Gaussian Smoothing:

```
def GaussianSmoothing(img_in, k_size, sigma):
    guassKernel = g_kernel(k_size, sigma) # Calculate the gaussain kernel
    result = cv2.filter2D(img_in, cv2.CV_64F, guassKernel) # Uses convolution fro
m cv2; Use CV_64F
    return result

def g_kernel(size, sigma):
    i = j = size
    gauss = np.zeros((i,j))
    i = size//2
    j = size//2

    for x in range (-i, i+1):
        for y in range (-j, j+1):
            # Equation of the 2D Gaussian
            equation = (1/(2 * np.pi * (sigma**2))) * (np.exp(-
(x**2+y**2) / (2*sigma**2)))
            gauss[x+i, y+j] = equation
    return gauss
```

The GaussianSmoothing function calls gaussKernel which then calculates the gaussian kernel via the size of the kernel and the sigma user inputs. The equation is from the lecture slides which will be applied to size of the kernel. This will be passed back to the equation (GaussianSmoothing) and will call the cv2 filter2D. This will use the newly calculated gaussian kernel and instead of -1 for the depth (which will default to uint8), I used cv2.CS_64F to make sure it will return in float 64.

Sobel Operator:

```
def ImageGradient(s):
    sobelGradX = np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]])/8
    sobelGradY = sobelGradX.T # Transpose of X-direction Array

    GradX = cv2.filter2D(s, cv2.CV_64F, sobelGradX)
    GradY = cv2.filter2D(s, cv2.CV_64F, sobelGradY)

    mag = np.hypot(GradX, GradY) # Hypotenuse = sqrt(a^2+b^2)
    mag *= 255.0 / mag.max()
    # mag = mag.astype(np.uint8) # Convert to uint8 to show img

    theta = np.arctan2(GradY, GradX) # arctan2 gives theta between -
pi to pi {Radian!!!}
    theta = np.rad2deg(theta) + 180 # Convert to Degrees between 0 to 360 {Degree
s}
    # theta = theta.astype(np.uint8) # Convert to uint8 to show img

    # print(theta)
    return (mag, theta)
```

This function was not as complex and straight forward. What it does is it takes in the image with the gaussian filter (make sure it is in uint8). Create the gradient x array (from lecture slides). For gradient y array, the matrix of y direction from lecture did not seem to work therefore, I just did a transpose of the x direction matrix. After the creation of the array, we pass each of the array into cv2 filter2D (CV_64F, keeping it as float). There are many ways to calculate magnitude, but I realized that sqrt(x^2+y^2) is the same as finding the hypotenuse. Therefore, I used np.hypotenuse to find the magnitude. Theta in the other had gave me a lot of issues because I never realized that it return radian instead and it is ranged from -pi to pi which is -180 to 180. So, I convert it to degrees and added 180 for the range to become 0 – 360.

Non-Maxima Suppression:

```
def NonmaximaSuppress(mag_image, theta):
    height = mag_image.shape[0]
    width = mag_image.shape[1]
    result = np.zeros(mag_image.shape)

    # Iterate through the pixels
    # Compare neighbor between 16 positions [between each of the 8 sections]
    for x in range (1, width-1):
        for y in range (1, height-1):
            angle = theta[x, y]
            if np.any(angle >= 0) and np.any(theta < 22.5) or np.any(angle >= 157
.5) and np.any(angle < 180): #0 - 22.5
                # Compare left-right
                a = mag_image[x, y-1]
                b = mag_image[x, y+1]
            elif np.any(angle >= 22.5) and np.any(angle < 67.5): #22.5 - 67.5
                # Compare diagonally
                a = mag_image[x-1, y+1]
                b = mag_image[x+1, y-1]
            elif np.any(angle >= 67.5) and np.any(angle < 112.5): #67.5 - 112.5
                # Compare top-bottom
                a = mag_image[x-1, y]
                b = mag_image[x+1, y]
            elif np.any(angle >= 112.5) and np.any(angle < 157.5): #112.5 - 157.5
                # Compare diagonally
                a = mag_image[x+1, y+1]
                b = mag_image[x-1, y-1]

            if np.any(mag_image[x, y] >= a) and np.any(mag_image[x, y] >= b):
                result[x, y] = mag_image[x, y]
    result = result.astype(np.uint8) # Convert to uint8 to show img
    print(result)
    return result
```

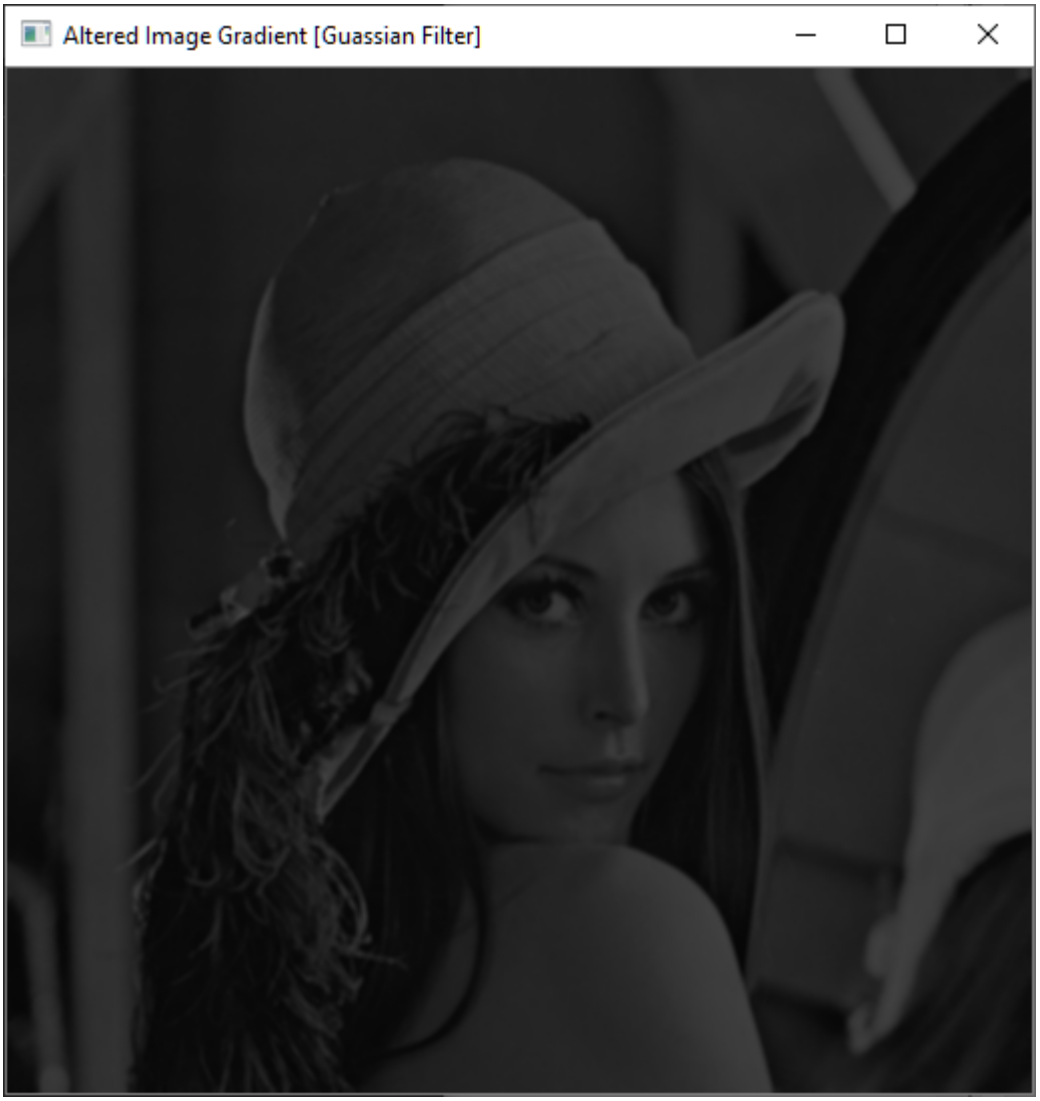
Non-maxima suppression was not the hardest, but I spent the longest on it. This is because I forgot to convert float back to uint8 to show the image, so I kept rewriting the code and using different ways to write the same logic.

First find the height and width of the image and create a container with the same size as the original. Then traverse through all the pixels of the original image and compare from 0 – 180 degrees, since anything over 180 is a repeat. To compare correctly, we split the 8 sections into 16 sections to compare (left-right, top-bottom, diagonally). Since I am only comparing from 0 – 180, which is 360/2, I also divided 16 sections by 2. Therefore, the comparison is (x*180/8) where x is ranged from 1 - 8

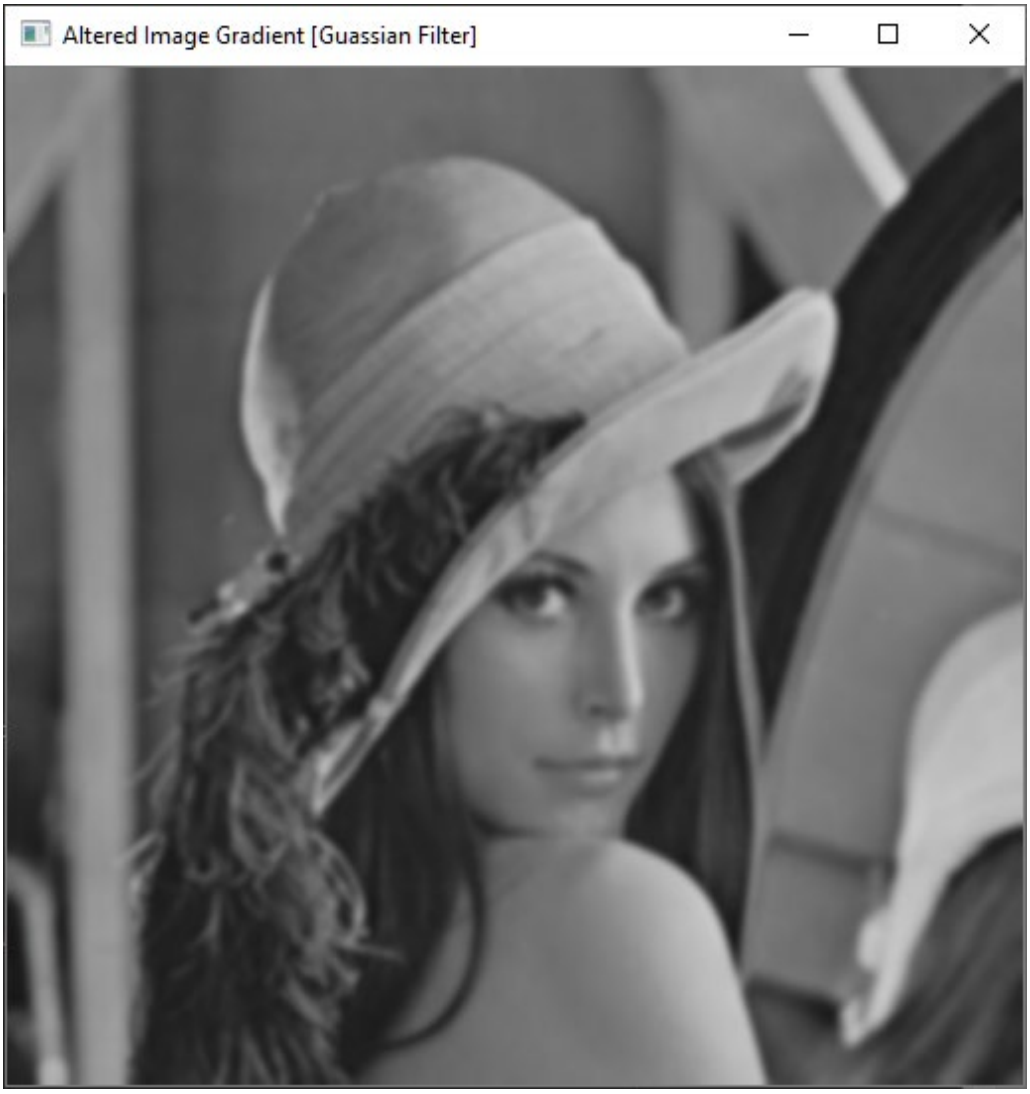
Below you will find comparison of different values for gaussian smoothing and output of non-maxima suppression and theta gradient.

Comparisons:

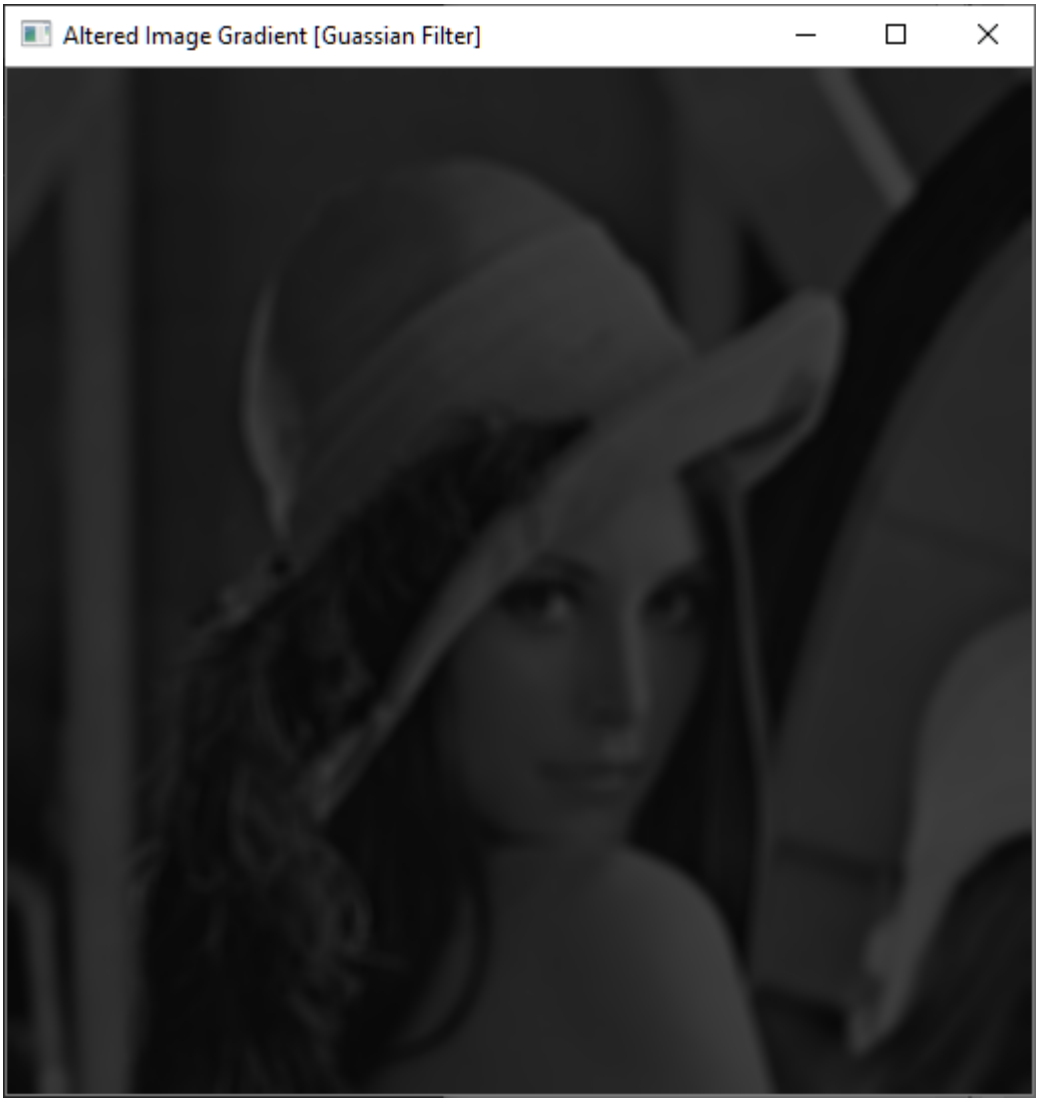
Gaussian Smoothing (K_size = 3, Sigma = 2):



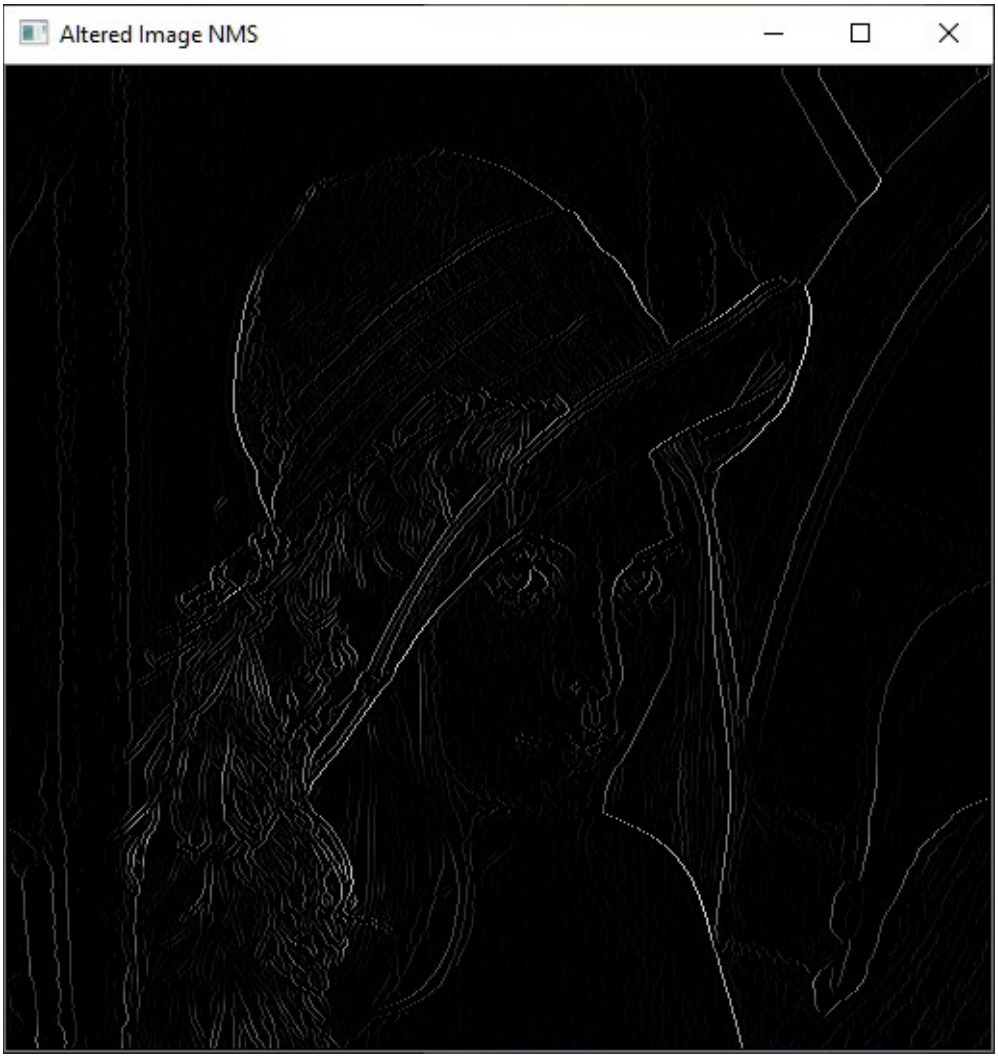
Gaussian Smoothing (K_size = 9, Sigma = 2):



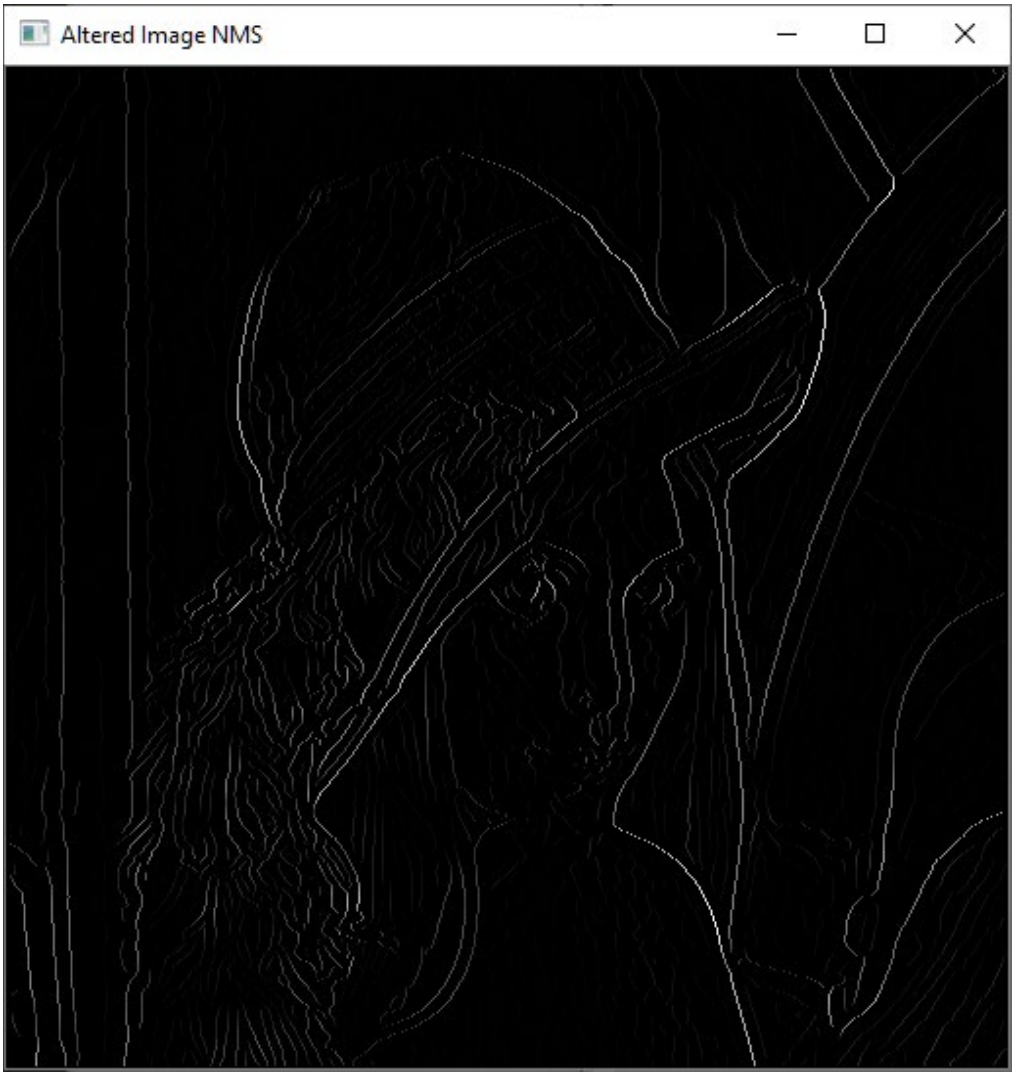
Gaussian Smoothing (K_size = 9, Sigma = 6):



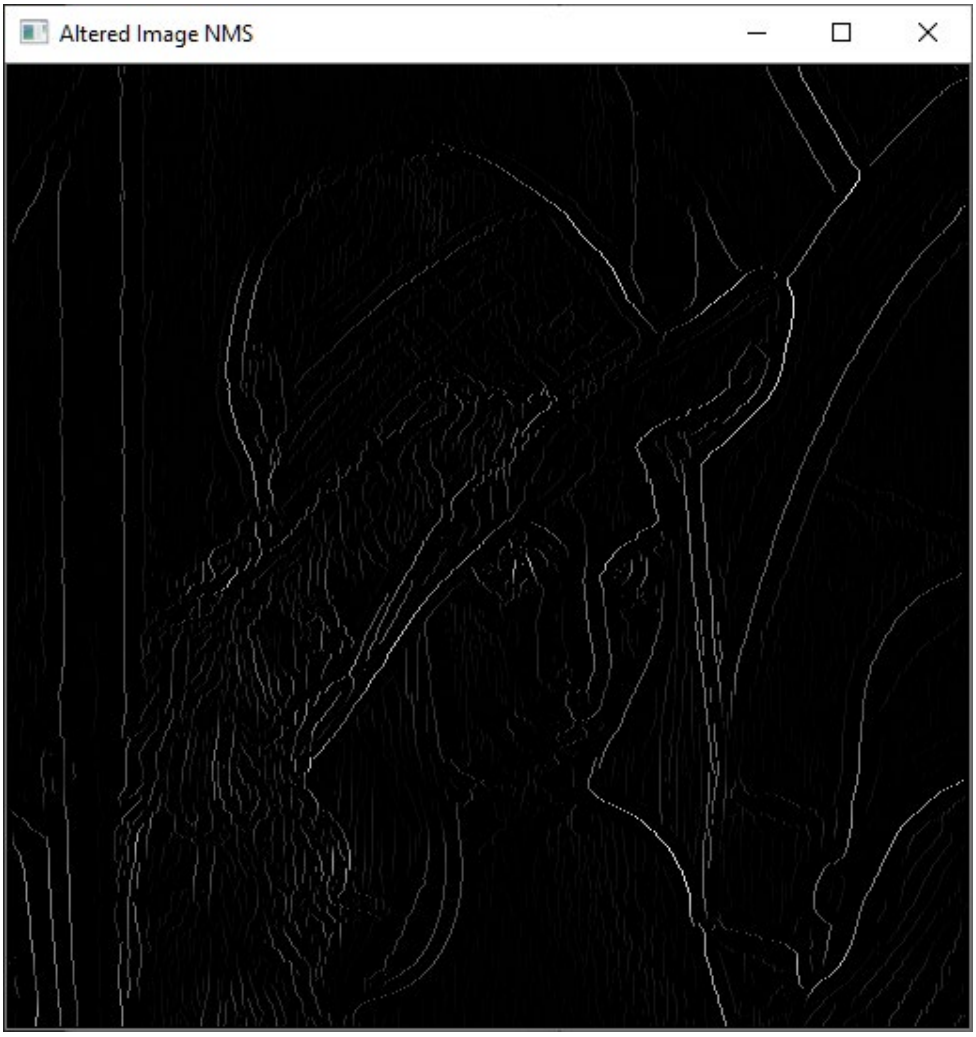
Non Maxima Suppression (K_size= 3 | Sigma = 2):



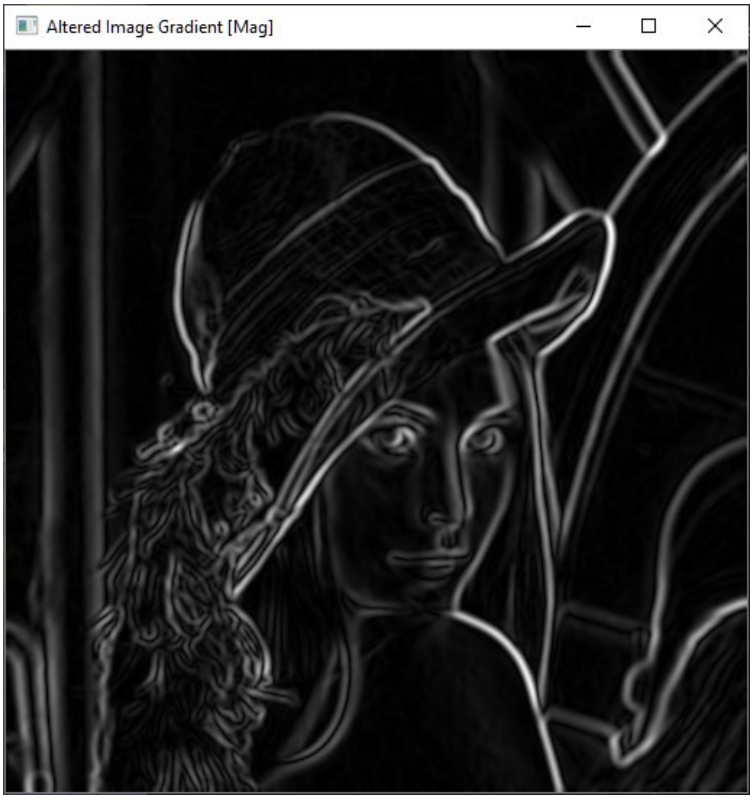
Non Maxima Suppression (K_size= 9 | Sigma = 2):



Non Maxima Suppression (K_size= 9 | Sigma = 6):



Sobel Operator Magnitude:



Sobel Operator Theta: