PLEASE NOTE! I was granted a 24-hour deadline extension by Professor Wu.
I cc'd the TAs in the email conversation.

For my results, you can open ***main.ipynb*** via Jupyter Notebook. You will see everything laid out very neatly. You can run any of the functions available, they are pretty self-explanatory, but the final function is canny(), which takes in 4 parameters: the file name, the desired kernel size, the sigma value, and the percentage of non-edge. An example function is the following:

Image.fromarray(canny('lena.bmp', 5, 1.75, 0.8))

(where 'lena.bmp' is the file name, 5 is the kernel size, 1.75 is sigma, and 0.8 is the percentage)
Optionally, manual low threshold and high threshold hold values may be entered in after the percentage parameter. If they are not entered, then the function will assume to use a cumulative distribution function to calculate the Threshold values, as instructed in the homework guidelines.

This week we studied edge detectors, and for our homework, we built a Canny Edge Detector. The Canny Edge Detector was developed by John Canny in 1986 that was designed to detect edges and suppress noise around an image. Previous edge detectors created by Roberts, Sobel or Prewitt contained quite a large amount of noise (I show examples at the end of this essay). Canny is still quite popular today and is a multi-stage algorithm designed to slowly filter out the noise from an image, calculate an intensity gradient, suppress unwanted pixels, and only keep certain edges that meet a desired threshold intensity based off the intensity gradient (a process called hysteresis).

I began this machine problem by converting all images to grayscale and then built a Gaussian kernel. The Gaussian kernel follows the following formula, and can take any odd-sized kernel:

$$H_{ij} = \frac{\exp\left(-\frac{(i-(k+1))^2 + (j-(k+1))^2}{2\sigma^2}\right)}{2\pi\sigma^2}$$

It is an odd-sized kernel because it is a 2d shape of (2k+1)*(2k+1).

This Gaussian kernel is required in order for us to apply a Gaussian filter over the image and smooth (blur) it. Once we convolve the kernel over the image, we got the following result as Figure 1:

Figure 1: Side-by-side comparison of the original image and a gaussian-smoothed image
(settings for this specific filtering used a kernel of size 7 and sigma of 1)

The next step was to calculate an image gradient of the smoothed image and extract both a magnitude and direction of the edge. In order to do this, I used one of the Sobel filters Professor Wu provided in class and calculated the magnitude as the hypotenuse of the convolution of the smoothed image with the Sobel X and Y filters. The theta was simply the arctangent of these two convolutions. Now I ran into a problem. It was necessary for me to create a histogram of the magnitude values so that I can use it for the next step (finding the correct threshold values), but there seems to be an anomaly when trying to find the magnitude when the image is read as a float and when the image is not read as a float. The correct way is by reading the image as a float, as so I got the following histogram (Figure 2).

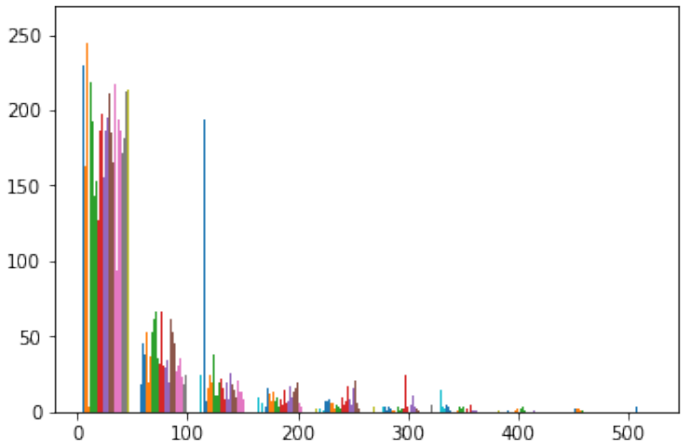


Figure 2: Histogram of this run's magnitude (part of the Image Gradient phase)

The next step looked at taking the result of the histogram and transforming it into a cumulative distribution function. From it, I could then assign a percentage of the distribution to mark as a threshold. Any magnitude values that fell above this line were considered to be in the high-threshold range, and any that fell above half of this threshold were assigned to a different bucket for later evaluation (last step, hysteresis). Of course, we can map the image gradient here too, even though it's not our desired result, but it gives us an idea of what gradient intensity maps look like (Figure 3):

Figure 3: Side-by-side comparison of the original image and an image showing gradient intensity

As you can see, there are thick and thin lines, but we want something else. In order to suppress this further we employed a Nonmaximum Suppression. Essentially, we want to eliminate those pixels that don't lie on the most important edges. An interesting technique here is to bucket or (or quantize) the theta values that we received from our image gradient function. Because theta values are 360˚ in nature, we can group them in 4's, based off their angle. If any pixel had less magnitude than its neighbor, I would remove it. This way I can have very crisp lines of only the highest intensity. The result of this step is illustrated in Figure 4, below.

Figure 4: Side-by-side comparison of the original image and the results of our nonmaxima suppression.

As one can see from the above image, there is still a little bit of an intensity difference along the edges of this image, even though they are crisp. Now comes the importance of our previous thresholding step. I created a 2d-array of zeros of all points in the image, where if a pixel location was previously assigned to the "high threshold" bucket, that corresponding location would receive a 1. I took this newly constructed binary matrix and added on all pixel locations that fell above the "low threshold". This way, I now had some values of 1's, and others of 2's (weak edges and strong edges, respectively). Recursively, if any weak edge was directly connected to a strong edge in an 8-connectivity pattern, it was considered strong, otherwise it was removed from the matrix. The result got me the following image (Figure 5):

Figure 5: Side-by-side comparison of the original image and the results of the hysteresis (edge-linking)

The image doesn't look perfect, so I went ahead and created multiple variations of this image with my Canny Edge Detector using different parameters values for their kernel sizes and sigma values. They're compared below in Figure 6:

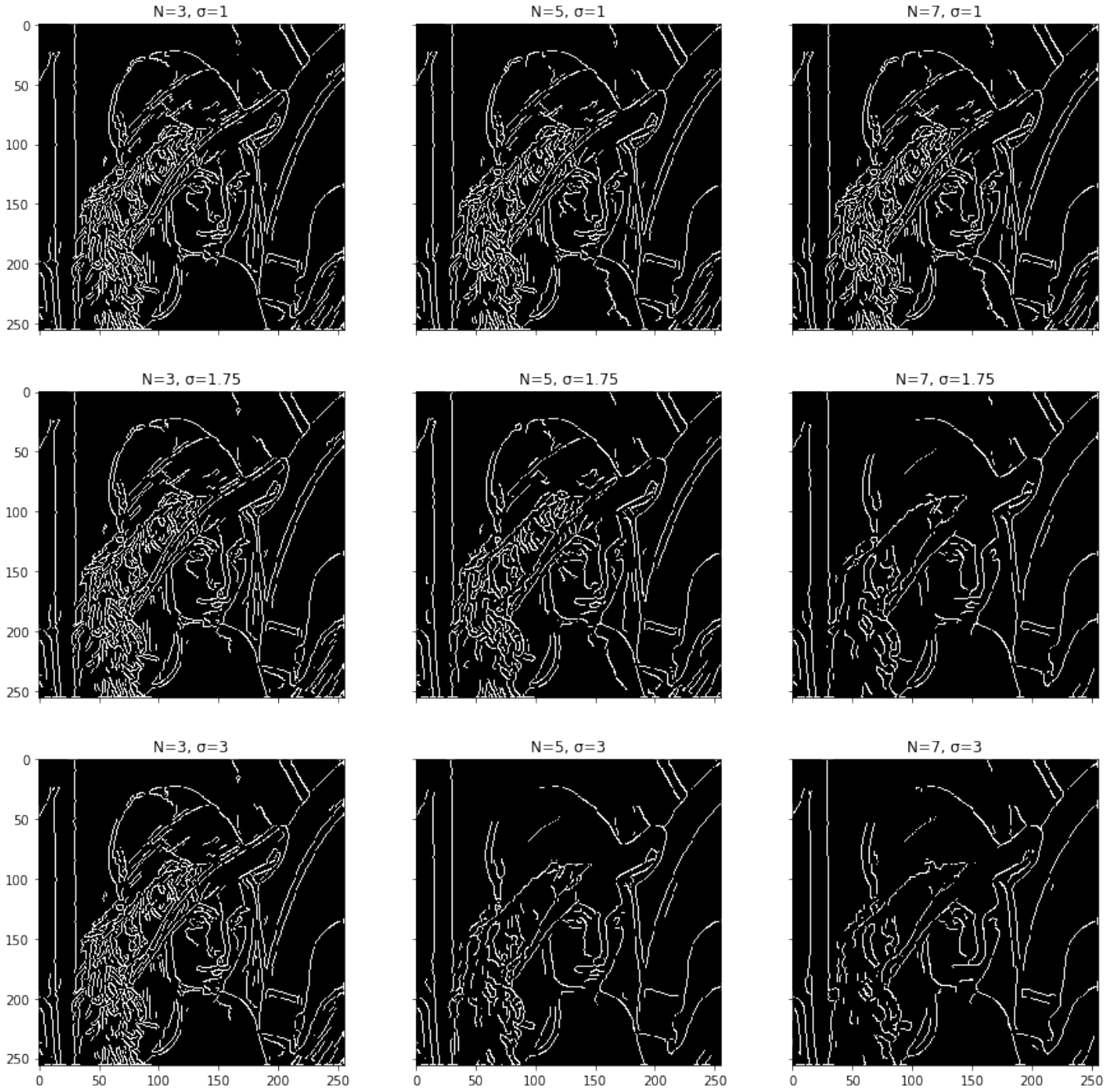


Figure 6: Comparison of different Canny settings.
Kernel size increases from left to right (3, 5, 7). Sigma increases from top to bottom (1, 1.75, 3).

Personally, I think the middle image has a decent enough look to it. Figure 7 below compares their varying percentage parameter that we assigned during our thresholding stage.

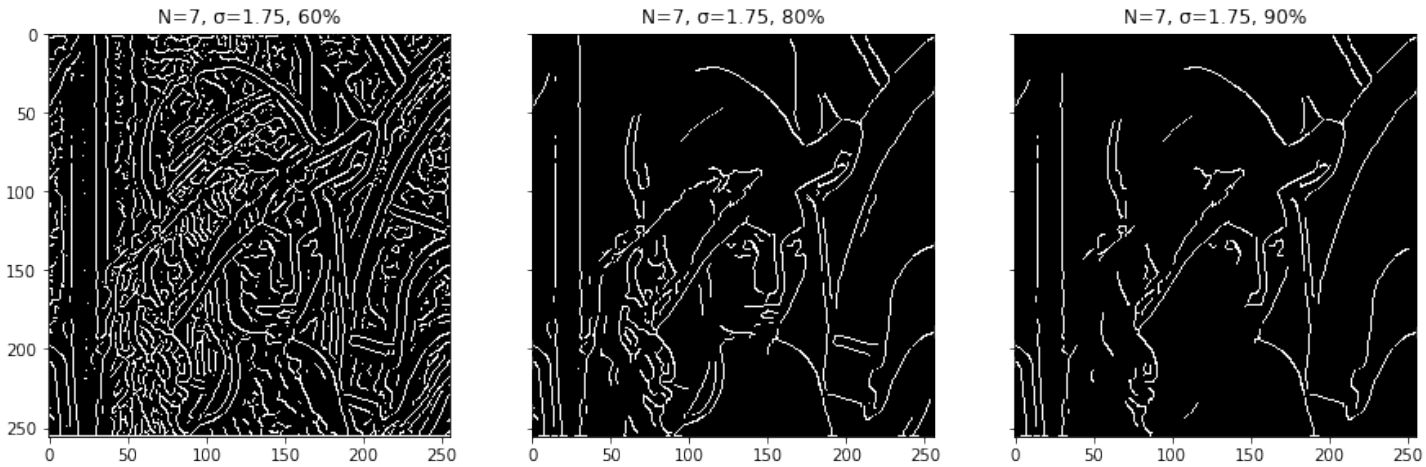
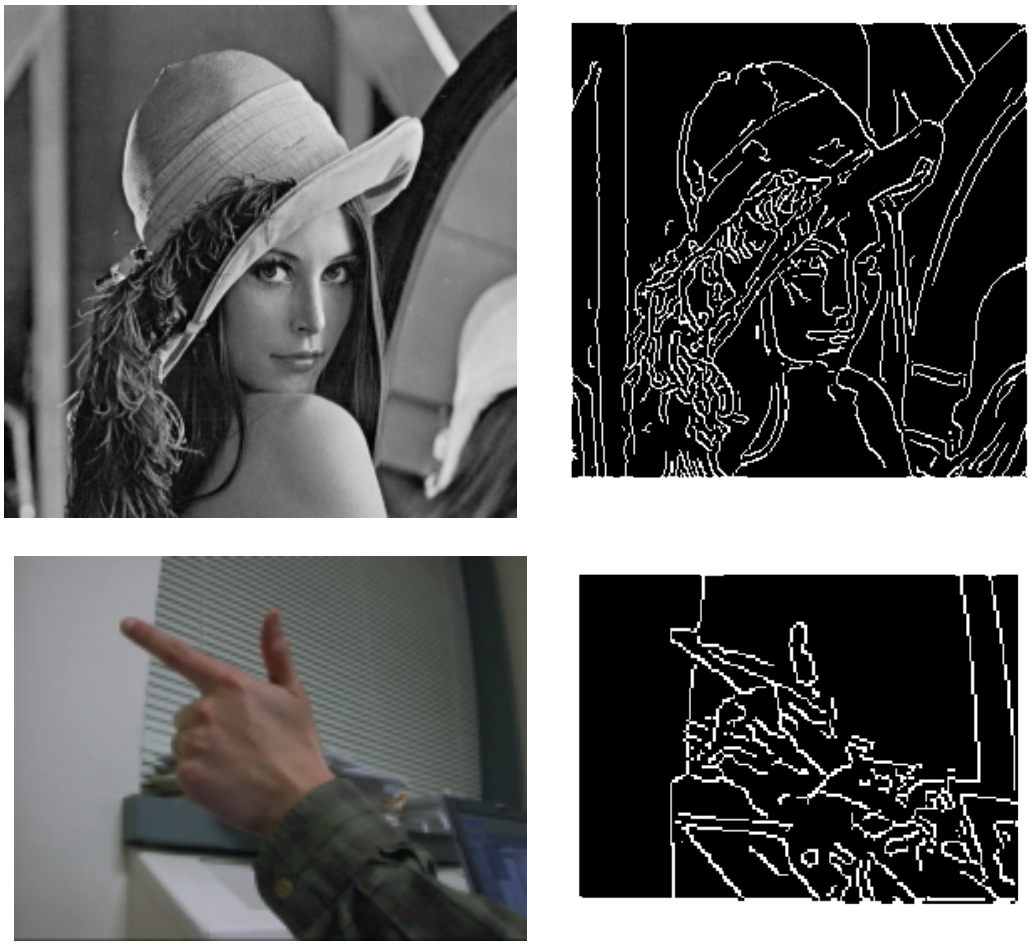
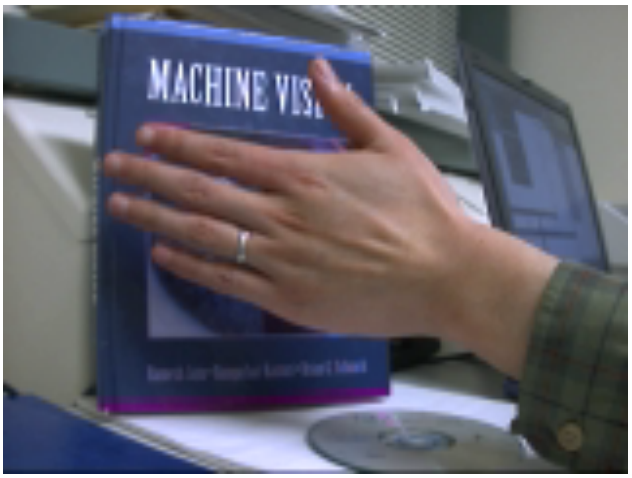
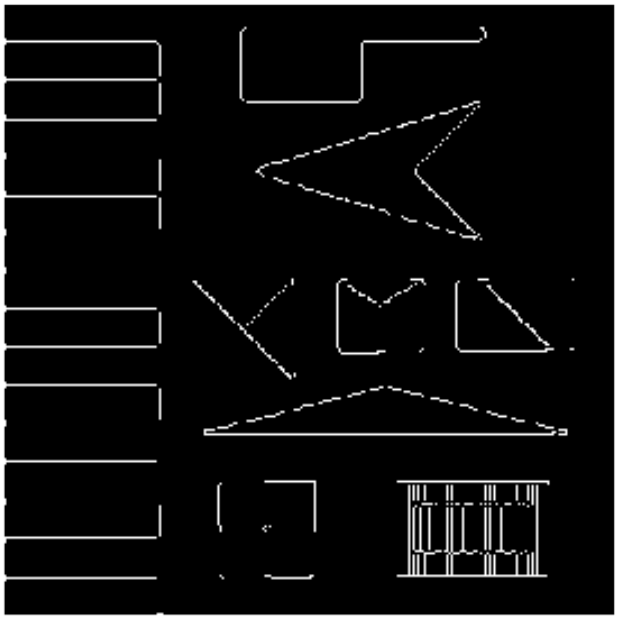


Figure 7: Comparison of different percentage of non-edge
Percentage increases from left to right (60%, 80%, 90%)

As one can see, the percentage makes a large difference, especially as it was determined for a cumulative distribution divided into a small number of bins. If the number of bins had been increased, it would definitely make a difference here. Of course, I ran this Canny Edge Detector program on all the test images provided, as you can see from the following images.




MACHINE VIS



Finally, I make sure to compare the program I've built to other famous edge detectors. This is represented below in Figure 8. As I mentioned earlier, previous edge detectors have been quite noisy, and the Canny Edge Detector really does a great job at detecting edges with low error rates, finding the center of those edges, and then marking it only once and not multiple times over.

Figure 8: A comparison of different edge detectors.
(From left to right: Sobel, my Canny, and Roberts)

While I might personally consider other edge detectors, each serve their own useful purpose and it really does depend on the case. Regardless, Canny Detection is proven to be unique and still popular and useful to this day.