**Detecting Lines, Features and**

**Experimenting with Color**

CSE 4422 Lab Assignment I

Aysar Khalid

10/04/2014

**Introduction**

In computer vision, edge detection is a widely used method that allows the identification of sharp intensity changes or discontinuities. It builds on the simple premise that at the most fundamental level, an edge is just a line segment with a sharp change in brightness. Edge detection is great for learning about an image and its formations, this is also known as feature detection. Edge detection algorithm design is not a trivial task since it is not just a matter of thresholding and comparing adjacent intensity differences. The Canny edge detection algorithm was used in this lab to explore better understand and appreciate its methodology.

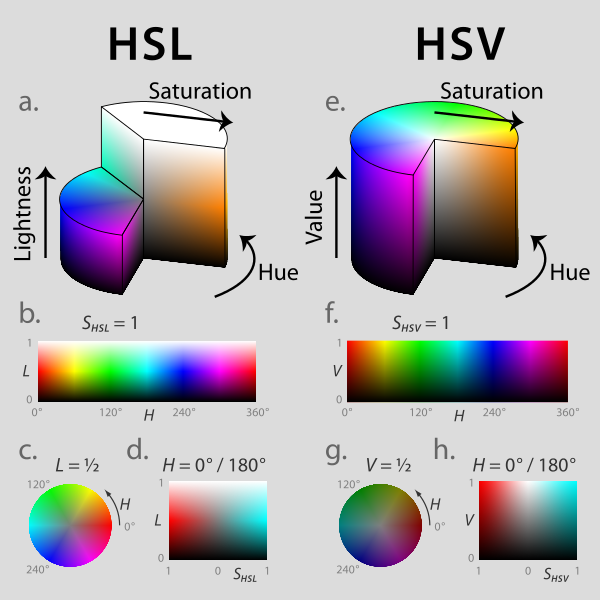
One type of color representation in use today is the HSV (hue-saturation-value). It allows the ease of perception to the human by using a cylindrical-coordinate system of the RGB color model. Whereby, the hue (uniqueness of color) is the angle around the vertical axis, the saturation (how colorful is it?) is the distance from the center and the value (brightness) is the height of the cylinder.

Figure A cut-away of the HSV model

**1. Detecting Lines**

In this first problem, we were tasked with developing a variant of the RANSAC (Random Sample Consensus) algorithm on the output of the Canny edge detector algorithm (using default values). This meant we first needed to randomly select a minimum number of points required to determine a model and its parameters. In this case, since we are dealing with a 2D image, it means we needed to choose two random edge points to form a line (our model). Using the cross product, we solved for our model to find a line that goes through the two points. As a sanity check, we use the dot product of the line and either point and ensure it equates to zero to ensure both points belong to the line. Next we determined, based on a threshold- D, how many points from the set of all points (in our case, edge points) that fit our model. To ensure we attain a high probability of finding the best model, we iterate this process N times. After various experimentation, it can be concluded that the higher the N (>100), the better the final model fitting.

Using least squares, to best represent the inlier-edge points, a fitting line was found as outlined in green in the figure 2. As we can see, the algorithm attempts to find the largest feature edges, in this case it’s the rightmost longest side of the rectangle. Due to outlier artifacts, noise and a moderate N value, it can sometimes detect only parts of the edge whole.

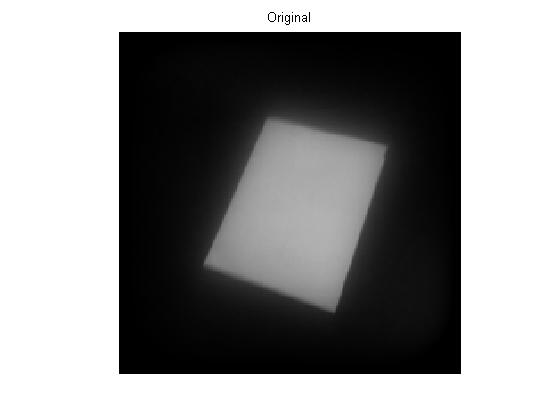
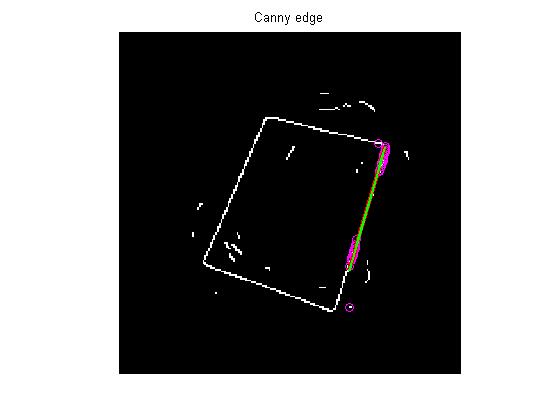
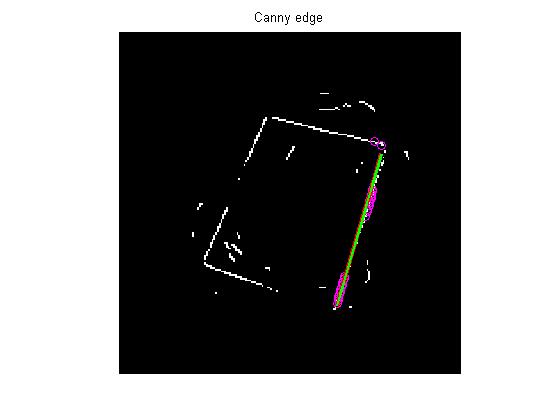
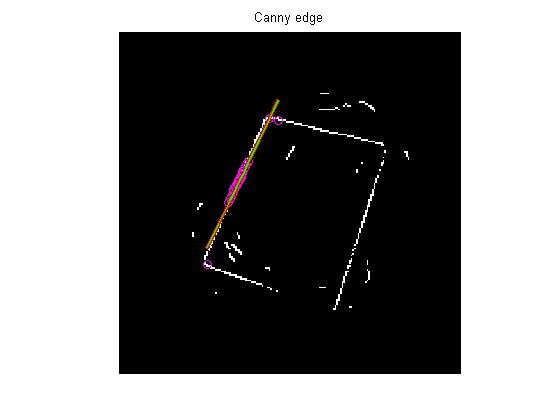
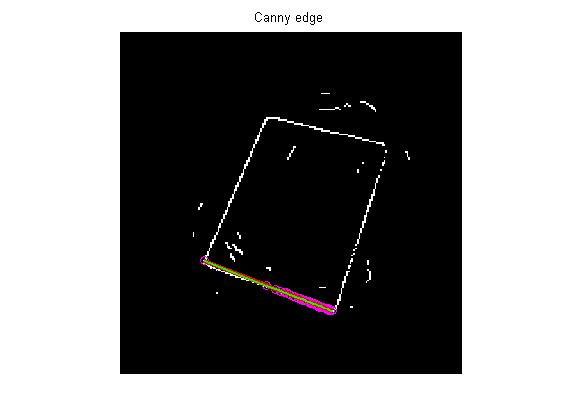


Figure The output of the Canny edge detection algorithm. In addition to our algorithm, the green line is the final line after least squares fitting. The magenta spots are the most edge points found after N iterations (in this example N=100, D=3).

Figure A picture of a white rectangle on a dark background is first taken

Once we have one line, we can then remove it from the image and rerun the algorithm again. The following figures show the three additional reruns of the algorithm. We can see that it begins to detect the next largest feature edge in the image. By the end of this process, we are able to detect the rectangle itself on the image thanks to this form of feature detection algorithm. We can see below, that the algorithm correctly builds on the next largest possible feature edge in the image.

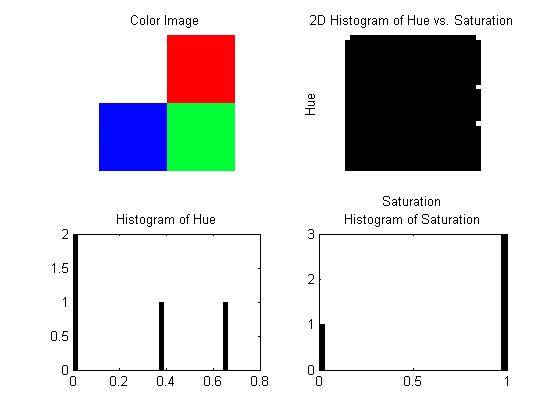


**2. Playing with Color**

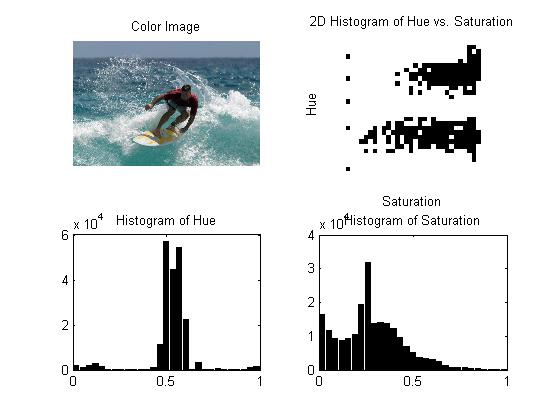
To compute the HSV (hue-saturation-value), a specific formulaic process was used, as follows:

Where V is the value, C is chroma, S is saturation, and H is hue. This was inside of a forloop where we iterate over each color space of the image (Red, Green, Blue) over each pixel. To test our solution, we first check input a pure red color, and check that the output hue matrix is all zero. Which is correct since red is at the 0th degree of the HSV model. In addition, white color has a zeroed saturation matrix and black has zeroed value matrix.

Next we were tasked with developing a 2D histogram; this essentially builds on the previous task with some additional plotting. Below we see our first test image that shows correct output in our standalone hue and saturation histograms.

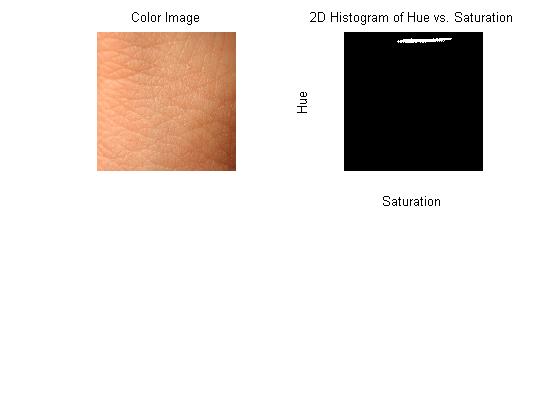


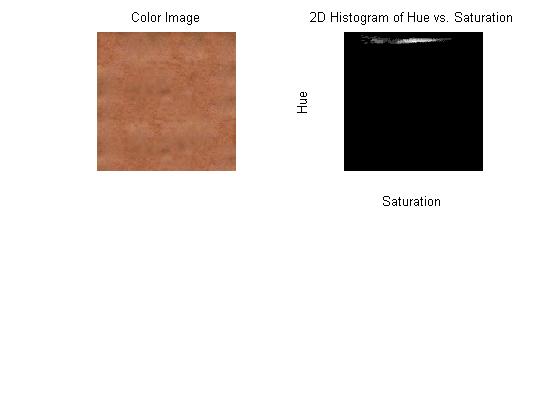
To demonstrate our correct 2D histogram implementation, take the following example. Whereby we see an image with a lot of blues, meaning a lot of midpoint hues and we see in the 2D histogram most of the white pixels are in the horizontal middle of the graph. To further illustrate the point, we see a lot of low saturation points, these show clearly on the 2D histogram too. Note: white pixels are votes in the 2D histogram.

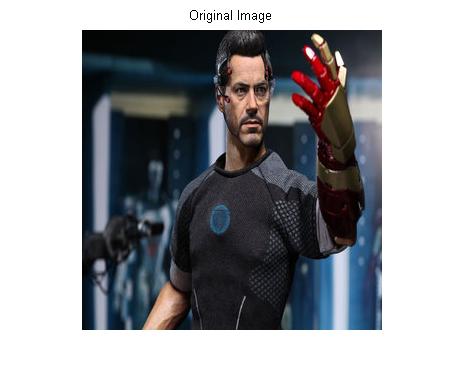


In the final tasks we needed to taken a series of skin-image histograms and compare it with another image that has some skin too. Essentially we are highlighting skin color that is the same as the skin images of the histogram. The problem proved to be difficult due to the ambiguity of the question, however, using the hue/saturation histogram one can deduce the color of skin to be highlighted. To improve our results, one can smoothen the image using a Gaussian filter to allow for better balance of the image and the skin. Another way to improve results is to collect a database of skin patches from different images, this would allow us to find skin under different light conditions. Then we can deduce a more suitable color space for skin detection.

Below we see a small but clear example of the power of such a dataset. The first figure shows a 2D histogram of just one sample skin image. As we added more and more skin images, we see the 2D histogram grow larger in width as it covers more saturations of skin and a bit of height in hue (skins aren’t that diverse in color after all!). In the end, we found eight different skin patches to be good enough for our experiments.









**Conclusion**

**User Manual**

All implementations were done using MATLAB 2013.

**1. Detecting Lines**

Open p1a.m and set D, N, F. Then run it.

**2. Playing with Color**

For part (1), run the rgb\_to\_hsv.m. You will need to input an image path.

For part (2), run the p2.m. You will need an image path and a histogram, you can also just use zeros(30).

For part (3), run the p3.m.

**References**

"MATLAB and Octave Functions for Computer Vision and Image Processing." Peter's Functions for Computer Vision. N.p., n.d. Web. 5 Oct. 2014. <http://www.csse.uwa.edu.au/~pk/Research/MatlabFns/>.

"HSL and HSV." Wikipedia. Wikimedia Foundation, 10 Apr. 2014. Web. 5 Oct. 2014. <http://en.wikipedia.org/wiki/HSL\_and\_HSV>.

"Edge detection." Wikimedia Foundation, 14 Sept. 2014. Web. 5 Oct. 2014. <http://en.wikipedia.org/wiki/Edge\_detection>.