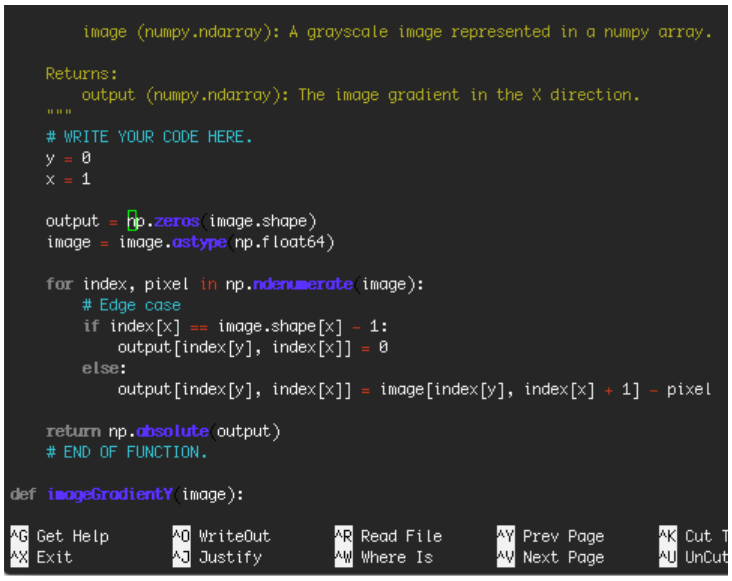
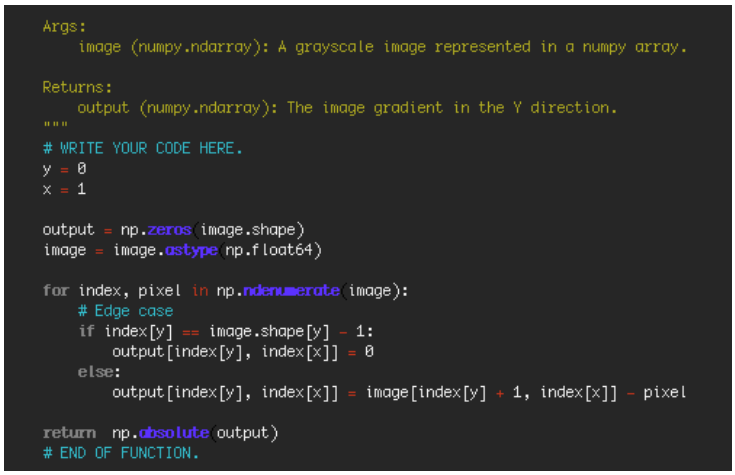
John Radice
GTID 903095161
Assignment 5

1. To implement the imageGradientX() function I used numpy.ndenumerate() function to iterate over the image. This function returns an iterator with pairs of array indices and values. I used the indices to capture the value of the next pixel in the x direction, which I then used to calculate the change. I set the pixels along the edges to 0 indicating that there is no change because there are no more pixels to compare. After calculating the gradient I used numpy.absolute() function to convert that to the magnitude of the gradient.



2. The imageGradientY() function was almost exactly the same except I used the indices to capture the value of the next pixel in the y direction, which I then used to calculate the change. I found I needed to convert the image array to float64 when my initial attempts resulted in noise. This was due to overflow errors as the input array was uint8.



John Radice
GTID 903095161
Assignment 5

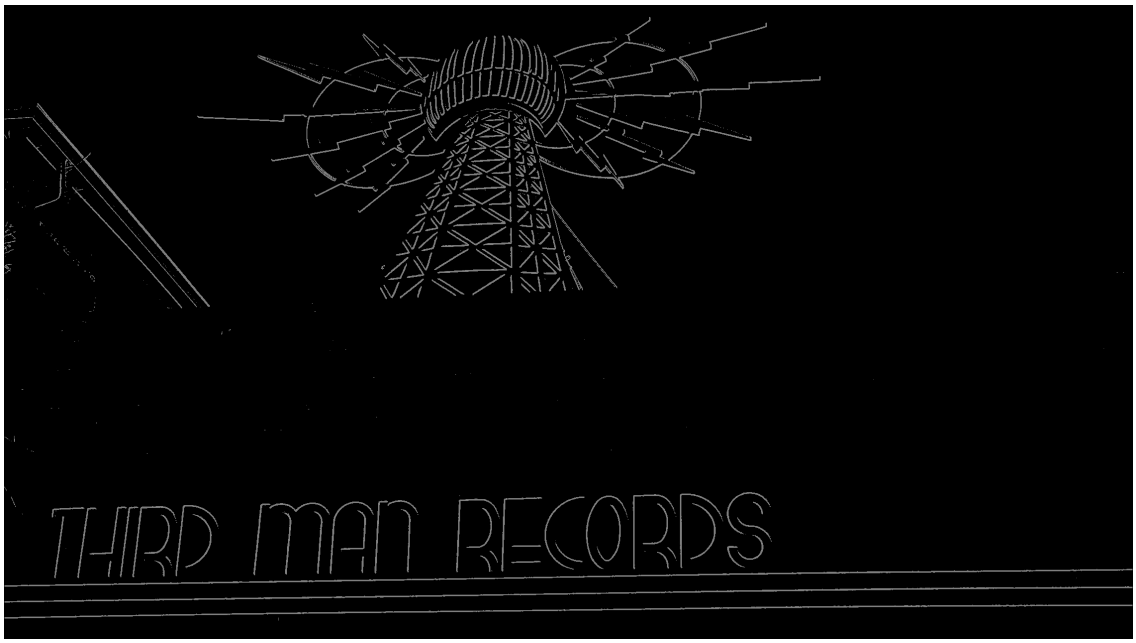
3. To implement the computeGradient() function I again used the numpy.ndenumerate() function to iterate through the image as well as the kernel. I first checked the bounds of the image to ensure the cross-correlation would take place in the bounds of the image. I then looped through the kernel using the image and kernel indices to capture the proper pixels to perform the cross-correlation on.

```
    Returns:
        output (numpy.ndarray): The computed gradient for the input image.
    """
    # WRITE YOUR CODE HERE.
    output = np.zeros(image.shape)
    image = image.astype(np.float64)

    y = 0
    x = 1

    for imageIndex, imagePixel in np.ndenumerate(image):
        # offset the kernel to prevent out of bounds exception
        if imageIndex[x] > 0 and imageIndex[y] > 0 \
           and imageIndex[x] < image.shape[x] - 1 \
           and imageIndex[y] < image.shape[y] - 1:

            sum = 0
            for kernelIndex, kernelValue in np.ndenumerate(kernel):
                u = kernelIndex[x] - 1
                v = kernelIndex[y] - 1

                sum += image[imageIndex[y] + v, imageIndex[x] + u] * kernelValue

            output[imageIndex[y], imageIndex[x]] = sum

    return output
    # END OF FUNCTION.

^G Get Help   ^O WriteOut   ^R Read File   ^Y Prev Page   ^K Cut Text
^X Exit       ^J Justify    ^W Where Is    ^V Next Page   ^U UnCut Text
```

4. I started with the following image while experimenting with edge detection:

John Radice
GTID 903095161
Assignment 5

After experimenting with several kernels, left/right average derivative, Prewitt, Sobel, Gaussian derivative, etc. I found the Sobel kernel provided the most accurate depiction of the edges. After taking the X and Y gradients I converted them to black and white. I used a threshold of 96 when converting the gradient in the X direction and used a threshold of 128 when converting the gradient in the Y direction. To create the final image shown below I averaged the X and Y gradients after applying the threshold.



For reference I have included the cv2.Canny() result of my original image: