**The Problem**

The problem for this assignment is to perform edge detection on an image by performing the Canny enhancer, non-maximum suppression, and hysteresis thresholding (in that order). Once hysteresis thresholding is completed, the output image should contain the edges of the original image. To better define the problem, the images below are the data set on which we will be operating.

****

**The Process**

The edge detection process can be split into three stages: Canny enhancement, non-maximum suppression, and hysteresis thresholding.

Canny Enhancement

The purpose of Canny enhancement is to retrieve the image’s edge strength as well as the image’s edge normal orientations. In order to accomplish this, we must first filter the image to remove any noise that may interfere with the edge detection process. We must then take the gradient of the image in both the X and Y directions. The way we take the gradient in the X direction is by convolving the image with the vector [1 0 -1]. This vector subtracts the pixel to the right of the center from the pixel to the left of the center, thus getting the directional change in the intensity/color of these pixels in the X direction. Similarly, we convolve the transpose of [-1 0 1] with the original image in order to get the gradient in the Y direction. Once we have these two convolved images, we can begin building our edge strength and edge orientation images. We begin by iterating through each pixel in both gradient images (since they both have the same dimensions). For each pixel element we visit, we will compute:

One important thing to note is that MATLAB’s inverse tangent function returns radians. Therefore, we must convert the output to degrees by multiplying the result with .

Non-Maximum Suppression

The purpose of non-maximum suppression is to thin the edges in the edge strength image, in order to increase the locality of the edges. The way we accomplish this, is by traversing each pixel in the edge strength image and for each pixel we visit, we retrieve that pixel’s corresponding edge orientation value. We then compare the immediate neighbors’ edge strength values along the edge orientation to our current pixel’s edge strength value. If the current pixel’s edge strength value is less than at least one of the two neighbors, we set the current pixel’s edge strength to zero. Otherwise, we leave the edge strength value in that pixel as it is.

Since there are only eight possible directions we can move in from any one pixel (left, right, up, down, up-left, up-right, down-left, down-right), we must discretize the angles in the edge orientation image (since the output of arc-tangent ranges from -90 to 90). The best angles for us to discretize the values in the edge orientation matrix to would be 0 (left, right), 45 (down-left, up-right), 90 (up, down), and 135 (down-right, up-left). Therefore, any angle between 0 and 22.5 degrees and 157.5 and 180 degrees are all classified as 0 degrees, 22.5 to 67.5 degrees is classified as 45 degrees, 67.5 to 112.5 degrees is classified as 90 degrees, and 112.5 to 157.5 degrees is classified as 135 degrees.

Hysteresis Thresholding

The purpose of hysteresis thresholding is to find and connect edges – until now, we’ve been working with edge pixels. There are several ways to approach this problem. One of the simpler solutions is approaching this in a recursive manner.

The first thing we should do is create a new zeros matrix of the same dimensions as the thinned edge strength image. This way, we can keep track of the pixels we have and have not visited. We then begin traversing the thinned edge strength image and comparing each value against our high threshold. If the pixel’s value is greater than the high threshold, we will then pass these indices for the row and column to a recursive helper function. This helper will find the rest of the connected edge. The first thing we must check is if the current pixel value is zero (in our new image). If it is, we then retrieve the corresponding edge strength and check if the edge strength is greater than the lower threshold. If it is, we then set the pixel value in the new image to a value like 255 (white), check the edge orientation, offset the angle by 90 degrees (to get the edge direction), discretize the angle, and move along the edge direction to our immediate neighbors. This function is recursive because we must move in both directions along the edge direction. This process will stop once we’ve visited all the pixel values in our new image. Once the process does stop, we will have all our connected edges.

For this assignment, there is one extra step – we must fill different edges with different colors, to distinguish them. In order to do this, we create a new RGB image by defining the rows and columns as before with the zeros matrix, but we give it a third dimension as well. We do the process same as before, filling in each pixel with 255 as the default color. The difference, however is that we keep track of the x and y coordinates for each pixel in the edge so that we can see which edges are the longest. Since we can now compare lengths of each edge against all the others, we create vector containers that hold the x and y coordinates for the top five longest edges. Therefore, after we complete the hysteresis thresholding process, we have the five longest edges and can hard code their colors (while all the other edges will be white).

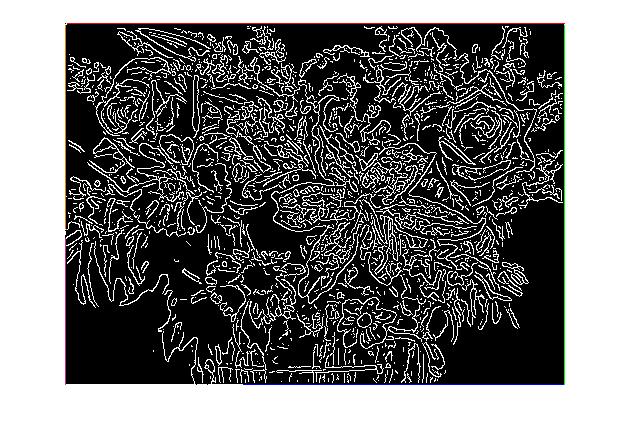
**The Results**

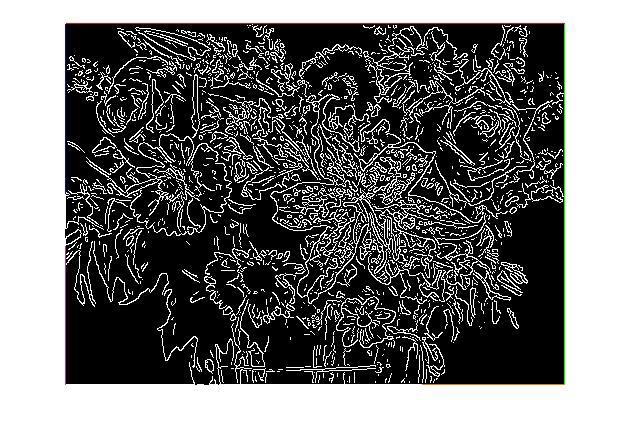
Syracuse\_01

 The first image that we will observe is Syracuse\_01. I performed edge detection on this image inputting two different parameters each time. The image on top has a Gaussian kernel size of 3 with mean 1 standard deviation of 2. The low threshold for the top image is 20 and the high threshold for the top image is 30. The lower image has a Gaussian kernel size of 5, with a mean of 3, and standard deviation 3. The low threshold for the lower image is 10 and the high threshold is 20. The results of these two images are comparable – I suspect this is the case because the ratio of kernel size to mean to standard deviation is not vastly different. My MATLAB code seems to have identified all the edges correctly, but for some reason classifies the bounds of the image as the longest edges and colors them (if you look close enough you’ll see the color). However, in the lower image one of the pillars was colored, meaning it has one of the longest lengths.

Syracuse\_02

The next image that we will observe is Syracuse\_01. I performed edge detection on this image inputting two different parameters each time. The image on top has a Gaussian kernel size of 5 with mean 3 standard deviation of 3. The low threshold for the top image is 6 and the high threshold for the top image is 7. The lower image has a Gaussian kernel size of 3, with a mean of 1, and standard deviation 5. The low threshold for the lower image is 30 and the high threshold is 60. The reason I chose these numbers (for the second image) is because I wanted to show what happens when the thresholds are too high and the standard deviation is also high. Clearly the first image (of the two) is better since the thresholds are proportional to the amount of smoothing that was done for that image. As in Syracuse\_01 the bounds of the image have been classified as the longest edges and, therefore, have been colored.

Flowers

The third image that we will observe is Flowers. I performed edge detection on this image inputting two different parameters each time. The image on top has a Gaussian kernel size of 5 with mean 1 standard deviation of 2. The low threshold for the top image is 5 and the high threshold for the top image is 7. The lower image has a Gaussian kernel size of 3, with a mean of 1, and standard deviation 3. The low threshold for the lower image is 10 and the high threshold is 15. The reason I chose these numbers is because (for the most part) they’re similar to the numbers that were used in Syracuse\_01 and those outputs turned out good. There isn’t a big difference between the first of these two images and the second. However, looking at the two, there is a few more edges in the first image, that aren’t present in the second image. This is most likely because the thresholds were higher when processing the second image. Therefore, not as many edge pixels passed the test when searching for edges. Again, like the other two images we’ve discussed so far, the bounds of the image have been classified as the longest edges and, therefore, have been colored.

Kakashi

The last image that we will observe is an image of Kakashi Hatake. I performed edge detection on this image inputting two different parameters each time. The image on top has a Gaussian kernel size of 5 with mean 3 standard deviation of 5. The low threshold for the top image is 10 and the high threshold for the top image is 11. The lower image has a Gaussian kernel size of 5, with a mean of 3, and standard deviation 6. The low threshold for the lower image is 15 and the high threshold is 20. Immediately there is a clear difference between the two images. The same phenomenon is occurring here as it did in the Syracuse\_02 image. In the Syracuse\_02 image, the second of the two images was smoothed more than the first of the two. This coupled with higher thresholds meant that there were fewer edges detected. As we can see here, the bottom of the face mask is not defined in the second Kakashi image, as was defined in the first Kakashi image. Therefore, the first of the two Kakashi images is better than the second of the two Kakashi images. To state this again, the bounds of the Kakashi image were found to be the longest edges and hence were colored.

Concluding Remarks

While analyzing the outputs of my edge detection function, I found that the greater the kernel size/mean/standard deviation, the less the edges are defined and so there are fewer edges in after running edge detection. Therefore, the more we smooth the image, the lower the thresholds have to be to achieve the same output as an image that has not been smoothed as much.