CSE 400: Image and Video Processing
Assignment 2: Edge Detection
Geri Madanguit

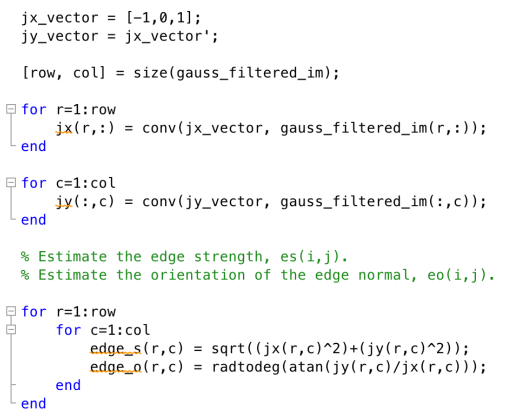
In this assignment, we handled edge detection, specifically the canny edge detector. In the canny edge detection algorithm, the implementation is through three fundamental steps: Canny Enhancer, Non-Max Suppression, and Hysteresis Thresh. Moving through the course, I have already gained an understanding of noise filtering. With image smoothing, it is possible to move on to edge detection. This report will describe my thought process and how I problem solved through the elements of the Canny edge detection algorithm. This case assignment makes use of the 3 images below:

Edge detection starts with the assumption that the noise is Gaussian and suppresses it as much as possible, without destroying the true edges. Then the filter needs to be designed to respond to the edges positively. The output should be large at the edge pixels and low everywhere else, so that the edges can be located as the local maxima in the filter's output. After enhancing the edges, the last part of the edge detection process is localizing the edges. There is a decision about which local maxima are edges and which are just caused by noise. This part of the process includes thinning wide edges to 1-pixel width (a.k.a. NONMAX_SUPPRESSION algorithm) and establishing the minimum value to declare a local maxima as an edge (a.k.a. HYSTERESIS_THRESH algorithm).

Canny Enhancer

In this algorithm, I convolved the smoothed image with a 1D vector [-1, 0,1] to obtain the gradient component Jx and then convolved the same image with the transpose vector for the Jy gradient component. These variables are important for the Canny Enhancer to get the edge strength, denoted by $e_s(i,j)$ and the orientation of the edge normal, denoted by $e_o(i,j)$.

Their computation is shown in Fig 1.



```
jx_vector = [-1,0,1];
jy_vector = jx_vector';

[row, col] = size(gauss_filtered_im);

for r=1:row
    jx(r,:) = conv(jx_vector, gauss_filtered_im(r,:));
end

for c=1:col
    jy(:,c) = conv(jy_vector, gauss_filtered_im(:,c));
end

% Estimate the edge strength, es(i,j).
% Estimate the orientation of the edge normal, eo(i,j).

for r=1:row
    for c=1:col
        edge_s(r,c) = sqrt((jx(r,c)^2)+(jy(r,c)^2));
        edge_o(r,c) = radtodeg(atan(jy(r,c)/jx(r,c)));
    end
end
```

← Fig. 0.

$$e_s(i,j) = \sqrt{J_x^2(i,j) + J_y^2(i,j)}$$

$$e_o(i,j) = \arctan \frac{J_y}{J_x}$$

Fig. 1

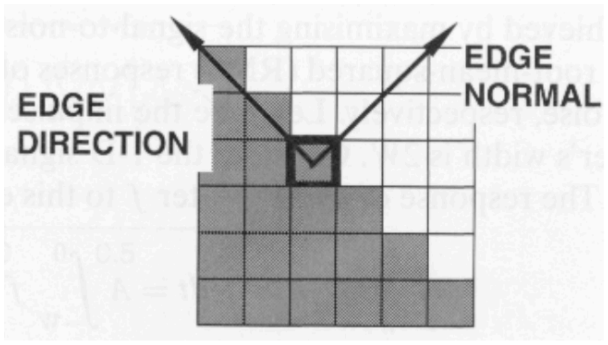


Fig. 2

The edge normal is a unit vector in the direction of maximum intensity change. The edge direction is a unit vector to perpendicular to the edge normal. The edge position of center is the image position at which the edge is located. The edge strength is related to the local image contrast along the normal, the magnitude of the gradient provides this information. All these terms signify important detectors for edges. The values of intensity are directly affected and changed within the environment of the image. For example, things like depth, surface color, and texture, or surface orientation and reflection of light all come into play. Edges can be categorized into four different intensity profiles: Step edge, Ramp edge, Ridge edge, and Roof edge.

Non-max Suppression

This part of the algorithm was more challenging to think through. When getting the edge orientation values through the inverse tangent, I had a lot of problems with the output values with approximation of direction through the NONMAX_SUPPRESSION algorithm. I realized that the output of the inverse tangent was spitting out values in radians and I was making test cases in degrees, therefore rendering my code obsolete. Fixing this was an easy fix by using a matlab function radtodeg() that translated my output to degrees. Another issue I faced with was that I assumed when the degrees were negative, that I just needed the absolute value to test with. That was not the case and instead of -45 degrees corresponding to 45 degrees, it actually computed to 135 degrees in the interest of finding neighboring pixels later in the algorithm. Although the values of the edge orientation aren't exactly 0, 45, 90, or 135 degrees exactly, the algorithm poses a range of values for the edge orientation to categorize into these specific angles.

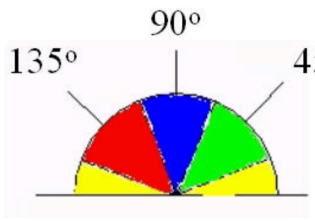


Fig. 3

For specificity, ant edge direction falling in the **yellow range**, (0 to 22.5 and 157.5 to 180 degrees) is set to 0 degrees. Any edge direction falling in the **green range** (22.5 to 67.5

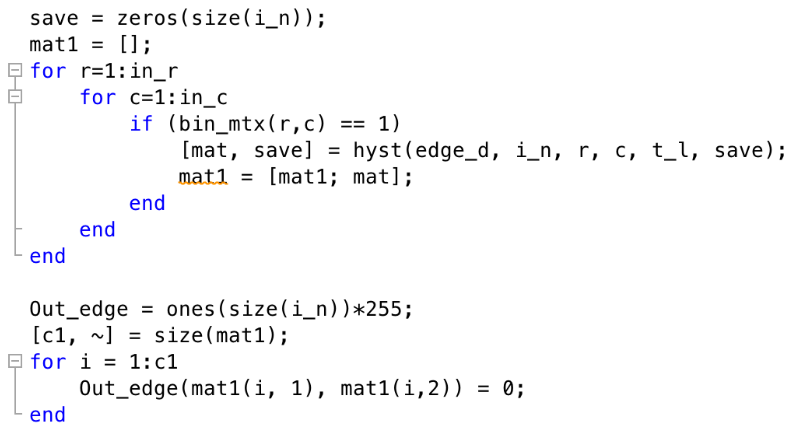
degrees) is set to 45 degrees. Any edge direction falling in the **blue range** (67.5 to 112.5 degrees) is set to 90 degrees. And finally, any edge direction falling within the **red range** (112.5 to 157.5 degrees) is set to 135 degrees. This range is depicted in Fig. 3.

The hardest part to code was considering the physical bounds of the image, including corners and edges. Writing a lot of test code was very meticulous, but it wasn't a waste for I got the output I desired.

The last part was suppressing the value of the pixels in the edge strength image if the neighbors along the categorized edge direction were greater.

Hysteresis Thresh

The last part of this algorithm was connecting all the points of the edge, which includes edge tracking and finding chains in connected contours. This process was very tedious and required a higher-level of coding. So, a lot of this code required more than a thinking process. To implement this algorithm, I used a recursive function that returned the visited points of the algorithm and saved a list of the locations of all points in the connected contour found.



```
save = zeros(size(i_n));
mat1 = [];
for r=1:in_r
    for c=1:in_c
        if (bin_mtx(r,c) == 1)
            [mat, save] = hyst(edge_d, i_n, r, c, t_l, save);
            mat1 = [mat1; mat];
        end
    end
end

Out_edge = ones(size(i_n))*255;
[c1, ~] = size(mat1);
for i = 1:c1
    Out_edge(mat1(i, 1), mat1(i,2)) = 0;
end
```

Fig. 4

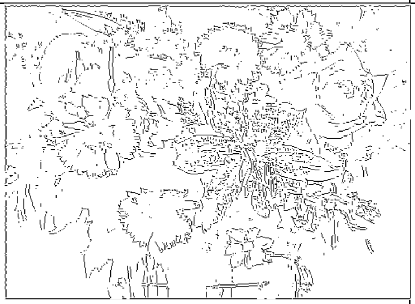
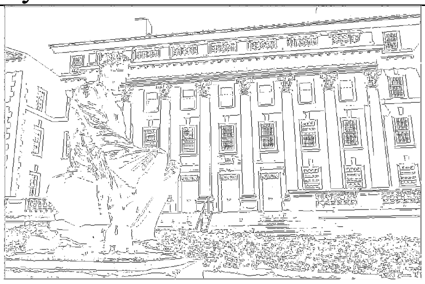
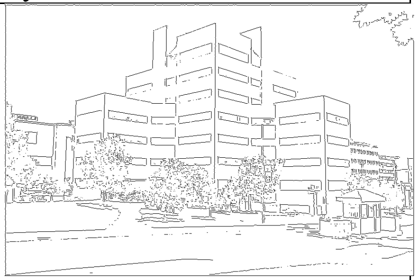
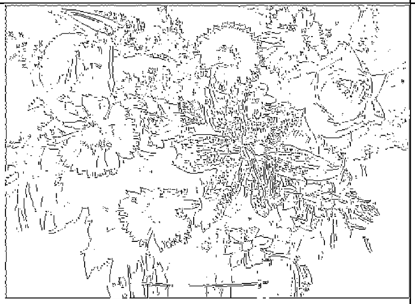
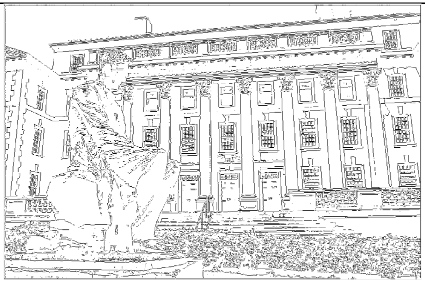
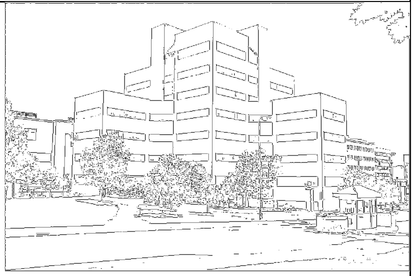
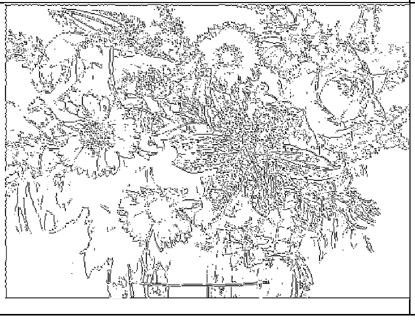
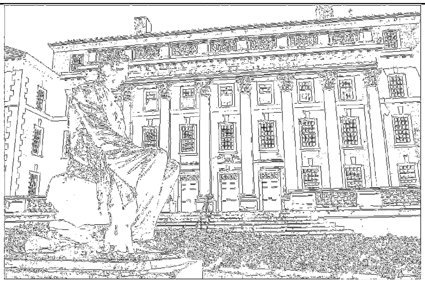
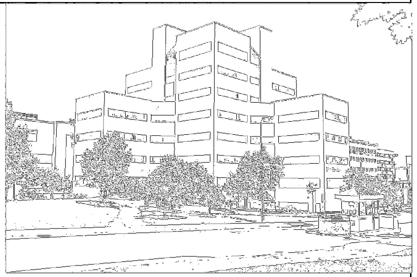
In my recursive function, called 'hyst', the parameters I took in were the edge orientation image, the 1-pixel wide image (output from the non-max suppression algorithm), the coordinates for where the point that in the image that has a greater edge strength value than the set threshold value, the low threshold value, and a matrix of values of which coordinates were visited. What is returned in my recursive function is a 2-column matrix that contains all the points along the edge contour.

In the specific figure, I go through a loop through a binary matrix I previously initialized with values that contains 0's, and 1's if the edge strength at the specific coordinate surpasses the high threshold value. When going through the loop, a conditional statement is passed when reaching 1, and once reached is able to call the 'hyst' recursive function.

When printing out the final image, I decided to draw the edge with 0 (black) values and have the background as a 255 value (white). Having the converse would be the same affect. It was a personal preference for me to have the contrast of white background and black edge contour.

The best thresholds I had that best looped for all images was from 2 to 10. I noticed that the larger my standard deviation, the more edge localization were applied. I found the best combination of low threshold and high threshold was usually high thresh = 2 * low thresh.

<u>Comparison Between Test Images with low threshold = 10, high threshold = 2</u>

Gaussian	Flowers	Syracuse 1	Syracuse 2
Sig = .5			
Sig = 1			
Sig = 1.5			

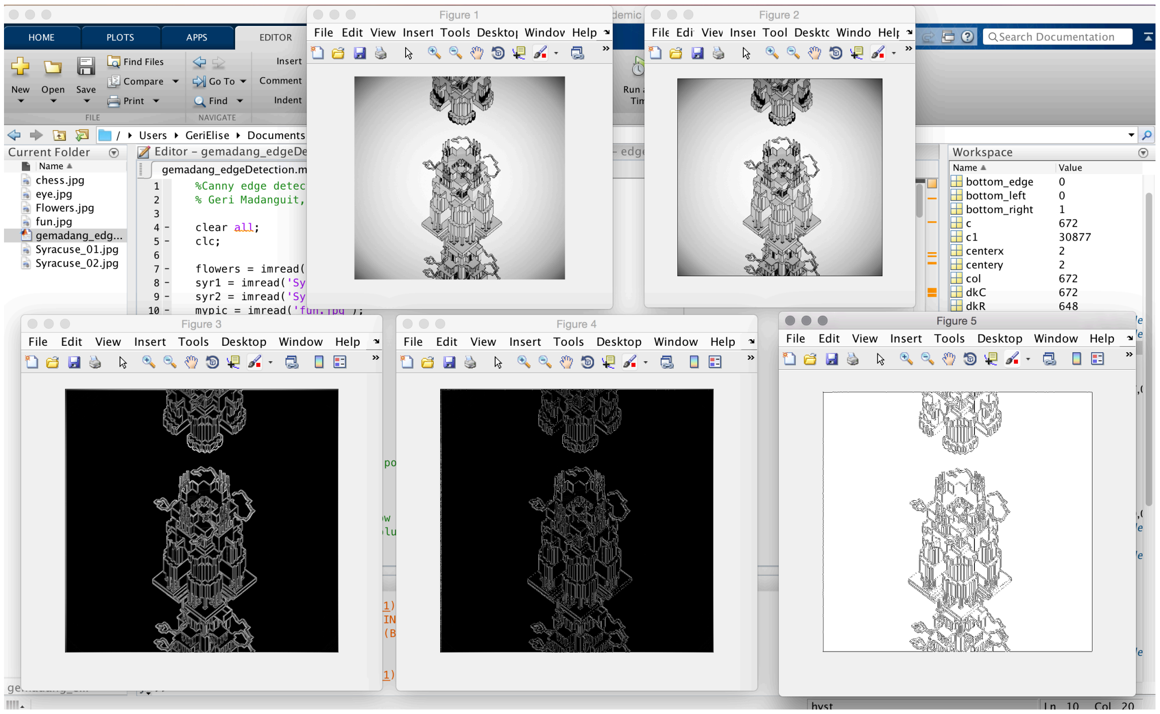
<u>Closing Remarks</u>

Through this assignment, I had a much more difficult time thinking through the problems because of the learning curve of the algorithm from the previous assignment. I learned to just draw out the problem and work through it and try to go through the code line by line while thinking of all possible cases. I learned that for optimal edge detection you need good detection, good localization, and a single-response constraint. Good detection includes minimizing the probability of false positives, as well as false negatives (missing real edges). Good localization defines coming as close as possible to the true edges. Lastly, single response constraint returns one point for each true edge point (greater than high threshold value). This means to minimize the number of local maxima around the true edge (created by noise).

Some practical issues faced within this assignment were that the differential masks act as high-pass filters, which tend to amplify noise. To reduce the effects, the image needs to be smoothed first with a low-pass filter. So this results in the noise suppression-localization tradeoff. That means where there is a larger filter, reducing noise, it worsens localization and vice versa. I also observed that the larger the image, the longer it took for the code to spit out the image, probably due to the number of recursion needs to run because of the increased amount of pixels.

Edge detection is very important for the science of computer vision. The goal of computer vision include the ability to produce a line drawing of a scene from an image of that scene, to extract important features from the edges of an image (corners, lines, curves). These goals can be used for recognition tools.

In addition, below is my personal favorite picture that I tested the algorithm on. What I noticed was that I had to convert the color image to gray-scale even when the image already seemed grey. If I didn't convert the image to gray-scale, my code wouldn't compile on Matlab.



```
%Canny edge detec
% Geri Madanguit,

clear all;
clc;

flowers = imread(
syr1 = imread('Sy
syr2 = imread('Sy
mypic = imread('fun.jpg');
```

Name	Value
bottom_edge	0
bottom_left	0
bottom_right	1
c	672
c1	30877
centerx	2
centery	2
col	672
dkC	672
dkR	648