DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING, UCLA
EC ENGR 188: THE FOUNDATIONS OF COMPUTER VISION

INSTRUCTOR: Prof. Achuta Kadambi
TA: Rishi Upadhyay

NAME: Simon Lee
UID: 505310387

HOMEWORK 1

PROBLEM	TYPE	TOPIC	MAX. POINTS
1	Analytical	LSI Systems	10
2	Coding	2D-Convolution	5
3	Coding	Image Blurring and Denoising	15
4	Coding	Image Gradients	5
5	Analytical + Coding	Image Filtering	15
6	Analytical	Interview Question (Bonus)	10

Motivation

The long-term goal of our field is to teach robots how to see. The pedagogy of this class (and others at peer schools) is to take a *bottom-up* approach to the vision problem. To teach machines how to see, we must first learn how to represent images (lecture 2), clean images (lecture 3), and mine images for features of interest (edges are introduced in lecture 4 and will thereafter be a recurring theme). This is an evolution in turning the matrix of pixels (an unstructured form of "big data") into something with structure that we can manipulate.

The problem set consists of:

- analytical questions to solidify the concepts covered in the class, and
- coding questions to provide a basic exposure to image processing using Python.

You will explore various applications of convolution such as image blurring, denoising, filtering and edge detection. These are fundamental concepts with applications in many computer vision and machine learning applications. You will also be given a computer vision job interview question based on the lecture.

Homework Layout

The homework consists of 6 problems in total, with subparts for each problem. There are 2 types of problems in this homework - analytical and coding. All the problems need to be answered in the Overleaf document here: `https://www.overleaf.com/read/wtqbbvbgqrzn#e962bb`. Make a copy of the Overleaf project, and fill in your answers for the questions in the solution boxes provided.

For the analytical questions you will be directly writing their answers in the space provided below the questions. For the coding problems you need to use the Jupyter notebook provided here: `https://colab.research.google.com/drive/1Te890U1Q6yf4cRBAYeB-YFaIPY6Wv9js?usp=sharing` (see the Jupyter notebook for each sub-part which involves coding). After writing your code in the Jupyter notebook you need to copy paste the same code in the space provided below that question on Overleaf. For instance, for Question 2 you have to write a function 'conv2D' in the Jupyter notebook (and also execute it) and then copy that function in the box provided for Question 2 here in Overleaf. In some questions you are also required to copy the saved images (from Jupyter) into the solution boxes in Overleaf. For instance, in Question 3.2 you will be visualizing the gaussian filter. So you will run the corresponding cells in Jupyter (which will save an image for you in PDF form) and then copy that image in Overleaf.

Submission

Submission will be done via Gradescope. You will need to submit three things: (1) this PDF with answers filled out (from Overleaf), (2) Your .ipynb file, (3) a PDF printout of your .ipynb file with all cells executed. You do not need to create a folder, you will be able to upload all three files directly to gradescope.

Software Installation

You will need Jupyter to solve the homework. You may find these links helpful:

- Jupyter (https://jupyter.org/install)
- Anaconda (https://docs.anaconda.com/anaconda/install/)

1 Image Processing

1.1 Periodic Signals (1.0 points)

Is the 2D complex exponential $x(n_1,n_2) = \exp(j(\omega_1 n_1 + \omega_2 n_2))$ periodic in space? Justify.

To check if the 2D complex exponention $x(n_1,n_2) = \exp(j(\omega_1 n_1 + \omega_2 n_2))$ is periodic in space we need to find an N_1 and N_2 where $x(n_1,n_2) = x(n_1+N_1,n_2)$ and $x(n_1,n_2) = x(n_1,n_2+N_2)$.

We therefore plug in $x(n_1,n_2) = x(n_1+N_1,n_2)$ to the equation and obtain

$$\exp(j(\omega_1(n_1+N_1)+\omega_2 n_2))$$

Using distribution and properties of exponents, we can further distribute and simplify the equation:

$$\exp(j\omega_1 n_1 + j\omega_1 N_1 + j\omega_2 n_2)$$

Now, we split the exponent into two parts and simplify the left equation to $x(n_1,n_2)$

$$\exp(j\omega_1 n_1 + j\omega_2 n_2)\cdot\exp(j\omega_1 N_1) = x(n_1,n_2)\cdot\exp(j\omega_1 N_1)$$

Therefore in order for $x(n_1,n_2) = x(n_1+N_1,n_2)$, the following must occur

$$x(n_1,n_2)\cdot\exp(j\omega_1 N_1) = x(n_1,n_2)$$
$$\exp(j\omega_1 N_1) = \frac{x(n_1,n_2)}{x(n_1,n_2)} = 1$$

This implies that:

$$\exp(j\omega_1 N_1) = 1$$

If we repeat this in the n_2 direction we should find that

$$\exp(j\omega_2 N_2) = 1$$

Therefore in order to find conditions that meet the following criteria:

$$\exp(j\omega_1 N_1) = 1$$

$$\exp(j\omega_2 N_2) = 1$$

$\omega_1 \cdot N_1 = 2\pi \cdot i_1$, the value $\omega_1 \cdot N_1$ must be an integer mutliple of 2π.

Therefore for a function to be periodic in the n_1 direction, ω_1 must be:

$$\omega_1 = \frac{2\pi \cdot i_1}{N_1}$$

.

n_2 direction follows a similar format and ω_2 comes out to:

$$\omega_2 = \frac{2\pi \cdot i_2}{N_2}$$

.

1.2 Working with LSI systems (3.0 points)

Consider an LSI system $T[x] = y$ where x is a 3 dimensional vector, and y is a scalar quantity. We define 3 basis vectors for this 3 dimensional space: $x_1 = [1,0,0]$, $x_2 = [0,1,0]$ and $x_3 = [0,0,1]$.
(i) Given $T[x_1] = a$, $T[x_2] = b$ and $T[x_3] = c$, find the value of $T[x_4]$ where $x_4 = [5,4,3]$. Justify your approach briefly (in less than 3 lines).
(ii) Assume that $T[x_3]$ is unknown. Would you still be able to solve part (i)?
(iii) $T[x_3]$ is still unknown. Instead you are given $T[x_5] = d$ where $x_5 = [1,-1,-1]$. Is it possible to now find the value of $T[x_4]$, given the values of $T[x_1], T[x_2]$ (as given in part (i)) and $T[x_5]$? If yes, find $T[x_4]$ as a function of a,b,d; otherwise, justify your answer.

(i) We can find $T[x_4]$ by taking a linear combination of the inputs which should be the same linear combination as the outputs:

$$T[x_4] = T[5x_1 + 4x_2 + 3x_3] = 5T[x_1] + 4T[x_2] + 3T[x_3] = 5a + 4b + 3c$$

(ii) If we do not know $T[x_3]$ we cannot solve the system since $3T[x_3]$ will be unknown.

(iii) we can express x_5 as a combination of x_1, x_2, x_3. which can then help us find the value of x_3

$$x_5 = x_1 - x_2 - x_3 =$$

$$x_3 = x_1 - x_2 - x_5$$

Now we can plug it back into the equation from part (i) and distribte:

$$T[x_4] = T[5x_1 + 4x_2 + 3(x_1 - x_2 - x_5)] = T[8x_1 + 1x_2 - 3x_5]$$

Lastley we can apply the linear combination like we did in part (i):

$$8a + b - 3d, \text{where } d \text{ represents new } x_5$$

1.3 Space invariance (2.0 points)

Evaluate whether these 2 linear systems are space invariant or not. (The answers should fit in the box.)
(i) $T_1[x(n_1)] = 2x(n_1)$
(ii) $T_2[x(n_1)] = x(2n_1)$.

A linear system is space invariant if it follows the following property:

$$T[x(n-k)] = T[x(n)] - k$$

The above equation is saying that if there is a shift in the input signal, the output signal shifts by the same amount, without any change to the signal.

(i) For T_1: - Original system: $T_1[x(n_1)] = 2x(n_1)$ - Apply a shift k to the input: $x(n_1 - k)$ - Output with shifted input: $T_1[x(n_1 - k)] = 2x(n_1 - k)$

Since $2x(n_1 - k)$ is exactly the shifted version of $2x(n_1)$ by k, system T_1 is space-invariant.

(ii) For T_2: - Original system: $T_2[x(n_1)] = x(2n_1)$ - Apply a shift k to the input: $x(n_1 - k)$ - Output with shifted input: $T_2[x(n_1 - k)] = x(2(n_1 - k)) = x(2n_1 - 2k)$

The output $x(2n_1 - 2k)$ is a scaled version of the shift, specifically twice the shift applied to the input. Therefore this one is not space-invariant

1.4 Convolutions (4.0 points)

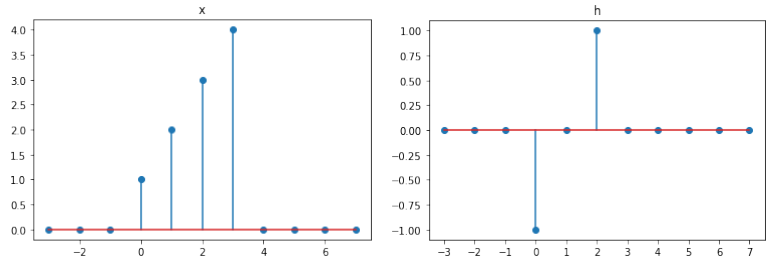


Figure 1: (a) Graphical representation of x (b) Graphical representation of h

Consider 2 discrete 1-D signals $x(n)$ and $h(n)$ defined as follows:

$$\begin{aligned} x(i) &= i+1 \quad \forall i \in \{0,1,2,3\} \\ x(i) &= 0 \quad \forall i \notin \{0,1,2,3\} \\ h(i) &= i-1 \quad \forall i \in \{0,1,2\} \\ h(i) &= 0 \quad \forall i \notin \{0,1,2\} \end{aligned} \tag{1}$$

(i) Evaluate the discrete convolution $h * x$.
(ii) Show how you can evaluate the non-zero part of $h * x$ as a product of 2 matrices H and X. Use the commented latex code in the solution box for typing out the matrices.

(i) The discrete convolution of two signals $x(n)$ and $h(n)$ is defined as:

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

We also know x(i) and $h(i)$ from our probelm.

$$x(i) = \{1,2,3,4\} \text{ for } i \in \{0,1,2,3\}$$
$$h(i) = \{-1,0,1\} \text{ for } i \in \{0,1,2\}$$

Now, we calmculate the convolution $y(n) = h * x$:

$$y(n) = \sum_{k=0}^{2} x(n-k) \cdot h(k)$$

Let's calculate this for each n from 0 to 5:

- For $n = 0$:
$$y(0) = x(0) \cdot h(0) = 1 \cdot (-1) = -1$$
- For $n = 1$:
$$y(1) = x(1) \cdot h(0) + x(0) \cdot h(1) = 2 \cdot (-1) + 1 \cdot 0 = -2$$
- For $n = 2$:
$$y(2) = x(2) \cdot h(0) + x(1) \cdot h(1) + x(0) \cdot h(2) = 3 \cdot (-1) + 2 \cdot 0 + 1 \cdot 1 = -2$$
- For $n = 3$:
$$y(3) = x(3) \cdot h(0) + x(2) \cdot h(1) + x(1) \cdot h(2) = 4 \cdot (-1) + 3 \cdot 0 + 2 \cdot 1 = -2$$
- For $n = 4$:
$$y(4) = x(3) \cdot h(1) + x(2) \cdot h(2) = 4 \cdot 0 + 3 \cdot 1 = 3$$

- For $n = 5$:

$$y(5) = x(3) \cdot h(2) = 4 \cdot 1 = 4$$

So the convolution $y = h * x$ is:

$$y = \{-1, -2, -2, -2, 3, 4\}$$

(ii) To represent the convolution operation as a matrix multiplication, we can create a Toeplitz matrix (Wikipedia - Toeplitz matrix) from $h(i)$ and multiply it with the signal $x(i)$ (he mentioned the Toeplitz matrix in class briefly).

Matrix H (Toeplitz matrix) will be of size 6×4 (since the convolution result has 6 elements and x has 4 elements), and matrix X will be a column vector with 4 elements (elements of x).

$$H = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$X = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

The matrix multiplication HX gives:

$$HX = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} -1 \\ -2 \\ -2 \\ -2 \\ 3 \\ 4 \end{bmatrix}$$

This is the same result as part *(i).

2 2D-Convolution (5.0 points)

In this question you will be performing 2D convolution in Python. Your function should be such that the convolved image will have the same size as the input image i.e. you need to perform zero padding on all the sides. (See the Jupyter notebook.)

This question is often asked in interviews for computer vision/machine learning jobs.

Make sure that your code is within the bounding box below.

```
def conv2D(image: np.array, kernel: np.array = None):
    """
    Perform 2D convolution on an image ensuring
    the output has the same size as the input.

    Author: Simon Lee
    """
    try:
        # If no kernel is provided, raise an error
        if kernel is None:
            raise ValueError("Kernel cannot be None. Please \
            provide a valid kernel.")

        # Like performing a convolution graphically,
        # we begin by flipping the kernel for convolution
        kernel = np.flipud(np.fliplr(kernel))

        # We define the padding sizes around all the sides of the image
        pad_height = (kernel.shape[0] - 1) // 2
        pad_width = (kernel.shape[1] - 1) // 2

        # Pad the input image with zeros on all sides
        padded_image = np.pad(image,
                              ((pad_height, pad_height),
                              (pad_width, pad_width)),
                              mode='constant',
                              constant_values=0
                              )

        # We need to initialize the output by creating a
        # np.array of size
        image
        convolved_image = np.zeros_like(image)
```

```
        # Convolution loop (From Lecture)
        for x in range(image.shape[0]):
            for y in range(image.shape[1]):
                # Extract the region to perform convolution
                region = padded_image[x:x+kernel.shape[0],
                                      y:y+kernel.shape[1]]
                # Perform element-wise multiplication and sum the result
                convolved_image[x, y] = np.sum(region * kernel)

        return convolved_image

    # except blocks for ValueError and Other Errors
    except ValueError as ve:
        print(f"Value Error: {ve}")
    except Exception as e:
        print(f"An error occurred: {e}")
```

3 Image Blurring and Denoising (15.0 points)

In this question you will be using your convolution function for image blurring and denoising. Blurring and denoising are often used by the filters in the social media applications like Instagram and Snapchat.

3.1 Gaussian Filter (3.0 points)

In this sub-part you will be writing a Python function which given a filter size and standard deviation, returns a 2D Gaussian filter. (See the Jupyter notebook.)

Make sure that your code is within the bounding box.

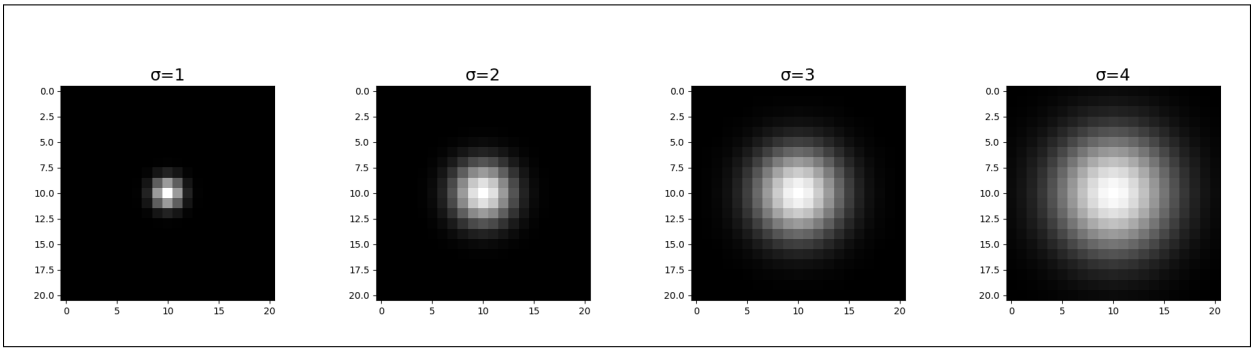
```
def gaussian_filter(size: int, sigma: float):
    """
    Generate a 2D Gaussian filter with given size and standard deviation.

    Author: Simon Lee
    """
    try:
      # Check for weird input
      if size % 2 == 0 or size <= 0:
          raise ValueError("Size must be a positive odd integer.")
      if sigma <= 0:
          raise ValueError("Sigma must be a positive number.")
      # We need to initialize the output by creating a np.array of
      # size x size image
      gaussian_filter = np.zeros((size, size))

      # Compute the mean (center of the filter)
      mean = (size - 1) / 2

      # sigma - std deviation
      # Compute the Gaussian filter
      for x in range(size):
          for y in range(size):
              gaussian_filter[x, y] = \
              (1 / (2 * np.pi * sigma**2)) * \
              np.exp(-(((x - mean)**2 + \
              (y - mean)**2) / (2 * sigma**2)))

      # Normalize the filter to make sure it sums to 1
      gaussian_filter /= np.sum(gaussian_filter)
```

```
        return gaussian_filter
    except ValueError as ve:
        print(f"Value Error: {ve}")
    except ZeroDivisionError:
        print("ZeroDivisionError: Sigma must not be zero.")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
```

3.2 Visualizing the Gaussian filter (1.0 points)

(See the Jupyter notebook.) You should observe that increasing the standard deviation (σ) increases the radius of the Gaussian inside the filter.

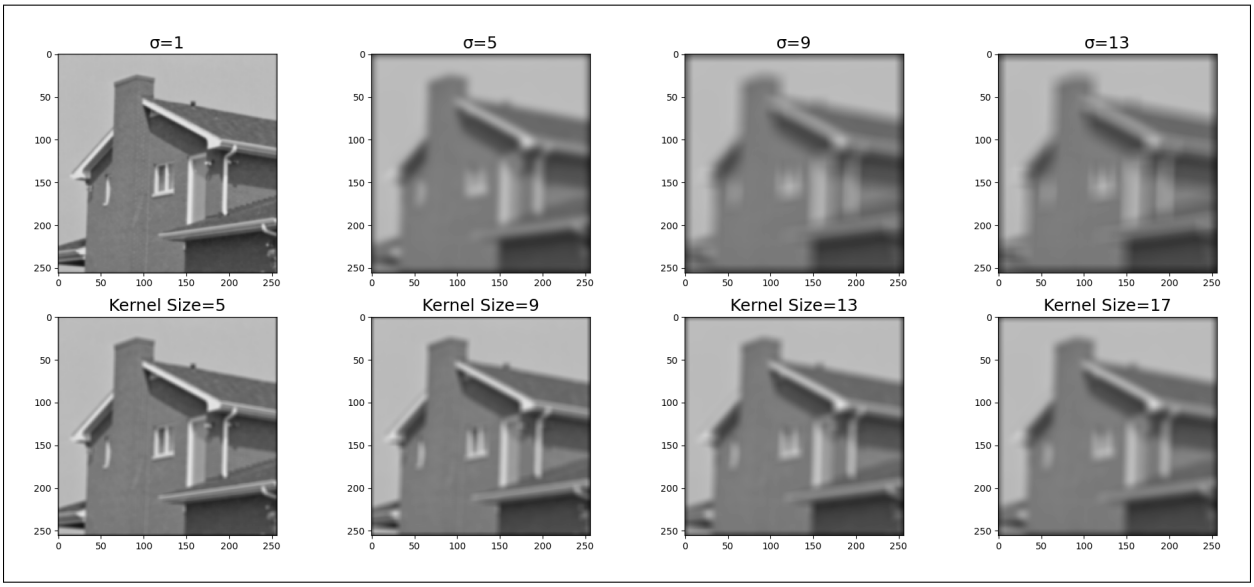
Copy the saved image from the Jupyter notebook here.



3.3 Image Blurring: Effect of increasing the filter size and σ (1.0 points)

(See the Jupyter notebook.) You should observe that the blurring should increase with the kernel size and the standard deviation.

Copy the saved image from the Jupyter notebook here.



3.4 Blurring Justification (2.0 points)

Provide justification as to why the blurring effect increases with the kernel size and σ?

Kernel Size: When we increase the kernel size, we effectiveky increase the space over which the averaging takes place. Therefore finer details will be blurred as a result.

Standard Deviation σ: Similar to kernel size, the standard deviation governs the width of the Gaussian. Therefore the distant pixels influence the blurring much more and the filter itself mixes the values over the large distance creating a blurring type effect which increases with the larger values of σ.

3.5 Median Filtering (3.0 points)

In this question you will be writing a Python function which performs median filtering given an input image and the kernel size. (See the Jupyter notebook.)

Make sure that your code is within the bounding box.

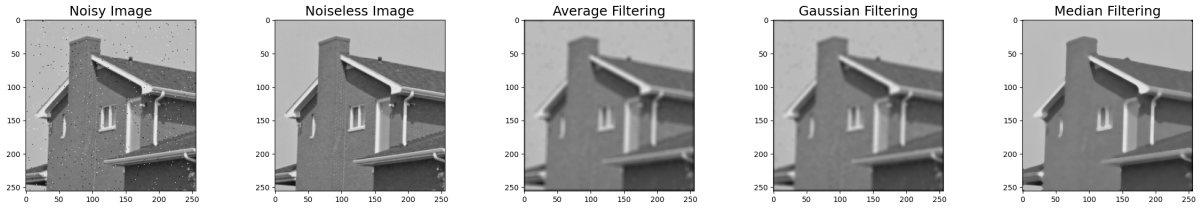
```
def median_filtering(image: np.array, kernel_size: int = None):
    """
    Perform median filtering on an image with given kernel size.

    Author: Simon Lee
    """
    try:
```

```
        # Checks for weird input
        if kernel_size % 2 != 1 or kernel_size <= 0:
            raise ValueError("Kernel size must be a \
                              positive odd number.")

        # Compute padding size
        pad_size = kernel_size // 2

        # Similar to 2dconv function pad the input
        # image with zeros on all sides
        padded_image = np.pad(image,
                              pad_size,
                              mode='constant',
                              constant_values=0)

        # Initialize output
        median_filtered_image = np.zeros_like(image)

        # Median filtering loop
        for x in range(image.shape[0]):
            for y in range(image.shape[1]):
                # get region
                region = padded_image[x:x + kernel_size,
                                      y:y + kernel_size]
                # Compute the median of the region
                median_filtered_image[x, y] = np.median(region)

        return median_filtered_image

    except ValueError as ve:
        print(f"Value Error: {ve}")
    except TypeError as te:
        print(f"Type Error: {te}")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
```

3.6 Denoising (1.0 points)

(See the Jupyter notebook.)

Copy the saved image from the Jupyter notebook here.



3.7 Best Filter (2.0 points)

In the previous part which filtering scheme performed the best? And why?

The median filtering appeared to perform the best out of the three filtering systems. One reason is that the median is a robust statistic and this is evident in the presence of salt-and-pepper noise. The reason behind this is that the median is less sensitive to potential outliers or extreme values compared to the mean. Meanwhile the Gaussian and average filtering shcemes are both based on the mean.

3.8 Preserving Edges (2.0 points)

Which of the 3 filtering methods preserves edges better? And why? Does this align with the previous part?

Another reason median filter worked the best is because it is much better at edge preservation where the median is more likely to preserve the real value of the edges versus the mean which can again be moved by either noise or extreme values. These values can in turn blur the edges making Gaussian and average filtering worse performance in this case.

4 Image Gradients (5.0 points)

In this question you will be visualizing the edges in an image by using gradient filters. Gradients filters, as the name suggests, are used for obtaining the gradients (of the pixel intensity values with respect to the spatial location) of an image, which are useful for edge detection.

4.1 Horizontal Gradient (1.0 points)

In this sub-part you will be designing a filter to compute the gradient along the horizontal direction. (See the Jupyter notebook.)

Make sure that your code is within the bounding box.

```
# Author: Simon Lee
# Define the Prewitt operator for horizontal gradient
# source: https://en.wikipedia.org/wiki/Prewitt_operator
gradient_x = np.array([[1, 0, -1],
                       [1, 0, -1],
                       [1, 0, -1]])
```

4.2 Vertical Gradient (1.0 points)

In this sub-part you will be designing a filter to compute the gradient along the vertical direction. (See the Jupyter notebook.)

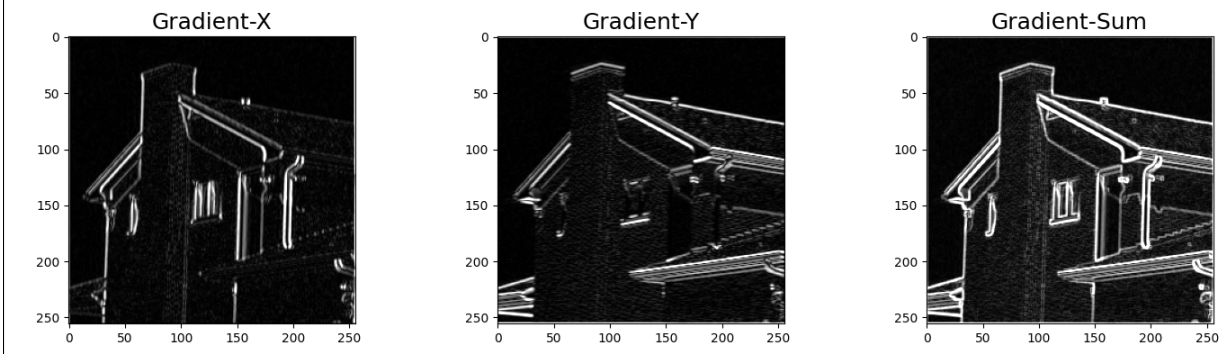
Make sure that your code is within the bounding box.

```
# Author: Simon Lee
# Define the Prewitt operator for vertical gradient
# source: https://en.wikipedia.org/wiki/Prewitt_operator
gradient_y = np.array([[1, 1, 1],
                       [0, 0, 0],
                       [-1, -1, -1]])
```

4.3 Visualizing the gradients (1.0 points)

(See the Jupyter notebook.)

Copy the saved image from the Jupyter notebook here.



4.4 Gradient direction (1.0 points)

Using the results from the previous part how can you compute the gradient direction at each pixel in an image?

If you look at the wikipedia page for Prewitt operator, it says explicitly you can calculate the gradient direction at each pixel using the following forumla:

$$\Theta = \arctan 2(G_y, G_x)$$

Source: Wikipedia - Prewitt Operator

4.5 Separable filter (1.0 points)

Is the gradient filter separable? If so write it as a product of 1D filters.

Yes, the gradient filter is separable. You can rewrite the prewitt operators (both horizontal and vertical) as two separate one dimensional filters in succession.:

$$\text{Prewitt}_x = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = v \cdot h = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$$

$$\text{Prewitt}_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} = v \cdot h = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$$

5 Beyond Gaussian Filtering (15.0 points)

5.1 Living life at the edge (3.0 points)

The goal is to understand the weakness of Gaussian denoising/filtering, and come up with a better solution. In the lecture and the coding part of this assignment, you would have observed that Gaussian filtering does not preserve edges. Provide a brief justification.

[Hint: Think about the frequency domain interpretation of a Gaussian filter and edges.]

In the frequency domain, a Gaussian filter is a low-pass filter, which means it allows low-frequency components to pass through while attenuating high-frequency components.

Therefore high frequencies like noise, details, and edges are especially affected causing a blur like effect in those areas. I found a nice quote on trying to understand edges sourced from this textbook from USF: Textbook. It states, "An edge in an image is a significant local change in the image intensity" implying it is of high frequency.

5.2 How to preserve edges (2.0 points)

Can you think of 2 factors which should be taken into account while designing filter weights, such that edges in the image are preserved? More precisely, consider a filter applied around pixel p in the image. What 2 factors should determine the filter weight for a pixel at position q in the filter window?

1. **Distance:** One factor could be the proximity of pixels. The weights of the the pixels can be determined by how close pixel p is to pixel q. If pixels are next to each other, they are likely to be from a similar region.
2. **Difference:** Another factor could be the difference in the intensity of the pixels, defined by: $p - q$. The weights of the filter can be determined by the intensity of pixel p and its intensity difference from pixel q. This ensures that pixels with similar intensities receive higher weights, while pixels with large differences receive lower weights. Such a weighting scheme allows the filter to preserve edges effectively, as the low weights assigned in areas of high intensity difference (like edges) prevent blurring across these areas.

5.3 Deriving a new filter (2.0 points)

For an image I, we can denote the output of Gaussian filter around pixel p as

$$GF[I_p] = \sum_{q \in S} G_\sigma(\|p - q\|) I_q.$$

I_p denotes the intensity value at pixel location p, S is the set of pixels in the neighbourhood of pixel p. G_{σ_p} is a 2D-Gaussian distribution function, which depends on $\|p - q\|$, i.e. the spatial distance

between pixels p and q. Now based on your intuition in the previous question, how would you modify the Gaussian filter to preserve edges?

[Hint: Try writing the new filter as

$$BF[I_p] = \sum_{q \in S} G_\sigma(\|p-q\|) f(I_p, I_q) I_q.$$

What is the structure of the function $f(I_p, I_q)$? An example of structure is $f(I_p, I_q) = h(I_p \times I_q)$ where $h(x)$ is a monotonically increasing function in x?]

We can rewrite the new Gaussian filter as:

$$BF[I_p] = \sum_{q \in S} G_\sigma(\|p-q\|) f(I_p, I_q) I_q.$$

where (I_p, I_q), is the difference in intensity pixel of the central pixel p from q.

The structure of the function could be something along the lines of a **bilateral filter** which is known to have the previous properties stated (distance, difference). Bilateral Filter Source: Here

$$f(I_p, I_q) = \exp\left(-\frac{(I_p - I_q)^2}{2\sigma_r^2}\right).$$

This function takes the difference of intensity of pixel $p-q$ and then models a Gaussian distribution around the center point p. The way this equation works is when the pixels are identicial the filter should have the largest effect/weight (e.g. $e^0 = 1$), but when the pixels differ the negative exponential will lower the weight of that (e.g. $e^{-0.8} = 0.449$). Its important to know the σ parameter also acts as how fast this functions value increases/decreases governing the exponent.

We also want to include some type of normalization to our results by dividing by the combined weights (distance [kernel] * difference) [*I added this last step thanks to one of the parameters in question 5.5*].

5.4 Complete Formula (3.0 points)

Check if a 1D-Gaussian function satisfies the required properties for $f(.)$ in the previous part. Based on this, write the complete formula for the new filter BF.

5.5 Filtering (3.0 points)

In this question you will be writing a Python function for this new filtering method (See the Jupyter notebook.)

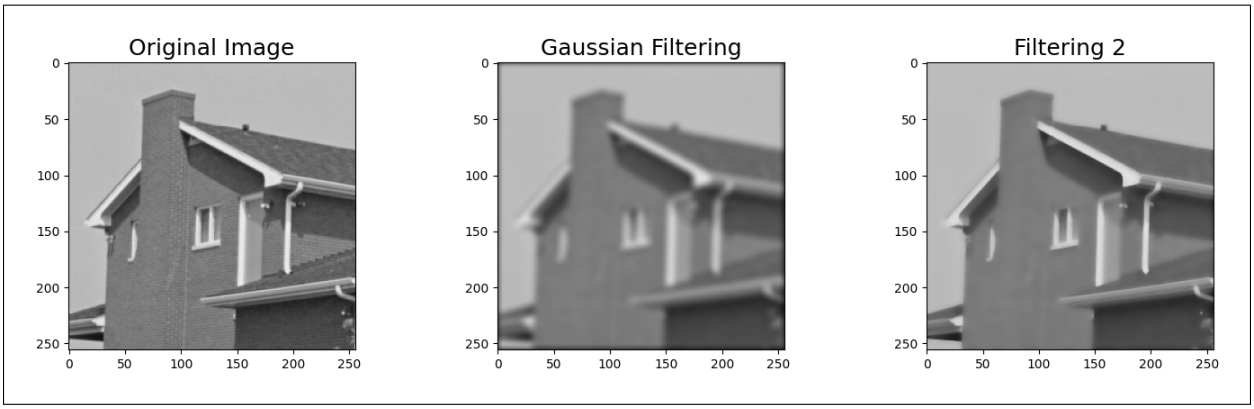
Make sure that your code is within the bounding box.

```
def filtering_2(image: np.array,
                kernel: np.array = None,
                sigma_int: float = None,
                norm_fac: float = None):
    """
    Apply the second Gaussian Filter modeled in Question 5.4. Basically
    a bilateral filter

    source: https://en.wikipedia.org/wiki/Bilateral_filter

    Author Simon Lee
    """
    try:
        # check for input errors
        kernel_size = kernel.shape[0]
        if kernel_size % 2 != 1:
            raise ValueError("Kernel size must be an odd number.")
        if sigma_int <= 0:
            raise ValueError("sigma_int must be positive.")
        if norm_fac <= 0:
            raise ValueError("norm_fac must be positive.")

        # padding size
        pad_size = kernel_size // 2

        # pad input with zeros
        padded_image = np.pad(
                              image,
                              pad_size,
                              mode='constant',
                              constant_values=0)

        # output image matrix
        filtered_image = np.zeros_like(image)

        # Compute new filter
        for i in range(pad_size,
```

```
                    padded_image.shape[0] - pad_size):
            for j in range(pad_size,
                           padded_image.shape[1] - pad_size):
                # region
                region = padded_image[
                                      i-pad_size:i+pad_size+1,
                                      j-pad_size:j+pad_size+1
                                      ]
                # Compute intensity weights
                intensity_weights = np.exp(-((region -\
                                              image[
                                                  i-pad_size,
                                                  j-pad_size])**2) /
                                           (2 * sigma_int**2))
                # Compute combined weights
                combined_weights = kernel * intensity_weights
                # Compute sum of Equation from Question 5.4.
                # Includes normalization at the end
                filtered_image[i-pad_size, j-pad_size] = \
                                           np.sum(
                                               combined_weights * region) / \
                                           np.sum(combined_weights)

        return filtered_image

    except ValueError as error:
        print(f"Value Error: {error}")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
```

5.6 Blurring while preserving edges (1.0 points)

Copy the saved image from the Jupyter notebook here.



5.7 Cartoon Images (1 points)

Natural images can be converted to their cartoonized versions using image processing techniques. A cartoonized image can generated from a real image by enhancing the edges, and flattening the intensity variations in the original image. What operations can be used to create such images? [Hint: Try using the solutions to some of the problems covered in this homework.]

Figure 2: (a) Cartoonized version (b) Natural Image

- **Edge Detection:** You can find the facial boundaries using some edge detection
- **Using Gaussian filter with edge preservation:** Building off of the first point, you can run a Gaussian filter to reduce the detail in the face while preserving the edges of the face.

6 Interview Question (Bonus) (10 points)

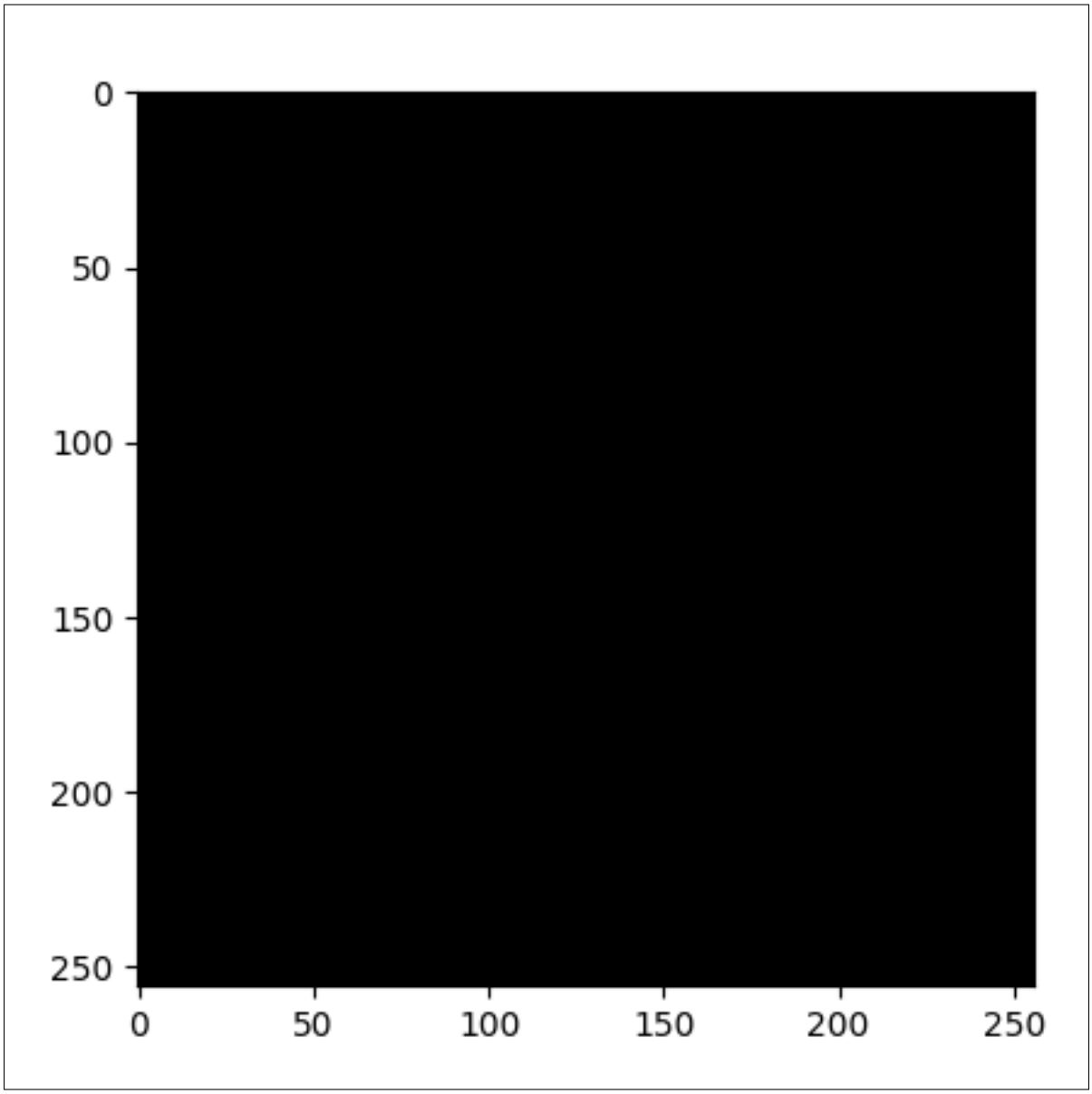
Consider an 256×256 image which contains a square (9×9) in its center. The pixels inside the square have intensity 255, while the remaining pixels are 0. What happens if you run a 9×9

median filter infinitely many times on the image? Justify your answer.

Assume that there is appropriate zero padding while performing median filtering so that the size of the filtered image is the same as the original image.

With each iteration, you will begin to see the pixels zero out since the number of 0's dominate the median filter and slowly erode the pixels in the center. I also coded up a little simulation running 100 iterations (with early stopping) to see what would physically happen to the image over iterations:

```
# Author: Simon Lee
from tqdm import tqdm
image = np.zeros((256, 256), dtype=np.uint8)
center = (image.shape[0] // 2, image.shape[1] // 2)
image[center[0]-4:center[0]+5, center[1]-4:center[1]+5] = 255

# Apply a 9x9 median filter multiple times to simulate infinite loop
# 100 iterations to simulate the effect of infinite applications
for i in tqdm(range(100)):
    image = median_filtering(image, kernel_size=9)
    # Early termination if the square has vanished
    if np.all(image == 0):
        break

display_gray(image)
```



After two iterations we have a picture with pixels all intensity 0.