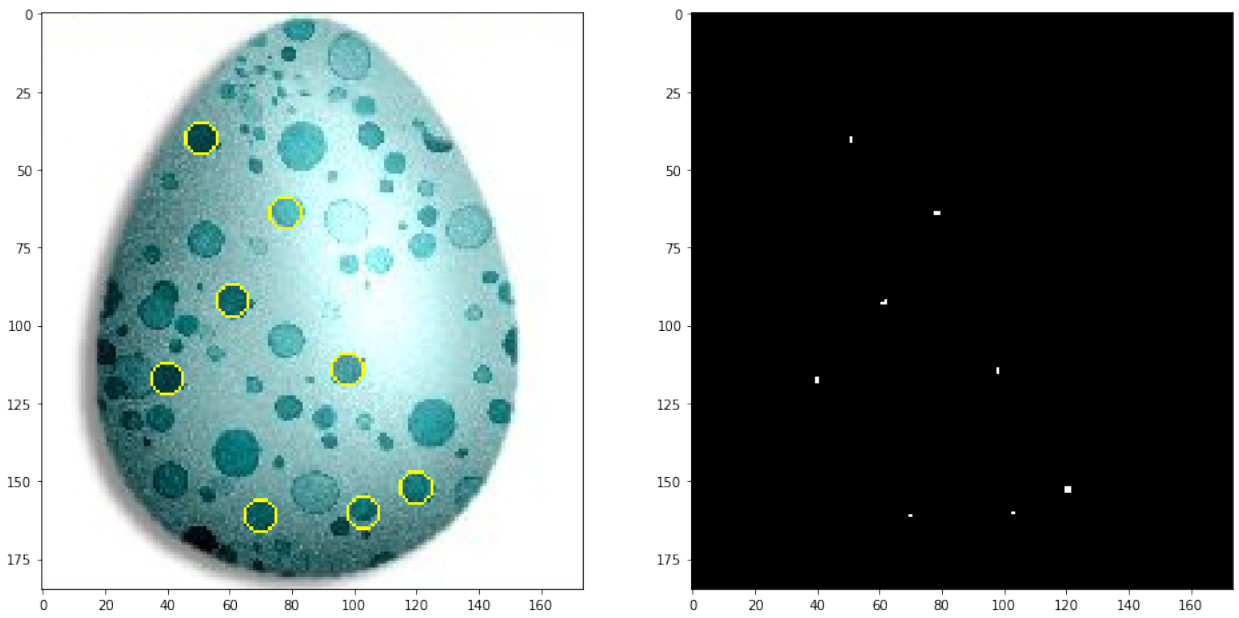
Code Option 2

My raw circle detection algorithm has two main drawbacks. First, it sometimes picks random noise from the image and outputs a non-existent circle, or it can also miss one instead. Second, it marks the same circle multiple times by counting centers that are really close to each other which in fact correspond to the same circle.

To tackle the first problem, I tweaked the voting threshold to find the sweet spot balancing between miss (circle not detected) and false alarm (non-existent circle found). It turned out a value between 0.6 - 0.8 is mostly desired, depending on the iamge content.

To resolve the second problem. I applied the optimal thresholding described above to produce a binary image and then ran connected component algorithem to count and identify blobs (because bins close to a maximal vote usually have high votes as well). To more accurately define the detected centers, I calculated the centroid of each blob and use that as the detected center. This ensures the algorithm only outputs one true center for each circle. In the image below, you will see there is no overlapping circle marks after the modification.

Image: `egg.jpg`; **Left**: Better Circle Marking; **Right**: Binary Image Centers



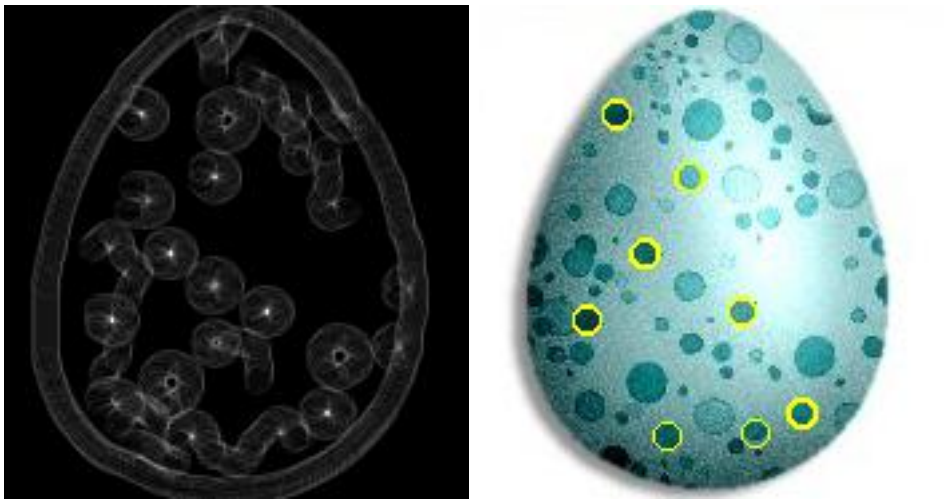
Program Output

```
Circle Count: 8
```

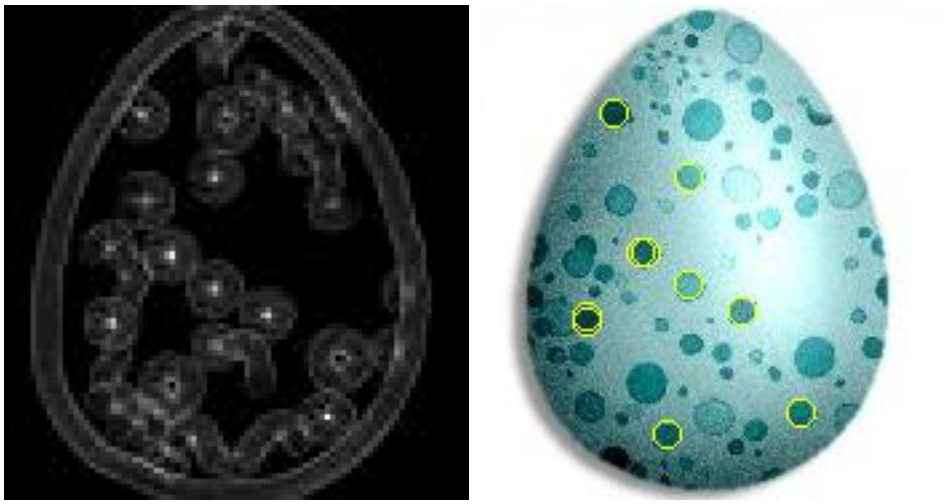
(5) For one of the images, demonstrate the impact of the vote space quantization (bin size).

Code Option 2, 4

Voting Space Shape = 1.0× Image Shape



Voting Space Shape = 0.5× Image Shape

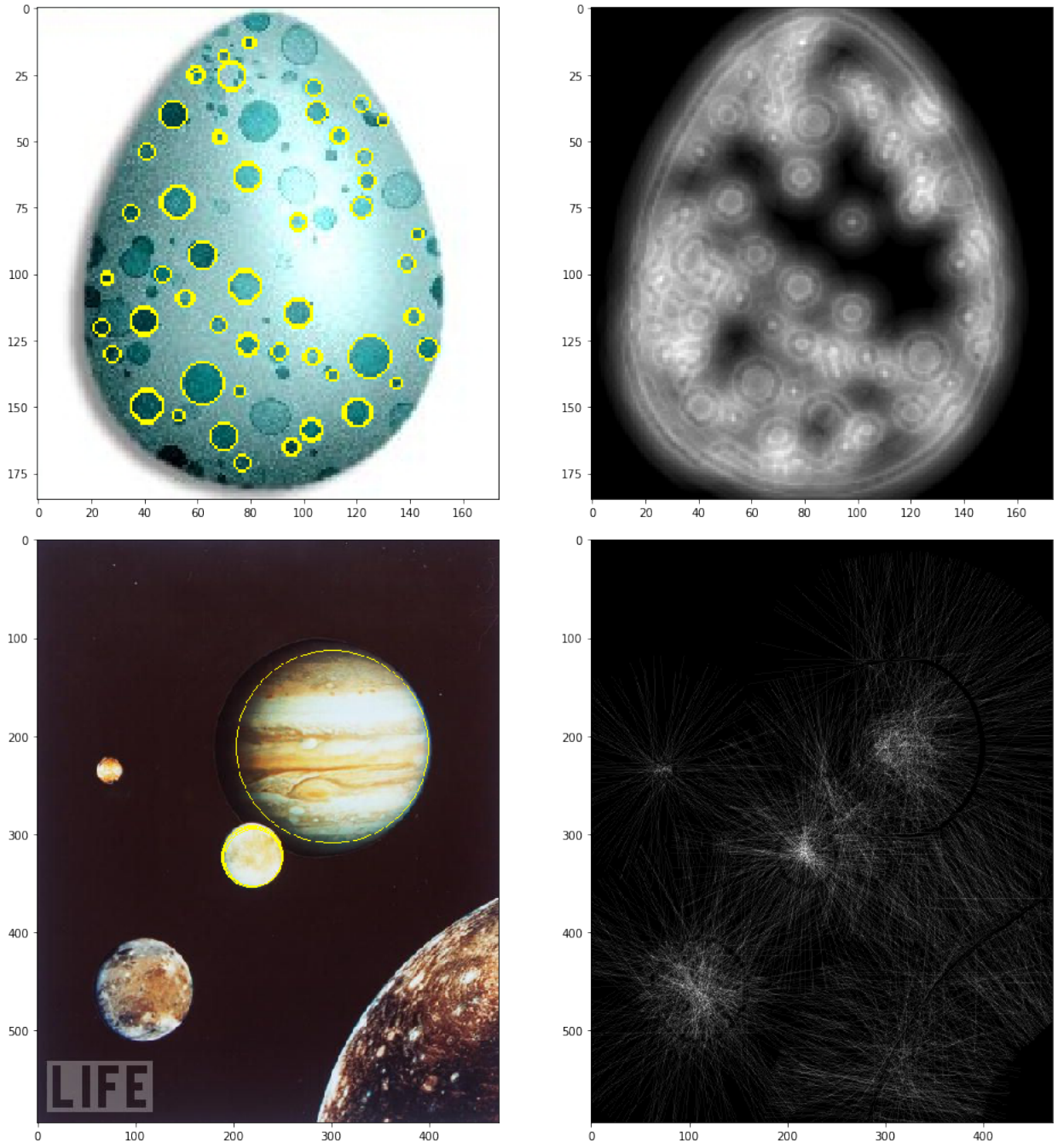


The image I chose was `egg.jpg`. I decreased the bin size to half of its original in each dimension (4 times smaller). The resulting accumulator array looks more pixelated than the original. In the result image the overlapping circles were going apart as a result of coarser granularity of the Hough space. However, we also observe that more circles tend to detected. This is because two votes that would have gone into adjacent bins now have a chance to end up in the same bin in a less fine Hough space, giving some candidate centers a high chance to survive the thresholding. In other words, we can expect to sacrifice precision for accuracy if we decrease bin size.

Problem 3

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

Code Option 5, 6



See explanation on the next page.

To enable the Hough circle detector to find circles of any radius, we need to parameterize radius in the Hough space as well. Therefore, we now have x-coordinate, y-coordinate, and radius as our parameters, and the Hough space becomes 3-dimensional. An edge point in the original image would become a cone in the Hough space, but the rest of the procedure is comparable with a fixed radius circle detector.

The images in the left column are the input images with circles marked in yellow. The images on the right are "planarized" accumulator arrays obtained by summing up votes over various radii at a particular center (x, y).

To detect circles in the `egg.jpg`, I searched over radius range $[2, 12]$ and did not use gradient. The detector produces mostly satisfactory result as it has captured most circles in the image despite missing a few.

To detect circles in the `jupiter.jpg`, I searched over radius range$[3, 110]$ and did use gradient because the image is relatively large. The detector could only identify Jupiter and its neighboring planet. However, the "planarized" accumulator array does reflect the presence of all circles in the image in the form of clusters. It is the noise in the array that makes it difficult for the algorithm to recognize all circles in the image.