Intro to Image Understanding (CSC420) Assignment 1

Due Date: Jan 24th, 2025, 11:59:00 pm
Total: 120 marks

General Instructions:

- You are allowed to work directly with one other person to discuss the questions. However, you are still expected to write the solutions/code/report in your own words; i.e. no copying. If you choose to work with someone else, you must indicate this in your assignment submission. For example, on the first line of your report file (after your own name and information, and before starting your answer to Q1), you should have a sentence that says: *"In solving the questions in this assignment, I worked together with my classmate [name & student number]. I confirm that I have written the solutions/code/report in my own words"*.
- Your submission should be in the form of an electronic report (PDF), with the answers to the specific questions (each question separately), and a presentation and discussion of your results. For this, please submit a file named **report.pdf** to MarkUs directly.
- Submit documented codes that you have written to generate your results separately. Please store all of those files in a folder called **assignment1**, zip the folder and then submit the file **assignment1.zip** to MarkUs. You should include a **README.txt** file (inside the folder) which details how to run the submitted codes.
- Do not worry if you realize you made a mistake after submitting your zip file; you can submit multiple times on MarkUs.

Collaboration

In solving the questions in this assignment, I worked together with my classmate (Yousef Al Rawwash, 1007684873). I confirm that I have written the solutions/code/report in my own words.

Part I: Theoretical Problems (60 marks)

[Question 1] Convolution (10 marks)

[**1.a**] (5 marks) Calculate and plot the convolution of $x[n]$ and $h[n]$ specified below:

$$x[n] = \begin{cases} 1 & -3 \le n \le 3 \\ 0 & \text{otherwise} \end{cases} \qquad h[n] = \begin{cases} 1 & -2 \le n \le 2 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

We can represent the functions as polynomials and do the polynomial multiplication method. The representations are

$$A[z] = \sum_{k=-\infty}^{\infty} a[k]z^k \tag{2}$$

where $a[z]$ is the signal in question. We then can compute the convolution between some a and b as $A[z] \cdot B[z]$. First compute the polynomial forms,

$$X[n] = z^{-3} + z^{-2} + z^{-1} + z^0 + z^1 + z^2 + z^3$$

$$H[n] = z^{-2} + z^{-1} + z^0 + z^1 + z^2$$

and then compute the product,

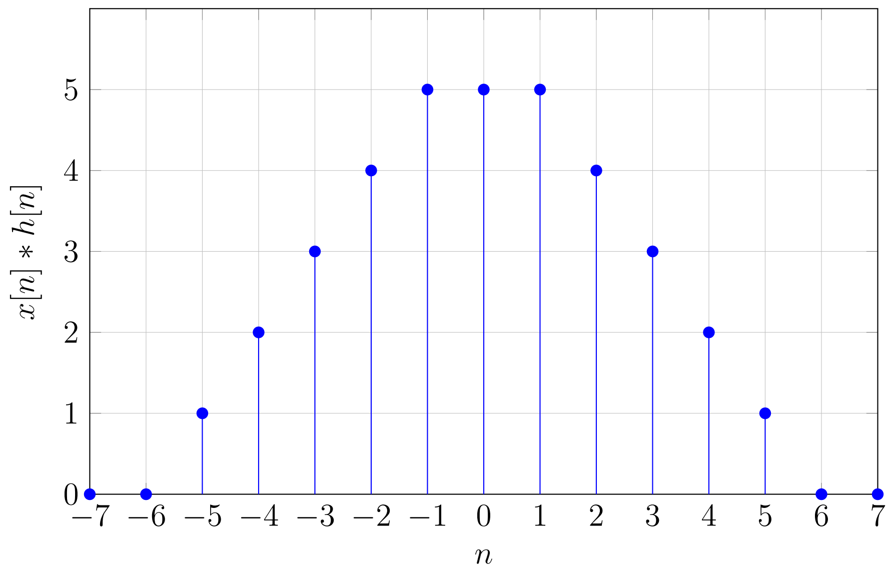
$$\begin{aligned} X[n] \cdot H[n] = (z^{-3} + z^{-2} + z^{-1} + z^0 + z^1 + z^2 + z^3)(z^{-2} + z^{-1} + z^0 + z^1 + z^2) \\ = z^{-5} + z^{-4} + z^{-3} + z^{-2} + z^{-1} + z^0 + z^1 \\ +z^{-4} + z^{-3} + z^{-2} + z^{-1} + z^0 + z^1 + z^2 \\ +z^{-3} + z^{-2} + z^{-1} + z^0 + z^1 + z^2 + z^3 \\ +z^{-2} + z^{-1} + z^0 + z^1 + z^2 + z^3 + z^4 \\ +z^{-1} + z^0 + z^1 + z^2 + z^3 + z^4 + z^5 \\ = z^{-5} + 2z^{-4} + 3z^{-3} + 4z^{-2} + 5z^{-1} + 5z^0 + 5z^1 + 4z^2 + 3z^z + 2z^4 + z^5. \end{aligned}$$

This form can be converted back to a signal using the same idea as in Equation 2 to get the following.

$$(x * h)[n] = \begin{cases} 1 & n = -5 \text{ or } n = 5, \\ 2 & n = -4 \text{ or } n = 4, \\ 3 & n = -3 \text{ or } n = 3, \\ 4 & n = -2 \text{ or } n = 2, \\ 5 & n = -1, 0, \text{ or } 1, \\ 0 & \text{otherwise.} \end{cases}$$

And the corresponding plot is shown below.

Figure 1: The plot of $x[n] * h[n]$ in part 1.a.



[1.b] (5 marks) Calculate and plot the convolution of $x[n]$ and $h[n]$ specified below:

$$x[n] = \begin{cases} 1 & -3 \le n \le 3 \\ 0 & \text{otherwise} \end{cases} \qquad h[n] = \begin{cases} 2 - |n| & -2 \le n \le 2 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Here we can use the same method from 1.a. since $x[n]$ does not change, its polynomial representation is preserved. However, for $h[n]$, we use Equation 2 to obtain,

$$H[n] = 0 \cdot z^{-2} + 1 \cdot z^{-1} + 2 \cdot z^{0} + 1 \cdot z^{1} + 0 \cdot z^{2}$$
$$= z^{-1} + 2z^{0} + z^{1} = z^{-1} + 2 + z^{1}$$

Using this, we can compute the polynomial multiplication as before.

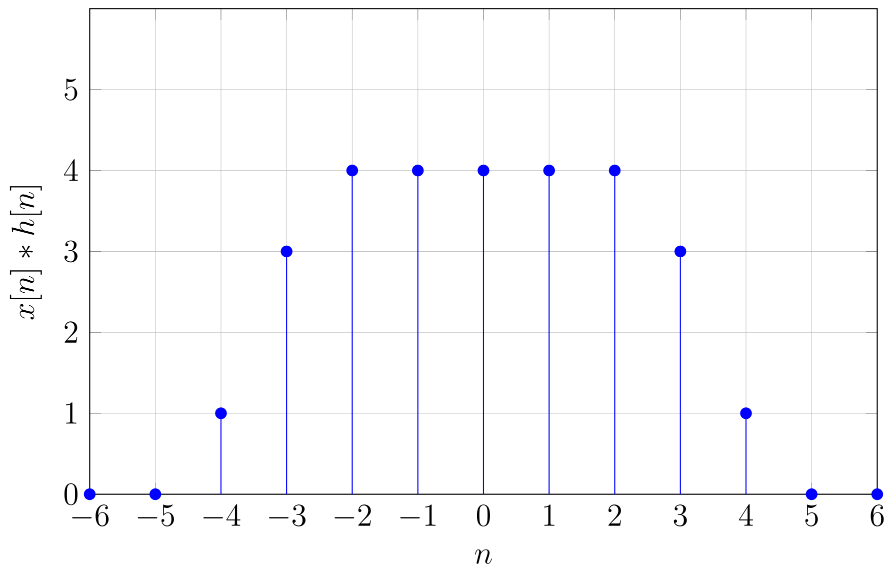
$$X[n] \cdot H[n] = (z^{-3} + z^{-2} + z^{-1} + z^{0} + z^{1} + z^{2} + z^{3})(z^{-1} + 2 + z^{1})$$
$$= z^{-4} + z^{-3} + z^{-2} + z^{-1} + z^{0} + z^{1} + z^{2}$$
$$+2z^{-3} + 2z^{-2} + 2z^{-1} + 2z^{0} + 2z^{1} + 2z^{2} + 2z^{3}$$
$$+z^{-2} + z^{-1} + z^{0} + z^{1} + z^{2} + z^{3} + z^{4}$$
$$= z^{-4} + 3z^{-3} + 4z^{-2} + 4z^{-1} + 4 + 4z + 4z^{2} + 3z^{3} + z^{4}$$

Then convert it back to a signal.

$$(x * h)[n] = \begin{cases} 1 & n = -4 \text{ or } n = 4, \\ 3 & n = -3 \text{ or } n = 3, \\ 4 & -2 \le n \le 2, \\ 0 & \text{otherwise.} \end{cases}$$

The final plot is show below.

Figure 2: The plot of $x[n] * h[n]$ in part 1.b.



[Question 2] LTI Systems (15 marks)

We define a *system* as something that takes an input signal, e.g. $x(n)$, and produces an output signal, e.g. $y(n)$. Linear Time-Invariant (LTI) systems are a class of systems that are both linear and time-invariant. In linear systems, the output for a linear combination of inputs is equal to the linear combination of individual responses to those inputs. In other words, for a system T, signals $x_1(n)$ and $x_2(n)$, and scalars a_1 and a_2, system T is linear if and only if:

$$T[a_1x_1(n) + a_2x_2(n)] = a_1T[x_1(n)] + a_2T[x_2(n)]$$

Also, a system is time-invariant if a shift in its input merely shifts the output; i.e. If $T[x(n)] = y(n)$, system T is time-invariant if and only if:

$$T[x(n - n_0)] = y(n - n_0)$$

[2.a] (5 marks) Consider a discrete linear time-invariant system T with discrete input signal $x(n)$ and impulse response $h(n)$. Recall that the impulse response of a discrete system is defined as the output of the system when the input is an impulse function $\delta(n)$, i.e. $T[\delta(n)] = h(n)$, where:

$$\delta(n) = \begin{cases} 1, & \text{if } n = 0, \\ 0, & \text{else.} \end{cases}$$

Prove that $T[x(n)] = h(n) * x(n)$, where $*$ denotes convolution operation.

Hint: represent signal $x(n)$ as a function of $\delta(n)$.
Following the hint, we write $x(n)$ as a combination of delta functions, scaled by the value of $x(n)$ at the point n. For a given point k, we have $x(k)$ as the scaling and use the impule

function shifted by k, $\delta(n-k)$ to have a non-zero at that point. This allows the below formulation.

$$x(n) = \sum_{k=-\infty}^{\infty} x(k) \cdot \delta(n-k)$$

We note that this is the convolution sum. Using the properties of linearity and time invariance of a LTI system, we get the following.

$$T[x(n)] = T\left[\sum_{k=-\infty}^{\infty} x(k) \cdot \delta(n-k)\right] = \sum_{k=-\infty}^{\infty} x(k) \cdot T[\delta(n-k)]$$

$$= \sum_{k=-\infty}^{\infty} x(k) \cdot h(n-k)$$

$$= x(n) * h(n)$$

As required. Note that $x(k)$ is a constant and since $T[\cdot]$ is time invariant, $T[\delta(n-k)] = h(n-k)$.

[2.b] (5 marks) Is Gaussian blurring linear? Is it time-invariant? Make sure to include your justifications.

Gaussian Blurring occurs when we convolve the Gaussian kernel with a given image. Let us assume 1-D convolutions.

$$y(n) = T[x(n)] = x(n) * g(n) = \sum_{k=-\infty}^{\infty} x(n) \cdot g(n-k)$$

We note it was shown in lecture that the convolution operation is itself linear; thus, a system that is a convolution must too itself be linear. Thus Gaussian Blurring is linear.

By the same reasoning, we notice that convolution is time-invariant and therefore so too must Gaussian Blurring be. Thus Gaussian Blurring is time-invariant.

[2.c] (5 marks) Is time reversal, i.e. $T[x(n)] = x(-n)$, linear? Is it time-invariant? Make sure to include your justifications.

For a system T, signals $x_1(n)$ and $x_2(n)$, and scalars a_1 and a_2, system T is linear if and only if:

$$T[a_1x_1(n) + a_2x_2(n)] = a_1T[x_1(n)] + a_2T[x_2(n)]$$

Thus to show linearity, is it sufficient to show that the above expression holds.

$$T[a_1x_1(n) + a_2x_2(n)] = (a_1x_1(n) + a_2x_2(n))(-n)$$

$$= a_1x_1(-n) + a_2x_2(-n)$$

$$= a_aT[x_1(n)] + a_2T[x_2(n)]$$

Thus time reversal is linear.

If $T[x(n)] = y(n)$, system T is time-invariant if and only if:

$$T[x(n-n_0)] = y(n-n_0)$$

A time reversed system is defined as $T[x(n)] = y(n)$ where $y(n) = x(-n)$. A shift in the input leads to the same shift in the output. Let's shift the input to the right by n_0. Before performing this, we note that the definition of time-invariance expects this shift in the input to be accompanied by the same shift to the right in the output (i.e. $T[x(n-n_0)] = x(-n-n_0)$).

$$T[x(n-n_0)] = x(-(n-n_0)) = x(-n+n_0)$$

This is a shift to the left of n_0 which contradicts the definition and out expectation.
Therefore the system is not time invariant.

[Question 3] Polynomial Multiplication and Convolution (15 marks)

Vectors can be used to represent polynomials. For example, 3^{rd}-degree polynomial $(a_3x^3 + a_2x^2 + a_1x + a_0)$ can by represented by vector $[a_3, a_2, a_1, a_0]$.
If $\mathbf{u}$ and $\mathbf{v}$ are vectors of polynomial coefficients, prove that convolving them is equivalent to multiplying the two polynomials they each represent.

Hint: You need to assume proper zero-padding to support the full-size convolution.

Let

$$U(x) = (u_nx^n + u_{n-1}x^{n-1} + \ldots + u_1x^1 + u_0)$$

and

$$V(x) = (v_mx^m + v_{m-1}x^{m-1} + \ldots + v_1x^1 + v_0)$$

represent the polynomials of degree n and m respectively. Then the vectors

$$\mathbf{u} = \begin{bmatrix} u_n \\ u_{n-1} \\ \vdots \\ u_1 \\ u_0 \end{bmatrix} \quad \text{and} \quad \mathbf{v} = \begin{bmatrix} v_m \\ v_{m-1} \\ \vdots \\ v_1 \\ v_0 \end{bmatrix}$$

represent the coefficients of the corresponding polynomials $U(x)$ and $V(x)$ respectively. Note that $\mathbf{u} \in \mathbb{R}^{n+1}$ and $\mathbf{v} \in \mathbb{R}^{m+1}$.

$$P(x) = U(x)V(x) = (u_nx^n + u_{n-1}x^{n-1} + \ldots + u_1x^1 + u_0)(v_nx^n + v_{n-1}x^{n-1} + \ldots + v_1x^1 + v_0)$$

$$= \left(\sum_{i=0}^{n} u_ix^i\right)\left(\sum_{j=0}^{n} v_jx^j\right)$$

$$= \sum_{i=0}^{n}\sum_{j=0}^{m} u_iv_jx^{i+j}$$

We note that this polynomial can be expressed as a vector with the coefficients being the sum of u_iv_j's where $i + j = k$.

$$\mathbf{p}_k = \sum_{i+j=k} u_iv_j$$

We can make the substitution $j = k - i$ to get the following.

$$\mathbf{p}_k = \sum_{i+j=k} u_i v_{k-i}$$

Assuming proper zero padding and degree of n, we get

$$\mathbf{p}_k = \sum_{i=0}^{n} u_i v_{k-i}$$

from which it follows that

$$\mathbf{p} = \mathbf{u} * \mathbf{v}$$

as required.

[Question 4] Laplacian Operator (20 marks)

The Laplace operator is a second-order differential operator in the "n"-dimensional Euclidean space, defined as the divergence (∇) of the gradient (∇f). Thus if f is a twice-differentiable real-valued function, then the Laplacian of f is defined by:

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

where the latter notations derive from formally writing:

$$\nabla = \left(\frac{\partial}{\partial x_1}, \ldots, \frac{\partial}{\partial x_n} \right).$$

Now, consider a 2D image $I(x, y)$ and its Laplacian, given by $\Delta I = I_{xx} + I_{yy}$. Here the second partial derivatives are taken with respect to the directions of the variables x, y associated with the image grid for convenience. Show that the Laplacian is in fact rotation invariant. In other words, show that $\Delta I = I_{rr} + I_{r'r'}$, where r and r' are any two orthogonal directions.

Hint: Start by using polar coordinates to describe a chosen location (x, y). Then use the chain rule.

We show that the Laplacian operator $\Delta I = I_{xx} + I_{yy}$ is rotation invariant, by showing that $\Delta I = I_{rr} + I_{r'r'}$ where r and r' are any two orthogonal directions.

First we note that the polar representation of the Cartesian coordinates is as follows.

$$x = r \cos \theta$$
$$y = r \sin \theta$$

Now performing a rotation by an angle α leads to the new coordinates.

$$r_x = r \cos(\theta + \alpha)$$
$$r_y = r \sin(\theta + \alpha)$$

We can use the angle sum identities we can write the new coordinates in terms of the old ones.

$$r_x = r\left(\cos\theta\cos\alpha - \sin\theta\sin\alpha\right) = (r\cos\theta)\cos\alpha - (r\sin\theta)\sin\alpha$$

$$r_x = x\cos\alpha - y\sin\alpha$$

$$r_y = r\left(\cos\theta\sin\alpha + \sin\theta\cos\alpha\right) = (r\cos\theta)\sin\alpha + (r\sin\theta)\cos\alpha$$

$$r_y = x\sin\alpha + y\cos\alpha$$

Returning to the original problem, we can now compute the partial derivatives of I w.r.t x and y in terms of r_x and r_y using chain rule. Note that we now switch to using r and r' instead of r_x and r_y.

$$\begin{aligned} I_x &= \frac{\partial I}{\partial r}\frac{\partial r}{\partial x} + \frac{\partial I}{\partial r'}\frac{\partial r'}{\partial x} = I_r\cos\theta - I_{r'}\sin\theta, \\ I_y &= \frac{\partial I}{\partial r}\frac{\partial r}{\partial y} + \frac{\partial I}{\partial r}\frac{\partial r'}{\partial y} = I_r\sin\theta + I_{r'}\cos\theta. \end{aligned}$$

Now we compute the second-order partial derivatives.

$$\begin{aligned} I_{xx} &= \frac{\partial}{\partial x}(I_x) = \frac{\partial}{\partial r}(I_x)\frac{\partial r}{\partial x} + \frac{\partial}{\partial r'}(I_x)\frac{\partial r'}{\partial x} \\ &= (I_{rr}\cos\theta - I_{rr'}\sin\theta)\cos\theta + (-I_{rr'}\cos\theta + I_{r'r'}\sin\theta)(-\sin\theta) \\ &= I_{rr}\cos^2\theta - 2I_{rr'}\sin\theta\cos\theta + I_{r'r'}\sin^2\theta. \\ I_{yy} &= \frac{\partial}{\partial y}(I_y) = \frac{\partial}{\partial r}(I_y)\frac{\partial r}{\partial y} + \frac{\partial}{\partial r'}(I_y)\frac{\partial r'}{\partial y} \\ &= (I_{rr}\sin\theta + I_{rr'}\cos\theta)\sin\theta + (I_{rr'}\sin\theta + I_{r'r'}\cos\theta)\cos\theta \\ &= I_{rr}\sin^2\theta + 2I_{rr'}\sin\theta\cos\theta + I_{r'r'}\cos^2\theta. \end{aligned}$$

Now we can add I_{xx} and I_{yy} to compute ΔI.

$$\Delta I = I_{xx} + I_{yy} = I_{rr}(\cos^2\theta + \sin^2\theta) + I_{r'r'}(\sin^2\theta + \cos^2\theta).$$

Noting that $\cos^2\theta + \sin^2\theta = 1$:

$$\Delta I = I_{rr} + I_{r'r'}.$$

Thus we have shown that the Laplacian operator is rotation invariant.

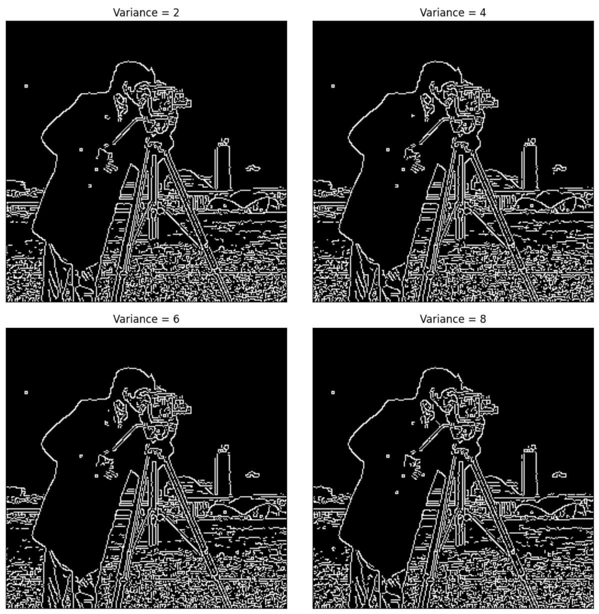
Part II: Implementation Tasks (60 marks)

[Question 5] Canny Edge Detector Robustness (10 marks)

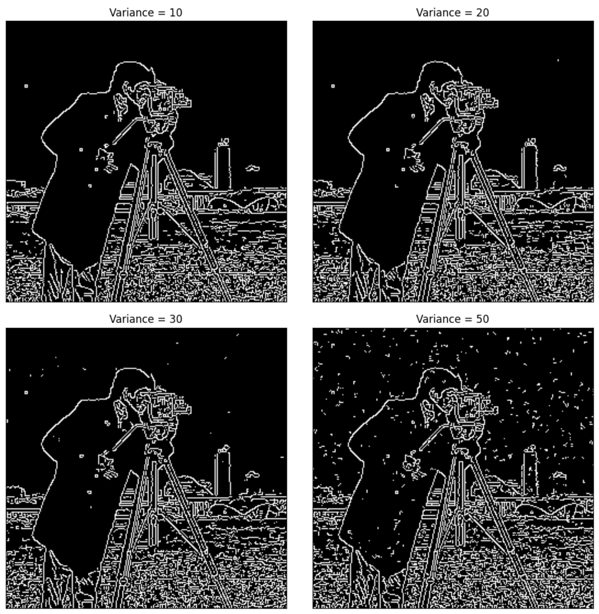
Using the sample code provided in Tutorial 2, examine the sensitivity of the Canny edge detector to Gaussian noise. To do so, take an image of your choice, and add i.i.d Gaussian noise to each pixel. Analyze the performance of the edge detector as a function of noise variance. Include your observations and three sample outputs (corresponding to low, medium, and high noise variances) in the report.

Using the code found in "assignment1/q5.py", IID Gaussian noise was added to each pixel

and it was plotted. Note that a mean of 0 is used for the discussion here but the results were verified for means 125 and 255 as well.
It is observed that as the variance increases, the edge detection degrades in performance significantly. With a mean of 0, and variance of 10, there is no noticeable change. As we increase the variance to 20 and beyond, it very quickly begins to fill with noise and detect nonexistent edges. 20 and 30 are quite reserved but 50 is rather noisy with many hallucinated edges in the background. Beyond 50, most of the image is white. The original edges are still maintained but it finds many nonexistent edges. Note that even at 200, the continuous lines help distinguish the actual edges from the Gaussian noise – though this requires trying really hard.
In Figure 2(b), we see that even with variance at 1000, the cameraman can still be made out. So we need to push further and use variance = 100000. This finally removes the cameraman and all edges from the image. It is purely noise.
The plots can be found in the assignment directory along with the code.

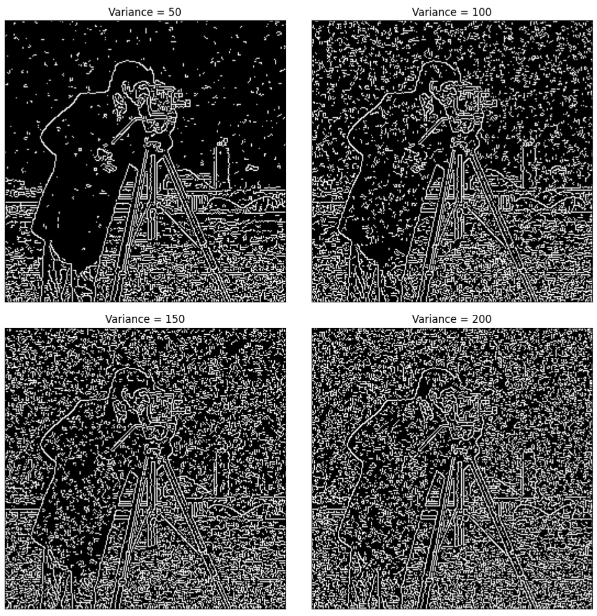


((a)) Variance from 2 to 8.

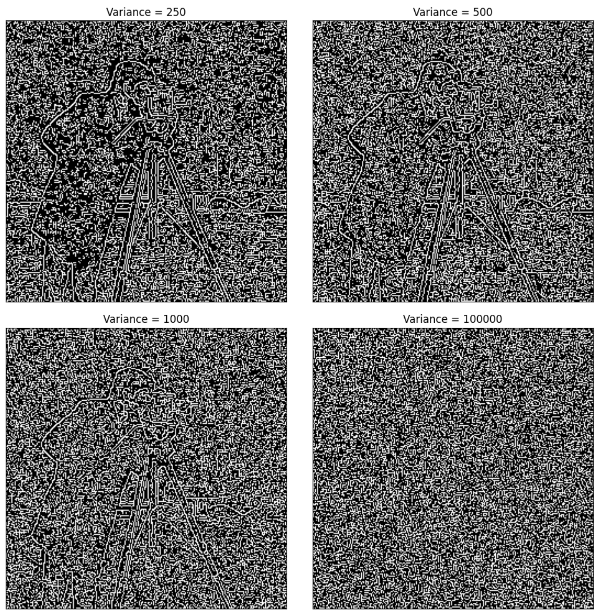


((b)) Variance from 10 to 50.

Figure 1: Combined plots with variance from 2 to 50.



((a)) Variance from 50 to 200.



((b)) Variance from 250 to 100,000.

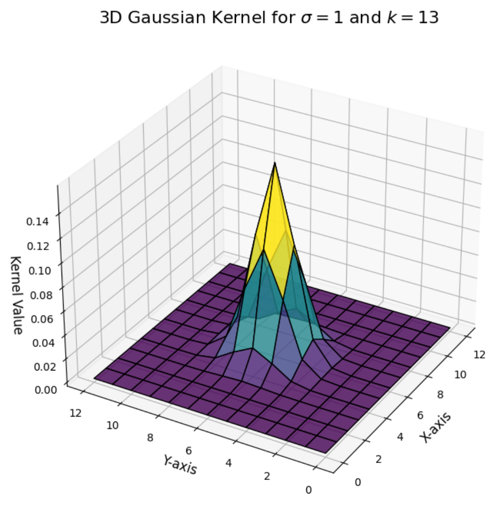
Figure 2: Combined plots with variance from 50 to 100,000.

[Question 6] Edge Detection (50 marks)

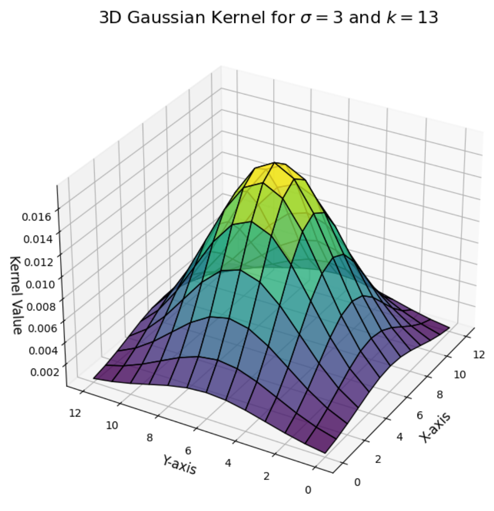
In this question, the goal is to implement a rudimentary edge detection process that uses a derivative of Gaussian, through a series of steps. For each step (excluding step 1) you are supposed to test your implementation on the provided image, and also on one image of your own choice. Include the results in your report.

Step I - Gaussian Blurring (10 marks): Implement a function that returns a 2D Gaussian matrix for input size and scale σ. Please note that you should not use **any** of the existing libraries to create the filter, e.g. cv2.getGaussianKernel(). Moreover, visualize this 2D Gaussian matrix for two choices of σ with appropriate filter sizes. For the visualization, you may consider a 2D image with a colormap, or a 3D graph. Make sure to include the color bar or axis values.

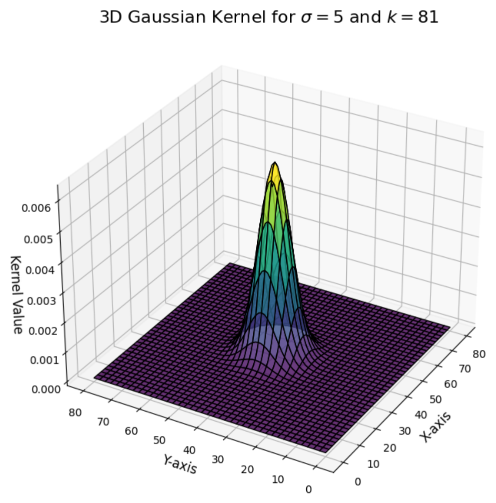
Figure 3 shows the plotted 3D Gaussian kernels for various values of σ and k. Clearly, increasing the standard deviation makes for a less sparse and smoother plot. Axes are labeled so a color plot bar is not included.
The code is located in "assignment1/q6.py" in lines 5 to 70. The parts of the main function wrapped in "`### STEP 1 ###`" and "`### END STEP 1 ###`" are relevant to running the code and generating the plots.



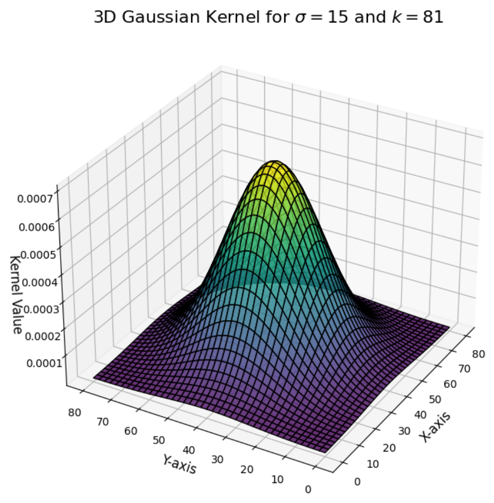
((a)) Kernel with $\sigma = 1$ and $k = 13$.



((b)) Kernel with $\sigma = 3$ and $k = 13$.



((c)) Kernel with $\sigma = 5$ and $k = 81$.



((d)) Kernel with $\sigma = 15$ and $k = 81$.

Figure 3: 3D Gaussian Kernel Plots.

Step II - Gradient Magnitude (10 marks): In the lectures, we discussed how partial derivatives of an image are computed. We know that the edges in an image are from the sudden changes of intensity and one way to capture that sudden change is to calculate the gradient magnitude at each pixel. The edge strength or gradient magnitude is defined as:

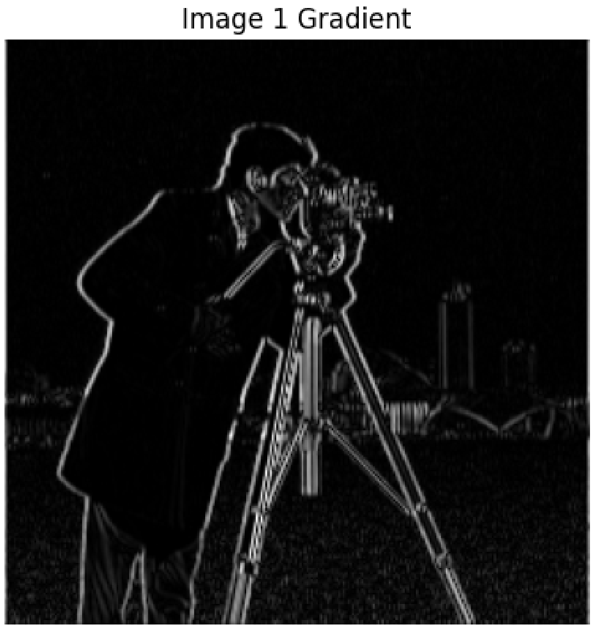
$$g(x, y) = |\nabla f(x, y)| = \sqrt{g_x^2 + g_y^2}$$

where g_x and g_y are the gradients of image $f(x, y)$ along x and y-axis direction respectively.

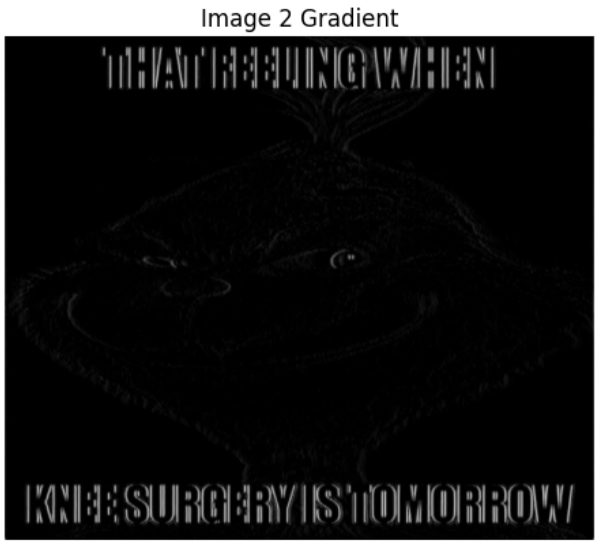
Using the Sobel operator, g_x and g_y can be computed as:

$$g_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * f(x,y) \text{ and } g_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * f(x,y)$$

Implement a function that receives an image $f(x,y)$ as input and returns its gradient $g(x,y)$ magnitude as output using the Sobel operator. You are supposed to implement the convolution required for this task from scratch, without using any existing libraries.
The code is located in "assignment1/q6.py" in lines 72 to 126. The parts of the main function wrapped in "`### STEP 2 ###`" and "`### END STEP 2 ###`" are relevant to running the code and generating the plots. The plots are in "assignment1/plots/q6/".



((a)) Gradient of the provided image.



((b)) Gradient of my image.

Figure 4: Step 2 Gradient Edge Plots.

Step III - Threshold Algorithm (20 marks): After finding the image gradient, the next step is to automatically find a threshold value so that edges can be determined. One algorithm to automatically determine image-dependent threshold is as follows:

1. Let the initial threshold τ_0 be equal to the average intensity of gradient image $g(x,y)$, as defined below:
$$\tau_0 = \frac{\sum_{j=1}^{h} \sum_{i=1}^{w} g(i,j)}{h \times w}$$
where h and w are the height and width of the image under consideration.

2. Set iteration index $i = 0$, and categorize the pixels into two classes, where the lower class consists of the pixels whose gradient magnitudes are less than τ_0, and the upper class contains the rest of the pixels.

3. Compute the average gradient magnitudes m_L and m_H of lower and upper classes, respectively.

4. Set iteration $i = i + 1$ and update threshold value as:

$$\tau_i = \frac{m_L + m_H}{2}$$

5. Repeat steps 2 to 4 until $|\tau_i - \tau_{i-1}| \leq \epsilon$ is satisfied, where $\epsilon \to 0$; take τ_i as final threshold and denote it by τ.

   Once the final threshold is obtained, each pixel of gradient image $g(x, y)$ is compared with τ. The pixels with a gradient higher than τ are considered as edge point and is represented as white pixel; otherwise, it is designated as black. The edge-mapped image $E(x, y)$, thus obtained is:

$$E(x, y) = \begin{cases} 255, & \text{if } g(x, y) \geq \tau \\ 0, & \text{otherwise} \end{cases}$$

Implement the aforementioned threshold algorithm. The input to this algorithm is the gradient image $g(x, y)$ obtained from step II, and the output is a black and white edge-mapped image $E(x, y)$.

Step IV - Test (10 marks): Use the image provided along with this assignment, and also one image of your choice to test all the previous steps (I to III) and to visualize your results in the report. Convert the images to grayscale first. Please note that the input to each step is the output of the previous step. In a brief paragraph, discuss how the algorithm works for these two examples and highlight its strengths and/or its weaknesses.

The code for this section can be found in "assignment1/q6.py" lines 128 to 174. The parts of the main function wrapped in "`### STEP 3 ###`" and "`### END STEP 3 ###`" are relevant to running the code and generating the plots. The plots are in "assignment1/plots/q6/". Figure 3 contains the plots for Step 1. Figure 4 contains the plots for Step 2. Figure 5 and Figure 6 demonstrate the threshold algorithm (Step 3).

The algorithm classifies pixels into two classes and updates the mean until the threshold is achieved. It stops when the update to τ is less than ϵ.

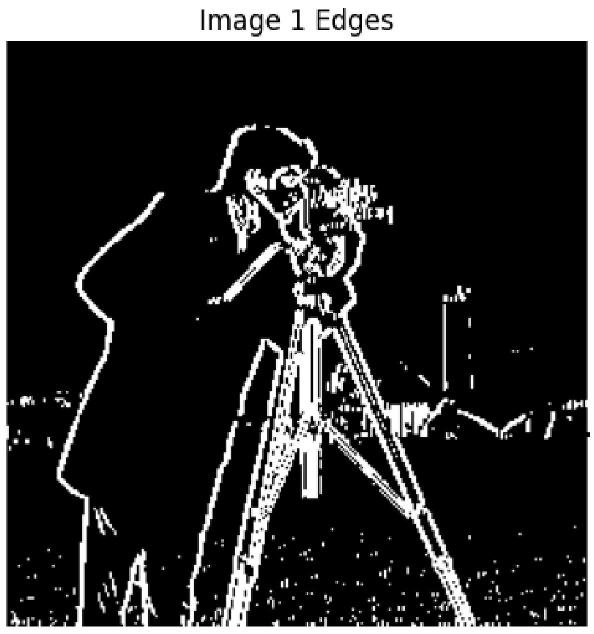
To see the impact of 1 iteration, we set $\epsilon = 100$, as demonstrated in Figure 5. The result of this is a thickening of the edges, enhancing both the lines and noise.
Setting $\epsilon = 1$, we see a noise reduction and thinning of the edges. The edges are still much thicker and more solid than the initial gradient. In my image, the Grinch has become difficult to see, save his eyes. This makes sense as this image has a consistent blend of colours on his furry face. It is also a synthetic image so we anticipate less noise. Plots for this are shown in Figure 6.
Pushing it to the other extreme, we set $\epsilon = 0$ and run 100 iterations. This leads to the figures

in Figure 7. There is little difference, if any, from $\epsilon = 1$. This is expected as the changes in τ approach 0, leading to convergence.

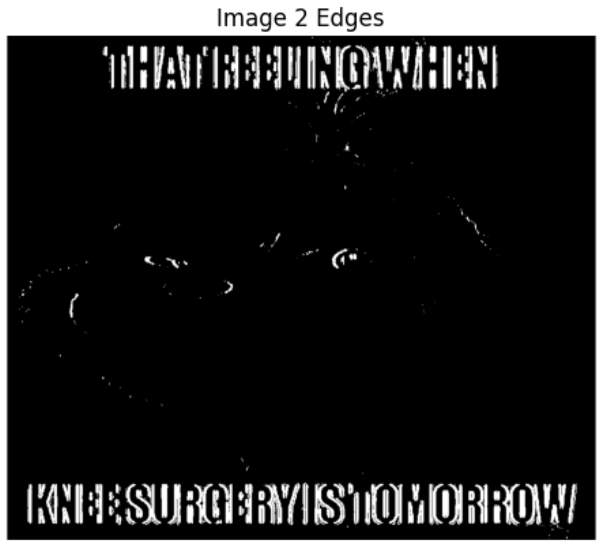
From these experiments, it is revealed that the thresholding algorithm has 2 functions; 1) it makes edges sharper by either setting them to 0 or 255, 2) it reduces noise.
The strengths of this algorithm are clearly that it can strengthen and thicken edges from image gradients. It does not require Fourier Transforms and can be easily implemented and run on CPUs. It also converges quickly and does not have any hyperparamenters, other than ϵ.

One of the main shortcomings is the struggle with low-visibility edges. In both the provided image and my image, dimmer edges are discarded as noise and set to 0. This might be a more pronounced issue in synthetic images with good blending as these images, such as the one I chose, have less pronounced gradients and might be overlooked as noise. I also note that the ϵ parameter is kind of useless as the algorithm converges so quickly that `max_iters` may be more meaningful.

Please note that neither the assignment nor piazza restricts our image of choice to be a natural image.

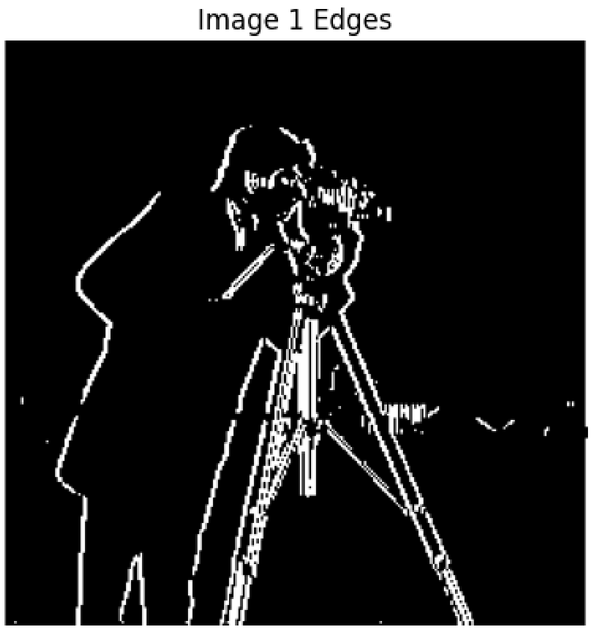


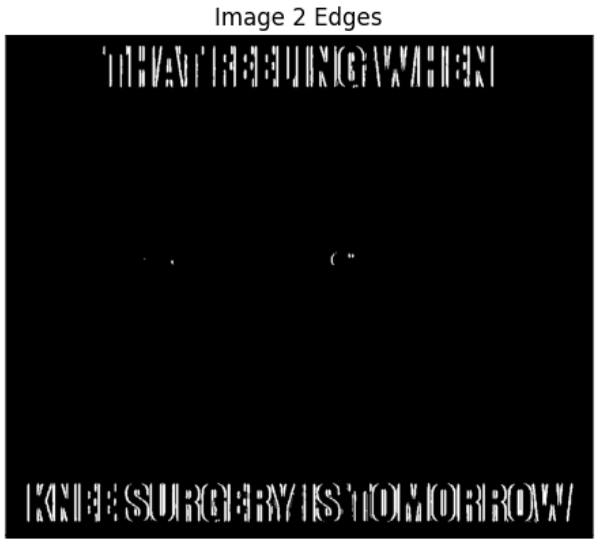
((a)) Provided image with $\epsilon = 100$ (1 iteration).



((b)) My image with $\epsilon = 100$ (1 iteration).

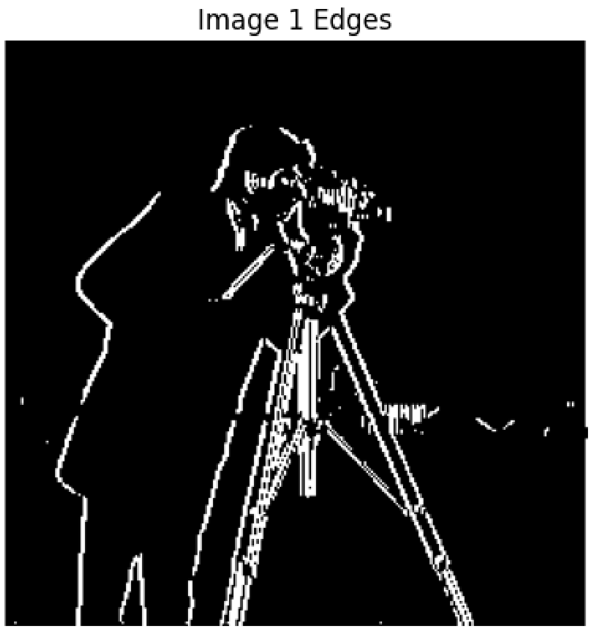
Figure 5: Threshold Algorithm Plots Of Gradients for $\epsilon = 100$ and $\epsilon = 10$.

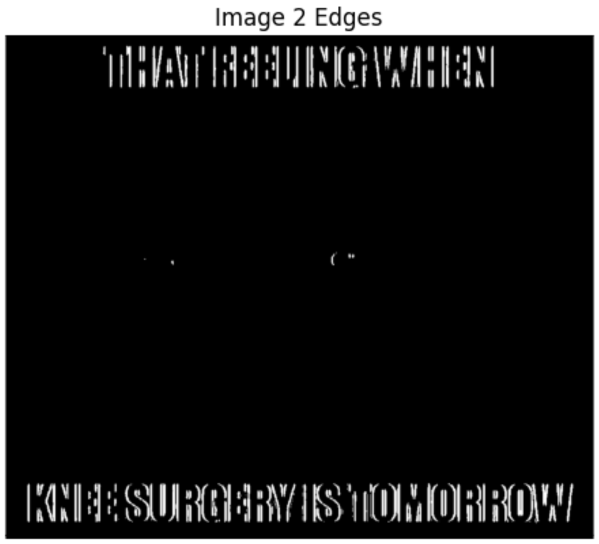




((a)) Provided image with $\epsilon = 1$ (7 iterations). ((b)) My image with $\epsilon = 1$ (7 iterations).

Figure 6: Threshold Algorithm Plots Of Gradients for $\epsilon = 1$.





((a)) Provided image with $\epsilon = 0$ (101 iterations). ((b)) My image with $\epsilon = 0$ (101 iterations).

Figure 7: Threshold Algorithm Plots Of Gradients for $\epsilon = 0$.