Assignment 3

Taylor Foxhall
tfoxhal1@binghamton.edu

November 13, 2015

1 Color Theory

1 6.5. The color corresponding to the pixel at N/2 has an RGB value of approximately (127, 255, 127). Therefore, someone would see a sickly pale greenish color that turns out is very hard to read.

2 6.16. Identify the gray levels in the given HSI images.

(a) This image is the intersection of all the primary colors. Thus, for hue values, we get $H(red) = 0$, $H(yellow) = 43$, $H(green) = 85$, and $H(cyan) = 128$. Since hue ranges from 0 to 360 and we need to scale it to 0 to 255, we can simply calculate the angle and scale it for the rest: $H(blue) = 170$, $H(magenta) = 213$ and $H(white) = 0$.

(b) Since we're working with straight primary and secondary colors, the saturation is 0 for black or white and 255 for anything else.

(c) By the intensity formula, the primary colors have an intensity of 85, the secondary colors have an intensity of 170, white has an intensity of 255, and black has an intensity of 0.

3 6.20. Describe the HSI components of the image:

(a) The HSI component images would appear as follows: the hue image would be four squares with the red square turning black (gray-level of 0), the green squares would turn dark gray (gray-level of 85), and the blue square would turn light gray (gray-level of 170); the saturation image would be completely white (gray-level of 255); the intensity image would be completely gray (gray-level of 85).

(b) Smoothing the saturation image would have no effect on it. The image would be unchanged.

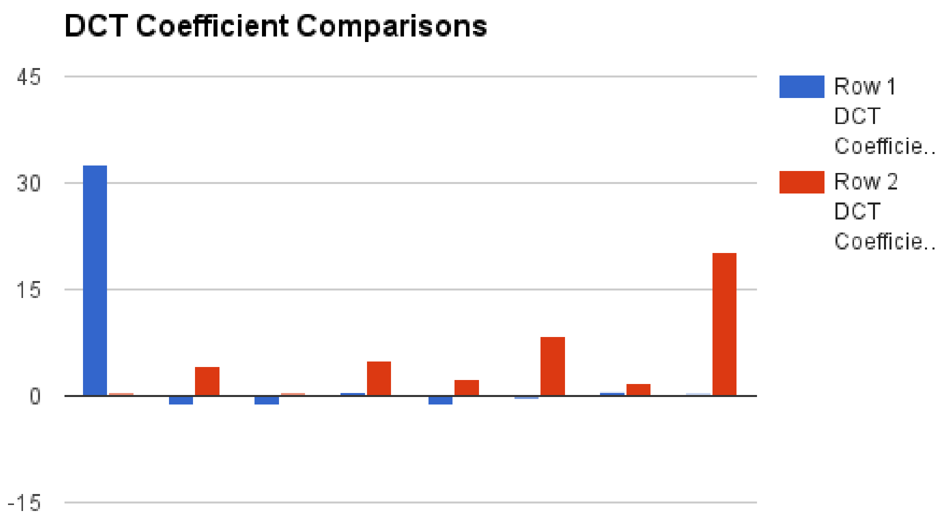
(c) Since the hue image isn't entirely one level, applying an averaging mask would make the borders between the squares blur, and you'd see a gradient transition between each of the squares.

2 DCT Based Image Compression

1 S. imulate DCT on some data.

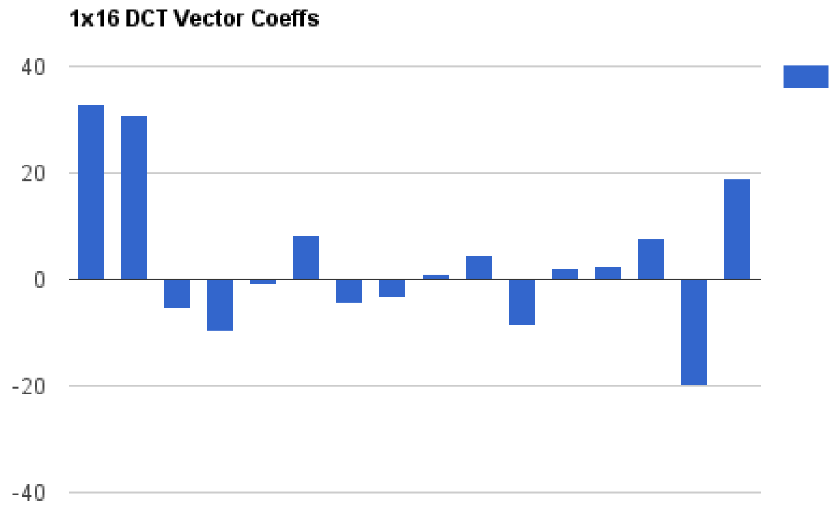
(a) 1x8 DCT:

$$\begin{bmatrix} 32.526 & -1.281 & -1.306 & 0.449 & -1.414 & -0.300 & 0.541 & 0.254 \\ 0.353 & 4.250 & 0.349 & 5.046 & 2.474 & 8.383 & 1.768 & 20.388 \end{bmatrix}$$



(b) 1x16 DCT:

$32.880 \; 30.799 \; -5.532 \; -9.838 \; -0.956 \; 8.537 \; -4.596 \; -3.632 \; 1.060 \; 4.466 \; -8.684 \; 2.274 \; 2.309 \; 7.773 \; -20.133 \; 19.182$

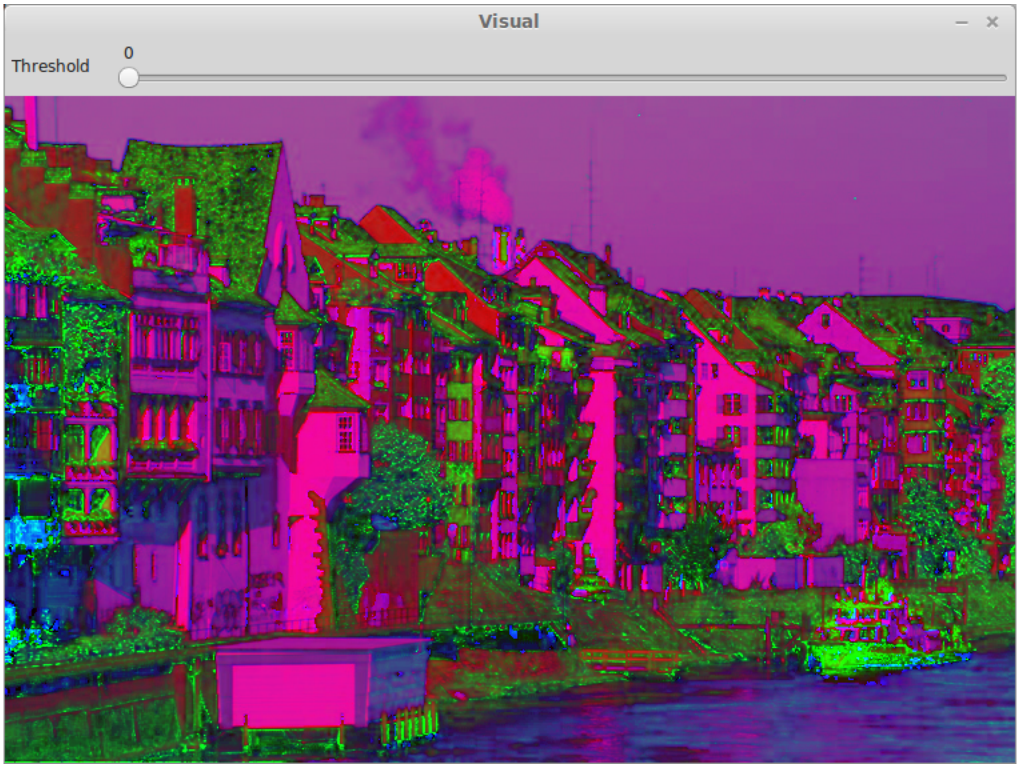


For this example, I think an 8-point DCT would be more useful in compression because if we used the 16-point model, when we compress we may lose a lot of information about the higher frequencies depending on how much we want to cut off. Say we want to compress the image by 50%. Then we drop the last 8 biggest frequency componenets would be harder to reconstruct the second row. So if our image happens to fluctuate between low and high intensities at that location, then we lose that important information and we get a very blurry picture over a large area. However, if we use the 8-point model, then we lose the high frequency component information in more local areas, so we get less blurring overall.

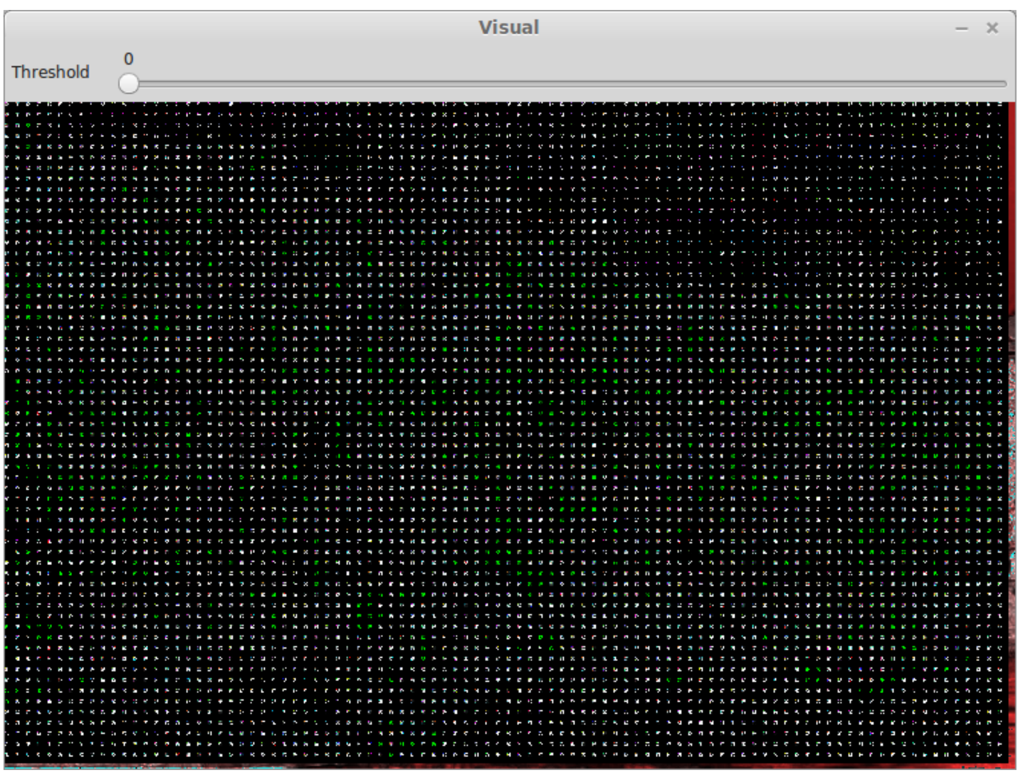
2.1 Programming

The algorithm behind DCT was rather simple, apply a DCT forward transform on an image, eliminate some of its high frequency components, and restore the image with a DCT back transform.

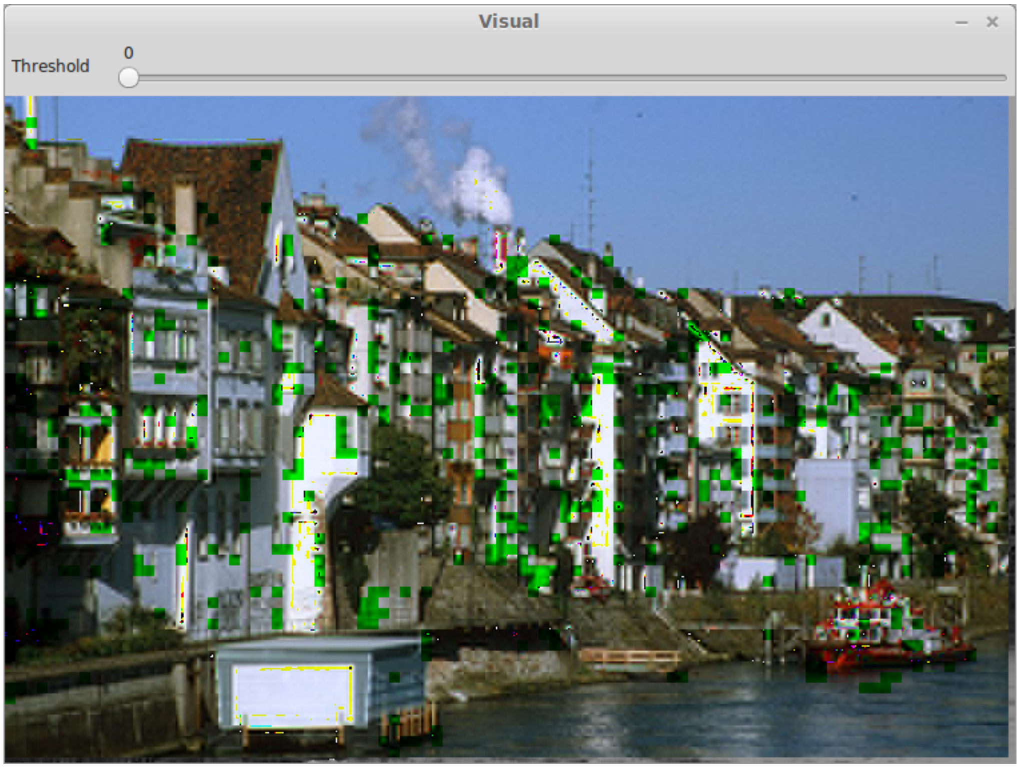
I started with transforming the image from BGR (the way OpenCV loads it) to HSI. The results are a bit psychedlic, but unsurprising given OpenCV still thinks it's a BGR image.



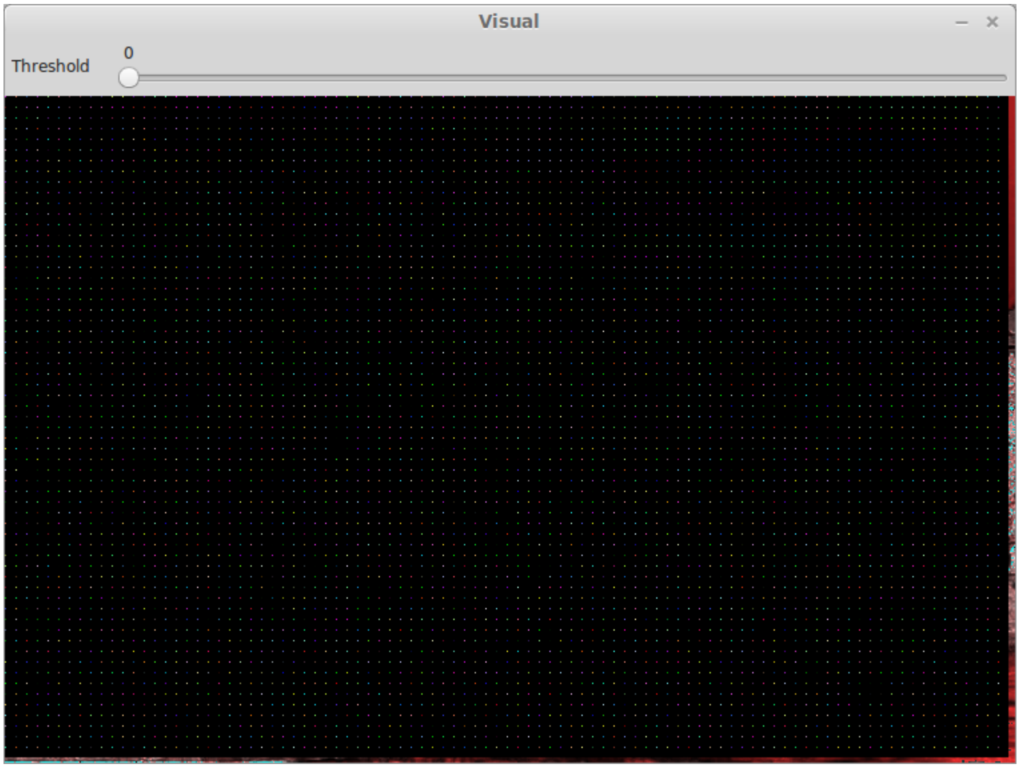
This is with hue, saturation, and intensity scaled to a range of $[0, 1]$. So, I ran it through a forward DCT transform:



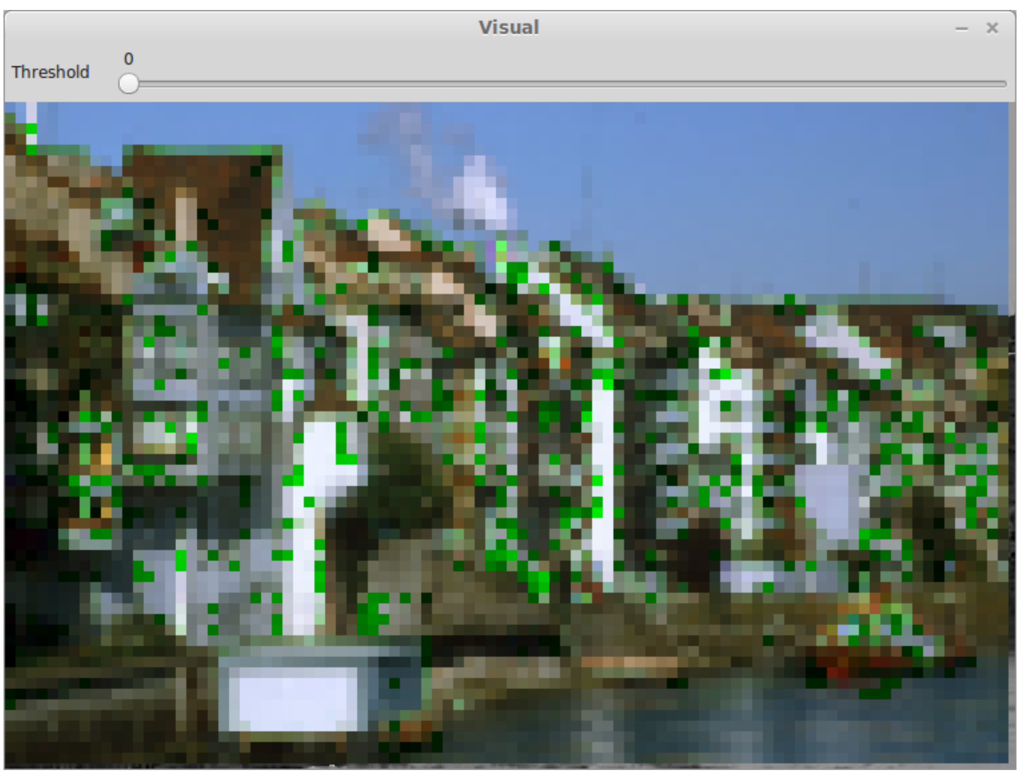
Again, this image looks odd, but it's actually unsurprising. The image itself is fairly meaningless because all we're doing is visualizing the coefficients of the lowest 9 frequencies of each HSI channel, placed into what OpenCV thinks is a BGR image. But you can clearly see that for each 8×8 segment, we're cutting out all but the lowest 9 frequencies. I performed a backward DCT transform on this image:



And now something looks fairly suspicious. Clearly, there are some unexpected green spots sprinkled across the image. To try to get a better idea of what went wrong I inspected the hue, saturation, and intensity channels separately. Saturation and intensity looked okay, but the hue channel seem to have some dark spots, approximately where the suspicious green squares are. I can only infer this means when the higher frequencies are ditched, critical hue values are lost, resulting in lost color information. I also ran this through the 1-frequency transform:

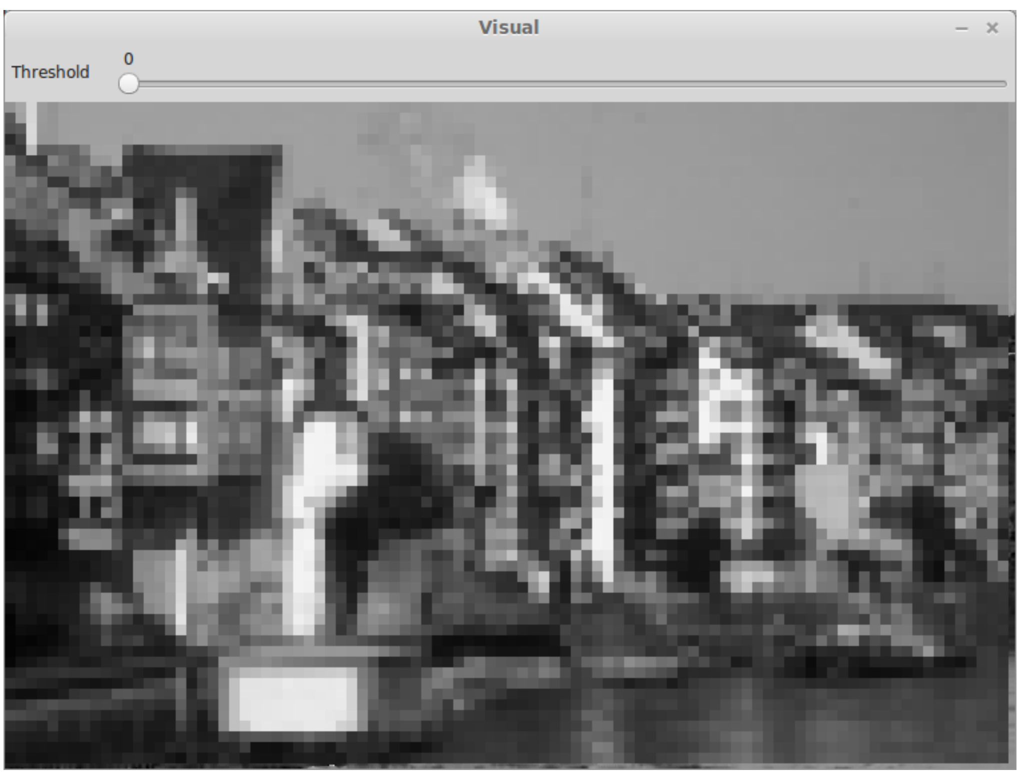


And the corresponding inverse DCT:



The same effects can still be seen. I provided the intensity channel output only, as it had a better effect:

And for the DC component transform:



Oddly enough this image looks fairly honest to the original, despite it's obvious low quality. Of course, I expected a lower quality than the first round of DCT since we took out most of the frequency components. R2 is clearly of higher quality than R1 since it retains more of the original information.

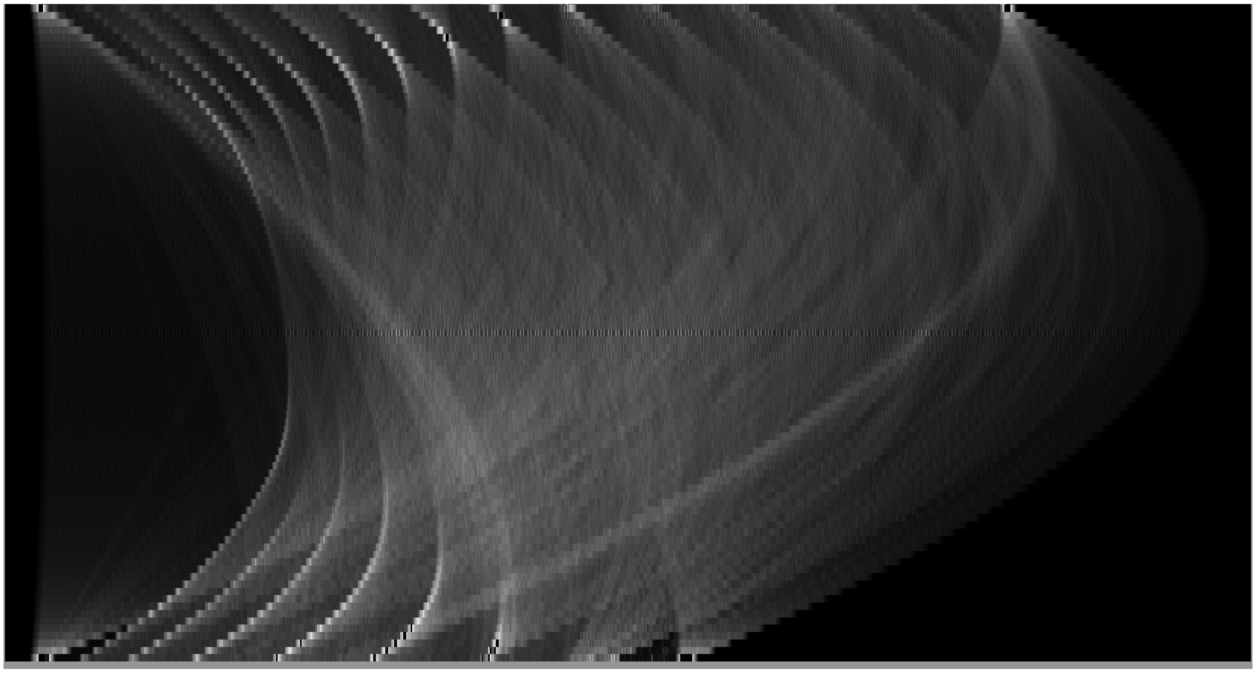
3 ROI Segmentation

3.1 Linear Segments

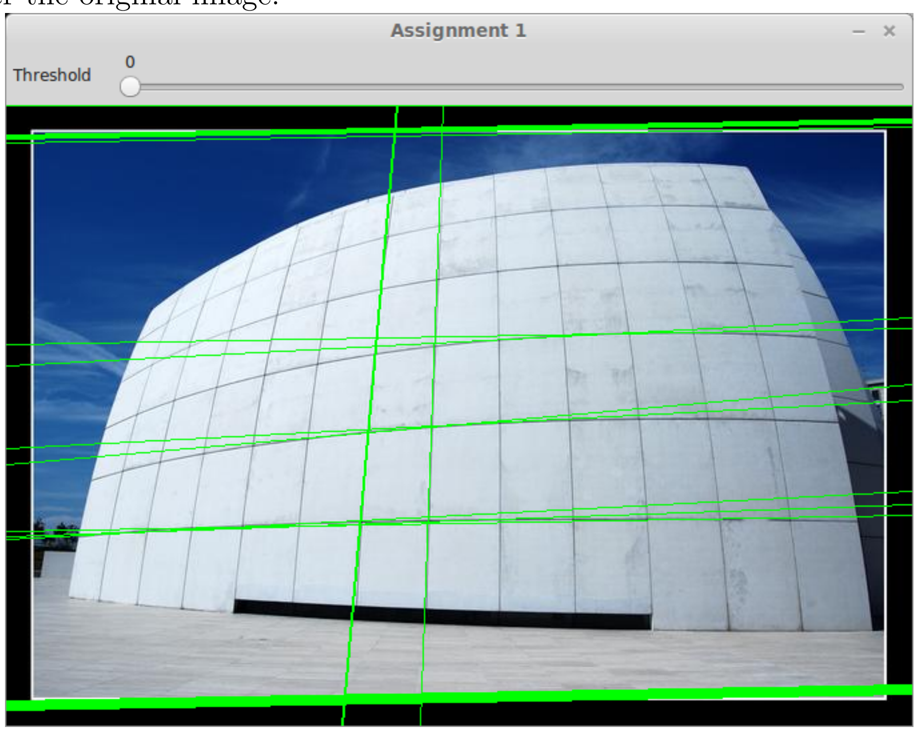
I implemented a linear Hough transform to detect lines in the image. Every line can be represented as $r = x\cos\theta + y\sin\theta$ where θ is the angle from the origin and r is the distance a perpendicular line is away from the origin at angle θ. Therefore, we can transform the xy-plane into an $r\theta$-plane.

Here is the image I will run a linear segmentation on:

Before I ran the Hough transform I first had to preprocess the image a bit. OpenCV reads in color images in BGR format, so I first converted the image to HSI and extracted the I channel, as it was the grayscale version of this image. Then I applied the Sobel edge detector to extract the edges out of this image. All values above a specific hard-coded threshold were then examined in the Hough transform. The resulting transformation is shown below:



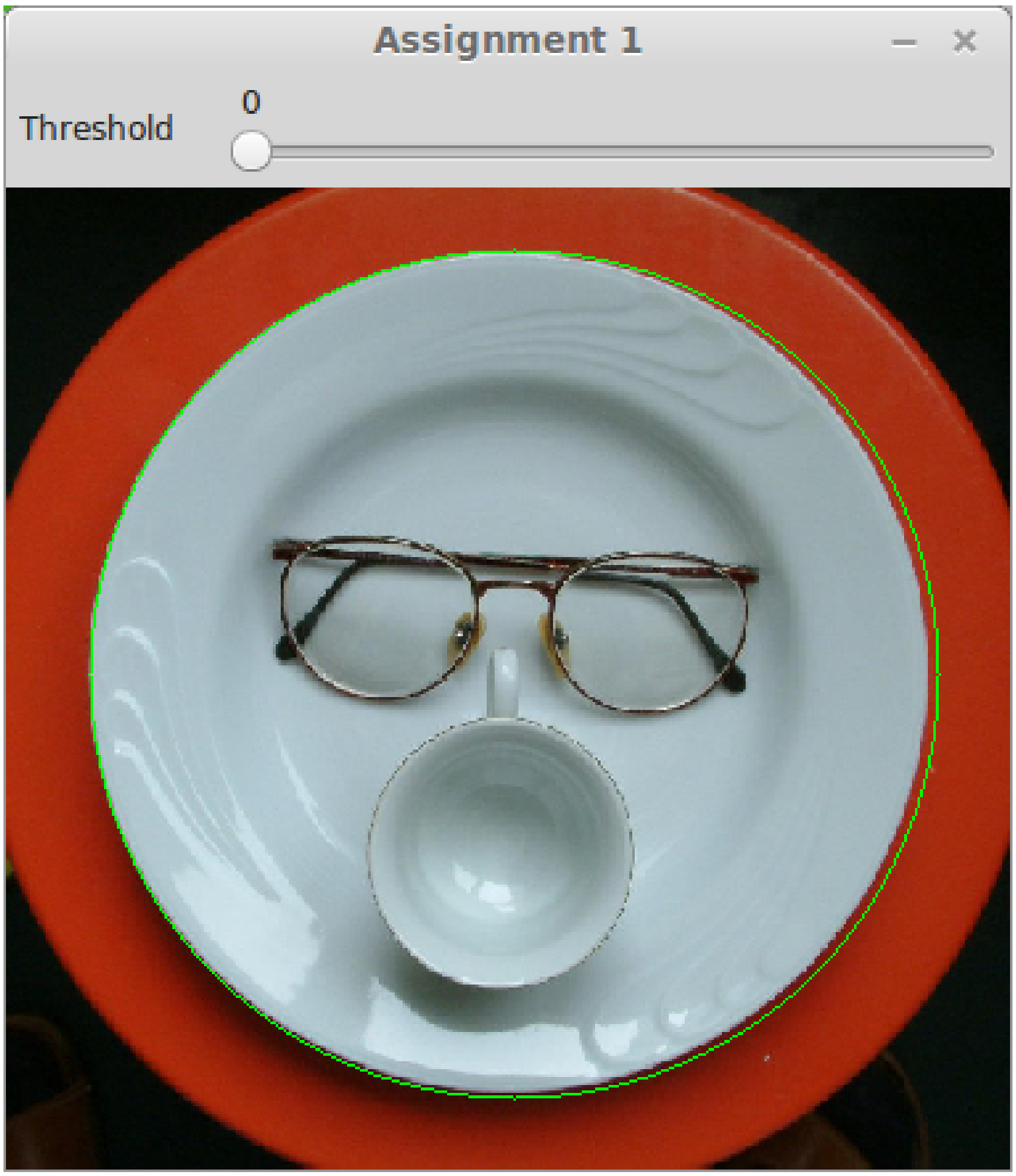
This image is a little small in height because I examined values of θ from 0 to 90, as values from 0 to 360 yielded similar results. But you can clearly see intersections of sinusodial curves, which are likely candidates for straight lines. I took the top 50 candidates and plotted them over the original image:



The resulting image clearly shows the algorithm has the right idea, though it's focusing a lot on the borders of the image, which are in fact, lines. For some reason the segmentation seems to prefer horizontal lines as well, but this may be specific to this image. I also hypothesize that I'd have achieved better results had I used a Canny edge detector instead, reducing some of the extra noise from the Sobel image.

3.2 Circular Segments

We can also use a Hough transform to detect circles. Similarly, since the equation of a circle is $r^2 = (x - x_0)^2 + (y - y_0)^2$, we can transform the xy-plane into x_0y_0r-space. This requires an extra dimension of memory, as a result induced a serious slow down on the region detection process. However, extrememly accurate results were achieved:



The global maxima from the entire transform yields this very clear circle around the white plate. Admittedly, the algorithm is fairly naive. The centers could be any valid point in the picture and the radius dimension was bounded from 0 to 1/2 the image width.

Some speedup could probably be achieved by increasing the lower bound for radii (after all, we're trying to detect big significant circles) and possibly by changing increasing the step change of the radius. Also for larger radii, we can gradually ignore checking the center of the of the image. This is because the edge of a circle with a raidus larger than either half the width or height couldn't possibly be located in the center, otherwise they'd reach outside the border of the image. Thus, the edges of the circle must be somewhere closer to the edges of the image. That would significantly reduce the memory used and benefit the time as well (it takes approximately 90–120 seconds to produce an output). Threading may also be an option for further increases in speed.

4 References

1. OpenCV 3.0 Documentation — http://docs.opencv.org/3.0.0/

2. Hough Transforms — https://www.youtube.com/watch?v=kMK8DjdGtZo

3. DCT — https://www.youtube.com/watch?v=_bltj_7Ne2c

4. Midpoint Circle Algorithm — https://en.wikipedia.org/wiki/Midpoint_circle_algorithm