Jason Kayser
Digital Image Processing 4650
jpk2f2

Assignment 3

For the third assignment I utilized python 3.7 alongside opencv2, numpy, scipy, and code from my assignment 2 solution to conduct canny edge detection on the given set of images and display images taken at various steps of the algorithm. The canny edge detection algorithm consists of a number of steps performed on the given image: gaussian smoothing, x gradient calculation, y gradient calculation, gradient magnitude calculation, gradient direction calculation, nonmaximum suppression, and hysteresis thresholding. The image is saved after each step and later displayed together as well as saved in a 'plots' folder.

I began by adding my *utility.py* module from assignment 2 to the project as well as copying my gaussian smoothing algorithm into the module. Then, in *main.py* I read in the given images as grayscale using opencv2's *imread()* function e.g.:

im_lenna = cv2.imread('resources/Lenna.png', 0)

I then created a function *action()* which is called to perform all the steps of the canny edge detection on a given image, and finally called *action()* on each image with my specified sigma and threshold values for the gaussian smoothing and hysteresis thresholding algorithms to respectively use.

The first step in the canny edge detection algorithm is to smooth the image with a gaussian filter. This removes noise from the image and helps minimize detection of false edges. Larger sigma values help reduce false edge detection, however they can also cause the loss of desired features in the image. An example can be seen below using the original "1016" image and the aftermath of being smoothed with a sigma 1, 3, and 5 gaussian as well as their final results (all using identical threshold values):

Figure 1: 1016 Original

Figure 2: 1016 Gaussian Sigma 1

Figure 3: 1016 Gaussian Sigma 3

Figure 4: 1016 Gaussian Sigma 5

Figure 5: 1016 Canny Edge Detection Result with Sigma 1

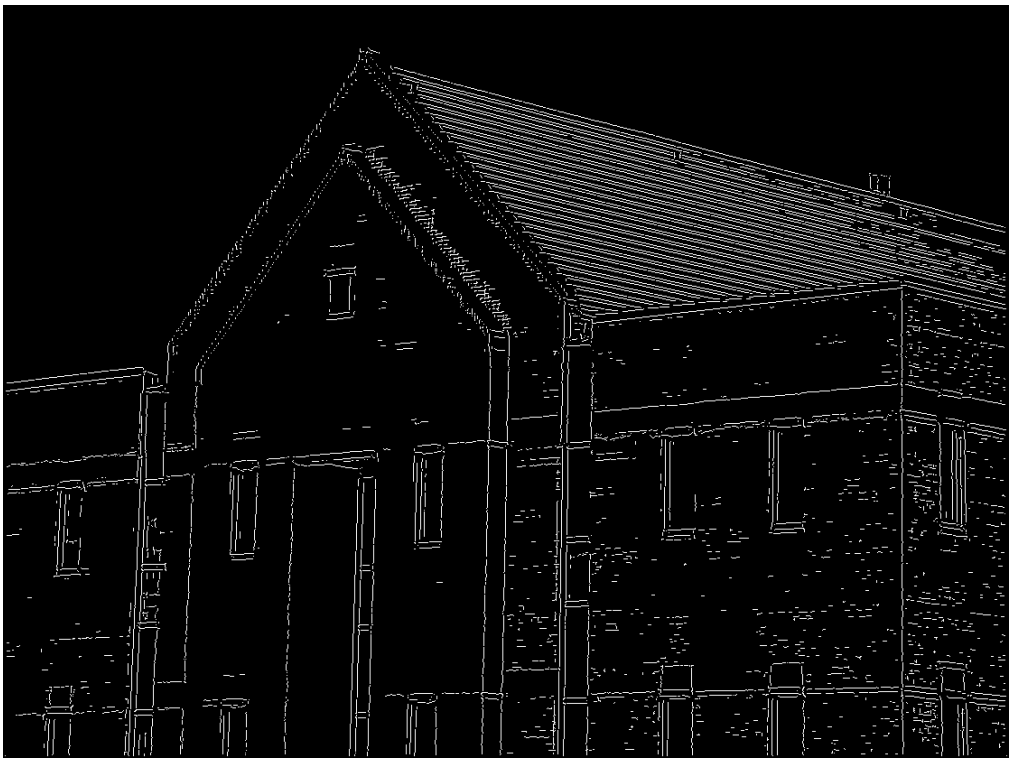


Figure 6: 1016 Canny Edge Detection Result with Sigma 3

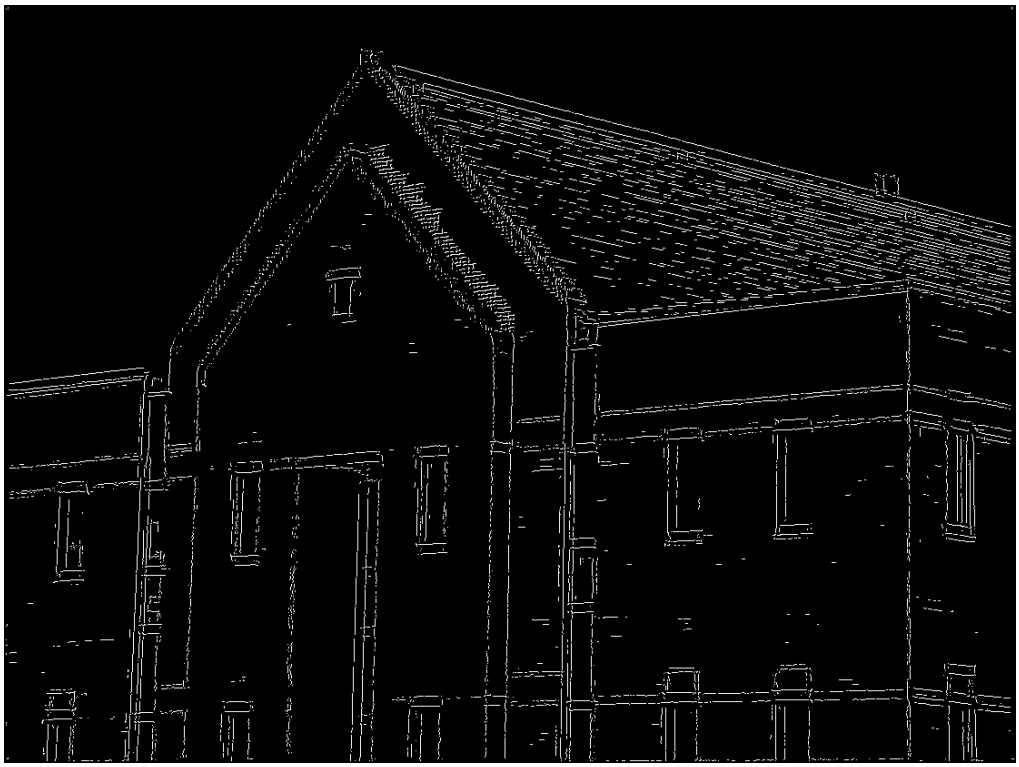


Figure 7: 1016 Canny Edge Detection Result with Sigma 5

As can be seen by comparing the final three images above, increasing sigma values decrease edges created from the texture and noise but reduce features as well.

For the next step, I calculated the gradient of the smoothed image in both the x and y directions as well as the magnitude and direction by calling my Sobel operator algorithm e.g.:

im_fx, im_fy, im_f, im_d = cED.sobel_filter(im_gauss)

which convolves the image with both the x and y Sobel operators:

fx = scipy.signal.convolve2d(im, SOBELX, 'valid')

fy = scipy.signal.convolve2d(im, SOBELY, 'valid')

It then uses the direction gradients to calculate the magnitude and direction:

```
f = np.sqrt(fx ** 2 + fy ** 2)

f = f / np.max(f) * 255

d = np.arctan2(fy, fx)
```

An example of the results can be seen below using the “Lenna” image with a sigma value of 1:

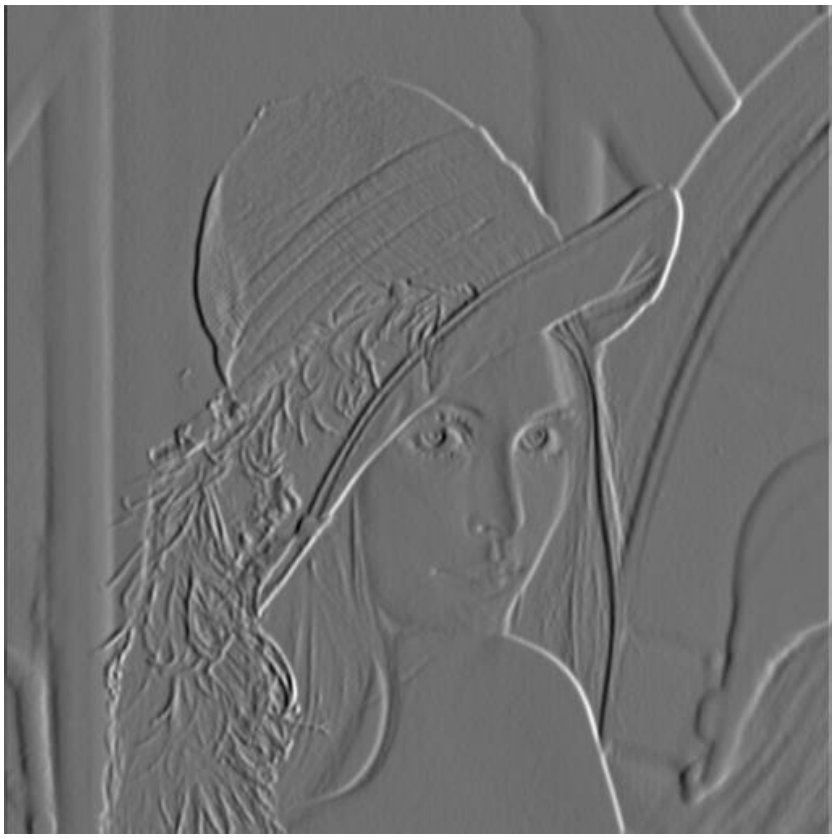


Figure 8: Lenna Fx image with Sigma 1

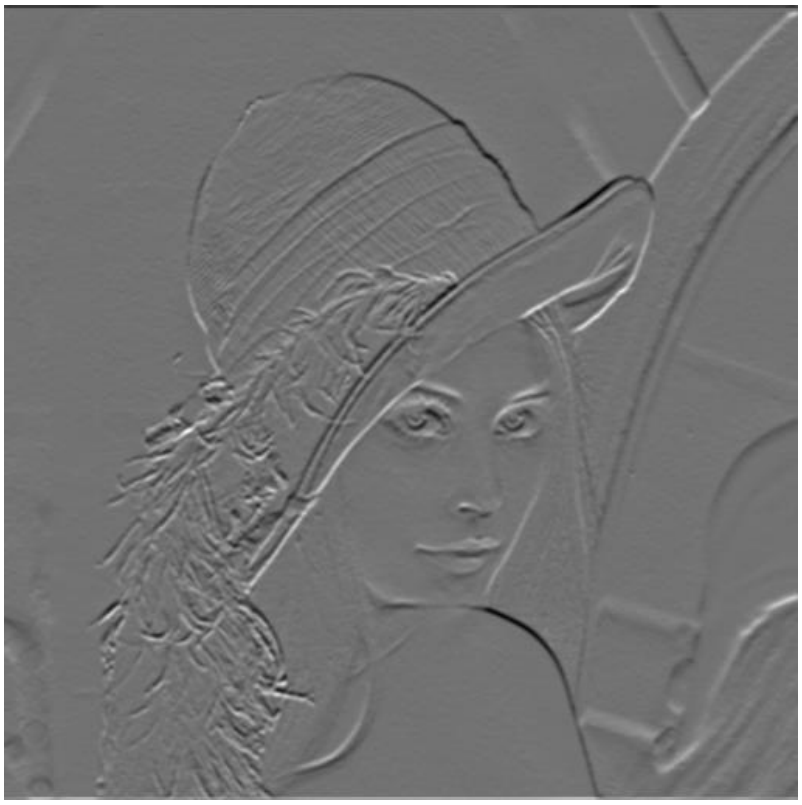


Figure 9: Lenna Fy image with Sigma 1

Figure 10: Lenna F(magnitude) image with Sigma 1

Figure 11: Lenna D(direction) image with Sigma 1

After calculating the gradient values, I performed nonmaximum suppression using the directional array and magnitude array in order to create a new, 'thinned' image. To do so, I called my *nonmax_supress()* function and passed it the directional image and magnitude image. This function begins by converting the directional image values to degrees between +/- 0 and +/- 180 e.g.:

*d = d * 180.0 / np.pi*

I then get the dimensions of the current image for use in the creation of the new 'thin' image as well as for looping through each pixel. Next I create a double nested loop for progressing through each directional value. At each step, I add 180 to the given degree if it's negative in order to convert all values to positive for simpler logic i.e. -45 is converted to +135:

angle = d[j, k] + 180 if d[j, k] < 0 else d[j, k]

Using the angles 0, 45, 90, and 135 the closest one is found for each directional value through the use of 22.5 degree increment comparisons and then the appropriate comparison magnitude values are selected e.g. for angle 0:

f (0 <= angle < 22.5) or (157.5 <= angle <= 180): # 0 degrees or +/- 180

n1 = im[j, k + 1]

n2 = im[j, k - 1]

Finally, the current location's magnitude is compared to its selected directional neighbors and either moved to the corresponding location in the new image or removed from said image based on the outcome e.g.:

if (im[j, k] <= n1) or (im[j, k] <= n2):

i[j, k] = 0

else:

i[j, k] = im[j, k]

Once all the values are process, the new image is returned for further use and display as seen below with the "lenna" image using sigma 1:

Figure 12: Lenna Nonmaximum Suppressed with Sigma 1

At this point, the image is finally ready for its final step – hysteresis thresholding. The thinned image is passed to my *hysteresis()* function along with the desired low and high threshold values. This function once again gets the image dimensions for new image creation and then uses the values to loop through the image and perform the thresholding. The algorithm sets values equal to or less than the low threshold to 0 and at each new pixel that is equal to or greater than the high threshold it sets to 255 and then checks neighboring pixels to set values between the high and low threshold values to 255. All "set" values are set in the new image rather than the old one, allowing for efficient processing without

repeating values and leaving behind unconnected values that were not set to 0. The main double loop can be seen below:

```
for i in range(2, dimensions[0] - 2):
    for j in range(2, dimensions[1] - 2):
        if im[i, j] <= t_l:
            im2[i, j] = 0
        elif im[i, j] >= t_h:
            im2[i, j] = 255
        elif im[i + 1, j] >= t_h or im[i - 1, j] >= t_h or im[i, j + 1] >= t_h or im[i, j - 1] >= t_h \
                or im[i - 1, j - 1] >= t_h or im[i - 1, j + 1] >= t_h or im[i + 1, j + 1] >= t_h \
                or im[i + 1, j - 1] >= t_h:
            im2[i, j] = 255
```

The final result for the previously used “lenna” image with a sigma of 1 and threshold values of 0.1 and 0.03(threshold values are multiplied by 255 to obtain the actual number used in processing) can be seen below:

Figure 13: Lenna after Hysteresis Thresholding with sigma 1 and low and high threshold values of 0.1 and 0.03, respectively

Additional examples on the same image using different threshold values with the same sigma can be seen below:

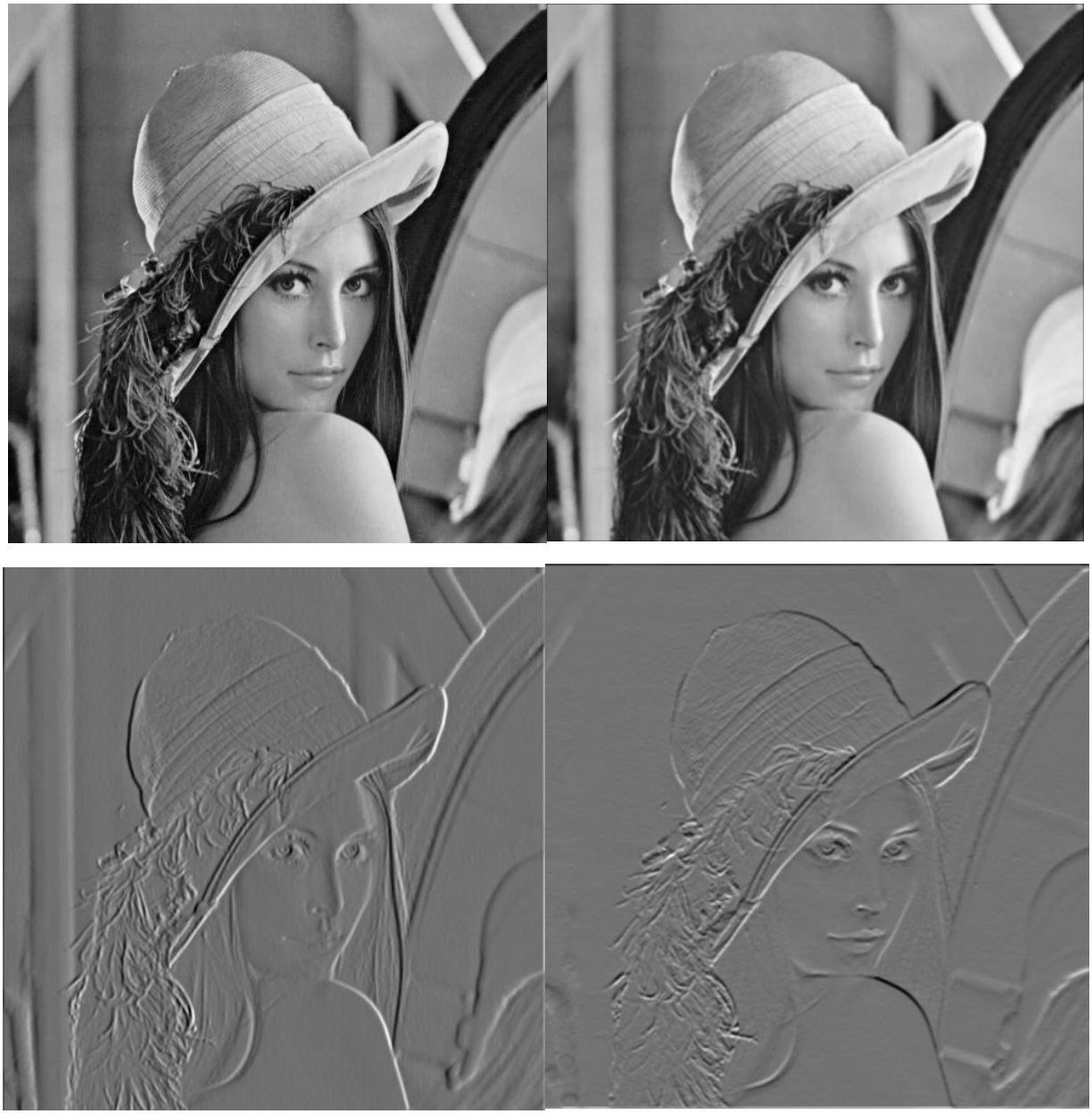
Figure 14: Lenna after Hysteresis Thresholding with sigma 1 and low and high threshold values of 0.2 and 0.1, respectively

Figure 15: Lenna after Hysteresis Thresholding with sigma 1 and low and high threshold values of 0.15 and 0.03, respectively

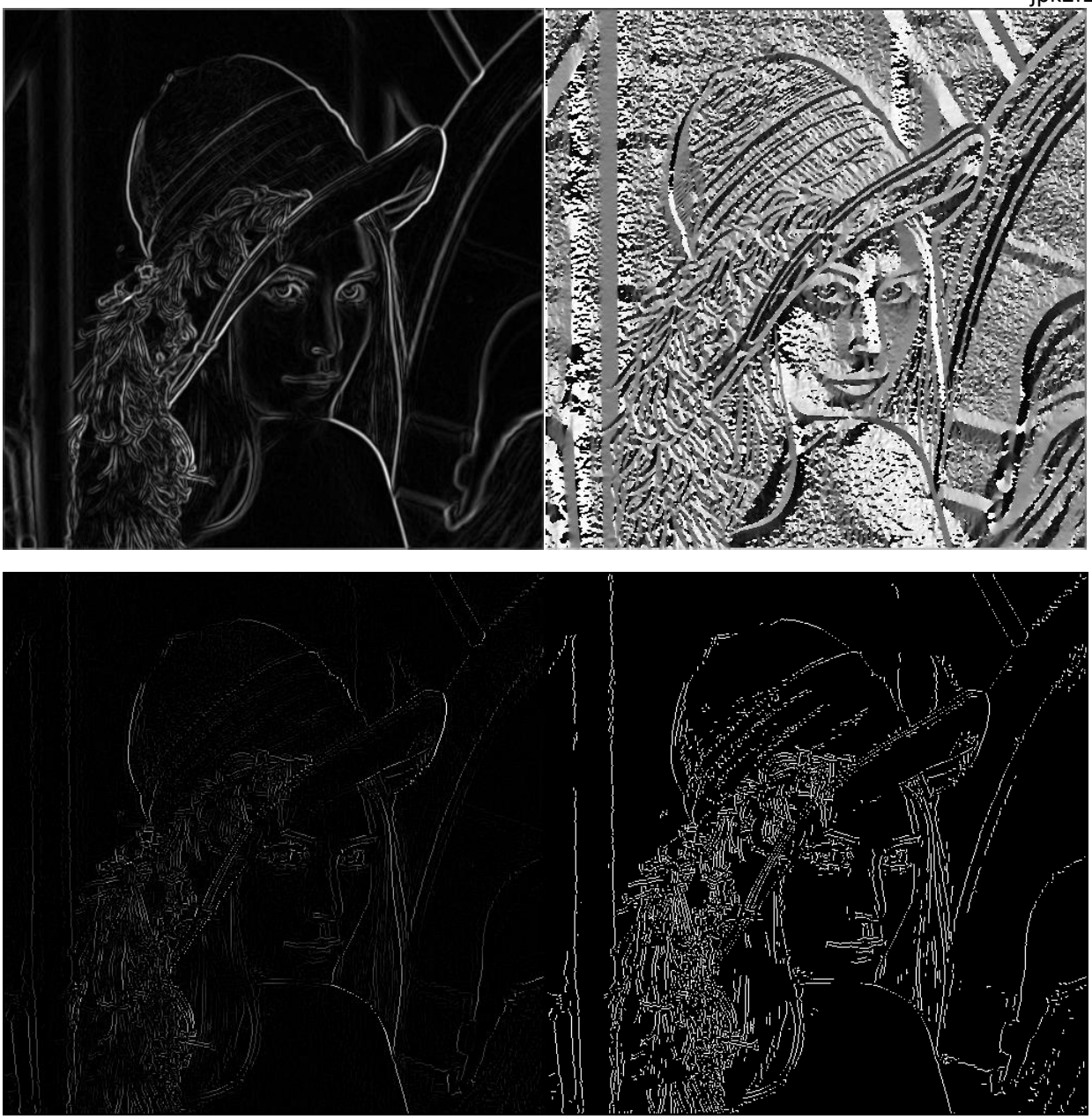
Jason Kayser
Digital Image Processing 4650
jpk2f2

An additional set of images are included below, these are the full set of images at every for “lenna”, “1001(a)”, and “1016”. All were run with identical sigma values of 1 and threshold values of 0.1 and 0.03. For each set, from left to right, top to bottom they are: original, gaussian smoothed, Fx, Fy, F, D, nonmaximum suppressed, hysteresis thresholded(final result).

Lenna

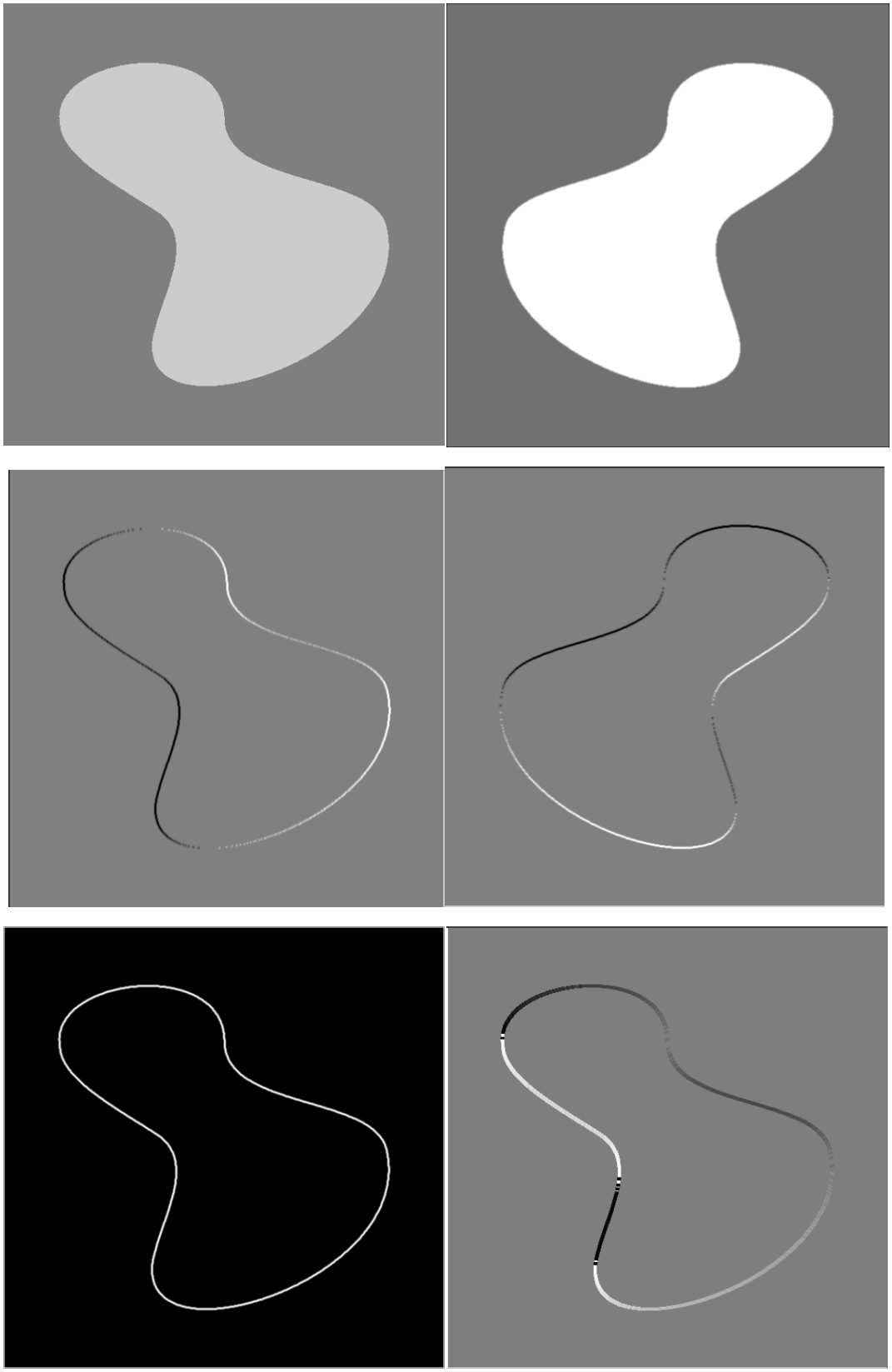


Jason Kayser
Digital Image Processing 4650
jpk2f2

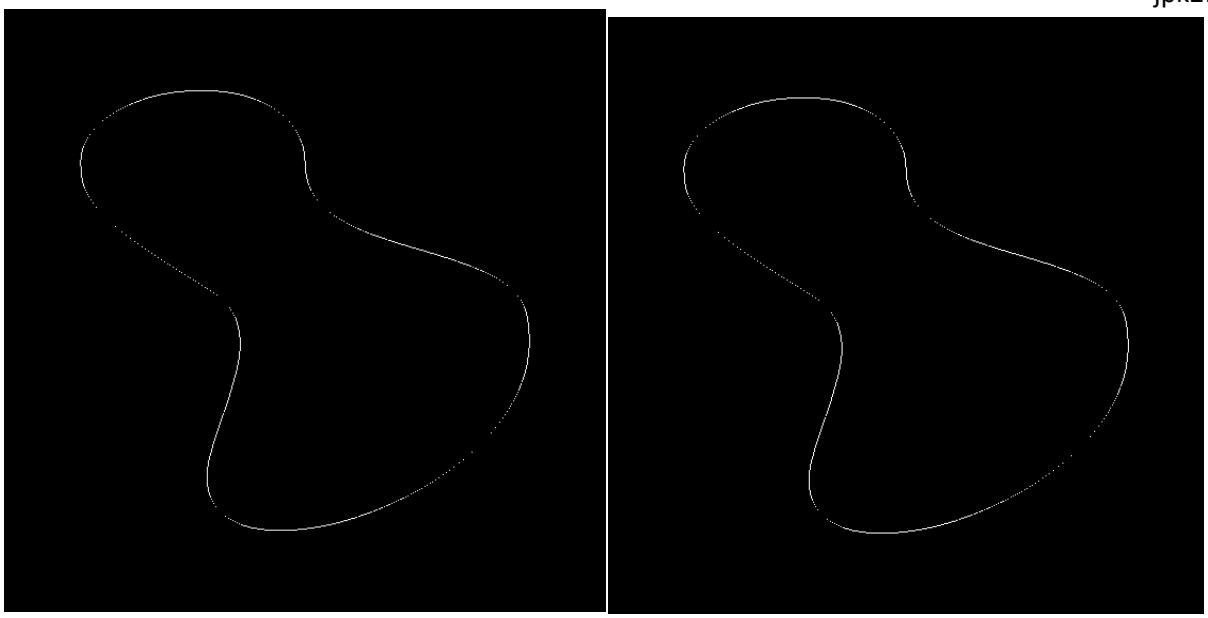


Jason Kayser
Digital Image Processing 4650
jpk2f2

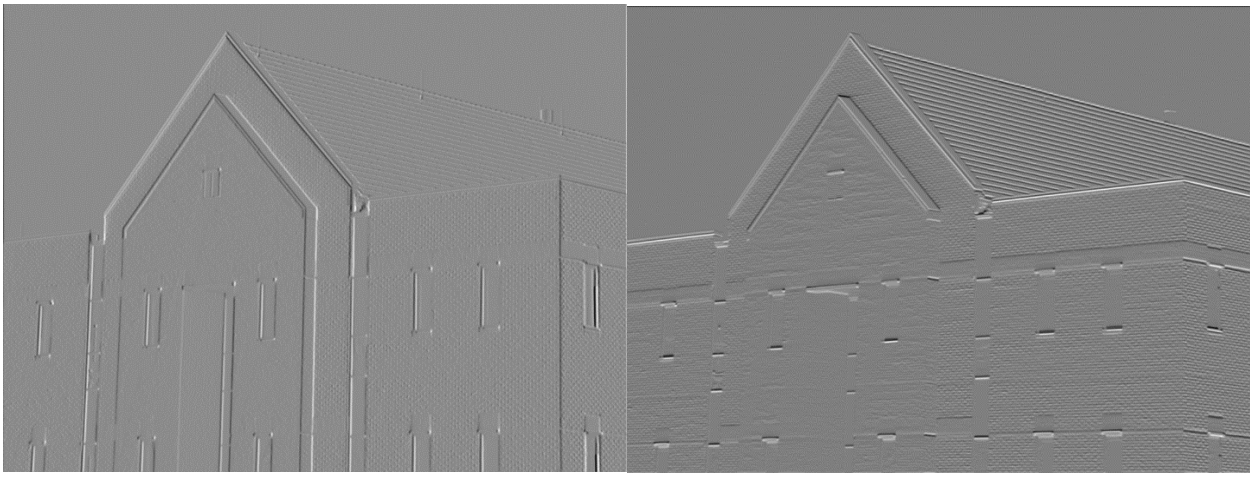
1001(a)



Jason Kayser
Digital Image Processing 4650
jpk2f2



1016



Jason Kayser
Digital Image Processing 4650
jpk2f2