

## Part I: Theoretical Problems

Q1 LTI system.

$$T[x(n)] = h(n) * x(n)$$

Any DT signal can be written as  $\sum_k x_k \delta[n-k]$

$$x[n] = \sum_{k=-\infty}^{\infty} x[k] \delta[n-k]$$

$$x[n] \xrightarrow{T} h(n)$$

$$\delta[n-k] \xrightarrow{T} h(n-k) \text{ (time invariant)}$$

$$x[n-k] \xrightarrow{T} x[n-k] \text{ (linearity)}$$

$$\text{Let } x = x(k)$$

$$x(k) \delta[n-k] \xrightarrow{T} x(k) h(n-k)$$

$$\sum_{k=-\infty}^{\infty} x(k) \delta[n-k] \xrightarrow{T} \sum_{k=-\infty}^{\infty} x(k) h(n-k)$$

$$x[n] \xrightarrow{T} \sum_{k=-\infty}^{\infty} x(k) h(n-k)$$

$$\begin{aligned} \text{In other word } T[x(n)] &= \sum_{k=-\infty}^{\infty} x(k) h(n-k) \\ &= x[n] * h[n] \\ &= h[n] * x[n] \quad (\text{commutative}) \end{aligned}$$



Q2 Polynomial Multiplication &amp; Convolution

consider 3rd-degree polynomial  $(a_3 x^3 + a_2 x^2 + a_1 x + a_0)$

this in vector form  $\rightarrow [a_3 \ a_2 \ a_1 \ a_0]$

lets create one other polynomial  $(b_3 x^3 + b_2 x^2 + b_1 x + b_0)$

w/ its vector form  $[b_3 \ b_2 \ b_1 \ b_0]$

Using table we can show multiplication of polynomial

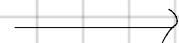
$a_0$	$a_1 x$	$a_2 x^2$	$a_3 x^3$
$b_0$	$a_0 b_0$	$a_1 b_0 x$	$a_2 b_0 x^2$
$b_1 x$	$a_0 b_1 x$	$a_1 b_1 x^2$	$a_2 b_1 x^3$
$b_2 x^2$	$a_0 b_2 x^2$	$a_1 b_2 x^3$	$a_2 b_2 x^4$
$b_3 x^3$	$a_0 b_3 x^3$	$a_1 b_3 x^4$	$a_2 b_3 x^5$

the result of multiplication between two polynomials could be obtained by adding all the terms in the table

the color above is grouped by the power of argument  $x$ .

if we assume  $c$  to be a resulting coefficient then

$$\begin{aligned}
 &c_0 && a_0 b_0 \\
 &+ c_1 x && + (a_0 b_1 + a_1 b_0) x \\
 &+ c_2 x^2 && + (a_0 b_2 + a_1 b_1 + a_2 b_0) x^2 \\
 &+ c_3 x^3 && = + (a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0) x^3 \\
 &+ c_4 x^4 && + (a_1 b_3 + a_2 b_2 + a_3 b_1) x^4 \\
 &+ c_5 x^5 && + (a_2 b_3 + a_3 b_2) x^5 \\
 &+ c_6 x^6 && + (a_3 b_3) x^6
 \end{aligned}$$



writing this in matrix form you get

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \end{bmatrix} = \begin{bmatrix} a_0 & 0 & 0 & 0 \\ a_1 & a_0 & 0 & 0 \\ a_2 & a_1 & a_0 & 0 \\ a_3 & a_2 & a_1 & a_0 \\ 0 & a_3 & a_2 & a_1 \\ 0 & 0 & a_3 & a_2 \\ 0 & 0 & 0 & a_3 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} b_0 & 0 & 0 & 0 \\ b_1 & b_0 & 0 & 0 \\ b_2 & b_1 & b_0 & 0 \\ b_3 & b_2 & b_1 & b_0 \\ 0 & b_3 & b_2 & b_1 \\ 0 & 0 & b_3 & b_2 \\ 0 & 0 & 0 & b_3 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

we could generalize the observation above to polynomial multiplication w/ other degrees

$$c_i = \sum_j a(j)b(i-j+1) = a_i * b_i$$

$\therefore$  convolving two vectors of polynomial coefficient is equal to multiplying the two polynomials they each represents.

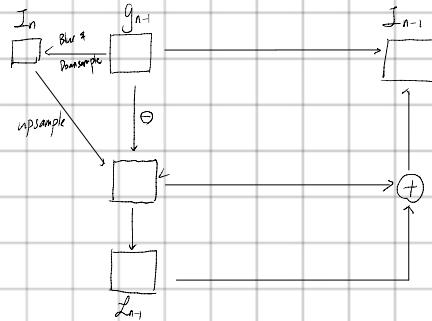


### [Q3] Image Pyramids

We should be able to perform at least one gaussian pyramid operation in order to reconstruct image

thus **Base case**: for  $I_0$  w/ size  $2^n \times 2^n$ ,  $g_{n-1}$  is base case, where image has size of  $2 \times 2$

let  $F(\cdot)$  be upsampling function



Above diagram could be expressed as

$$I_{n-1} = L_{n-1} + F(I_n) \quad (*)$$

Given all the  $L_i$ ,  $I_0$  is minimum information to get  $I_0$

containing w/ equation (\*)

$$\begin{aligned} I_{n-2} &= L_{n-2} + F(I_{n-1}) \\ &= L_{n-2} + F(L_{n-1} + F(I_n)) \\ &= L_{n-2} + F(L_{n-1}) + F^2(I_n) \end{aligned}$$

$$\begin{aligned} I_{n-3} &= L_{n-3} + F(I_{n-2}) \\ &= L_{n-3} + F(L_{n-2} + F(L_{n-1}) + F^2(I_n)) \\ &= L_{n-3} + F(L_{n-2}) + F^2(L_{n-1}) + F^3(I_n) \end{aligned}$$

$$\therefore I_{n-k} = L_{n-k} + \sum_{i=1}^k F^i(L_{n-k+i}) \quad \text{where base case: } k=1$$

$$\text{at } k=0 \rightarrow I_{n-1} = L_{n-1} + F(L_n) \quad \checkmark$$

$$\text{assume } k=k \text{ is true } I_{n-k} = L_{n-k} + \sum_{i=1}^{k+1} F^i(L_{n-k+i}) \quad (k+1)$$

$$\text{when } k=k+1 \quad I_{n-(k+1)} = L_{n-(k+1)} + \sum_{i=1}^{k+1} F^i(L_{n-(k+1)+i})$$

$$I_{n-(k+1)} = L_{n-(k+1)} + F(L_{n-k}) + \sum_{i=2}^{k+1} F^i(L_{n-k+i})$$

$$\text{from eqn } (*) \rightarrow L_{n-k} = I_{n-k} - \sum_{i=1}^k F^i(L_{n-k+i})$$

$$\begin{aligned} I_{n-(k+1)} &= L_{n-(k+1)} + F(I_{n-k}) - F\left(\sum_{i=1}^k F^i(L_{n-k+i})\right) \\ &\quad + \underbrace{\sum_{i=2}^{k+1} F^i(L_{n-k+i})}_{= \sum_{i=1}^k F^{i+1}(L_{n-(k+1)+i})} \\ &= \sum_{i=1}^{k+1} F^{i+1}(L_{n-(k+1)+i}) \end{aligned}$$

$$I_{n-(k+1)} = L_{n-(k+1)} + F(I_{n-k}) \quad \checkmark$$



Thus by mathematical Induction we have shown that

$$I_{n-k} = L_{n-k} + \sum_{i=1}^k F^i(L_{n-k+i}) \text{ is true for } 1 \leq k \leq n$$

#### [Q4] Laplacian Operator

$$\Delta f = \nabla^2 f = \nabla \cdot \nabla f = \sum_{i=1}^n \frac{\partial^2 f}{\partial x_i^2}$$

$$\text{when } k=n \rightarrow I_0 = L_0 + \sum_{i=1}^n F^i(L_i) \text{ where } L_n = I_n$$

$$\Delta I = I_{xx} + I_{yy}$$



In  $\mathbb{R}^2$ , matrix rotation that rotates vector  $v_0$  in the counter clockwise angle  $\theta$  is given by

$$R_\theta = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \quad \text{so } v' = R_\theta v_0$$

setting  $v_0 = \begin{bmatrix} x \\ y \end{bmatrix}$  we could obtain two expressions for coordinates in polar notations

$$\begin{cases} r = x \cos \theta - y \sin \theta \\ r' = x \sin \theta + y \cos \theta \end{cases} \quad I(r, r')$$

Now using chain rule,

$$\begin{aligned} f &= \frac{\partial I(x, y)}{\partial x} = \frac{\partial r}{\partial x} \frac{\partial I}{\partial r} + \frac{\partial r'}{\partial x} \frac{\partial I}{\partial r'} \\ &= \cos \theta \frac{\partial I}{\partial r} + \sin \theta \frac{\partial I}{\partial r'} \end{aligned}$$

$$\begin{aligned} \frac{\partial^2 I}{\partial x^2} &= \frac{\partial f}{\partial x} = \frac{\partial r}{\partial x} \frac{\partial f}{\partial r} + \frac{\partial r'}{\partial x} \frac{\partial f}{\partial r'} \\ &= \cos \theta \frac{\partial f}{\partial r} + \sin \theta \frac{\partial f}{\partial r'} \\ &= \cos \theta \frac{\partial}{\partial r} \left( \cos \theta \frac{\partial I}{\partial r} + \sin \theta \frac{\partial I}{\partial r'} \right) \\ &\quad + \sin \theta \frac{\partial}{\partial r'} \left( \cos \theta \frac{\partial I}{\partial r} + \sin \theta \frac{\partial I}{\partial r'} \right) \\ &= \cos^2 \theta \frac{\partial^2 I}{\partial r^2} + 2 \sin \theta \cos \theta \frac{\partial^2 I}{\partial r \partial r'} + \sin^2 \theta \frac{\partial^2 I}{\partial r'^2}. \end{aligned}$$

$$\begin{aligned} g &= \frac{\partial I(x, y)}{\partial y} = \frac{\partial r}{\partial y} \frac{\partial I}{\partial r} + \frac{\partial r'}{\partial y} \frac{\partial I}{\partial r'} \\ &= -\sin \theta \frac{\partial I}{\partial r} + \cos \theta \frac{\partial I}{\partial r'} \end{aligned}$$

$$\begin{aligned} \frac{\partial^2 I}{\partial y^2} &= \frac{\partial g}{\partial y} = \frac{\partial r}{\partial y} \frac{\partial g}{\partial r} + \frac{\partial r'}{\partial y} \frac{\partial g}{\partial r'} \\ &= -\sin \theta \frac{\partial g}{\partial r} + \cos \theta \frac{\partial g}{\partial r'} \\ &= -\sin \theta \frac{\partial}{\partial r} \left( -\sin \theta \frac{\partial I}{\partial r} + \cos \theta \frac{\partial I}{\partial r'} \right) \\ &\quad + \cos \theta \frac{\partial}{\partial r'} \left( -\sin \theta \frac{\partial I}{\partial r} + \cos \theta \frac{\partial I}{\partial r'} \right) \\ &= \sin^2 \theta \frac{\partial^2 I}{\partial r^2} - 2 \sin \theta \cos \theta \frac{\partial^2 I}{\partial r \partial r'} + \cos^2 \theta \frac{\partial^2 I}{\partial r'^2}. \end{aligned}$$

$$\begin{aligned}
 \Delta I &= I_{xx} + I_{yy} = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2} \\
 &= \cos^2 \phi \frac{\partial^2 I}{\partial r^2} + 2 \sin \phi \cos \phi \frac{\partial^2 I}{\partial r \partial r'} + \sin^2 \phi \frac{\partial^2 I}{\partial r'^2} \\
 &\quad + \sin^2 \phi \frac{\partial^2 I}{\partial r^2} - 2 \sin \phi \cos \phi \frac{\partial^2 I}{\partial r \partial r'} + \cos^2 \phi \frac{\partial^2 I}{\partial r'^2} \\
 &= \underbrace{\left( \sin^2 \phi + \cos^2 \phi \right) \frac{\partial^2 I}{\partial r^2}}_{=1} + \underbrace{\left( \sin^2 \phi + \cos^2 \phi \right) \frac{\partial^2 I}{\partial r'^2}}_{=1} \\
 &= \frac{\partial^2 I}{\partial r^2} + \frac{\partial^2 I}{\partial r'^2} = I_{rr} + I_{r'r'}
 \end{aligned}$$

$\therefore$  Laplacian is rotation invariant.



# A1\_P2\_Implementation\_Task

October 4, 2021

## 1 PART II: Implementation TASK

### 1.0.1 import necessary library for the assignment

```
[ ]: import numpy as np  
import matplotlib.pyplot as plt  
import cv2
```

#### 1.1 STEP1: Gaussian Blurring

```
[ ]: def gaussian_kernel(size: int, sigma: float) -> np.ndarray:  
    """This function returns 2-dimentional gaussian kernel  
  
    Parameters  
    -----  
    size : int, size of the kernel in square  
    sigma : float, standard derivation in gaussian distribution function  
  
    Returns  
    -----  
    out : np.ndarray, shape = (size, size)  
        an square matrix with centered gaussian kernel  
    """  
  
    # create equally spaced x which will inputed to gaussian formula  
    x = np.linspace(-(size // 2), size // 2, size)  
  
    # gaussian formula (results 1D gaussian kernel)  
    gaussian = np.exp(-np.square(x)/(2.*np.square(sigma)))  
  
    # perform outer product to obtain 2D gaussian kernal  
    gaussian_2d = np.outer(gaussian, gaussian)  
  
    # normalize and return  
    return gaussian_2d / gaussian_2d.sum()
```

- Note: 2D visualization with color map is done on STEP4(testing part)

## 1.2 STEP2: Gradient Magnitude

```
[ ]: def convolution(image: np.ndarray, filter: np.ndarray) -> np.ndarray:  
    """This function returns image that are convolved with filter(kernel)  
  
    Parameters  
    -----  
    image: np.ndarray, gray scaled image  
    filter: np.ndarray, kernel used to convolved the image with  
  
    Returns  
    -----  
    out: np.ndarray, shape(image.shape[0], image.shape[1])  
        an convolved result from image with filter  
    """  
    #perform horizontal and vertial flip of a filter.  
    kernel = np.fliplr(np.flipud(filter))  
  
    #obtain sub-matrices to speed up the process of convolution  
    →(size(sub_matrices) = size(kernels))  
    sub_matrices = np.lib.stride_tricks.as_strided(image,  
                                                    shape=tuple(np.  
    →subtract(image.shape, kernel.shape)+1)+kernel.shape,  
                                                    strides=image.strides*2)  
  
    #get result of convolution by applying scalar product between kernal and  
    →sub_matrices  
    return np.einsum('ij,klij->kl', kernel, sub_matrices)
```

```
[ ]: def gradient_magnitude(image: np.ndarray) -> np.ndarray:  
    """This function return gradient magnitude from the inputed image  
  
    Parameters  
    -----  
    image: np.ndarray, gray scaled image  
  
    Returns  
    -----  
    out: np.ndarray, shape(image.shape[0], image.shape[1])  
        gradient magnitude using the Sobel operator  
    """  
  
    # sober filters  
    s_x = np.array([[[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]]])  
    s_y = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])  
  
    # get derivates
```

```

d_x = convolution(image, s_x)
d_y = convolution(image, s_y)

# get gradient magnitude
res = np.linalg.norm(np.array([d_x,d_y]), axis=(0)) #same as np.sqrt(d_x**2
+ d_y**2)
return res

```

### 1.3 STEP3: Threshold Algorithm

```

[ ]: def threshold(grad_image: np.ndarray, epsilon=0.1)->np.ndarray:
    """This function returns black and white edge-mapped image

    Parameters
    -----
    grad_image: np.ndarray
    epsilon: variable for setting the threshold

    Returns
    -----
    out: np.ndarray, shape(grad_image[0], grad_image[1])
        edge-mapped image with pixel value of either 0 or 255
    """

    # calculate average intensity of grad image
    i_t = grad_image.mean()
    i_t_new = float('inf')

    #create loop that sets the threshold
    while np.abs(i_t- i_t_new) > epsilon:
        i_t_new = i_t
        l_class = grad_image.mean(where=grad_image < i_t_new)
        u_class = grad_image.mean(where=grad_image >= i_t_new)
        i_t = (l_class + u_class)/2

        # for all the pixels in grad_image, if pixel is larger than threshold make
        that
        # pixel 255 otherwise 0
        res = np.where(grad_image > i_t, 255, 0)

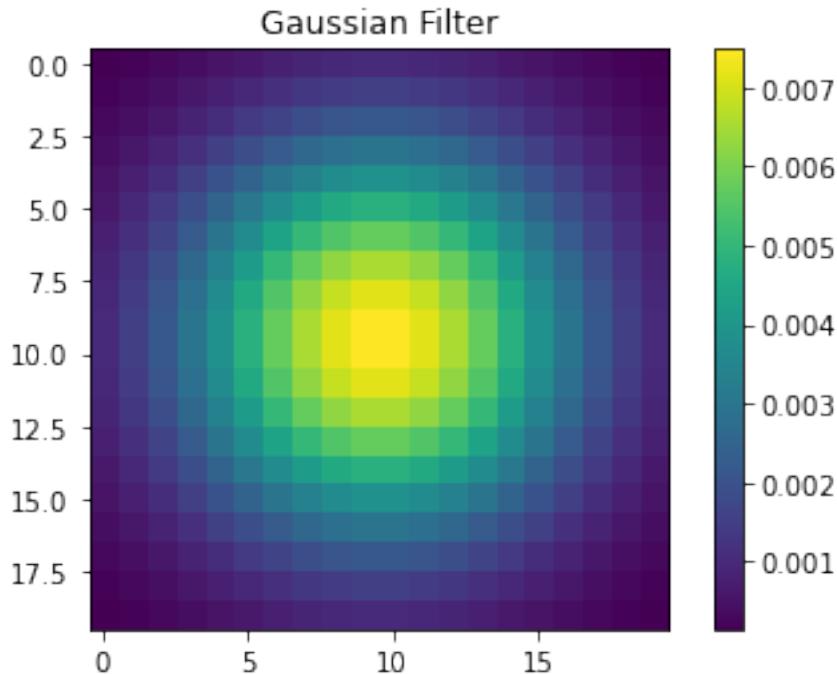
    return res

```

## 1.4 STEP4: TEST

```
[ ]: #generate gaussian_kernel (step1)
g_kernel = gaussian_kernel(20, 5)
plt.imshow(g_kernel, interpolation='none')
plt.colorbar()
plt.title('Gaussian Filter')
```

```
[ ]: Text(0.5, 1.0, 'Gaussian Filter')
```



### 1.4.1 For image1

```
[ ]: fig_1 = plt.figure(figsize=(200, 160))
ax1 = fig_1.add_subplot(141)
ax2 = fig_1.add_subplot(142)
ax3 = fig_1.add_subplot(143)
ax4 = fig_1.add_subplot(144)

#original image
image_1 = cv2.imread('./a1_image/image1.jpg')
gray_1 = cv2.cvtColor(image_1, cv2.COLOR_BGR2GRAY) #generate gray-scaled image
ax1.imshow(image_1)
ax1.set_title("Original Image", fontsize=100)
ax1.set_xticks([]), ax1.set_yticks([])
```

```

#apply gaussian filter on image
g_image_1 = convolution(gray_1, g_kernel)
ax2.imshow(g_image_1, interpolation='none', cmap='gray')
ax2.set_title("Gaussian Filter", fontsize=100)
ax2.set_xticks([]), ax2.set_yticks([])

#generate gradient magnitude of the image
image_grad_mag_1 = gradient_magnitude(gray_1)
ax3.imshow(image_grad_mag_1, cmap='gray')
ax3.set_title("Gradient Magnitude", fontsize=100)
ax3.set_xticks([]), ax3.set_yticks([])

#Show black and white edge-mapped image
threshold_1 = threshold(image_grad_mag_1)
ax4.imshow(threshold_1, cmap='gray')
ax4.set_title("Thresholding", fontsize=100)
ax4.set_xticks([]), ax4.set_yticks([])

```

[ ]: ([] , [])



```

[ ]: fig_2 = plt.figure(figsize=(200, 160))
bx1 = fig_2.add_subplot(141)
bx2 = fig_2.add_subplot(142)
bx3 = fig_2.add_subplot(143)
bx4 = fig_2.add_subplot(144)

#original image
image_2 = cv2.imread('./a1_image/image2.jpg')
gray_2 = cv2.cvtColor(image_2, cv2.COLOR_BGR2GRAY) #generate gray-scaled image
bx1.imshow(image_2)
bx1.set_title("Original Image", fontsize=100)
bx1.set_xticks([]), bx1.set_yticks([])

#apply gaussian filter on image
g_image_2 = convolution(gray_2, g_kernel)
bx2.imshow(g_image_2, interpolation='none', cmap='gray')

```

```

bx2.set_title("Gaussian Filter", fontsize=100)
bx2.set_xticks([]), bx2.set_yticks([])

#generate gradient magnitude of the image
image_grad_mag_2 = gradient_magnitude(gray_2)
bx3.imshow(image_grad_mag_2, cmap='gray')
bx3.set_title("Gradient Magnitude", fontsize=100)
bx3.set_xticks([]), bx3.set_yticks([])

#Show black and white edge-mapped image
threshold_2 = threshold(image_grad_mag_2)
bx4.imshow(threshold_2, cmap='gray')
bx4.set_title("Thresholding", fontsize=100)
bx4.set_xticks([]), bx4.set_yticks([])

```

[ ]: ([] , [])



```

[ ]: fig_3 = plt.figure(figsize=(200, 160))
cx1 = fig_3.add_subplot(141)
cx2 = fig_3.add_subplot(142)
cx3 = fig_3.add_subplot(143)
cx4 = fig_3.add_subplot(144)

#original image
image_3 = cv2.imread('./a1_image/image3.jpg')
gray_3 = cv2.cvtColor(image_3, cv2.COLOR_BGR2GRAY) #generate gray-scaled image
cx1.imshow(image_3)
cx1.set_title("Thresholding", fontsize=100)
cx1.set_xticks([]), cx1.set_yticks([])

#apply gaussian filter on image
g_image_3 = convolution(gray_3, g_kernel)
cx2.imshow(g_image_3, interpolation='none', cmap='gray')
cx2.set_title("Gaussian Filter", fontsize=100)
cx2.set_xticks([]), cx2.set_yticks([])

#generate gradient magnitude of the image
image_grad_mag_3 = gradient_magnitude(gray_3)

```

```

cx3.imshow(image_grad_mag_3, cmap='gray')
cx3.set_title("Gradient Magnitude", fontsize=100)
cx3.set_xticks([]), cx3.set_yticks([])

#Show black and white edge-mapped image
threshold_3 = threshold(image_grad_mag_3)
cx4.imshow(threshold_3, cmap='gray')
cx4.set_title("Thresholding", fontsize=100)
cx4.set_xticks([]), cx4.set_yticks([])

```

[ ]: ([], [])



#### 1.4.2 How does the edge detection algorithm works?

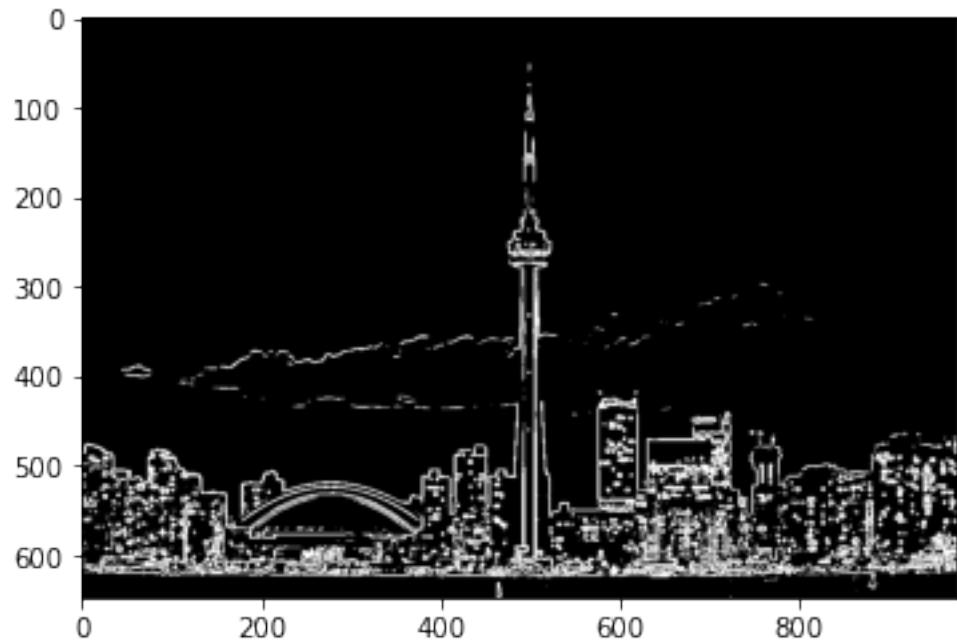
The four different images in the order of original, gaussian blur, gradient magnitude, thresholding were shown for each image we had. In order to detect the edge, we took the grayscale of the original image and applied convolution with a Sobel filter which produce a derivative of the image. Then we took the results from the convolution to obtain the image of gradient magnitude. Finally, we adopted the thresholding algorithm to filter out the points below the computed threshold to obtain the edge detected image.

#### 1.4.3 Highlight our edge detecting algorithm's strength and weakness

As you see from the 3 different edges detected images from the implemented algorithm, it does indeed produce perceivable edges. Without knowing how the original image looks like, we could still tell what each image is about. However, there are two weaknesses I could point out from looking at the result. First of all, some of the edges were lost (ex. edges on the CN tower due to cloud). It seems that the edges from the gradient magnitude were too weak to win the thresholds. Secondly, there were lots of noises in the edge detected images. However, trying to remove them would induce losing the important information about the edge. More detailed results of the edge detected image are shown below.

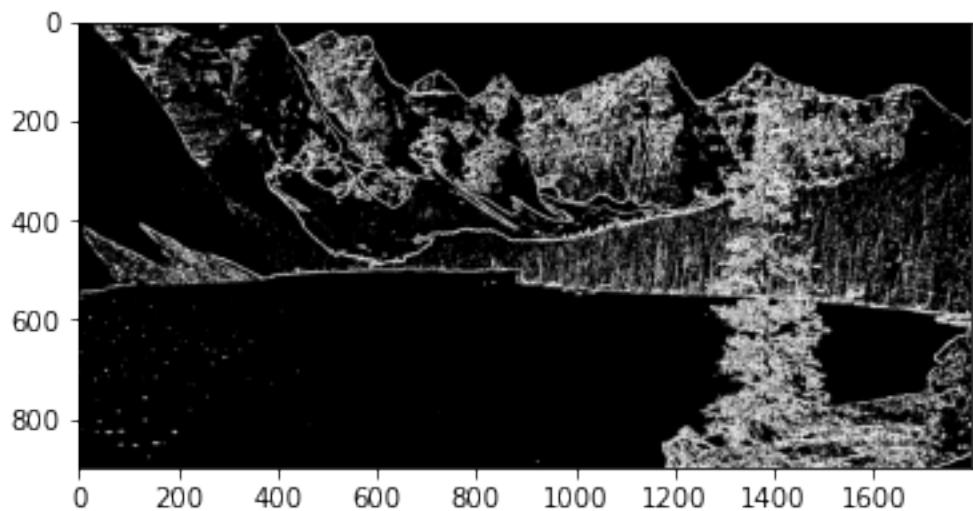
[ ]: plt.imshow(threshold\_1, cmap='gray')

[ ]: <matplotlib.image.AxesImage at 0x7f3d3cc92a60>



```
[ ]: plt.imshow(threshold_2, cmap='gray')
```

```
[ ]: <matplotlib.image.AxesImage at 0x7f3d3dc9e3d0>
```



```
[ ]: plt.imshow(threshold_3, cmap='gray')
```

```
[ ]: <matplotlib.image.AxesImage at 0x7f3d3cdec400>
```

