CS5670 - Computer Vision

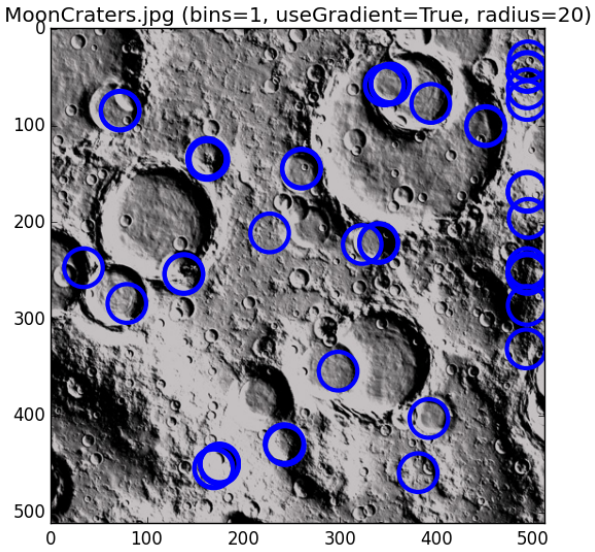
Homework 1 Write-up

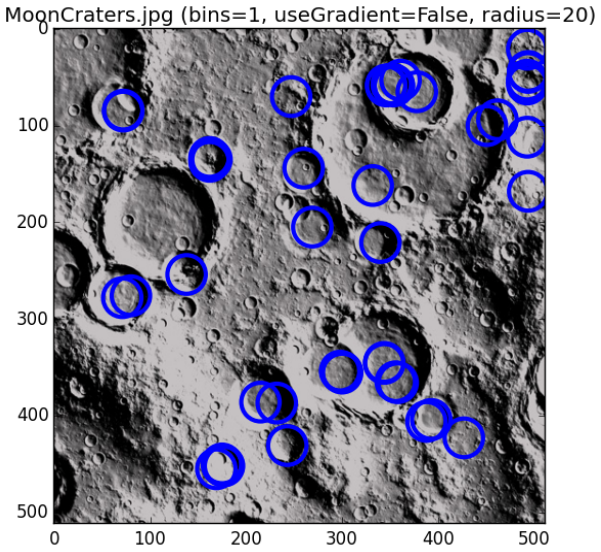
Jing Dai (jd837) - Muhammad Khadafi (mk2333)

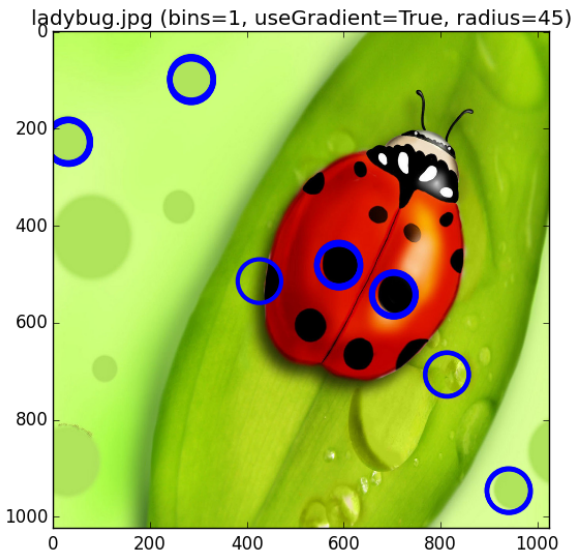
Task 1.

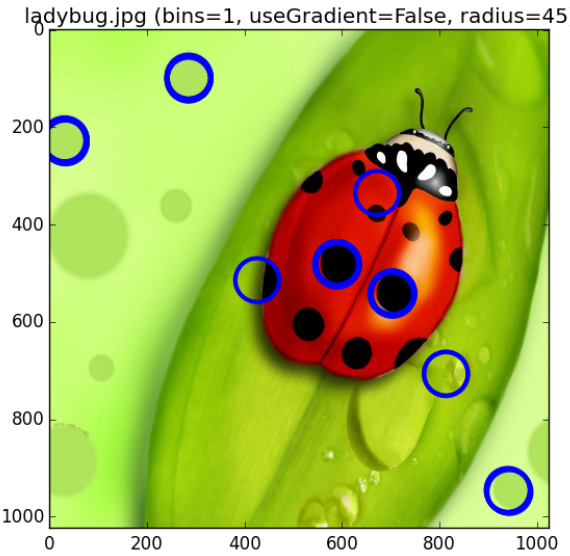
1. The detectCircles(im, radius, usegradient) functions work in several steps:
    - We read the image from file using scipy.misc.imread(filename, True), there true indicated flattening of the image to a single grayscale layer
    - We then blur the image using with gaussian filter of sigma 1
    - We then call getImageGradientandTheta(blurred_image) from helper.py, which basically convolve the image using the Sobel mask to get the gradient and the angle of each point
    - Using roundAngle(image_theta) from helper.py, we rounded the angles of each point to 0, 45, 90 and 135 degrees
    - Using supressNonMax(image_gradient, round_theta), we suppress the non maximum points depending on the direction of the line
    - This is where it differs from using image gradient and not. If we're not using image gradient, after suppressing the non-local maximum, we stop and consider everything that's left as an edge. We use the function thresholding(image_suppressed, threshold) from helper.py, which basically change everything above the threshold to 1 and the rest to 0, in this case using threshold value 0
    - Otherwise, if we are to use image gradient, we use the threshold derived from the formula 1.33*np.median(image_suppressed[image_suppressed != 0]) to get the threshold. Basically 1.33 times the median of the suppressed image without the 0s. I found this formula to be better in getting a good threshold compared to the usual 1.33*median(image), especially for the ladybug image where the distribution of values is so big that using that formula results in so few edges
    - Then, using hough(image_edges, radius) we loop through x, y and x_0 to get y_0 location in the Hough space using formula $y_0 = y - \text{sqrt}(r^2 - (x-x_0)^2)$, we vote up if (x,y) is an edge and in the stored angle of (x,y) is perpendicular to the line from (x0, y0) to (x,y) indicating that (x,y) is part of the circle centered in (x0,y0) with radius r
    - Lastly, we picked the top 50 circles with storeBestCircle(hough_bins, NUM_CIRCLES) from helpers.py, which grabbed the positions of the top 50 values from the Hough space bins
    - We return that as a Nx2 array of centers

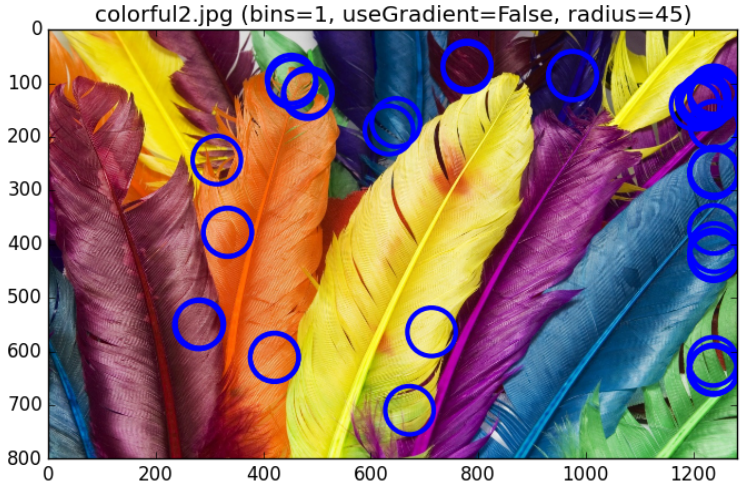
2. Comparison of circles, with and without gradient

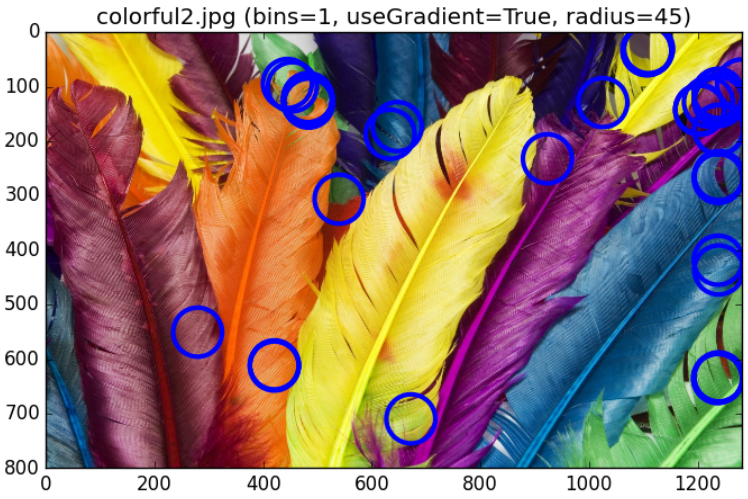


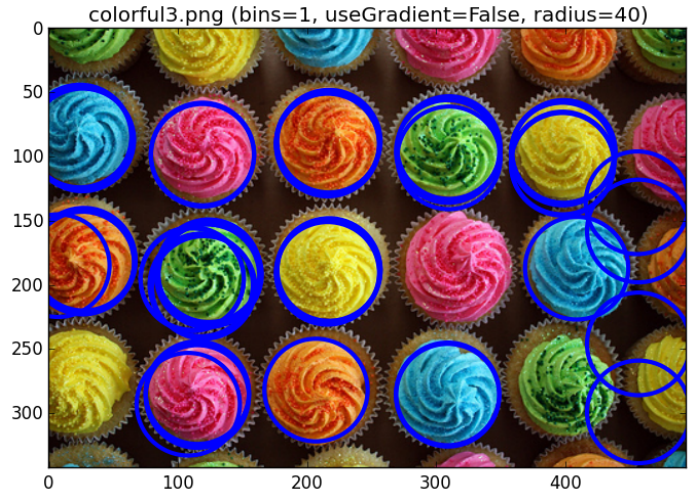


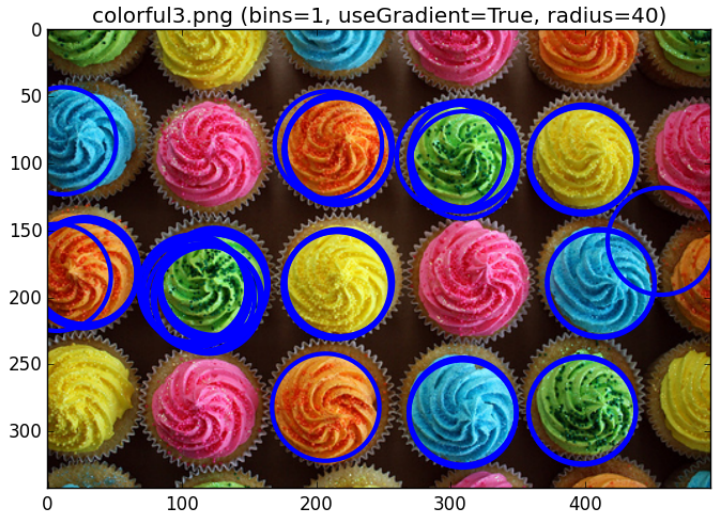


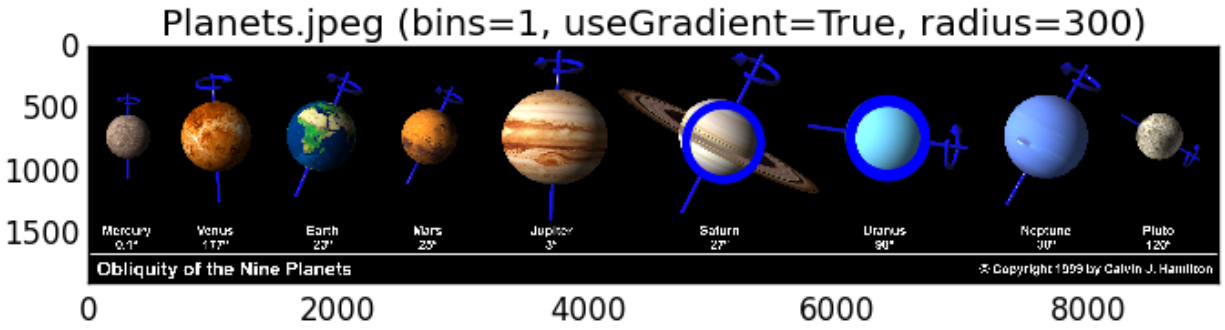


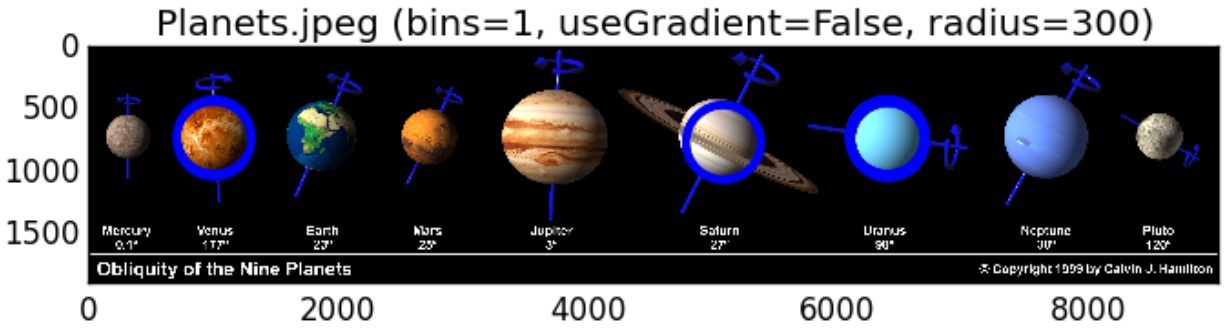



colorful2.jpg (bins=1, useGradient=True, radius=45)
0
100
200
300
400
500
600
700
800
0
200
400
600
800
1000
1200

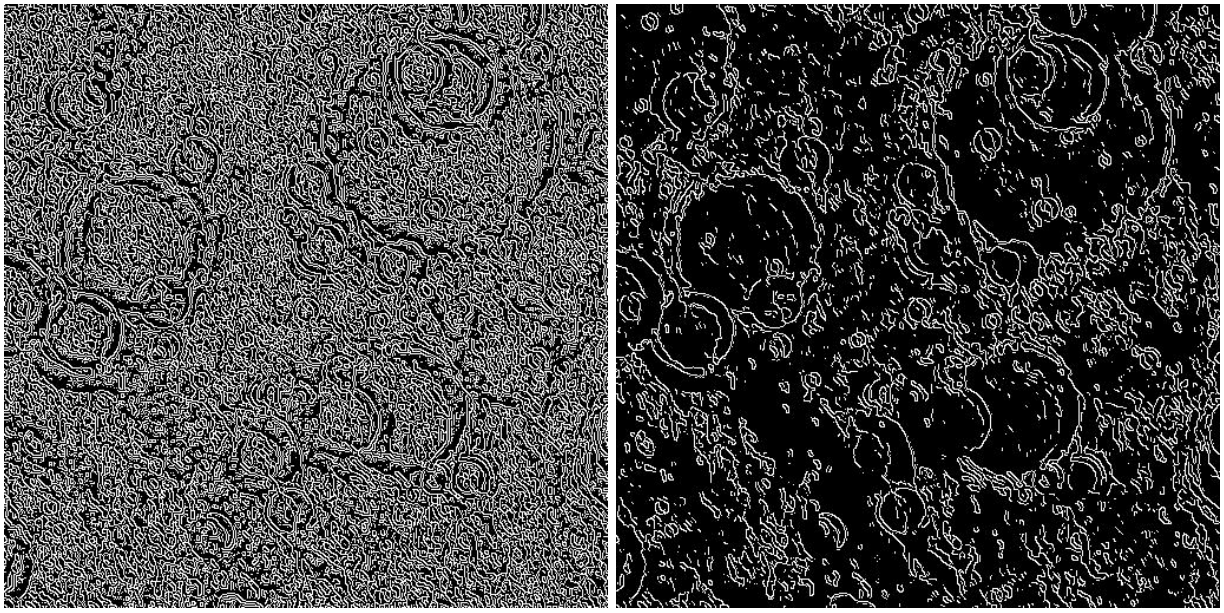

colorful3.png (bins=1, useGradient=False, radius=40)
0
50
100
150
200
250
300
0
100
200
300
400


colorful3.png (bins=1, useGradient=True, radius=40)
0
50
100
150
200
250
300
0
100
200
300
400



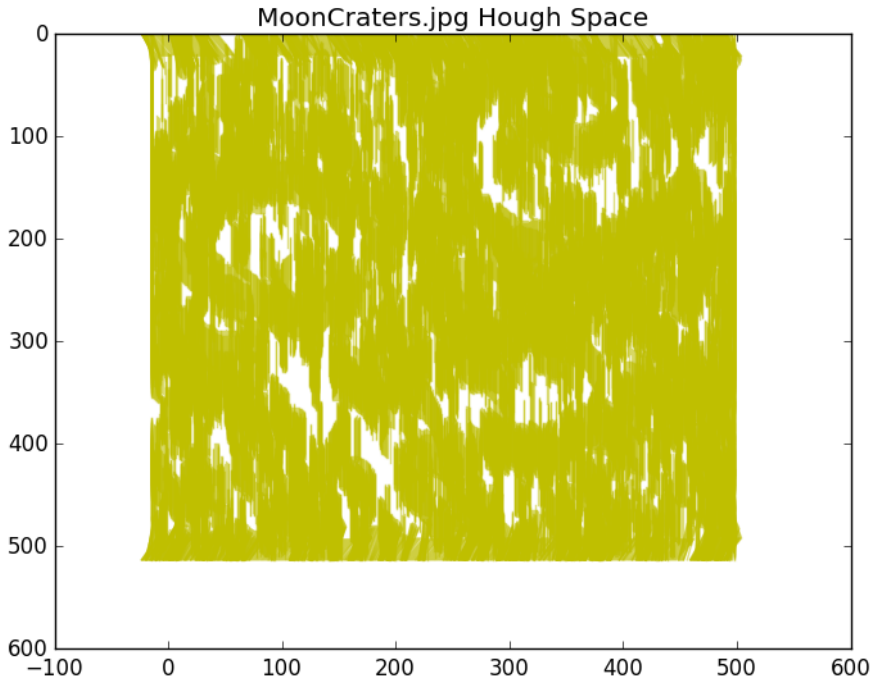


3. By using gradients, one of the benefit is that there will generally be much less edges to be considered during the Hough transform due to weaker edges being suppressed by the threshold. This will result in a faster computation during Hough transform. Another benefit is that it may reduce the number of false positives produced by circle detection. having a lot of edges may fool the function into considering unrelated edges as a circle just because they are there.
   The biggest disadvantage of using gradients is that the image might lose many of weak edges due to thresholding. Some of this edges might have formed a circle, and it map pass by undetected due to the edges being suppressed. A good example is the ladybug picture. When using the wrong thresholding (1.33*median(image)), the image lost a lot of information, including most of the circles, as shown below.



MoonCraters.jpg edges without gradient (left), with gradient (right)

4. Hough space
   What we see here are lines formed by the points (x_0, y_0) quantized from looping through x and y in the image space. Where the lines intersects the most are the top voted points in the Hough space, which is the center points of the circles. The Hough space created by the MoonCraters.jpg image (below) has too many lines, thus a little hard to see where the lines intersects most. However you can see some clusters where most of the circle centers will probably by

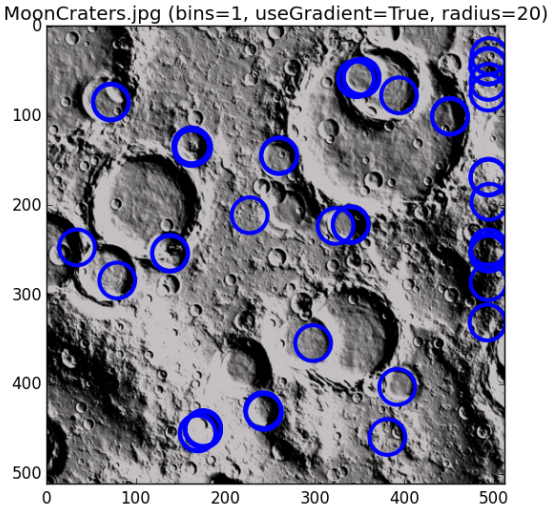


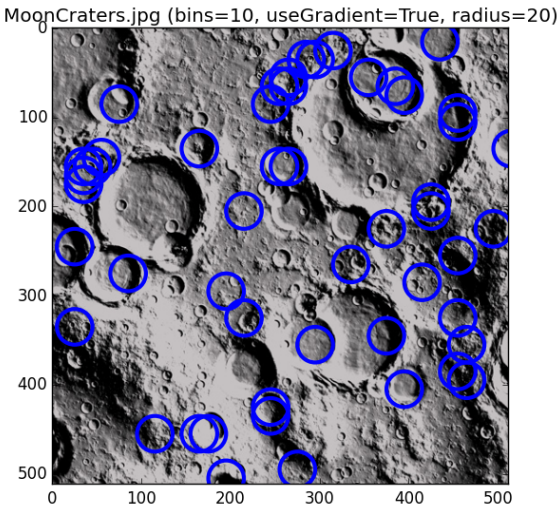
5. Different quantization

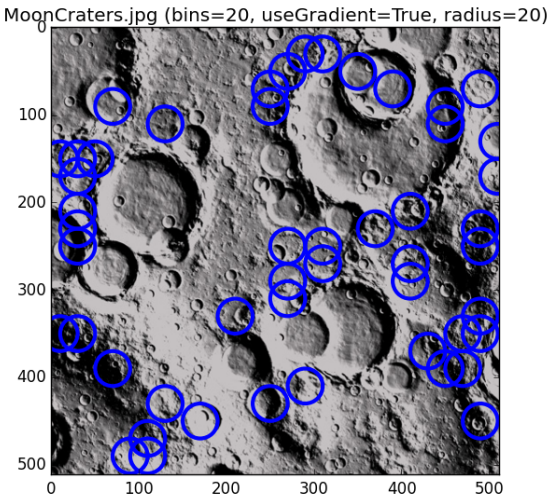
   In having different quantization, we basically combined some neighboring bins in the Hough space together. By doing this, we give more weight to thick circles that may have centers close to each other, and also reducing the resulting circles on the same spot. A good side by side comparison can be seen around (320, 50) of bins 1 and 10, where many circles are together in that spot for bin 1, and only one circle on bin 10. This also have the added benefit of lowering the processing needed to go through the bins to find the highest value.

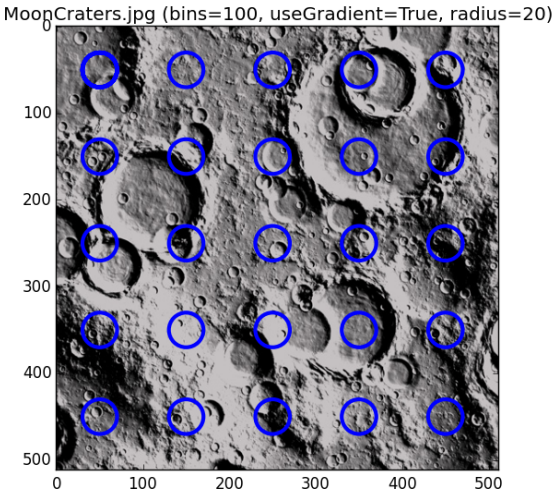
   However, the larger the bin may also result in high value bin on non-circles simply because there are many edges or part circles on a certain area, where combined, they will produce a high value. This is seen from the clusters around (100, 400) on the bon 20 image.

   Another disadvantage is that drawing the circle will require a non exact location as the center of the circle. The most logical location is at the center of the bin itself, but it might not always be the case. An example can be seen around (100, 100) of the bin 20 image, where the circle missed the actual image circle.

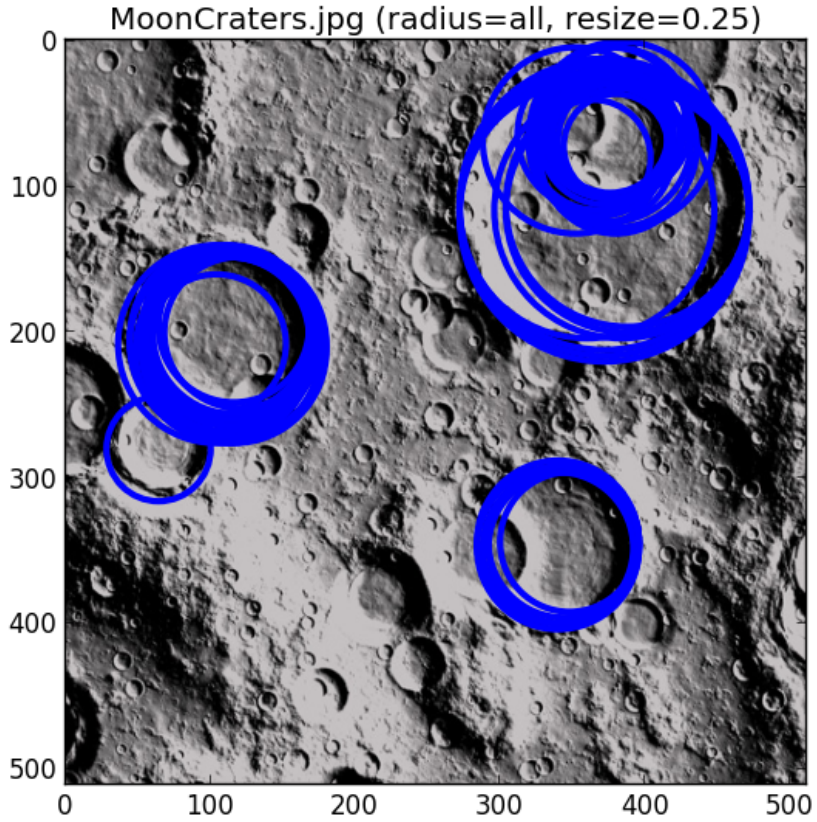








6. To modify the code and make it accept input without radius, we simply added another loop to loop though y_0 in addition to x, y and x_0. We then find the r that fulfils the requirements of $r^2 = (x-x_0)^2 + (y-y_0)^2$. We then vote up the 3D Hough space of $[x_o, y_o, r]$. This will expand the running time significantly but it produces circles of different sized from the image.
   Another big problem resulted by variable radius is that the big radius is unnecessarily voted up due to the sheer amount of edges they went through on their circumference. To counter this, we added code while detecting the circles to ignore circles with votes less than ¼ of the circumference (i.e the edges are just random edges, not making a circle).
   A method that we use to reduce complexity is to reduce the size of image and resizing the resultant circles later. This does result in reduced information of edges and reduced accuracy, which we need to take into account balancing accuracy with speed. The image below was Hough transformed as 1/4th of it's original image.

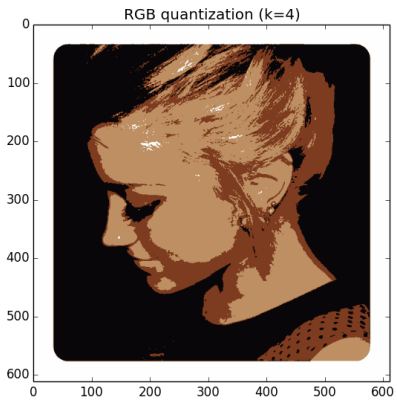
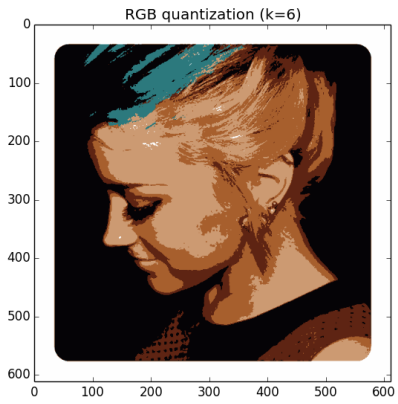


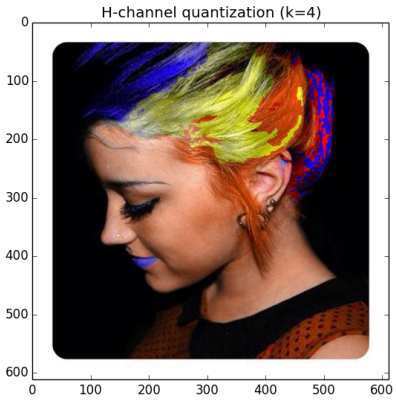
7. One way we can use the Hough transform to detect rectangles is to detect four perpendicular straight line intersections. Once we know all the line equations detected from the Hough transform using the (r, θ) space, we can assign 2 arrays to each line, one for all other lines that are parallel (same θ) and the other for lines that are perpendicular (θ_1-θ_2 = 90). We can eliminate those who don't have both. For the remaining lines, follow through, find start and end of the edge, continue with each perpendicular line, if they are within each other, then they are a rectangle

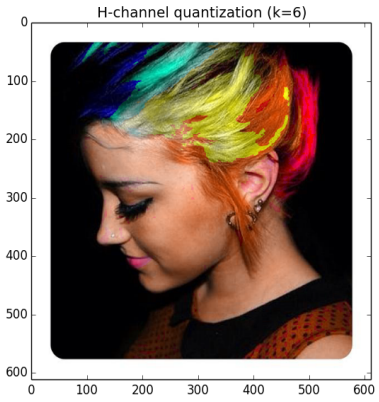
Task 2.
In the program, the **quantizeRGB( )** function quantizes the colors in the image using k-means, and output the quantized color at each pixel.
The **quantizeHSV( )** function convert the RGB color at each pixel to HSV color. It then quantizes the H channel, and convert the quantized H channel and the original S, V channels and convert the image back to RGB.
The **getSSD( )** function calculates the sum of squared distances of two RGB images.
The **histHChl( )** function generates the two histograms of the H channel of the images before and after quantization.

Results:

- Image: colorful1.jpg

K=4	K=6
RGB Quantization	
	
SSD: 66219614	SSD: 57271006
HSV Quantization	

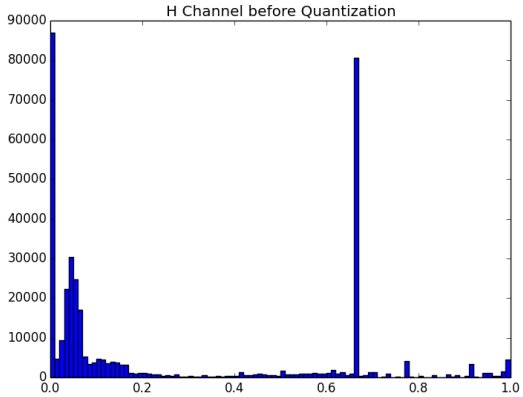




SSD: 13850042

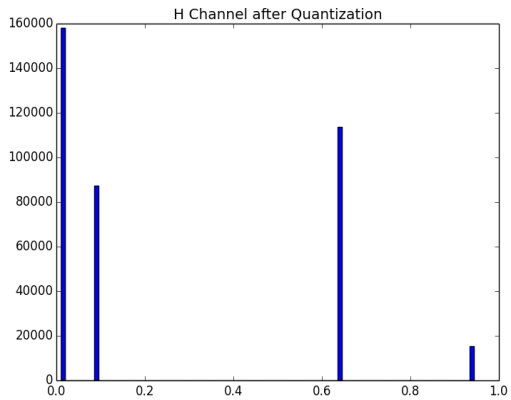
SSD: 11701459

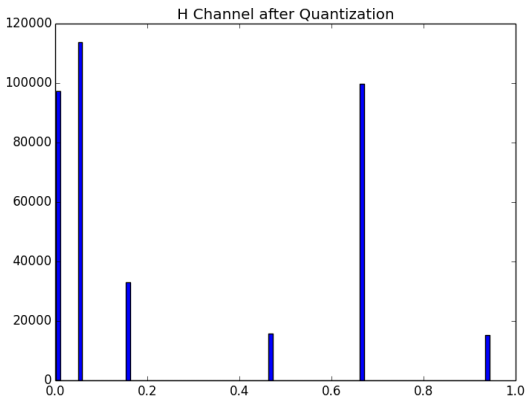
Histograms



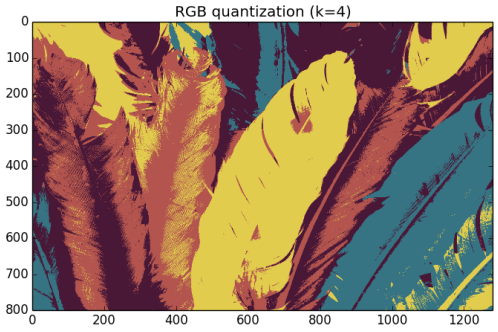
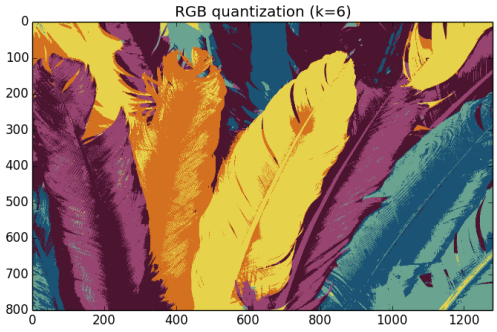
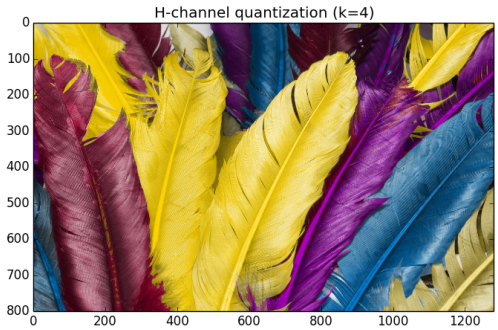
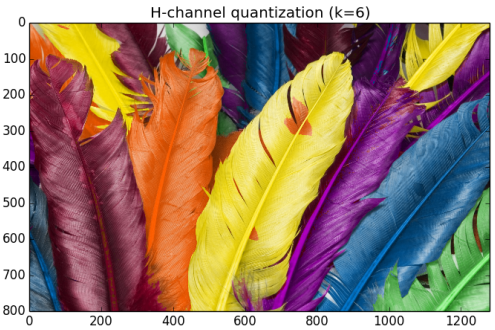
K=4

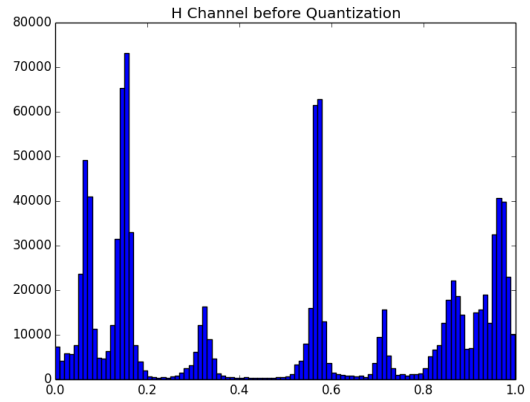
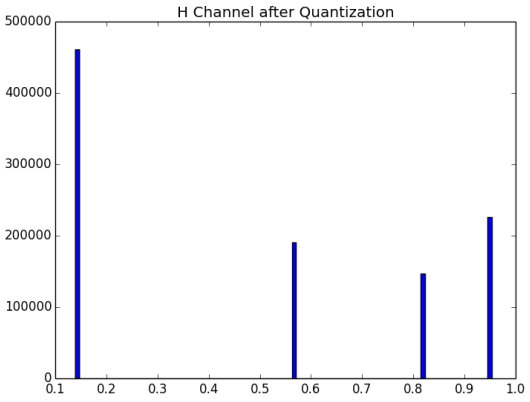
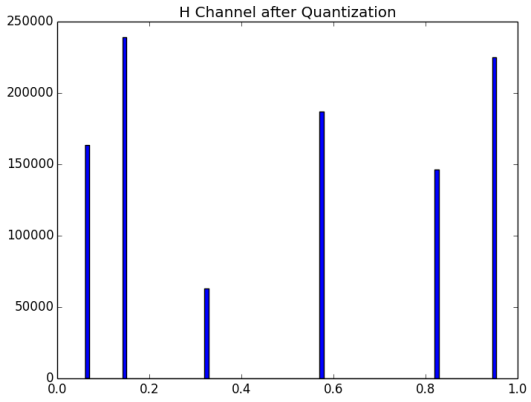
K=6





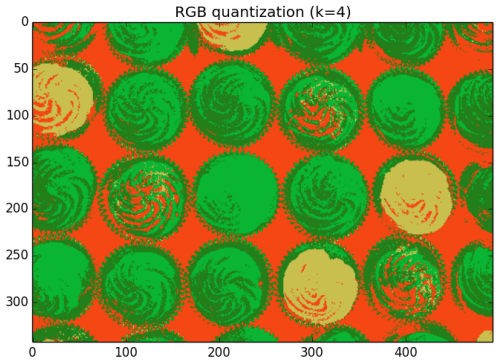
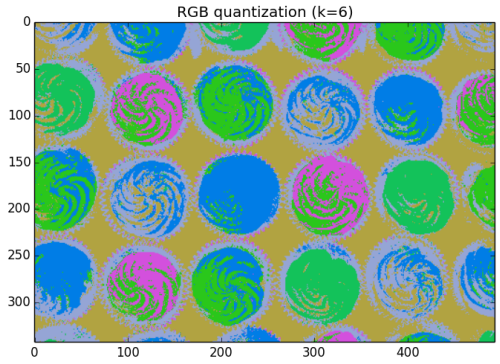
- Image: colorful2.jpg

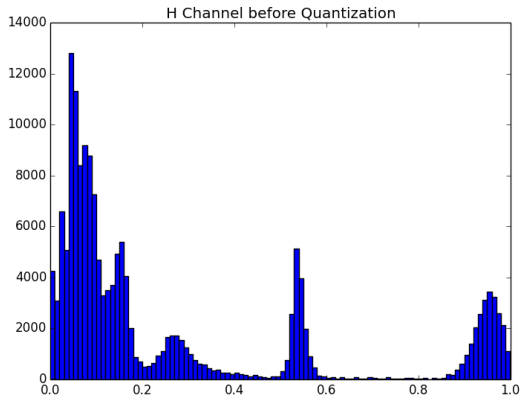
K=4	K=6
RGB Quantization	
	
SSD: 311472502	SSD: 295821554
HSV Quantization	
	
SSD: 97544829	SSD: 74039887
Histograms	

	
K=4	K=6
	

- Image: colorful3.png

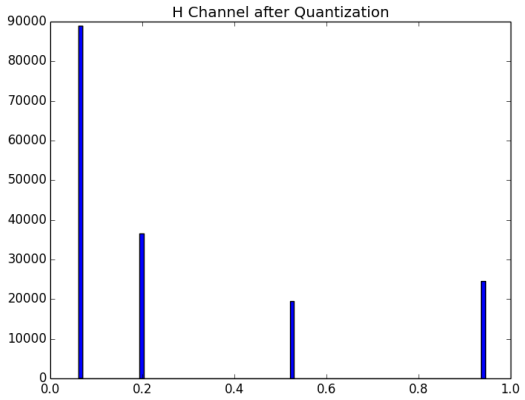
K=4	K=6
RGB quantization	

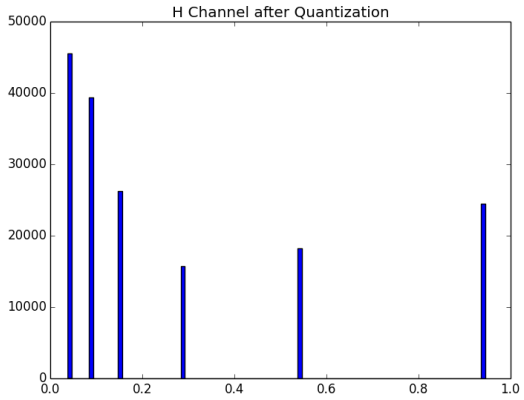
	
SSD: 6.36334e+08	SSD: 4.24475e+08
HSV Quantization	
	
SSD: 8.78179e+07	SSD:2.3971e+07
Histograms	



K=4

K=6





As shown in the above images, the H channel quantization method preserves the original images better than the RGB quantization method. Larger K gives a better quantization result and also smaller SSD.

The results are not the same for every execution because the kmeans method provided by **scipy** sets the maximum number of iterations to 20. Therefore in each execution, the H values chosen by kmeans can be different which results in different quantization results.