Collin Avidano

PS2 Answers

1. Suppose we form a texture description using textons built from a filter bank of multiple anisotropic derivatives of Gaussian filters at two scales and six orientations (as displayed below in Figure 1). Is the representation sensitive to orientation or is it invariant to orientation? Explain why.

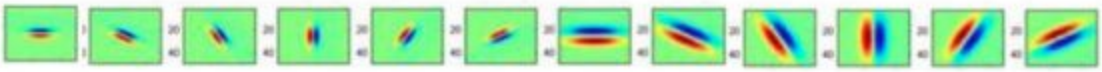
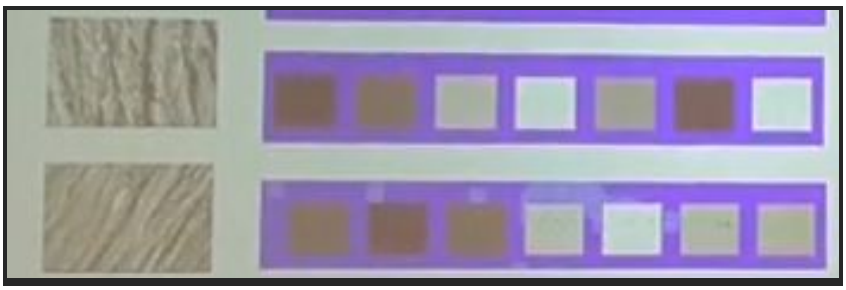


Figure 1: Filter bank

Well I would first note that the definition of anisotropic is quite literally:
(of an object or substance) having a physical property that has a different value when measured in different directions.

So yes the response of each of these filters to a particular image / texture will vary if that texture is rotated. Consider if we have an example similar to that given in class of the tree bark texture. The lines run vertical in one and the lines run at a -45 degree angle in the other view of the texture. In the -45 degree view the filters aligned with the edges will have a higher response. In the filter bak above that is the last two filters on the right and the and the smaller versions of those.



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.

My ml class also taught me this and I'm just sick of this miserable iterative existence.

Basically since k means seeks to minimize the sum of squared distances over all points not just one cluster it will end up dividing the points into the circle into two halves as if a line had been drawn through the origin of the circle. Depending on the initialization of the clusters themselves the line that divides the two clusters would rotate.

If you wanted to cluster a circle like this with kmeans you would probably need to do something like take the features as a function of their distance from the center of the image (eg transform to circular space and use the 1d axis you get to cluster)

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.

We can consider our model shapes as approximations of infinitely many individual data points in continuous space which are in the shape we are looking for. This will be helpful for linking back to explanations of clustering from class.

Much like with the example from class where we were trying to find the best line, in the parameter space we get a number of lines connecting each of our different edge points in the image space and the intersection among them is the explanatory parameters for our lines. If we tried to cluster the many explanatory points for these lines we may think a mean seeking clustering algo such as kmeans would always be a good choice to find the intersection or middle of each of these explanatory models. However in the presence of noise which will cause many outliers a mean seeking method like kmeans will be pulled away from these centers and find incorrect parameters.

We could consider something like normal cuts which would segment our image into fairly evenly sized regions however this is also not

what we want as this gives us a region or a set of points between which we should cut to form even regions not the point with the highest density of votes.

Then we consider mean shift clustering. Unlike k means mean shift clustering is mode or local maxima seeking not mean seeking which makes it much more resistant to noise than k means. Consider that when we were discretizing the space the point with the highest votes was the point with the highest density of points or the mode of points in a region. Mean shift is very good at finding this point as it constantly moves towards the greater density regions and is not affected by the shape of the data when trying to find this point. With this in mind it becomes fairly clear that mean shift is the most appropriate for finding the best parameters in a continuous 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.

We are trying to get a similarity of the shape of each of the blobs. I think the easiest way to do this is via the chamfer distance of the edges that define the border of these blobs.

Now I will discuss the algorithms for doing this.

```
def shape_grouping(blobs): # Blobs are separate binary images one for each blob

    blobs_edges = im height by im width by blobs size zeroed array
    For i, blob in blobs:
        Blobs_edges[i] = canny(blob) # for each blob we want a 1 pixel border for the edge to capture its shape
    Chamfer_distances = blobs size by blobs size array (similarity matrix)
    For i, template_blob in blobs_edges:
        for j, target_blob in target_blobs_edges:
            target_transform = distance_transform(target_blob) # get distance transform map for target blob
            Chamfer_distances[i,j] = Chamfer_distance(template_blob, target_transform) # find min weight
```

```
matching of our template blob to our target transform map which is the similarity of blob i to j
     # now we have a similarity graph where the edge weights are the chamfer distance. We dont want something like kmeans where we would group together all things of high similarity we want to group together the things that HAVE high similarity with each other and cut off things that have low similarity.
     # We use normalized cuts to achieve this cut other graph segmentation approaches can achieve this
     blob_groupings = normalized_cuts(chamfer_distance) # should return what vertices (blobs in the graph) are connected after cuts as an array for each grouping
     return blob_groupings
```

2.1 B

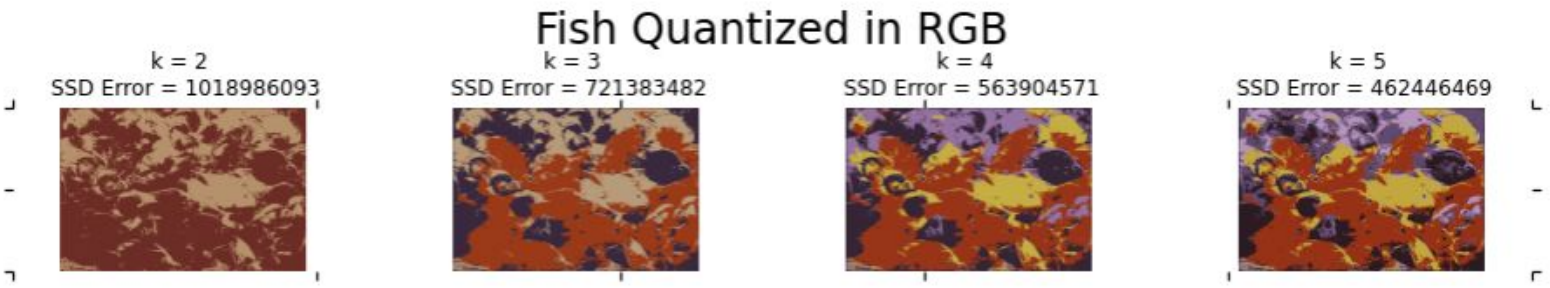
1: Experimentation Graphs

A. Fish quantized in RGB space

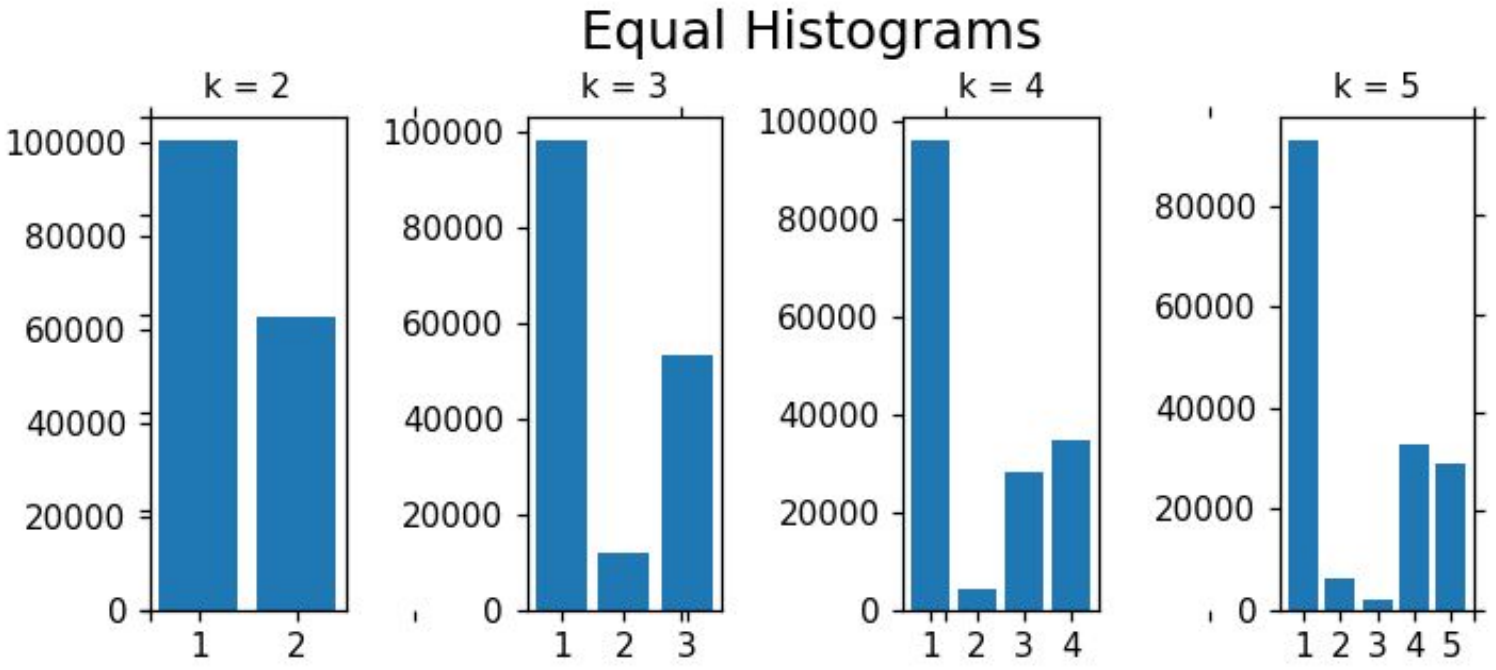
B. Fish quantized in HSV space

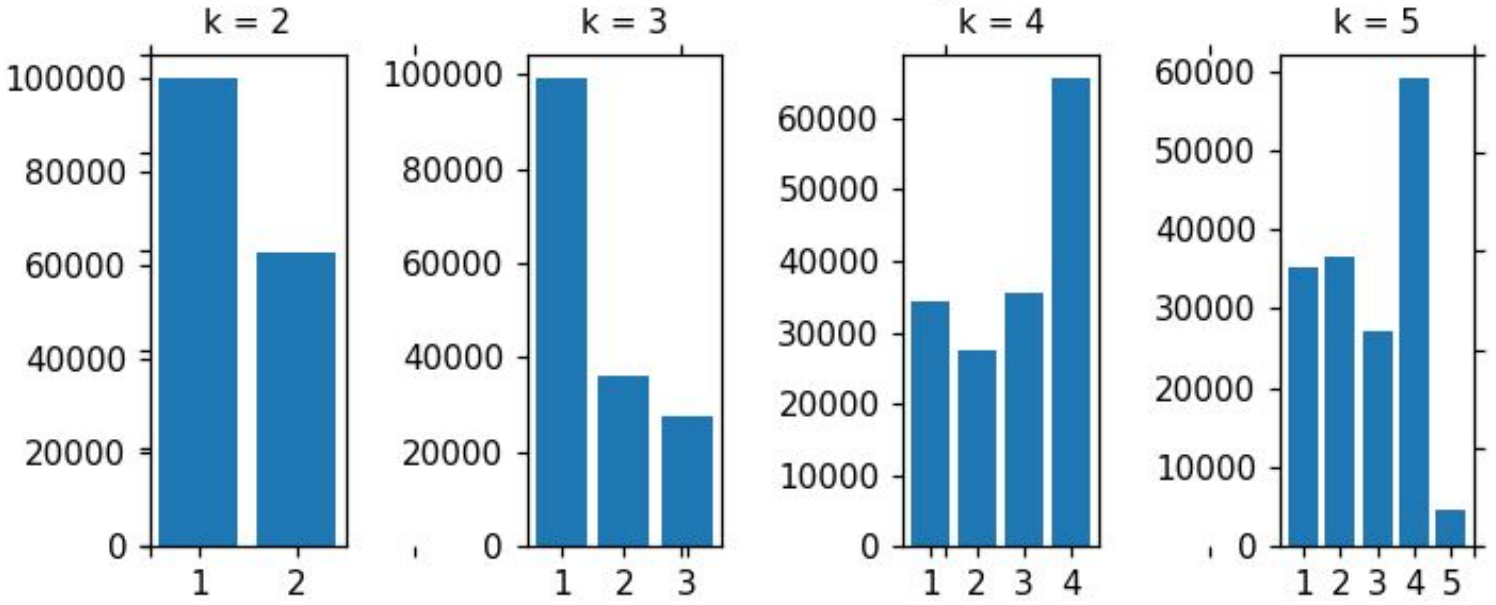
C. SSD error between original RGB image and RGB quantized image

D. SSD error between original RGB image and HSV quantized image



E. Equal bin size division of hue space histogram
F. KMeans cluster hue space histogram





2: Results Discussion

How do the two forms of histogram differ?

While both histograms are computed in the hue color space one is using bins of equal width across the whole space vs the other which is using the membership of hue values (1d array) to clusters which minimize the sum of distances squared in this 1d space. This has the effect of reducing the difference in number of pixels assigned to a particular bin vs another. In other words the number of assignments to any particular bin is more even overall and the bins themselves more closely fit the region of the hue spectrum with more pixels

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

By running k means with a value of n against all three components of the rgb space the output image only has n colors overall. This leads to the quite literally flat shaded regions of the image as no variability in these colors exists.

However with the HSV quantization we only clustered the Hue value. The saturation and value components were untouched and therefore can have a much wider range of colors. However since saturation is the “strength” of a particular color (hue sets a red value but that can be anywhere from a strong red to a fading grayish red) and value is the brightness of that color (value acts a lot like specular light, how much is reflected back gives you a range between black and white with the color in between), the colors we perceive are generally only brighter or dimmer, stronger or more faded versions of the hue value. So in the k = 2 case we only kept an orange and a purple hue value as the averages of our hue space and the others are the results of higher or lower saturation and value.

The value of k?

As noted before, increasing the value of K allows for more colors to be represented in both rgb and the hue quantization. As k grows back towards the number of pixels in the original image we would recover more and more of the original color. Since we only clustered and thus averaged the results of the hue field for the hue quantization this quantization would start looking like the original with a much lower k value than the rgb quantization.

Across different runs?

Since the convergence of k means is affected by the initial conditions of the clusters themselves (which are randomly assigned) it is very likely that different runs would result in various quanization outputs. Which would be most perceptively different at a lower k value.

2.2

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

My implementation follows the fairly standard method of object detection using hough space transforms. First a canny edge detector is run over the original image to harder edges than those that might arise from just texture. The result of a canny edge detection should be all 1 pixel wide binary valued pixels (basically 255 or 0, an edge or not).

At this point I check if the use_gradient argument was passed and if it was I find the directional x and y gradients by convolving the original image with the x and y prewitt filters. Then for each pixel location I take the the arctan of the value at that location in the y directional gradient over that same location in the x direction gradient matrix. This gives me a theta that describes the direction of the gradient at each pixel.

I initialize an accumulator array for the votes that is the same size as the image. As we will see this is because my parameter space and cartesian space are actually essentially the same.

Then I iterate over every pixel in the canny output. If a particular pixel x,y is an edge (eg non zero), I cast a vote for all the locations in the accumulator array that are on a circle of radius r with its origin at x,y. More concretely I iterate from 0 to 2pi in discrete steps and calculate the result of a = x + Rcos(theta) and b = y + Rsin(theta) at each step then increment the index at a,b by 1. The equations a = x + Rcos(theta) and b = y + Rsin(theta) are simply the parametric equations for a circle of radius R at x and y.

In the case that use_gradient is set to true rather than iterating from from 0 to 2pi and casting a vote at many small increments we only case 2 vote at 2 a,b locations by using the same equations as before, a = x + Rcos(theta) and b = y + Rsin(theta), taking the theta value from the gradient direction array initialized earlier. The reason I had to use two here was because of a shortcoming of arctan so I cast another vote for the original angle - math.pi.

The approach which sweeps from 0 to 2pi is effectively an exhaustive search whereas here we are using the gradient information of the original image to inform where the center of the circle might be.

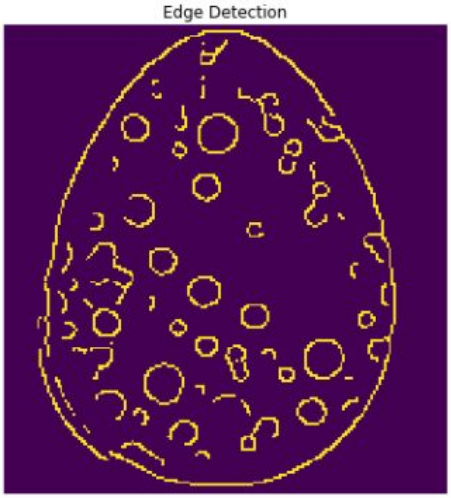
As you may be able to see the accumulator array is basically just counting where the most circles of radius R would intersect. Because this location was determined as the intersection of many different circles placed at edge locations in hough space this location ends up being the best explanation for these edges back in the cartesian space of our image. That is the origin of the circle we are trying to find in our image cartesian space is at the intersection of circles drawn at edges in our hough space. also and a,b are 1,1 with our original images x,y coordinates.

Although the threshold can be arbitrary for how many votes are needed at a location to return it as a result the way I decided was by setting the threshold of votes needed to be a percent of the max votes observed in the entire accumulator array.

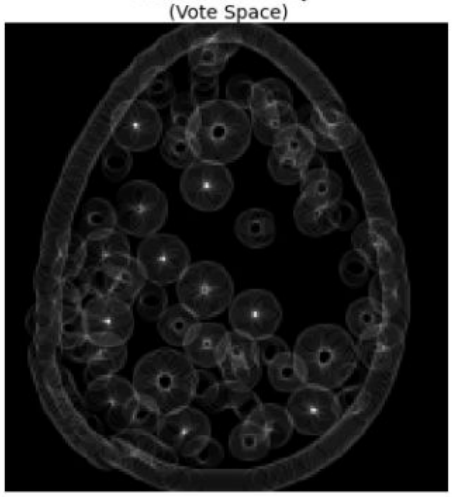
2. Demonstrate the function applied to the provided images jupiter.jpg and egg.jpg. Display the accumulator arrays obtained by setting use_gradient to True and False. In each case, display the images with detected circle(s), labeling the figure with the radius.

Egg Circle Detection
sigma = 2
use_gradient = False
radius = 5
Number of results: 8

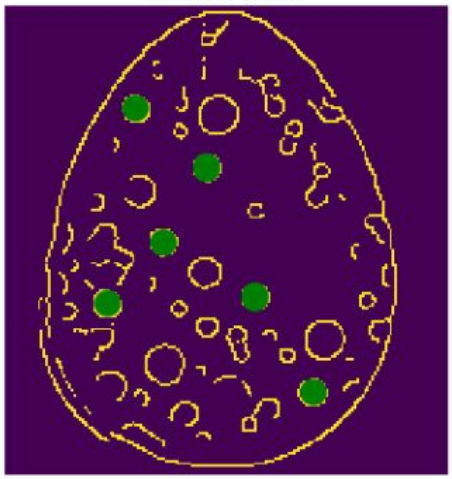
Edge Detection



Accumulator Array
(Vote Space)

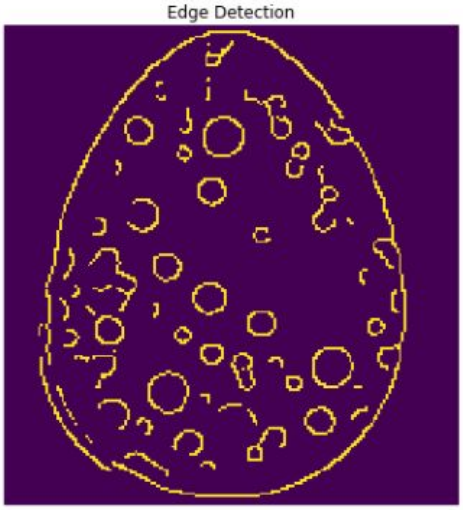


Detected Circles

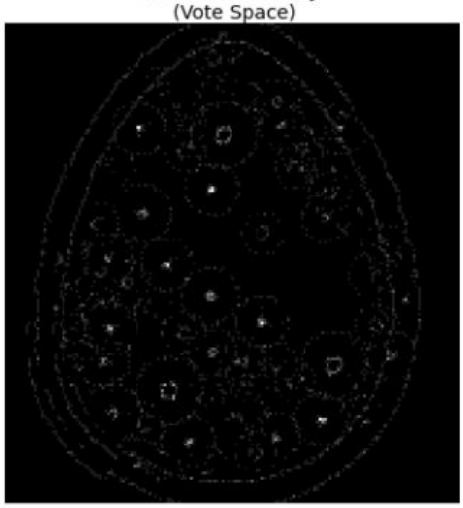


Egg Circle Detection
sigma = 2
use_gradient = True
radius = 5
Number of results: 9

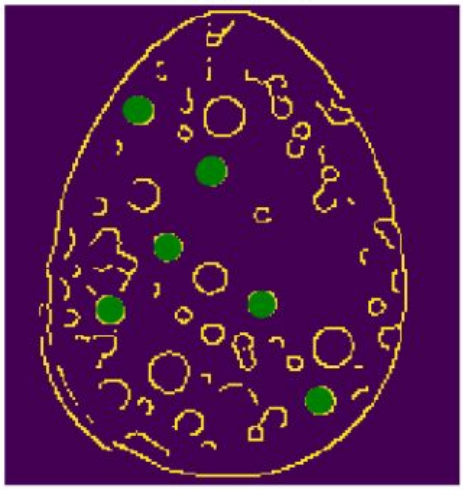
Edge Detection



Accumulator Array
(Vote Space)

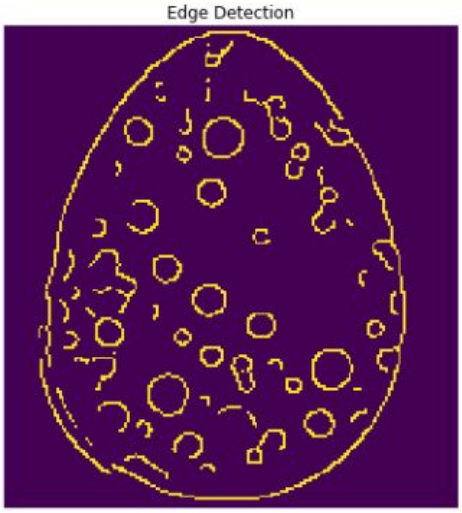


Detected Circles

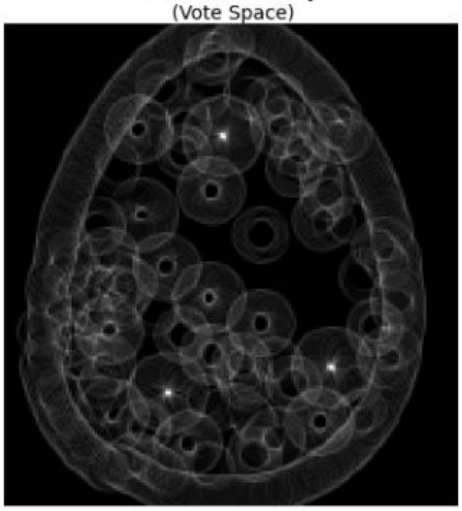


Egg Circle Detection
sigma = 2
use_gradient = False
radius = 8
Number of results: 6

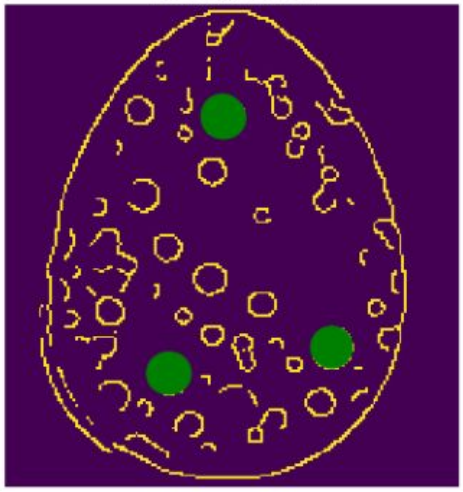
Edge Detection



Accumulator Array
(Vote Space)

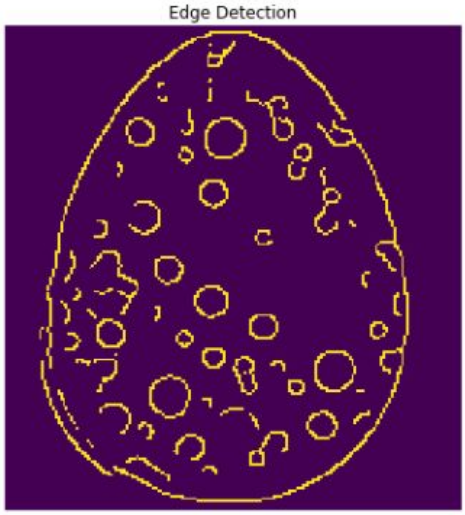


Detected Circles

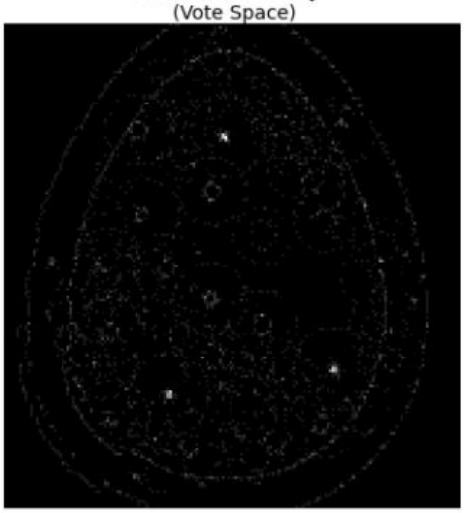


Egg Circle Detection
sigma = 2
use_gradient = True
radius = 8
Number of results: 5

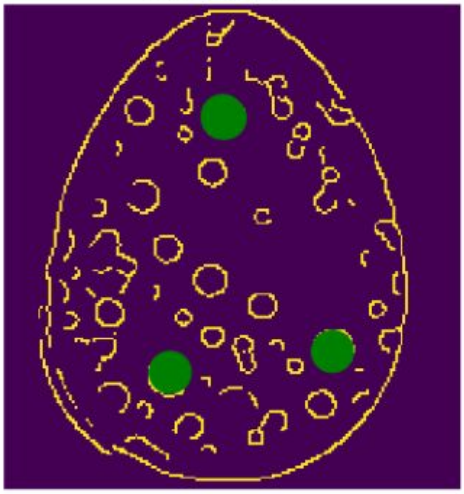
Edge Detection



Accumulator Array
(Vote Space)

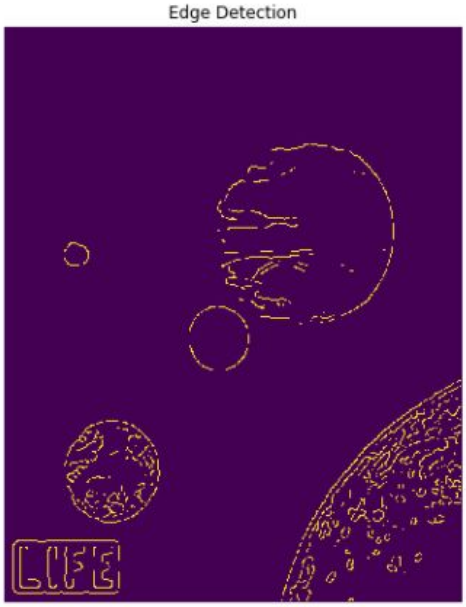


Detected Circles

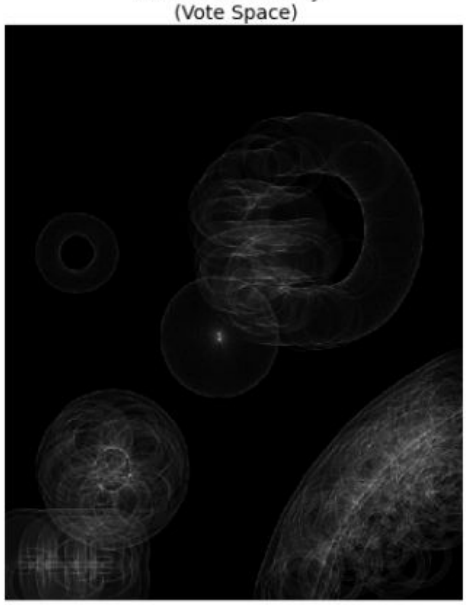


Jupiter Circle Detection
sigma = 2
use_gradient = False
radius = 30
Number of results: 3

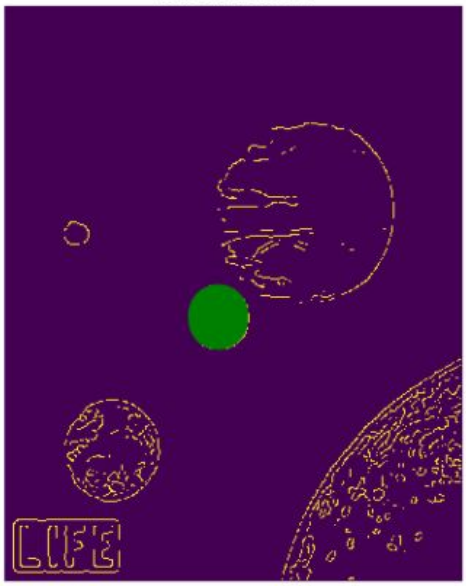
Edge Detection



Accumulator Array
(Vote Space)

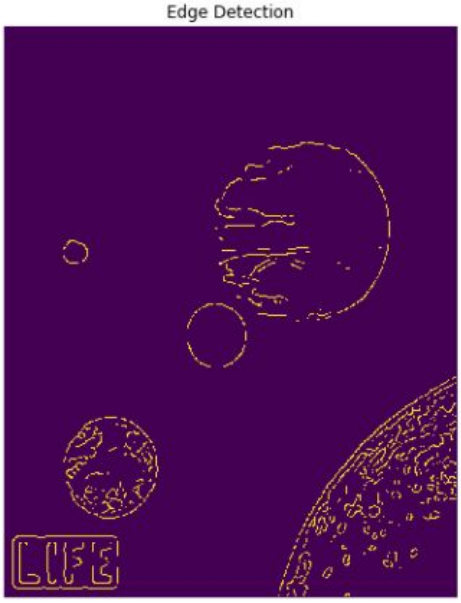


Detected Circles

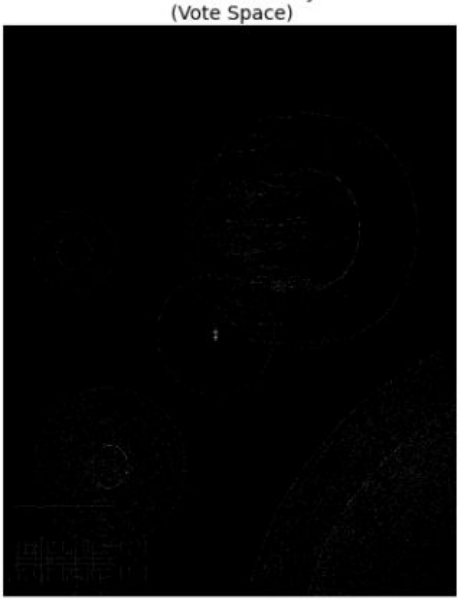


Jupiter Circle Detection
sigma = 2
use_gradient = True
radius = 30
Number of results: 2

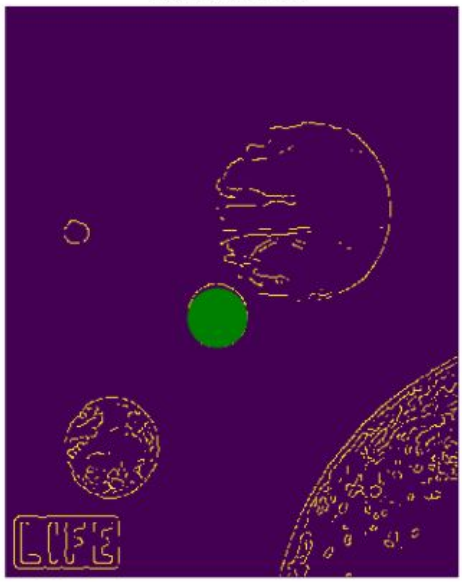
Edge Detection



Accumulator Array
(Vote Space)

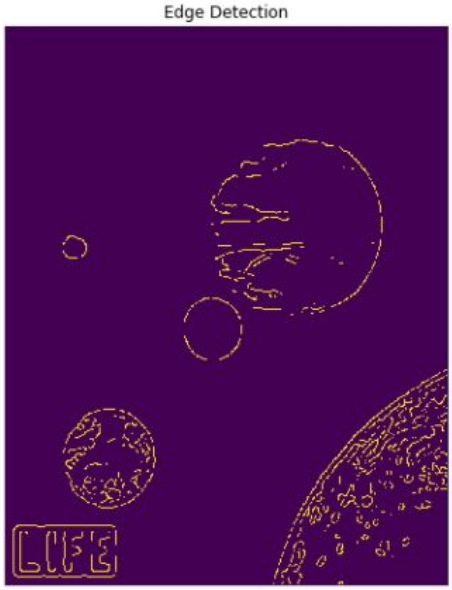


Detected Circles

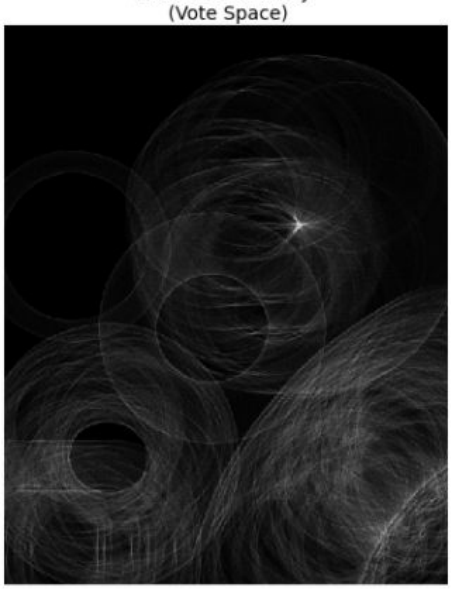


Jupiter Circle Detection
sigma = 2
use_gradient = False
radius = 90
Number of results: 24

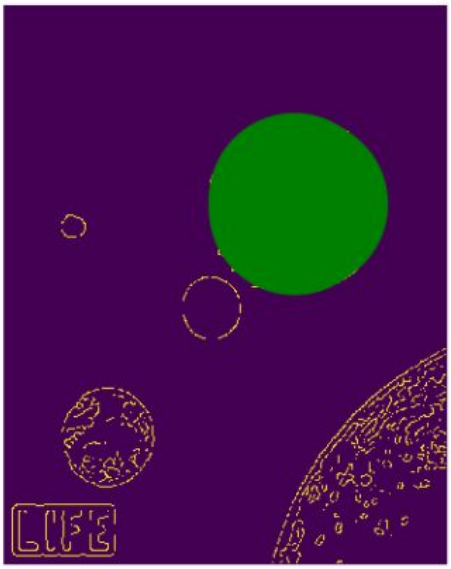
Edge Detection



Accumulator Array
(Vote Space)

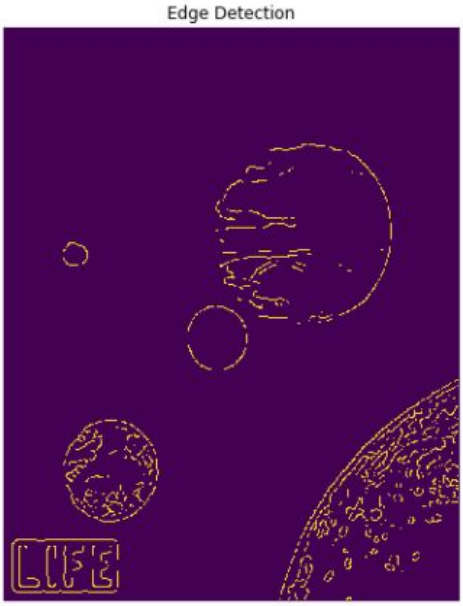


Detected Circles

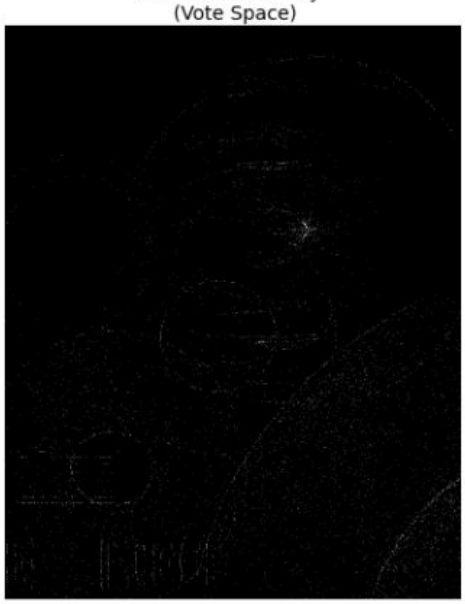


Jupiter Circle Detection
sigma = 2
use_gradient = True
radius = 90
Number of results: 3

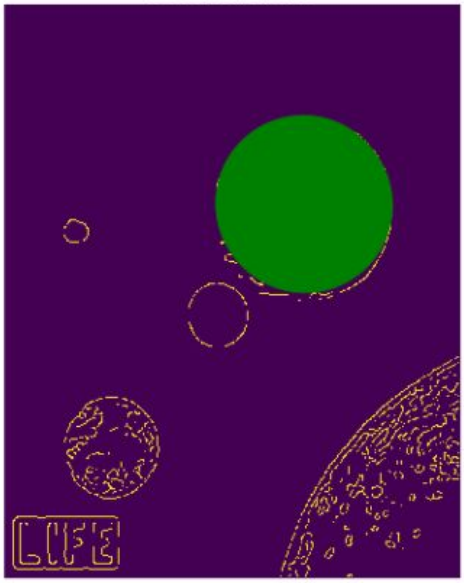
Edge Detection



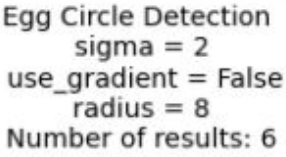
Accumulator Array
(Vote Space)

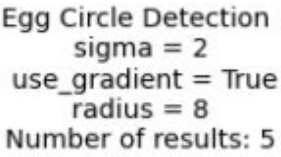


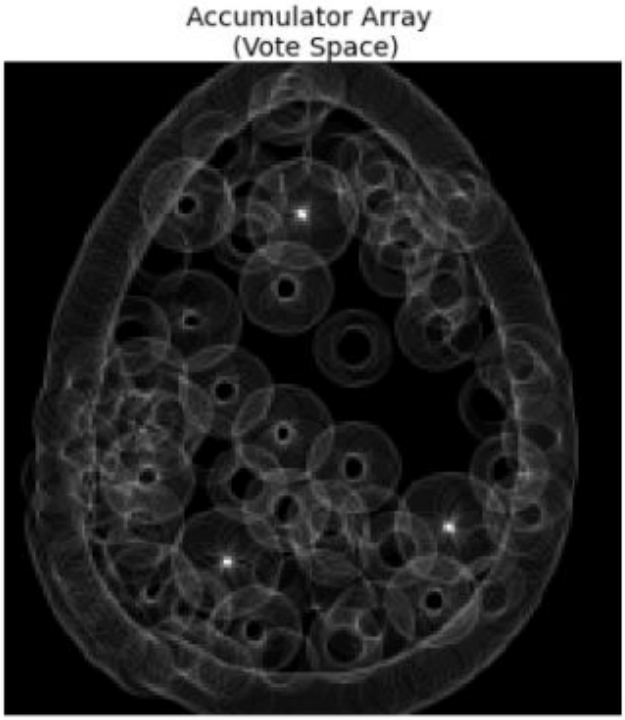
Detected Circles

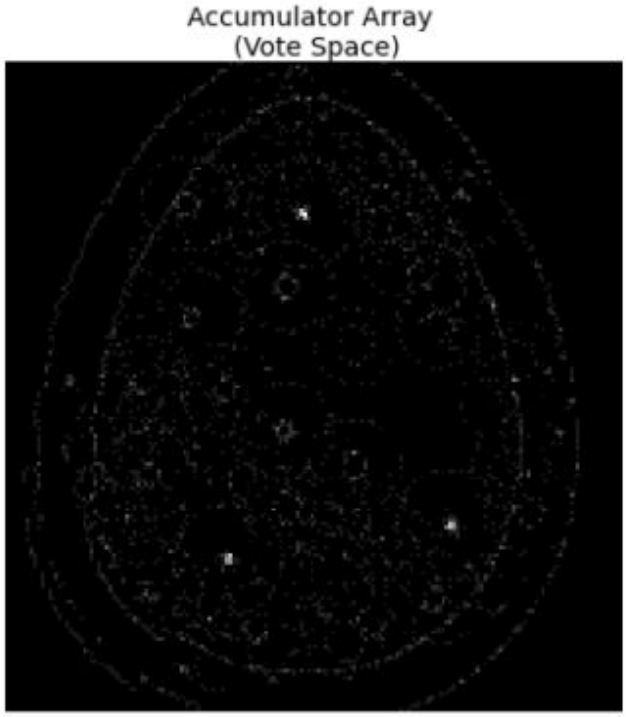


3. For one of the images, display and briefly comment on the Hough space accumulator array.









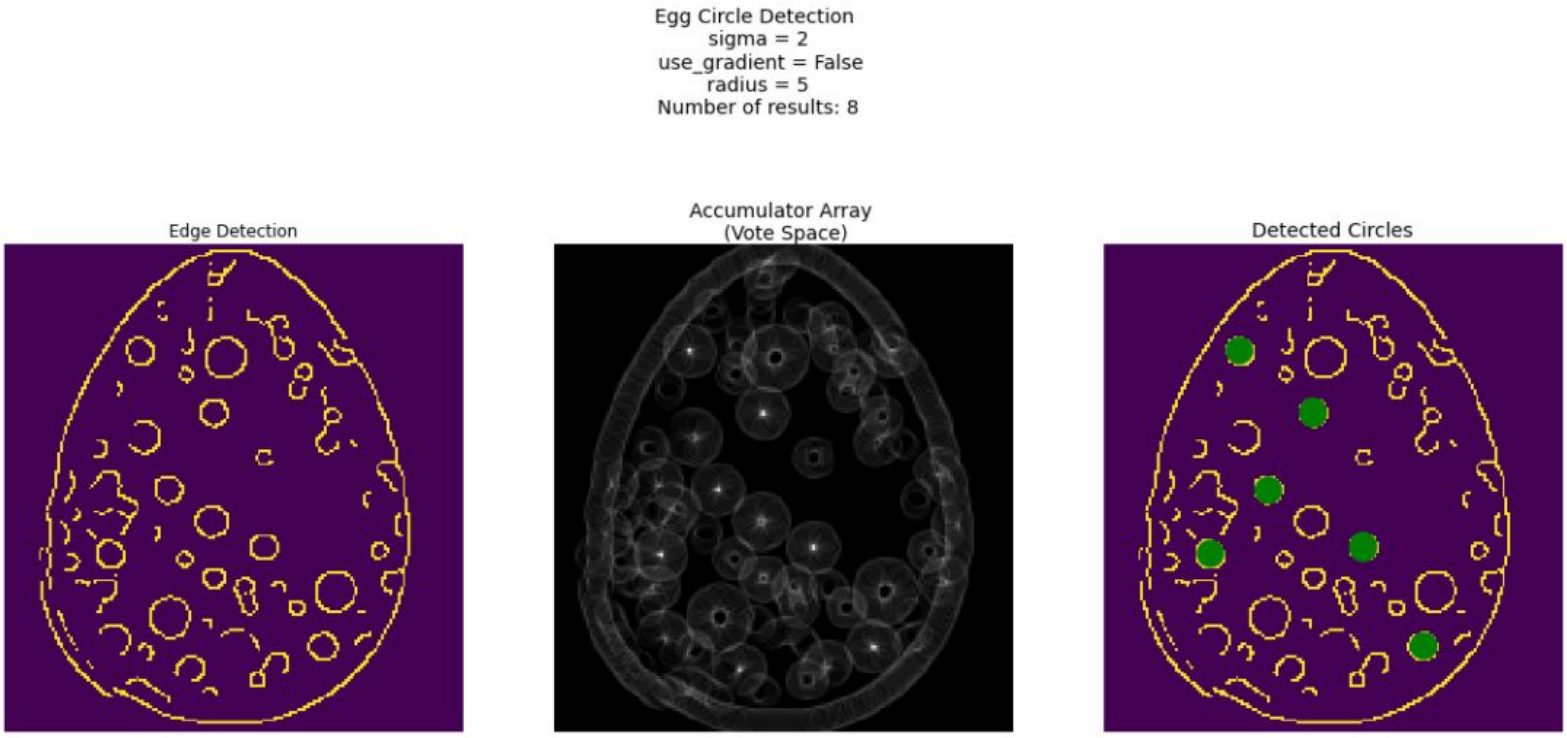
Here we see a Hough accumulator array from the egg tests at a radius of 8. Note when not use_gradient is not set we see the model shape (a circle) trace out around every single edge point in our original edge image. When use_gradient is set we no longer have circles as all accumulator points from zero to 2pi are not voted for, only those in one direction.

If we look at where many circles intersect those are of course much brighter as they have many more votes and will more than likely be returned as our final result. As you may note, the region of points ,not just a single point is bright. This is what leads to the many additional results we see.

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

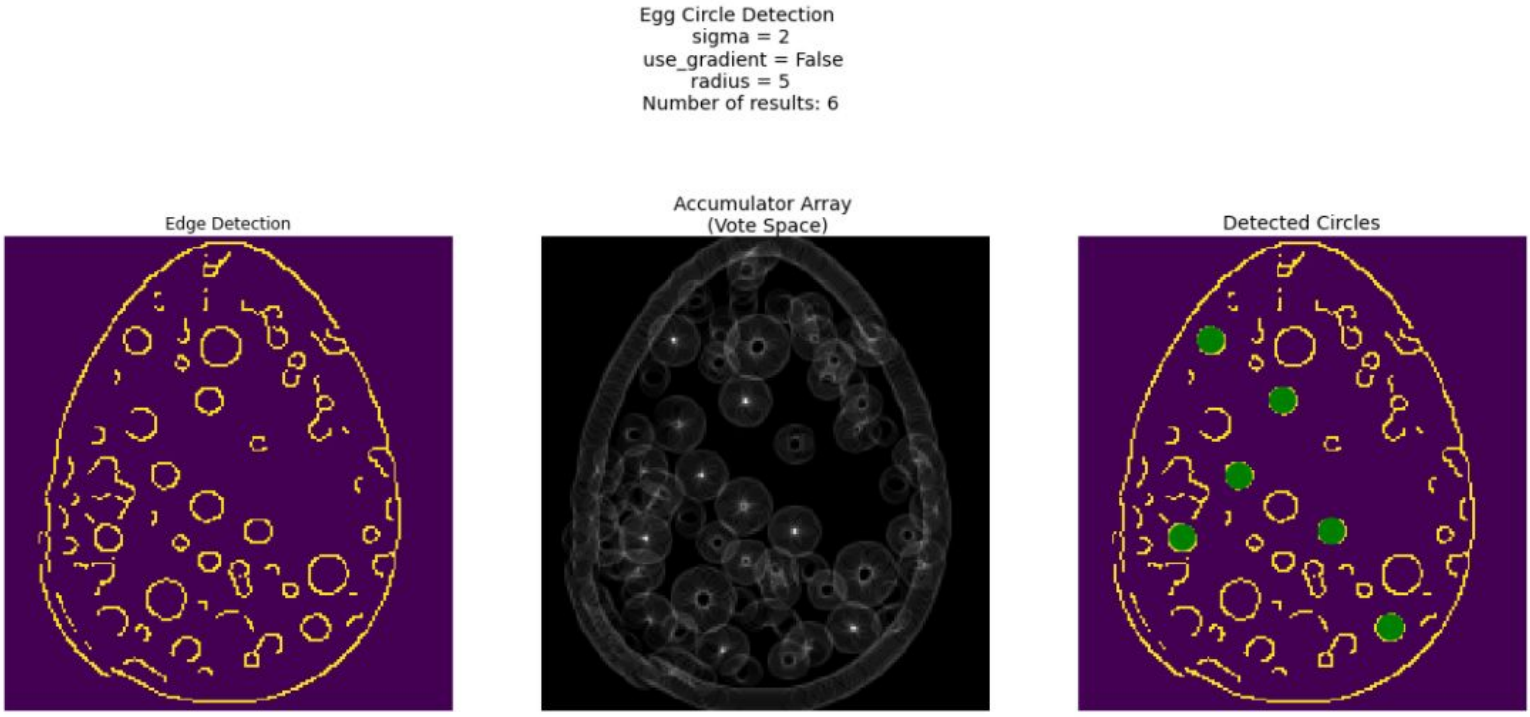
Original

```
Circle Results:
[[ 51  40]
 [ 79  63]
 [ 79  64]
 [ 62  93]
 [ 98 114]
 [ 98 115]
 [ 40 117]
 [121 152]]
```



Mean Shift clustering after results of above

```
Original points cluster assignments:
[4 1 1 3 0 0 5 2]
Cluster centers:
[[ 98.  114.5]
 [ 79.   63.5]
 [121.  152. ]
 [ 62.   93. ]
 [ 51.   40. ]
 [ 40.  117. ]]
```

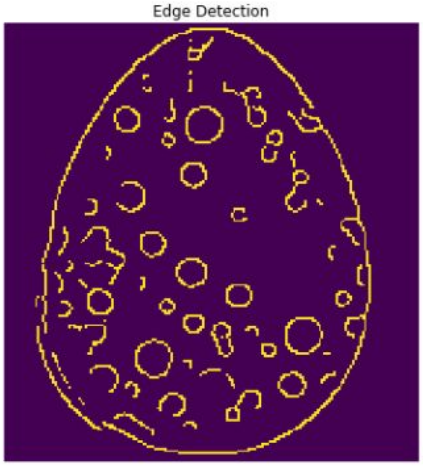


To optimize the number of circles found I knew I needed to merge overlapping or very close results into singular results. To do so I used the mean shift algorithm to determine cluster membership and the cluster means. In the example above this had the effect of merging 2nd and 3rd result as well as the 5th and 6th result yielding the 6 non overlapping circles we desire.

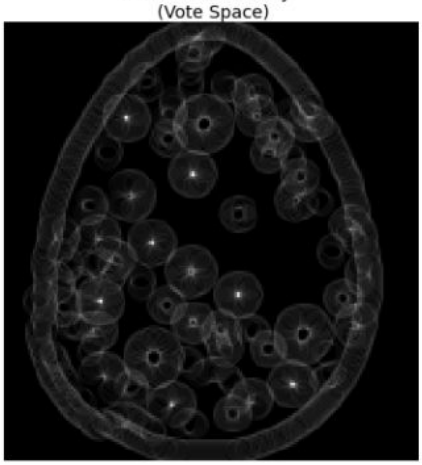
Below we see that this becomes increasingly useful if we reduce the voting threshold and thus many more circles are included. The examples below are done with a 0.5 voting threshold vs most others 0.8 voting threshold.

Egg Circle Detection
sigma = 2
use_gradient = False
radius = 5
Number of results: 28

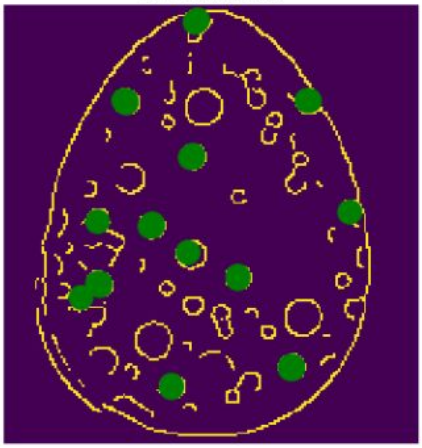
Edge Detection



Accumulator Array
(Vote Space)

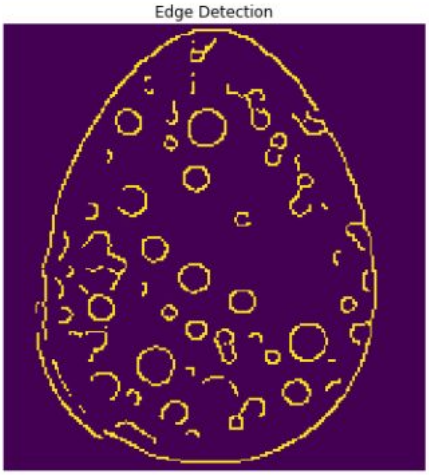


Detected Circles

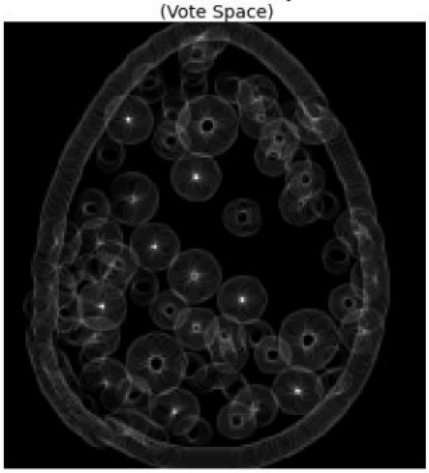


Egg Circle Detection
sigma = 2
use_gradient = False
radius = 5
Number of results: 13

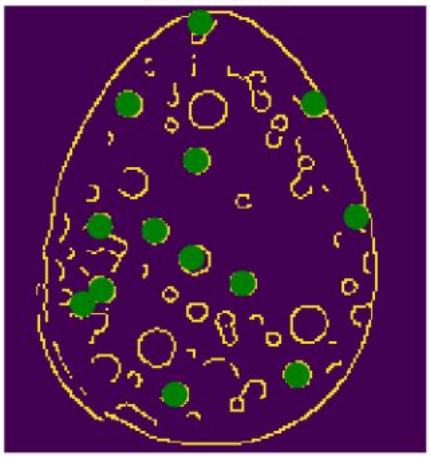
Edge Detection



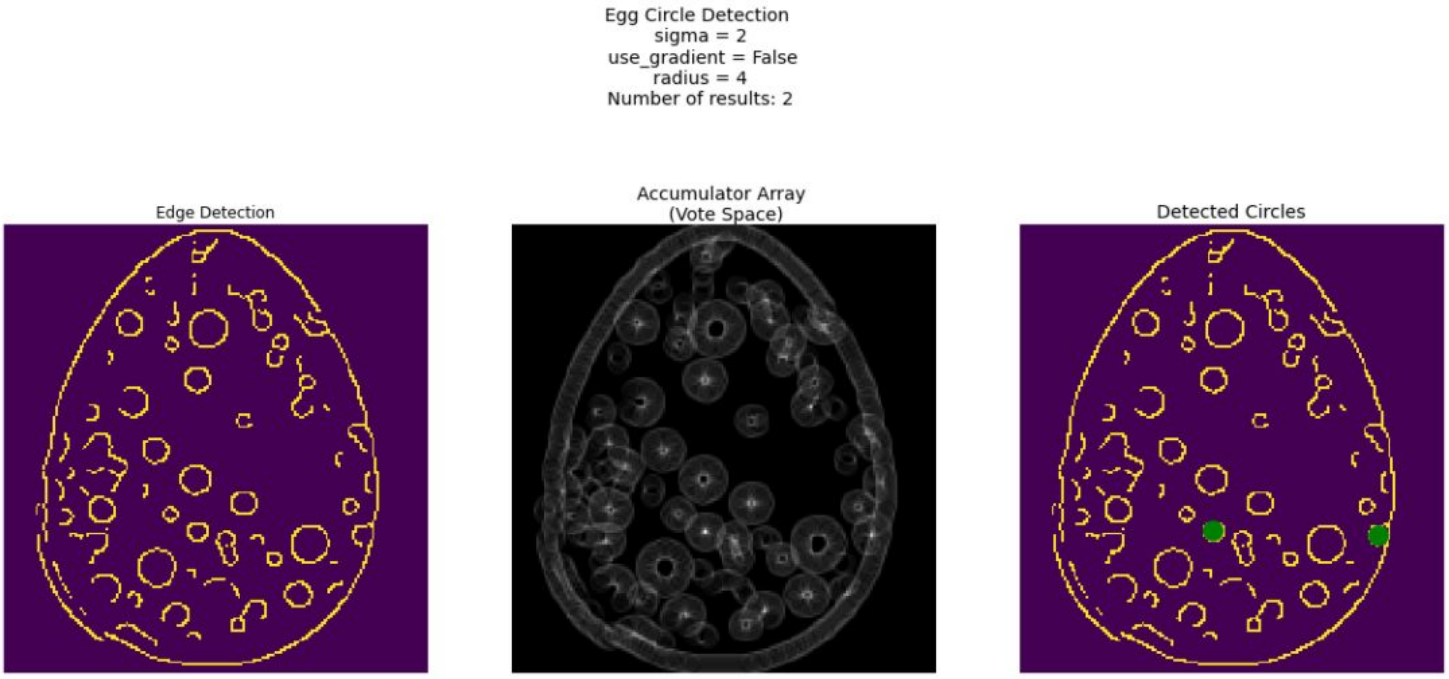
Accumulator Array
(Vote Space)

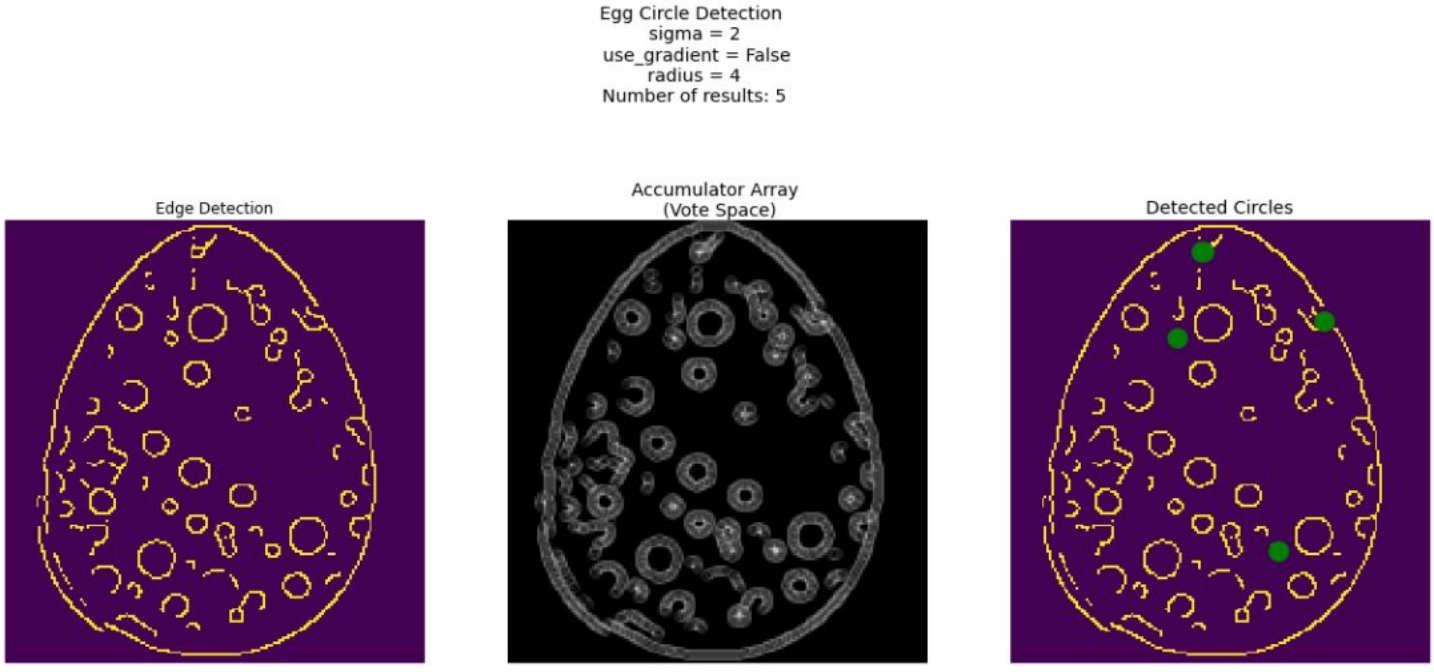


Detected Circles



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





Here we see the effect of a voting space that has been reduced to ¼ the size (½ in x and y). As expected the space is much lower

resolution and blocky. You may note that the new circles chosen at this low scale are far more oblong and warped than the original clean circles we were detecting at full resolution which is happening because our reduced voting space makes them appear more circular with less bins.