ECE 4984/5554: Computer Vision, Fall 2015

PS2

Christopher Mobley

Due: Monday, 05 OCT 15

1 Short answer problems [20 points]

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.

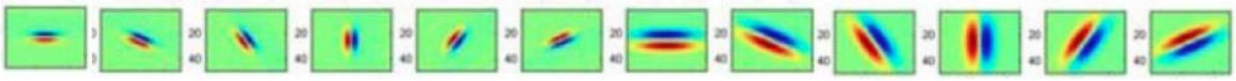


Figure 1: Filter bank

   a. The resulting representation is invariant of orientation due to the fact the filter bank includes filters that have orientation from 0 to 2π or in other words they include filters that have been rotated.

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.

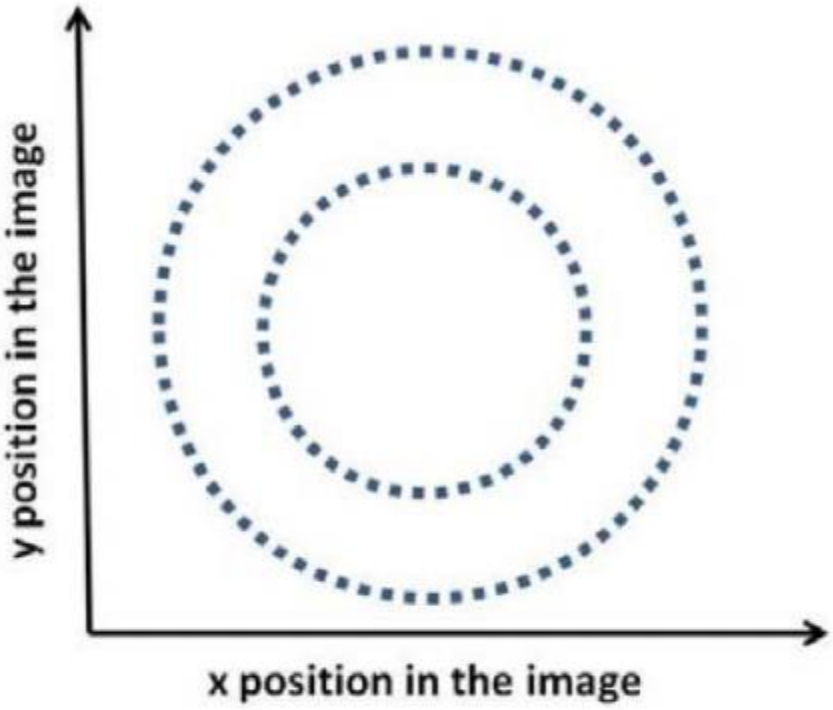


Figure 2: Edge points

a. With a k value of two the most likely clustering would be to cut the circles into equal halve through the circle's center. The k-mean algorithm iteratively solves for the cluster center in this case the cluster center would be the center of the two 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.

a. Mean-shift would be the appropriate grouping algorithm to recover the model parameter hypotheses from the continuous Hough space due to the fact that it will gradually move to center(s) of mass or to the point(s) of most votes. K-means would only find the center of a cluster(s), but not the center(s) of mass of the cluster or the point(s) with the most votes. While graph-cut just cut the space

according to the line that damages the space minimally, which would not result in center(s) of mass or to the point(s) of most votes.

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.

Circularity will be calculated using the following

mean_radial_distance = $$\mu_R = \frac{1}{K}\sum_{k=0}^{K-1}\|(r_k, c_k) - (\bar{r}, \bar{c})\|$$

variance_of_radial_distance = $$\frac{1}{K}\sum_{k=0}^{K-1}\left[\|(r_k, c_k) - (\bar{r}, \bar{c})\| - \mu_R\right]^2$$

circularity = mean_radial_distance/ variance_of_radial_distance

use k-mean to group blobs based on circularity

Kmeans(circularity of each pixel, user_inputed_k) = classification of each blob

2 Programming [80 points]

1. Color quantization with k-means. [40 points]

For this problem you will write code to quantize a color space by applying k-means clustering to the pixels in a given input image, and experiment with two different color spaces—RGB and HSV. Write Matlab(Python) functions as defined below. Save each function in a file called <function-name>.m(py) and submit all of them.

a. 5 points. Given an RGB image, quantize the 3-dimensional RGB space, and map each pixel in the input image to its nearest k-means center. That is, replace the RGB value at each pixel with its nearest cluster's average RGB value. Use the following form: **[outputImg, meanColors] = quantizeRGB(origImg, k)** where **origImg** and **outputImg** are **MxNx3** matrices of type **uint8**, **k** specifies the number of colors to quantize to, and **meanColors** is a **k x 3** array of the **k** centers. (You can use built-in k-means function in Matlab and Python)
b. 5 points. Given an RGB image, convert to HSV, and quantize the 1-dimensional Hue space. Map each pixel in the input image to its nearest quantized Hue value,

while keeping its Saturation and Value channels the same as the input. Convert the quantized output back to RGB color space. Use the following form: **[outputImg, meanHues] = quantizeHSV(origImg, k)** where **origImg** and **outputImg** are **MxNx3** matrices of type **uint8**, **k** specifies the number of 2 clusters, and **meanHues** is a **k x 1** vector of the hue centers. (You can use built-in k-means function in Matlab and Python)

c. 5 points. Write a function to compute the SSD error (sum of squared error) between the original RGB pixel values and the quantized values, with the following form: **function [error] = computeQuantizationError(origImg,quantizedImg)** where **origImg** and **quantizedImg** are both **RGB** images, and **error** is a scalar giving the total SSD error across the image.

d. 5 points. Given an image, compute and display two histograms of its hue values. Let the first histogram use equally-spaced bins (uniformly dividing up the hue values), and let the second histogram use bins defined by the k cluster center memberships (i.e., all pixels belonging to hue cluster i go to the i-th bin, for i=1,...k). Use the following form: **function [histEqual, histClustered] = getHueHists(im, k)** where **im** is an **MxNx3** matrix represeting an **RGB** image, and **histEqual** and **histClustered** are the two output histograms.

e. 5 points. Write a script **colorQuantizeMain.m(py)** that calls all the above functions appropriately using the provided image fish.jpg, and displays the results. Calculate the SSD error for the image quantized in both RGB and HSV space. Write down the SSD errors in your answer sheet. Illustrate the quantization with a lower and higher value of k. Be sure to convert an HSV image back to RGB before displaying with **imshow**. Label all plots clearly with titles.

f. 15 points. In your writeup, explain all 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?

The two histograms differ in bin size and thereby bin edges, as well as bin count. The histogram created by histEqual divides up the hue color space (0 to 1 in this case) into k equally spaced bins and assigns each pixel to a bin based on its hue value. While the histogram created by histClustered divided up the hule color space (0 to 1 in this case) into k bins based on kmeans clustered center membership and assign each pixel to a bin based on its hue value.

The results vary in color space due to the fact that in quantizeRGB the image is quantized to k specific RGB values, while for quantizeHSV only the hue channel is quantized to k specific values. Thus, leaving the saturation and value/brightness untouched. So, when the hue, saturation, and value channels are remerged and converted back into the RGB color space, same color space as the original image, significantly more RGB values are available.

As aforementioned, k determines the specific number of RGB values an image can take for quantizeRGB or the specific number of hues for quantizeHSV.

The results for each run depicted in Figures 3 through 5, show conclusively that the quantizeRGB has significantly more normalized SSD error (sum of square error) than that of quantizeHSV. This again is due to the fact that quantizeHSV allows the final image to take on more RGB values than k, which is the number of RGB values quantizeRGB is limited to. In addition, one can see that the normalized SSD error reduces as k increase, which make sense implicitly due to the fact that as k increases the quantized image can take on more and more RGB values and therefore have closer RGB values to the original image and therefore less error.

for k = 2
error_RGB = 108
error_HSV = 43
bin count for equally spaced bins is[100413. 62587.]
bin count for bins defined by the k cluster center membership[100124. 62876.]

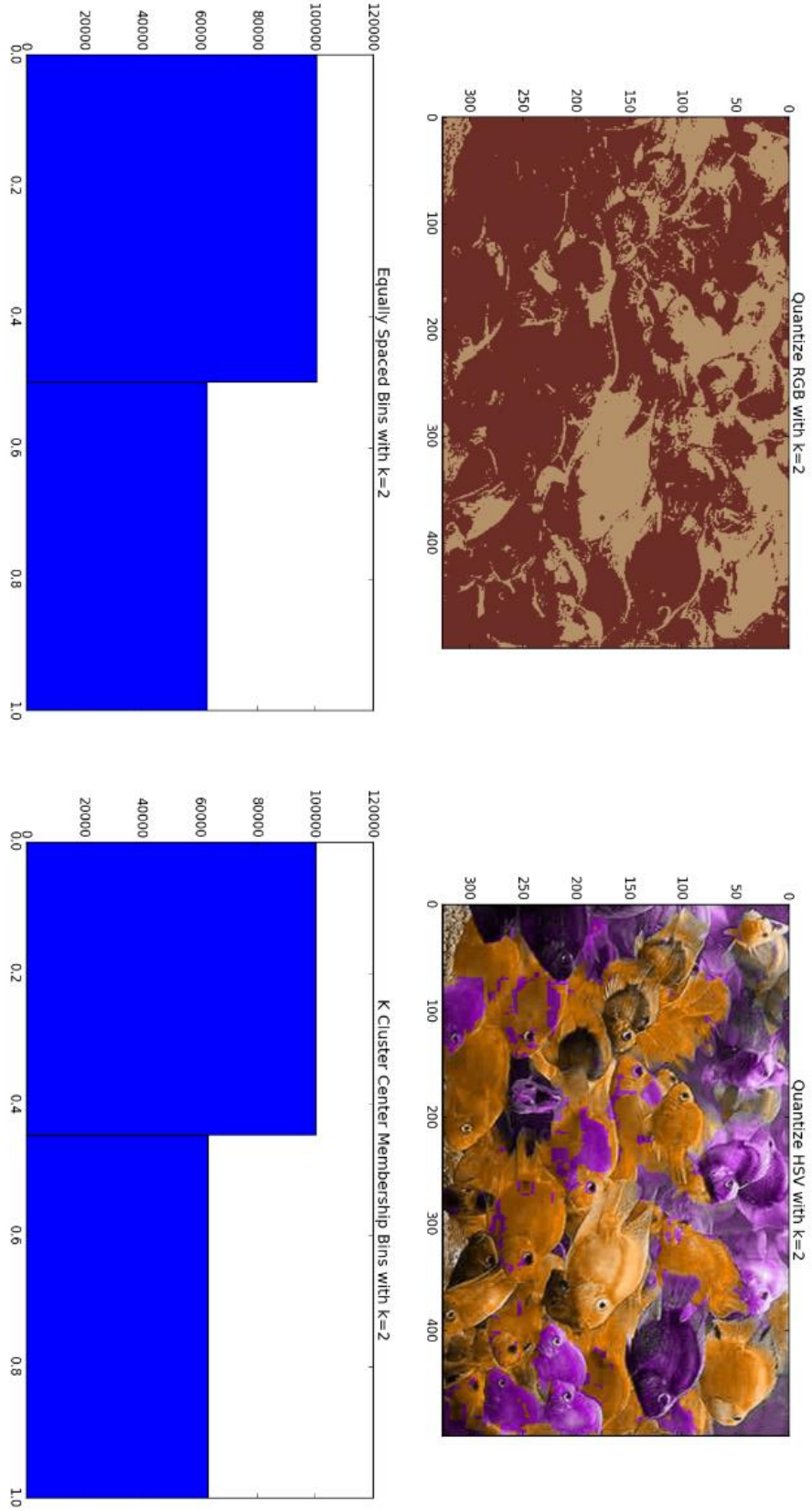


Figure 3: colorQuantizeMain with k = 2

for k = 4
error_RGB = 102
error_HSV = 32
bin count for equally spaced bins is[96138. 4275. 27912. 34675.]
bin count for bins defined by the k cluster center membership[65665. 34329. 35550. 27456.]

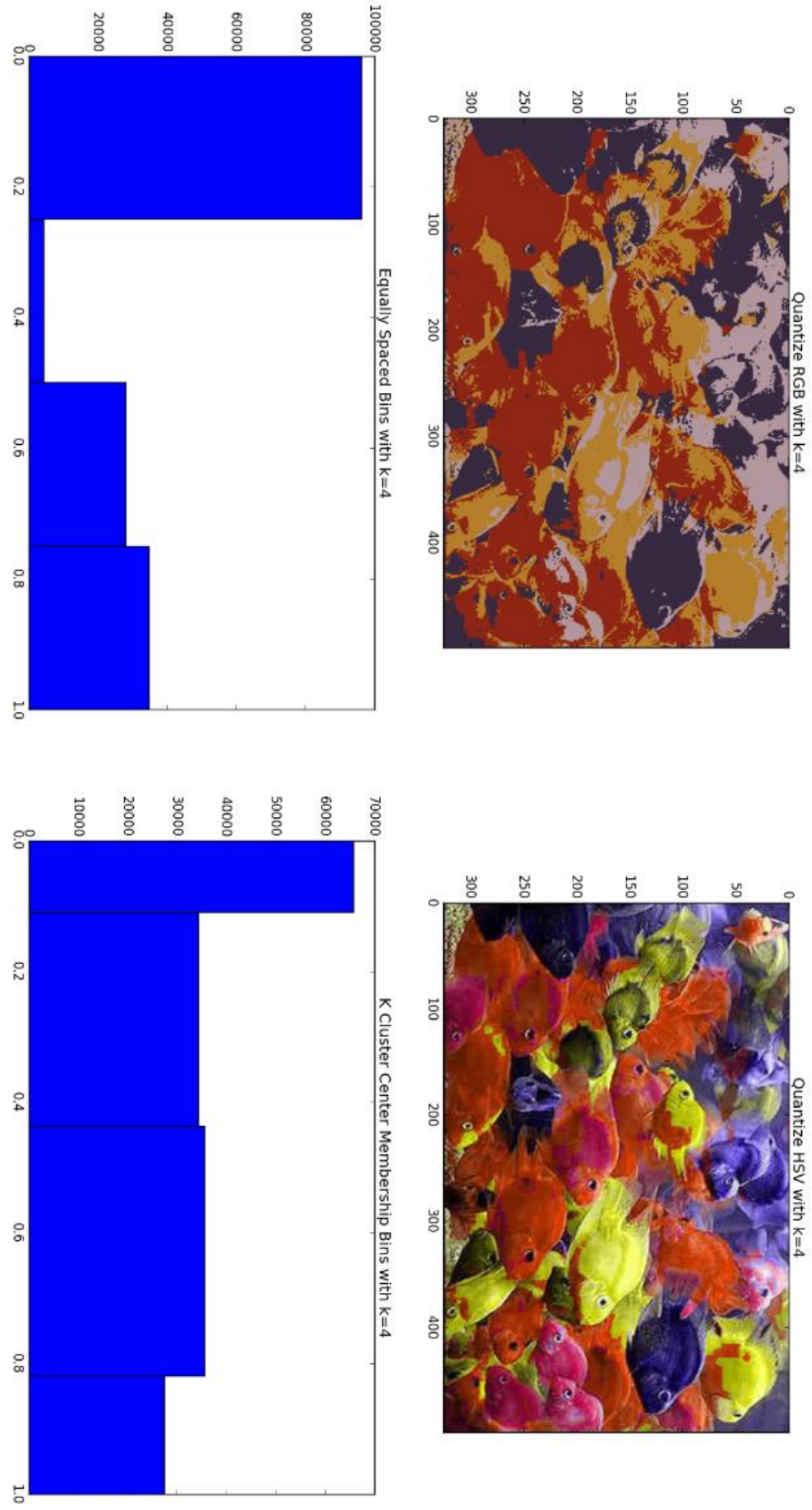


Figure 4: colorQuantizeMain with k = 4

for k = 6
error_RGB = 98
error_HSV = 30
bin count for equally spaced bins is[85888. 12342. 2183. 9452. 26737. 26398.]
bin count for bins defined by the k cluster center membership[51240. 29095. 17032. 3258. 35144. 27231.]

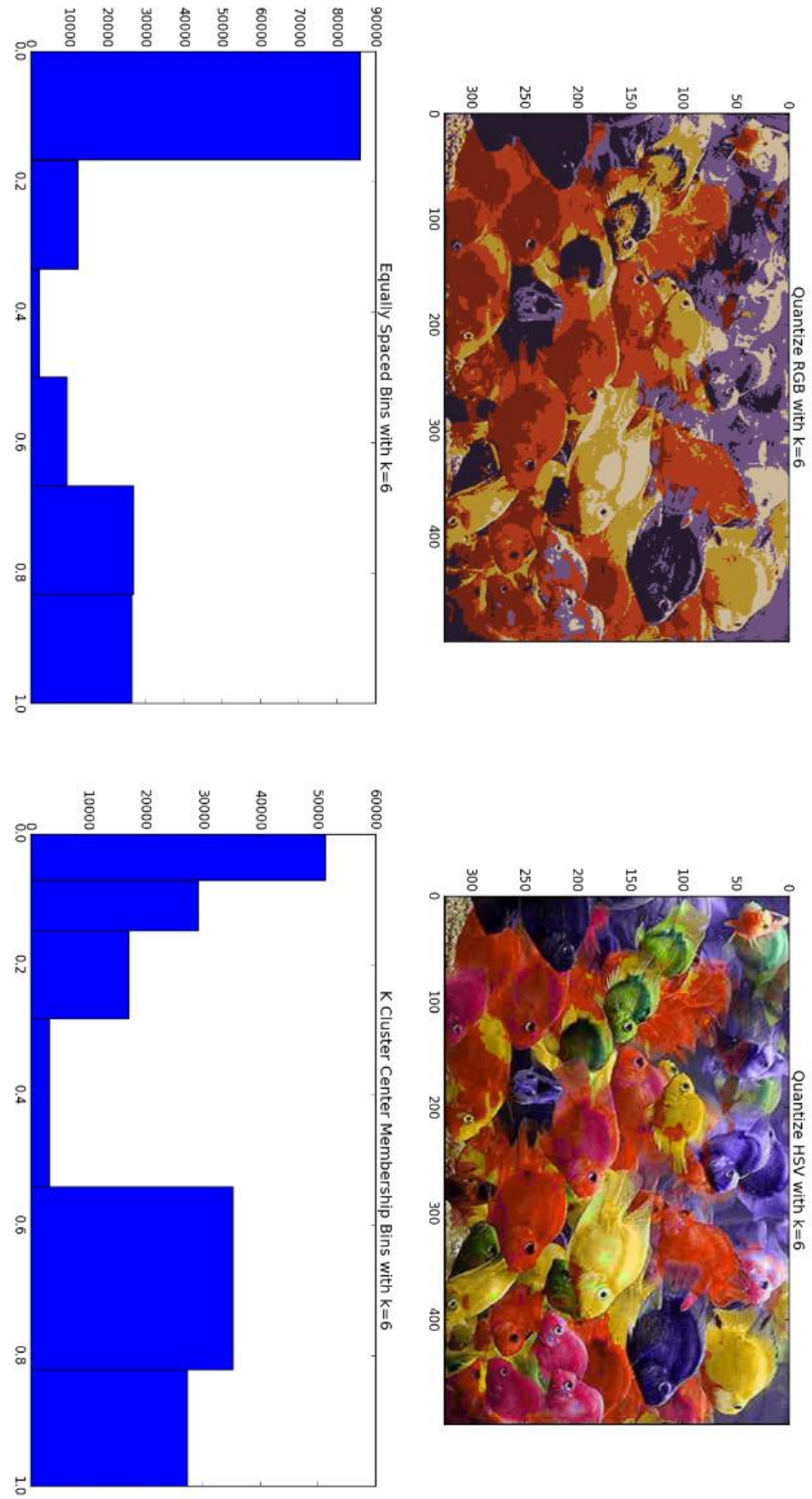


Figure 5: colorQuantizeMain with k = 6

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

   Implement a Hough Transform circle detector that takes an input image and a fixed radius, and returns the centers of any detected circles of about that size. Include a function with the following form: **[centers] = detectCircles(im, radius, useGradient)** where **im** is the input image, **radius** specifies the size of circle we are looking for, and **useGradient** is a flag that allows the user to optionally exploit the gradient direction measured at the edge points. The output centers is an **N x 2** matrix in which each row lists the (x,y) position of a detected circles' center. Save this function in a file called **detectCircles.m(py)** and submit it. Then experiment with the basic framework, and in your writeup analyze the following:

   a. 10 points. Explain your implementation in concise steps (English, not code).

For my implementation of the Hough Tranform, the input image is converted from an RGB image to a binary image. After which, Canny Edge detection with a sigma of 6 for jupiter.jpg and 3 for the other images was used in order to extract the main image edges. For each edge pixel, when useGradient is set to 0, a and b are calculated, using $a = x + rcos(\theta)$ and $a = y + rsin(\theta)$ for thetas sweeping from 0 to 2π, voting for that pixel [b, a] in the accumulator array or hough space. Thus creating a circle of the specified radius around each edge point. When useGradient = 1, the gradient direction at each edge point is used for theta in the equations for a and b given above. Since, one does know whether the gradient goes from light to dark or dark to light at each pixel, both theta and theta minus π are used. So, instead of sweeping from 0 to 2π making a circle at each edge point, just two points are voted for. After the votes have been cast by each edge pixel, creating the accumulator array or Hough space, the function takes all points above a specified magnitude in the accumulator array or Hough space, choosing which pixel will be circle centers. These circle centers are then outputted as a Nx2 matrix with the 0^{th} column being the x locations and the 1^{st} column being the y location. These (x, y) locations are then used as circle center to draw circles of a specified radius on the input image.

   b. 10 points. Demonstrate the function applied to the provided images jupiter.jpg and egg.jpg (and an image of your choosing if you like). Display the accumulator arrays obtained by setting **useGradient** to 0 and 1. In each case, display the images with detected circle(s), labeling the figure with the radius. For Matlab you can use **impixelinfo** to estimate the radius of interest manually. For Python, you can directly read the image pixel information in the plot.

Figures 6 through 15, show the Hough space and detected circles for each image for the specified radius setting useGradient to 0, False or that the gradient direction will not be used, and 1, True or that the gradient direction will be used.

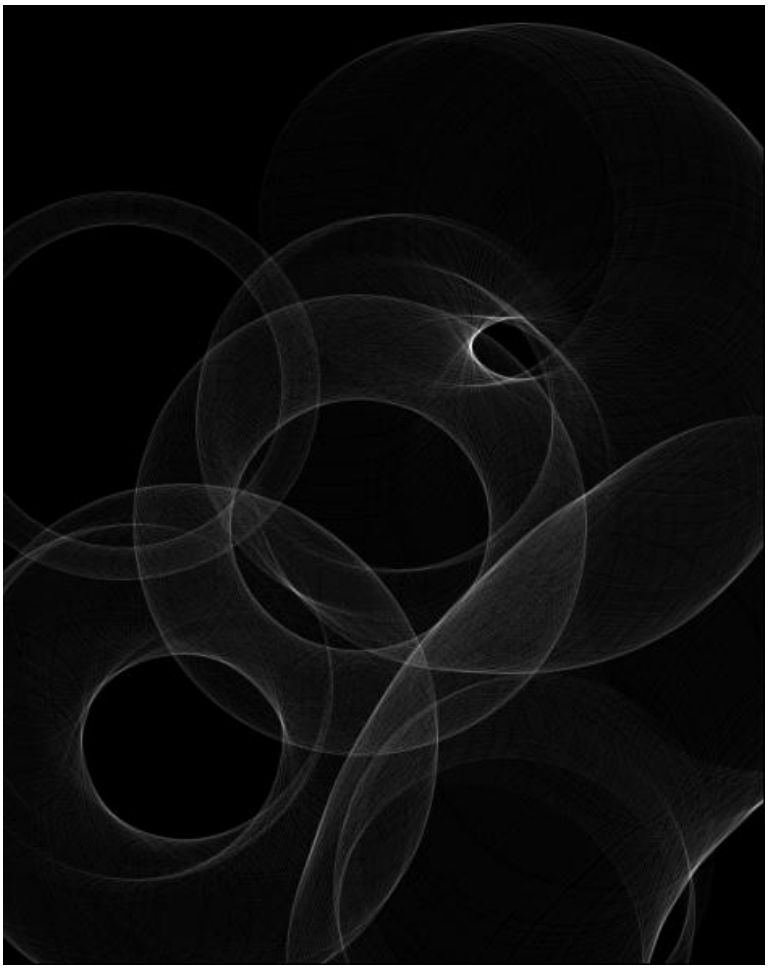


Figure 6: Jupiter.jpg Hough Space with radius = 110 and useGradient = 0

Figure 7: Jupiter.jpg detected circles with radius = 110 and useGradient = 0

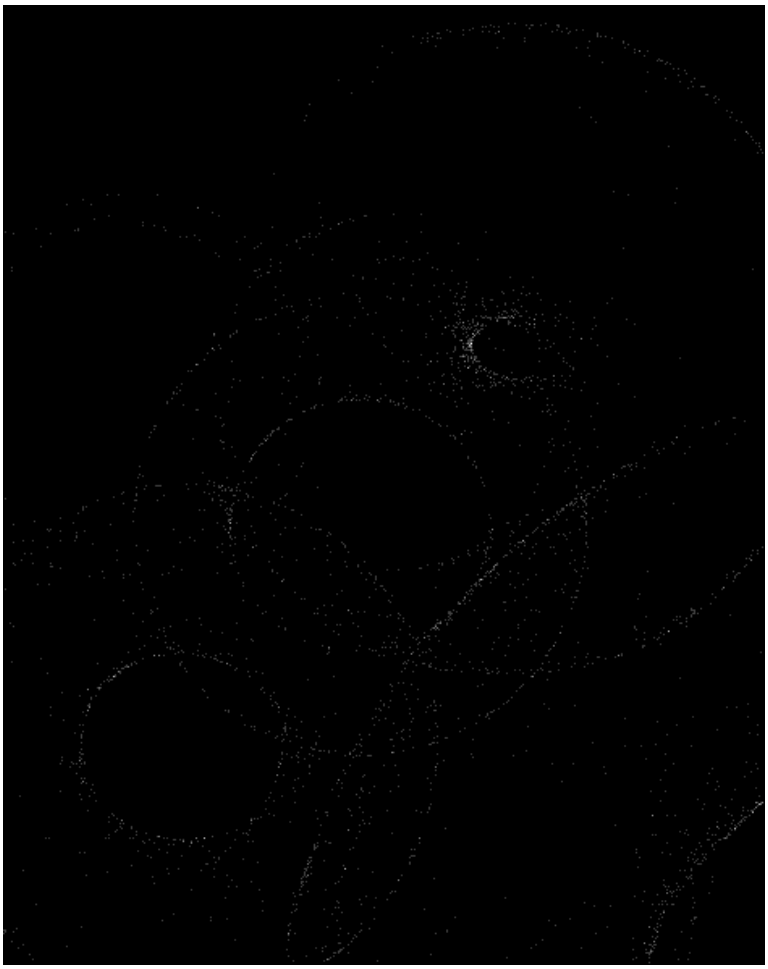
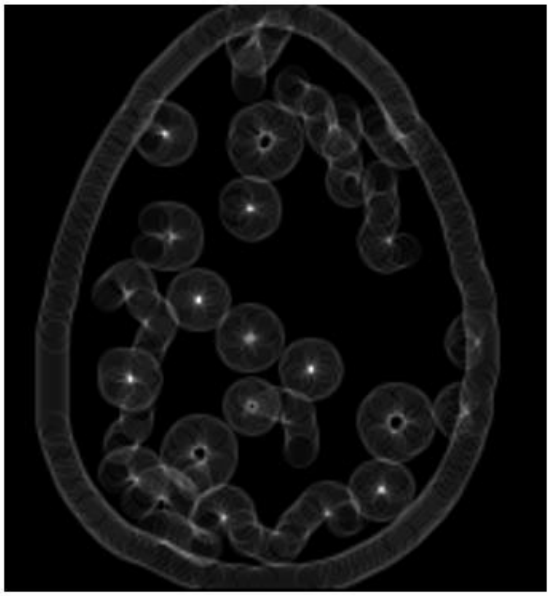


Figure 8: Jupiter.jpg Hough Space with radius = 110 and useGradient = 1

Figure 9: Jupiter.jpg detected circles with radius = 110 and useGradient = 1

Hough Space



Detected Circles

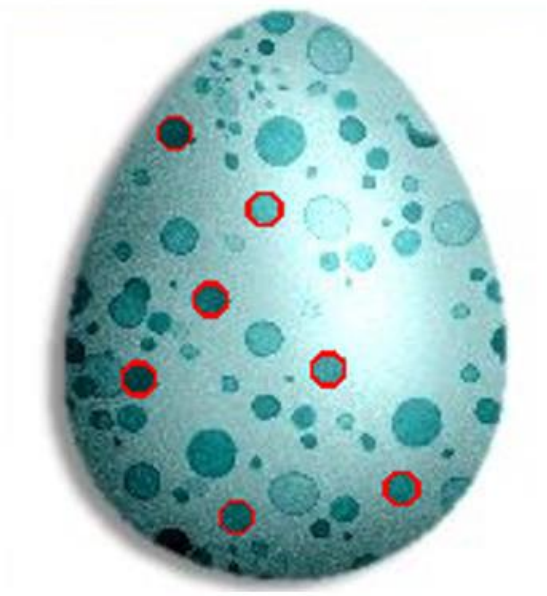


Figure 10: Egg.jpg Hough Space and detected circles with radius = 5 and useGradient = 0

Hough Space

Detected Circles

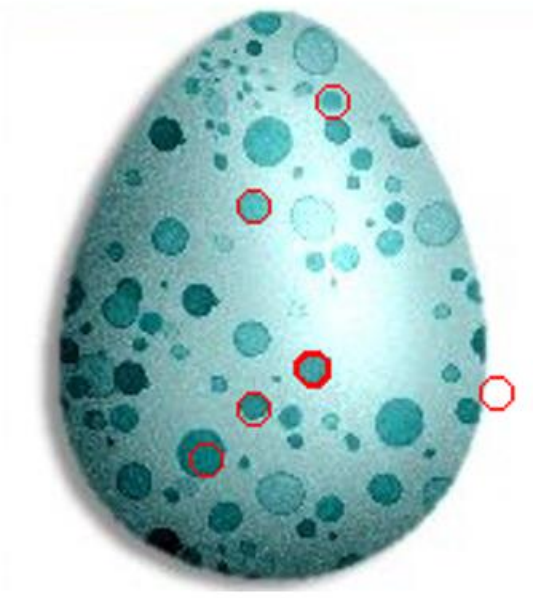


Figure 11: Egg.jpg Hough Space and detected circles with radius = 5 and useGradient = 1

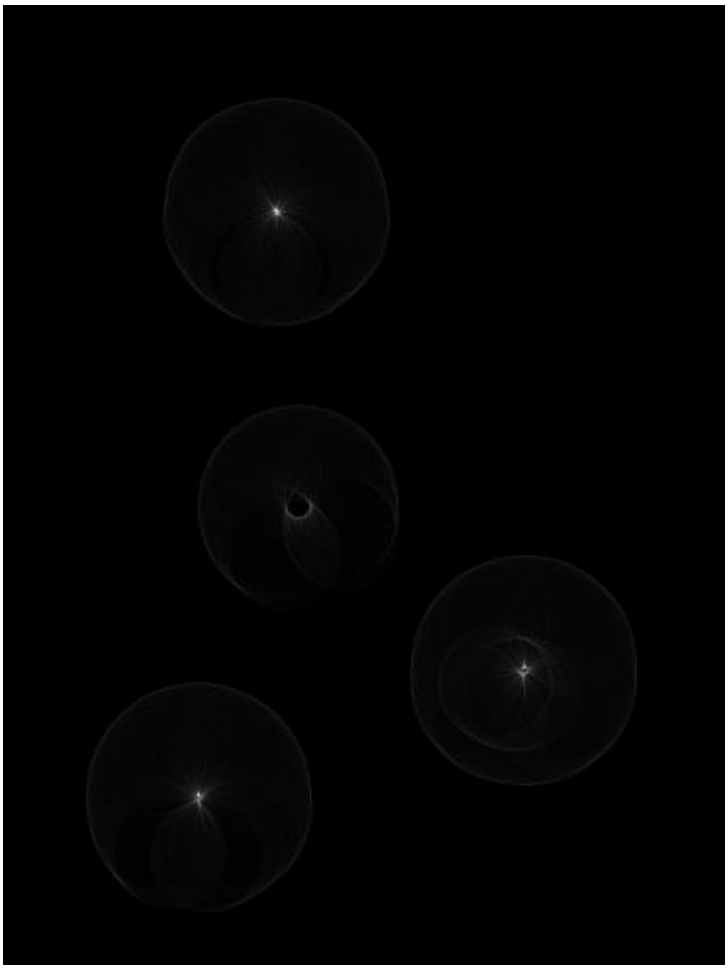


Figure 12: Coins.jpg Hough Space with radius = 33 and useGradient = 0

Figure 13: Coins.jpg detected circles with radius = 33 and useGradient = 0

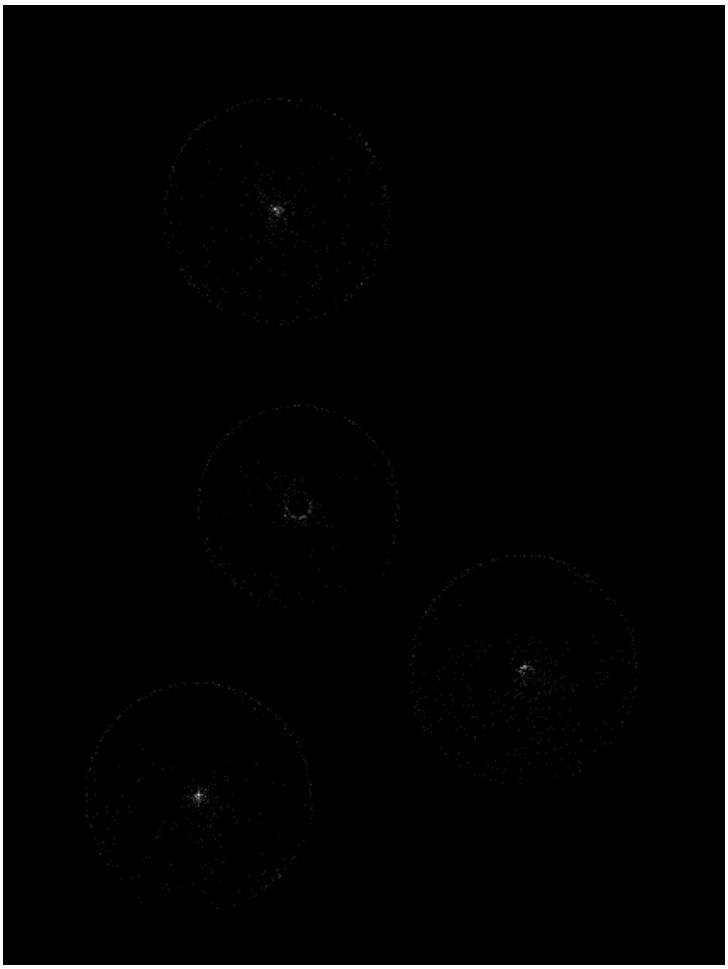


Figure 14: Coins.jpg Hough Space with radius = 33 and useGradient = 1

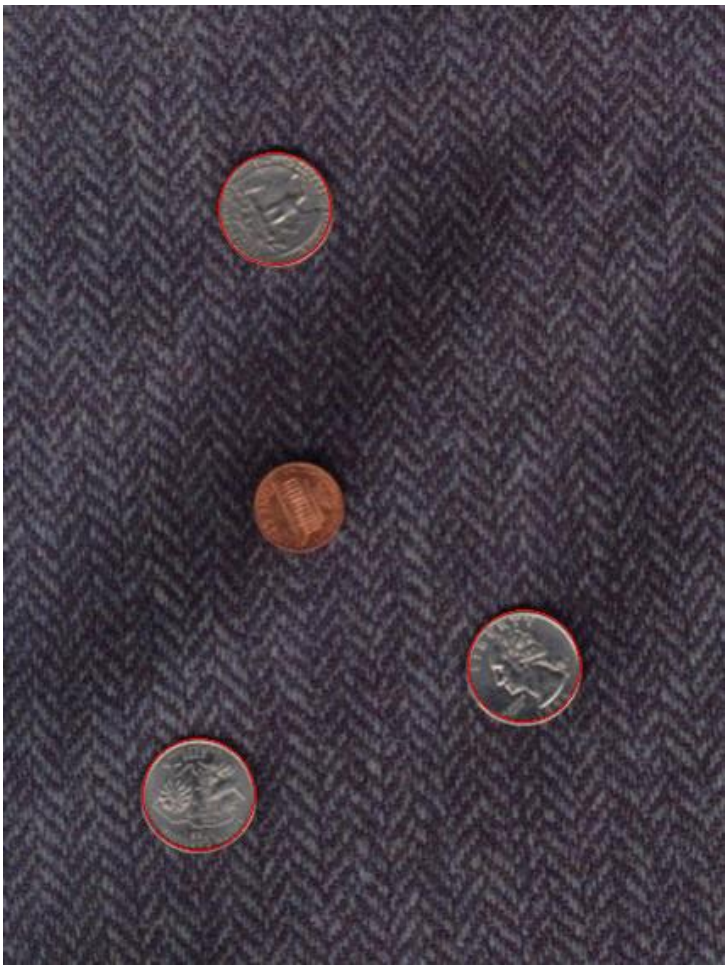


Figure 15: Coins.jpg detected circles with radius = 33 and useGradient = 1

c. 10 points. For one of the images, display and briefly comment on the Hough space accumulator array

Figures 16 and 17 show the accumulator array or Hough space for the jupiter.jpg image with useGradient set to both 0 and 1 respectively. Figure 16 shows circles made by each edge point, the overlapping circle create a maximum point which should be the center of Jupiter with the radius set to 110 pixels. While Figure 17, shows just two points for each edge point. Thus, the amount of points is drastically reduced. However, the maximum with a radius of 110 still appear to occur near or at center of Jupiter in the jupiter.jpg.

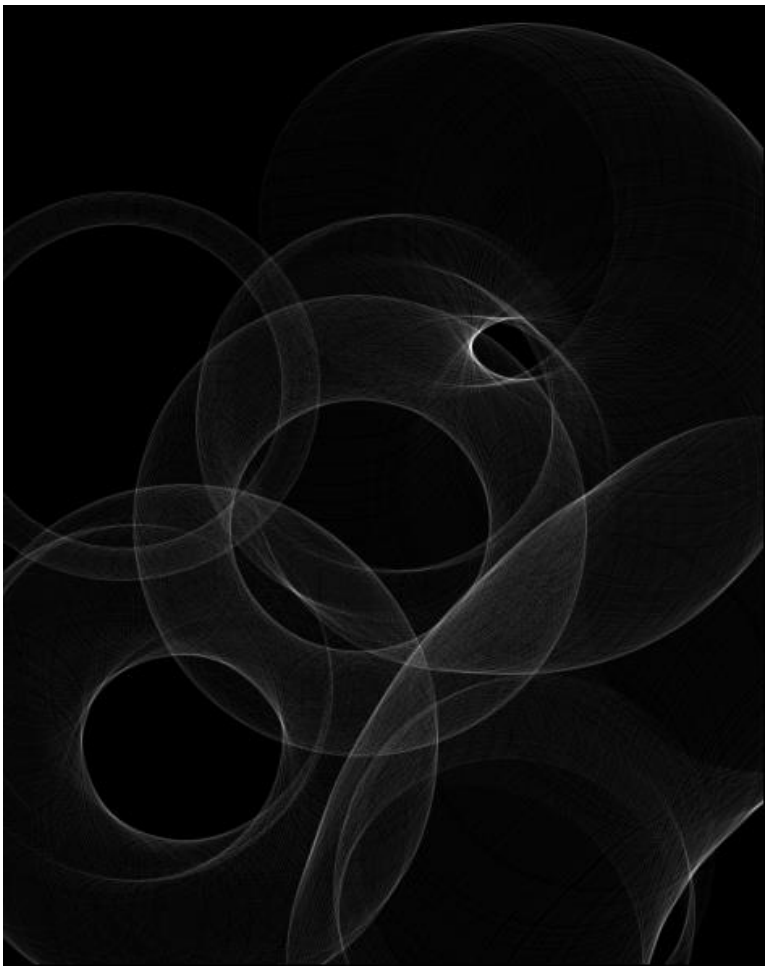


Figure 16: Jupiter.jpg Hough Space with radius = 110 and useGradient = 0

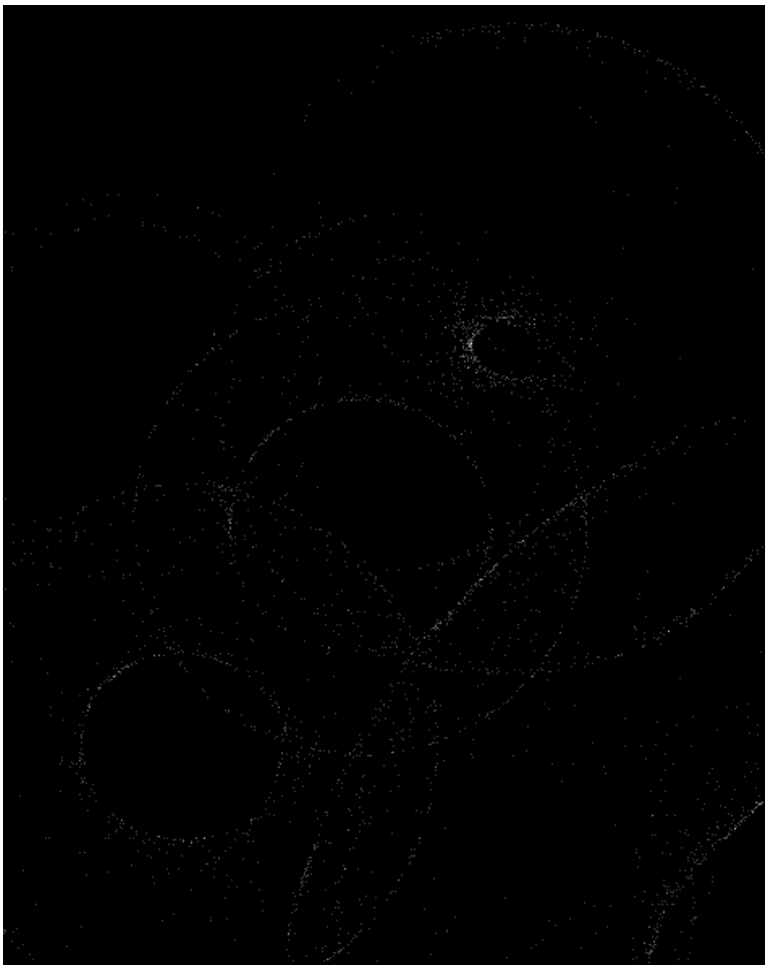


Figure 17: Jupiter.jpg Hough Space with radius = 110 and useGradient = 1

d. 5 points. Experiment with ways to determine how many circles are present by post-processing the accumulator array.

Figure 18 shows 6 different egg.jpg images with 5 pixel radius circle overlaid. Each of these images were created from the same Hough space. However, the values which results in a circle being created was varied from 40 to 90 percent of the max value in the Hough space.

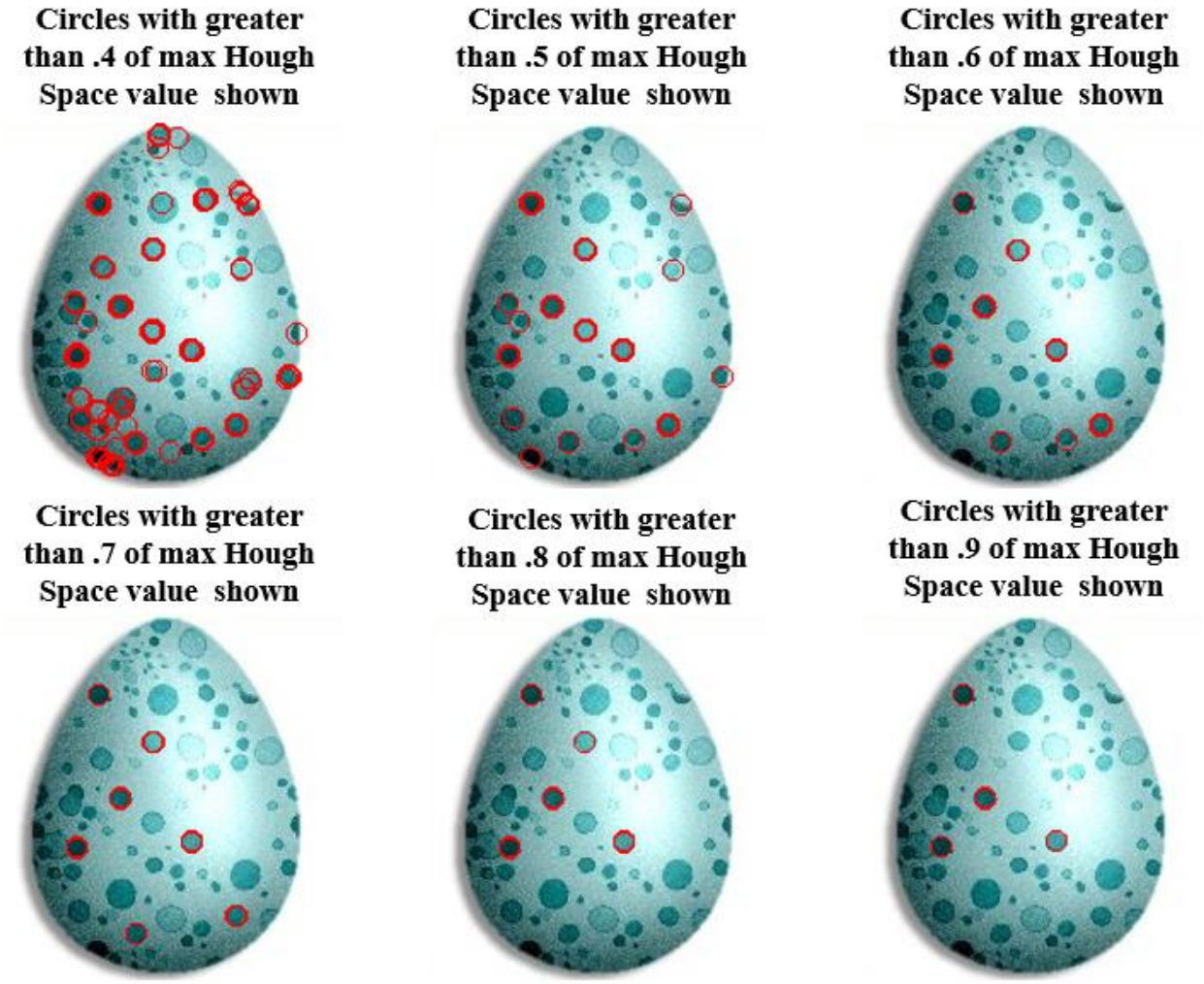


Figure 18: Egg.jpg detected circles with radius = 5, useGradient = 0, and varying intensites

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

Figures 19 through 22 show the results of doubling the bin size. Thereby, decreasing the Hough Space by half. Thus, decreasing the precision, which results in less precise placement in circles.

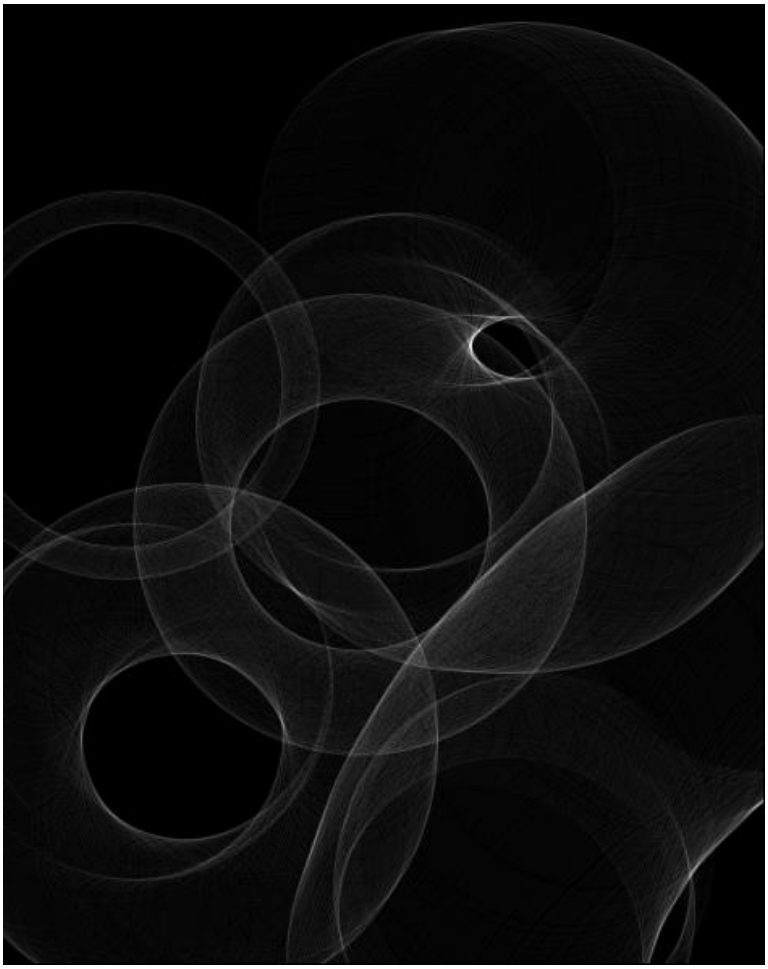


Figure 19: Jupiter.jpg original Hough Space with radius = 110 and useGradient = 0

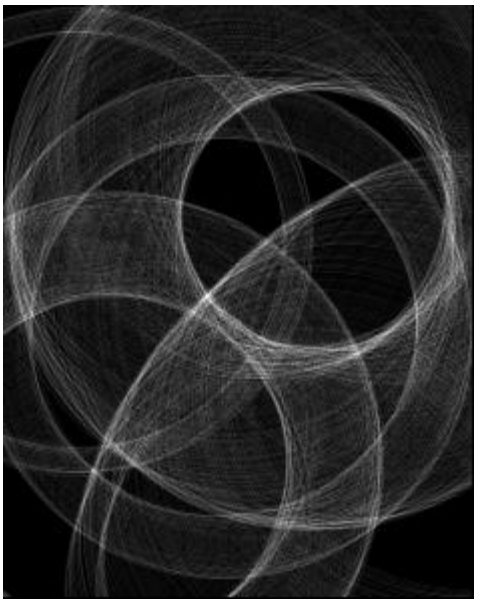


Figure 20: Jupiter.jpg quantize Hough Space with radius = 110 and useGradient = 0

Figure 21: Jupiter.jpg original detected circles with radius = 110 and useGradient = 0


LIFE

Figure 22: Jupiter.jpg quantize detected circles with radius = 110 and useGradient = 0

Note: I discussed this assignment with Murat Ambarkutuk, Evan Smith, Orson Lin, and Yi Tien