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

INSTRUCTOR: Prof. Achuta Kadambi
TA: Rishi Upadhyay

NAME: Krish Patel
UID: 605796227

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.

This can be further expanded using Euler's formula as follows:

$$\exp(j(\omega_1 n_1 + \omega_2 n_2)) = \exp(j(\omega_1(n_1+\Delta_1) + \omega_2(n_2+\Delta_2)))$$

where Δ_1, Δ_2 are some arbitrary values used to prove the periodicity

$$= \exp(j\omega_1(n_1+\Delta_1))\exp(j\omega_2(n_2+\Delta_2))$$

$$= (\cos(\omega_1(n_1+\Delta_1)) + j\sin(\omega_1(n_1+\Delta_1)))(\cos(\omega_2(n_2+\Delta_2)) + j\sin(\omega_2(n_2+\Delta_2)))$$

$$= (\cos(\omega_1 n_1 + \omega_1\Delta_1) + j\sin(\omega_1 n_1 + \omega_1\Delta_1))(\cos(\omega_2 n_2 + \omega_2\Delta_2) + j\sin(\omega_2 n_2 + \omega_2\Delta_2))$$

As cosine and sine are known periodic functions, where $\cos(\theta + 2\pi k) = \cos(\theta)$ and $\sin(\theta + 2\pi k) = \sin(\theta)$ for any integer k, and letting Δ_1, Δ_2 be a multiple of 2π, it can be shown that this function is periodic in space for all $\omega_1, \omega_2 \in \mathbb{Z}$. Therefore, $x(n_1,n_2)$ is a periodic function in both variables n_1 and 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) $[x_4] = [5,4,3]$ can be expressed as a linear combination of the basis vectors x_1, x_2, x_3. Thus, $x_4 = 5x_1 + 4x_2 + 3x_3$. Using linearity principles from LSI models, we can construct $T[x_4]$ as $T[x_4] = T[5x_1] + T[4x_2] + T[3x_3]$, which is equal to $5a + 4b + 3c$.Thus, the vector $T[x_4]$ can be constructed from the basis vectors.

(ii) No, this is because all three vectors are needed to cover the sample space, and we can't construct the vector x_4 from just x_1 and x_2. Thus we need a third vector that can be used to construct the third dimension as well.

(iii) x_3 can be written as a combination of the vectors x_1, x_2, x_5. $x_3 = x_1 - x_2 - x_5$, as $[1,0,0] - [0,1,0] - [1,-1,-1] = [0,0,1]$. Thus, using the equation from (i), plugging in this new equa-

tion of x_3 gives: $T[x_3] = c = a - b - d$. And using substitution, the answer is:

$$T[x_4] = 8a + b - 3d.$$

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) $T_2[x(n_1)] = x(2n_1)$.

(i) Linear system T_1 is space invariant because shifting the input signal $x(n_1)$ by n_0 results in a corresponding shift of the output signal $T_1[x(n_1 - n_0)] = 2x(n_1 - n_0)$, thus there in no change in the scaling factor.

(ii) Linear system T_2 is not space invariant because shifting the input signal $x(n_1)$ by n_0 results in a different transformation of the output signal $T_2[x(n_1 - n_0)] = x(2(n_1 - n_0)) = x(2n_1 - 2n_0)$, which is not the same as $T_2[x(n_1)]$.

1.4 Convolutions (4.0 points)

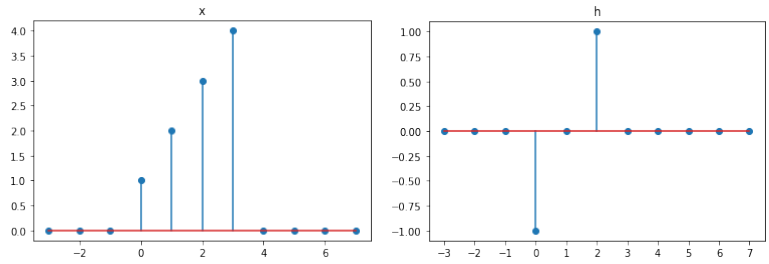


Figure 1: (a) Graphical representation of x (b) Graphical representation of h

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 $y[n] = (h * x)[n]$ is computed with the formula:

$$y[n] = (h * x)[n] = \sum_{m=-\infty}^{\infty} x(m)h(n-m) \tag{2}$$

The result of the convolution $y[n]$ for the signals given above is:

$$y[n] = \begin{cases} 1 \cdot -1 & \text{if } n = 0 \\ 2 \cdot -1 & \text{if } n = 1 \\ 3 \cdot -1 + 1 \cdot -1 & \text{if } n = 2 \\ 4 \cdot -1 + 2 \cdot -1 & \text{if } n = 3 \\ 3 \cdot 1 & \text{if } n = 4 \\ 4 \cdot 1 & \text{if } n = 5 \\ 0 & \text{else} \end{cases} \tag{3}$$

This results in the sequence:

$$y[n] = \begin{cases} -1 & \text{if } n = 0 \\ -2 & \text{if } n = 1 \\ -2 & \text{if } n = 2 \\ -2 & \text{if } n = 3 \\ 3 & \text{if } n = 4 \\ 4 & \text{if } n = 5 \\ 0 & \text{else} \end{cases} \tag{4}$$

(ii) The convolution $y = h * x$ can be represented as a product of two matrices H and X. The matrices are defined in this manner:

$$H = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad X = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \tag{5}$$

Therefore, the product HX = Y is:

$$Y = HX = \begin{bmatrix} -1 \\ -2 \\ -2 \\ -2 \\ 3 \\ 4 \end{bmatrix} \tag{6}$$

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):
  # Zero padding
  height, length = kernel.shape
  img_h, img_w = image.shape
  pad_height = height // 2
  pad_width = length // 2
  padded_img = np.pad(image, ((pad_height, pad_height),
            (pad_width, pad_width)), mode='constant')
  result = np.zeros((img_h, img_w))
  for x in range(img_h):
      for y in range(img_w):
          for i in range(-pad_height, pad_height + 1):
              for j in range(-pad_width, pad_width + 1):
                  result[x, y] += padded_img[x+pad_height+i,
                        y+pad_width+j] * kernel[i+pad_height,
                        j+pad_width]
  return result
```

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.)

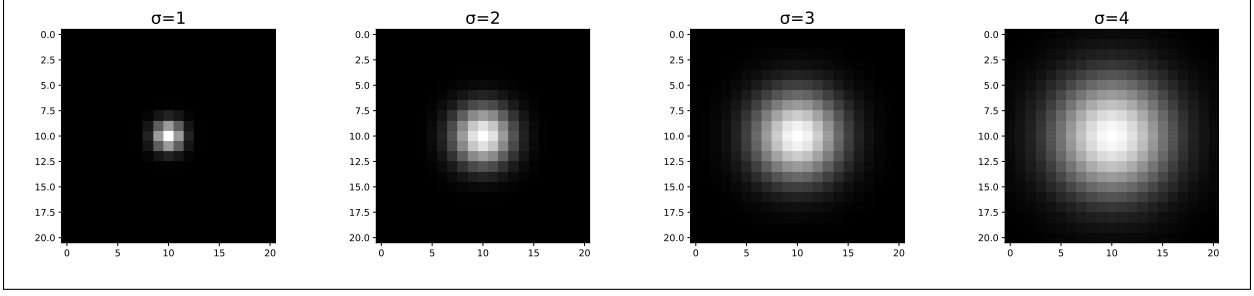
Make sure that your code is within the bounding box.

```
# Write your answer in this cell.
import math
def gaussian_filter(size: int, sigma: float) :
  filter = np.zeros((size, size))
  center_square = size//2
  for x in range(size):
    for y in range(size):
      filter[x][y] =(1/(2*np.pi*sigma**2))*np.exp(-((x-center_square)**2
                    +(y-center_square)**2)/(2 * sigma**2))
  filter = filter/filter.sum()
  return filter
```

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.

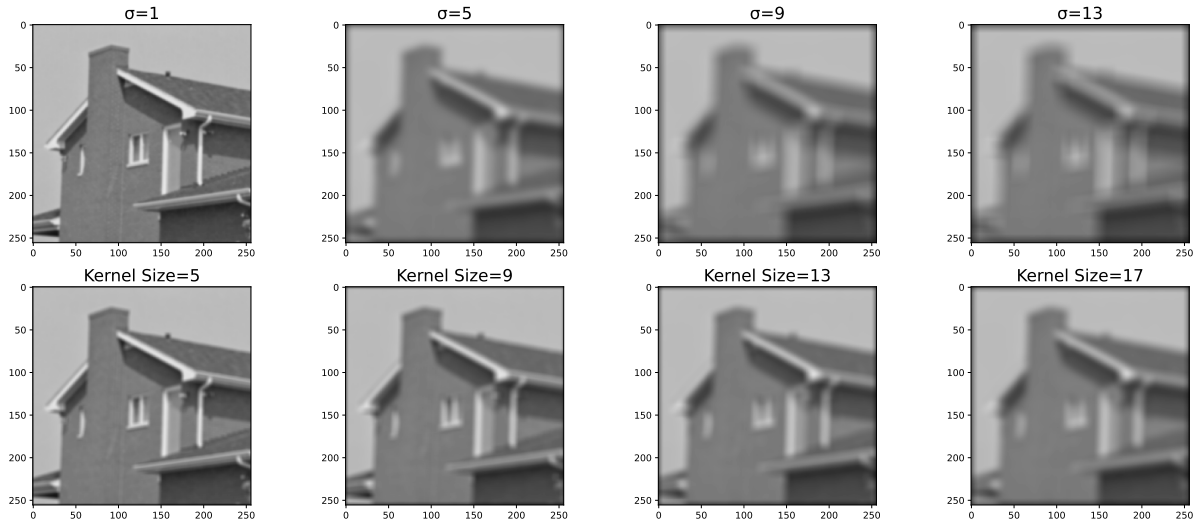
Copy the saved image from the Jupyter notebook here.



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?

The Blurring effect increases with the σ as when the standard deviation is larger, there is a larger spread or width of the curve, thus averaging it out even more(thus the blurring increases). For the kernel size, the larger the size of the kernel, the larger the neighbourhood pixel area and thus the averaging out increases, thus making the image more blurry.

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.

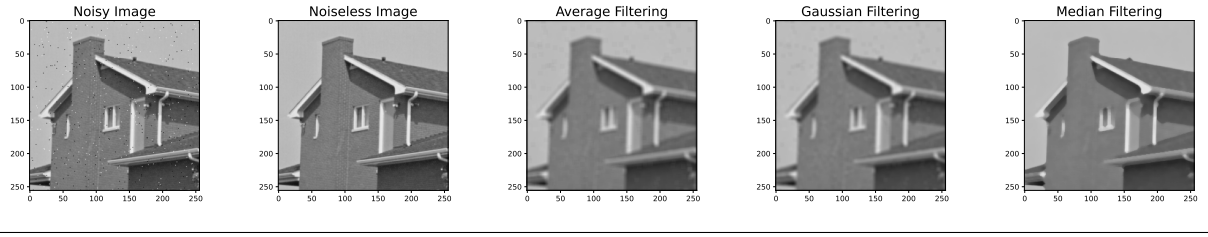
```
def check_bounds(edge: int, max_value: int):
        if edge < 0:
            return 0
```

```
        elif edge >= max_value:
            return max_value-1
        else:
            return edge

def median_filtering(image: np.array, kernel_size: int = 3):

    pad = kernel_size//2
    new_image =np.empty_like(image)
    for x in range(image.shape[0]):
        for y in range(image.shape[1]):
            bottom_x = check_bounds(x-pad,image.shape[0])
            left_y = check_bounds(y-pad,image.shape[1])
            top_x= check_bounds(x+pad+1,image.shape[0])
            right_y = check_bounds(y+pad+1,image.shape[1])
            # Calculate the median value of neighbourhood pxls
            median_value = np.median(image[bottom_x:top_x,
                                           left_y:right_y])
            new_image[x, y] = median_value
    return new_image
```

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?

According to the picture, the median filtering worked out best out of all the filtering schemes. This is because of the

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?

Average Filtering: Gaussian Filtering: Median Filtering:

In this example, median filtering outperforms the other two methods in preserving image edges. It achieves a lower relative absolute distance value of 0.032337645233158754, compared to 0.06525493187234266 for average filtering and 0.059951053670929684 for Gaussian filtering. This lower relative absolute distance value is the result of median filtering's effective noise reduction capability, which removes the noise while maintaining the clarity of edges.

Median filtering performs extremely well in preserving edges due to its ability to give priority to the removal of extreme pixel values. In areas with sudden transitions, it performs better than the other filters by retaining the fine details of edges. On the other hand, average filtering performs poorly as it uniformly blurs the entire image without considering the distances between pixels. This observation supports our earlier findings, confirming median filtering as the best choice for edge preservation.

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.

```
gradient_x = np.array([[-1,0,1],[-1,0,1],[-1,0,1]])
gradient_x = gradient_x/6
```

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.)

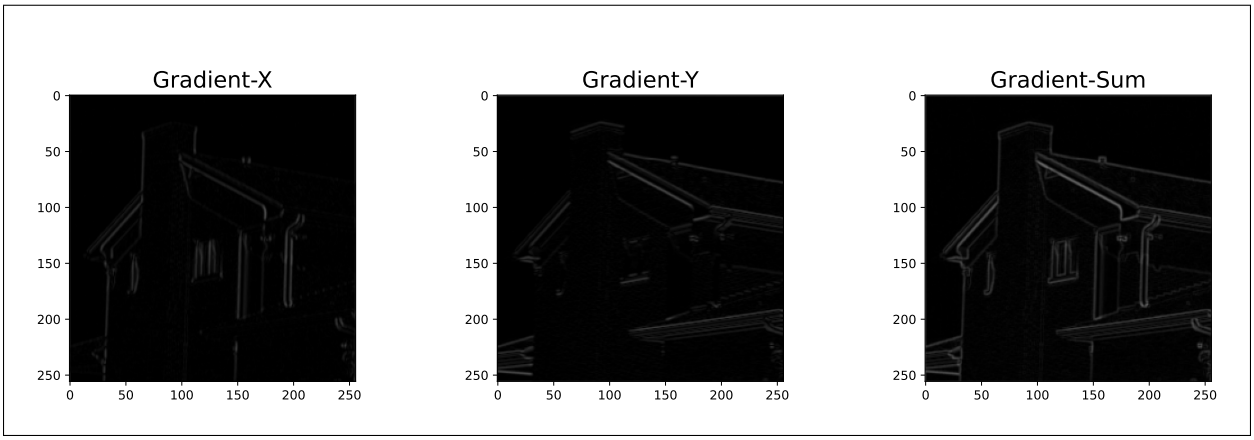
Make sure that your code is within the bounding box.

```
gradient_y = np.array([[1,1,1],[0,0,0],[-1,-1,-1]])
gradient_y = gradient_y/6
```

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?

This can be done using the gradient_x *and* gradient_y *convolution filters in the previous subparts*.

By convolving the image with these filters, we calculate G_x and G_y using basic trigonometry as:

$$\text{gradient} = \tan^{-1}\left(\frac{G_y(i,j)}{G_x(i,j)}\right)$$

where i and j are used to determine the gradient at the specific pixel.

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 filters are separable.

The Gradient for X can be calculated from the product of two 1D filters(example given):

$$\begin{bmatrix}-1 & 0 & 1\end{bmatrix} \times \begin{bmatrix}1\\2\\1\end{bmatrix} = \begin{bmatrix}-1 & 0 & 1\\-2 & 0 & 2\\-1 & 0 & 1\end{bmatrix}$$

Similarly, the Gradient for Y can also be calculated as the product of two 1D filters:

$$\begin{bmatrix}-1\\0\\1\end{bmatrix} \times \begin{bmatrix}1 & 2 & 1\end{bmatrix} = \begin{bmatrix}-1 & -2 & -1\\0 & 0 & 0\\1 & 2 & 1\end{bmatrix}$$

Therefore, both gradient filters can be represented as the product of 1D filters, making them separable.

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.]

Gaussian filtering is a technique used for smoothing images and reducing noise. It works by applying a weighted average to pixel values based on a Gaussian function. However, images contain edges that have high-frequency details caused by sudden changes in intensity or color. While Gaussian filters are effective at reducing high-frequency noise, they also blur these high-frequency edge details. In the frequency domain, Gaussian filters reduce the amplitudes of high-frequency components, leading to a loss of edge clarity. A better approach is to use edge-preserving filters like the bilateral filter. These filters can reduce noise while preserving edge sharpness by considering the intensity differences between neighboring pixels, and don't blur the edges and other noisy parts of the image similarly

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?

To preserve edges in an image, two key factors to consider when designing filter weights are:

1.Distance from Pixel p to Pixel q: The distance between the central pixel p and the neighboring pixel q within the filter window should influence the filter weight. Closer pixels should have higher weights to preserve local image details.

2.Intensity or Color Difference: The difference in intensity or color between pixel p and pixel q should also be a determining factor for the filter weight. Pixels with similar intensity or color to the central pixel p should receive higher weights to retain edge sharpness.

These two factors help ensure that the filter emphasizes pixels that are both spatially close and similar in intensity or color to the central pixel, which is important for edge preservation.

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

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

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?]

The Gaussian Filter is defined as:

$$GF[I_p] = \sum_{q \in S} G_\sigma(\|p-q\|) f(\|I_p - I_q\|) I_q. \tag{7}$$

Here, the function $f(I_p, I_q) = G_{\sigma_i}(\|I_p - I_q\|)$ where σ_i modifies the variance of the 1D-Gaussian.

The function $G_{\sigma_i}(|I_p - I_q|)$ consistently decreases as long as the inputs are: $|I_p - I_q|$).

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.

To see if a 1D Gaussian function works for $f(.)$, we know that a Gaussian function decreases as the difference increases. So, it's suitable for the edge-preserving function in $f(.)$. The complete formula for the new filter BF, which includes the edge-preserving function, is given by:

$$BF[I_p] = \sum_{q \in S} G_{\sigma_p}(\|p-q\|) G_{\sigma_i}(\|I_p - I_q\|) I_q. \tag{8}$$

This bilateral filter combines both spatial and intensity differences for preserving edges while smoothing.

5.5 Filtering (3.0 points)

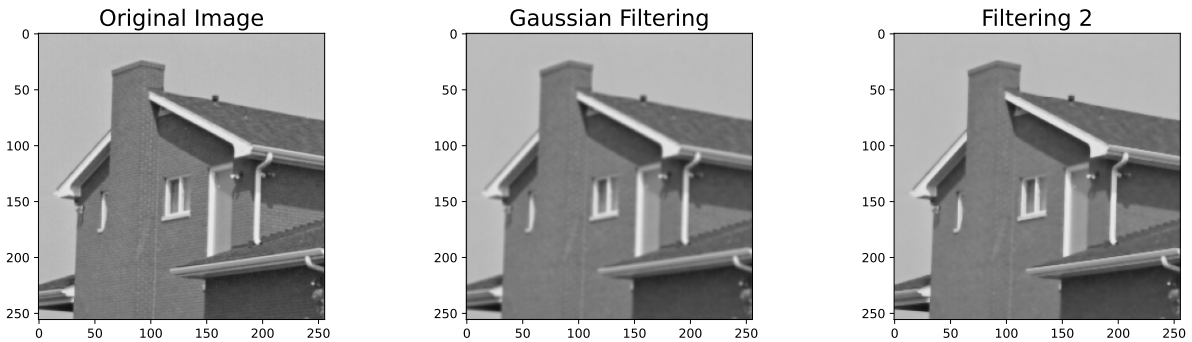
In this question you will be writing a Python function for this new filtering method (See the Jupyter notebook.)

Make sure that your code is within the bounding box.

```python
def filtering_2(image: np.array, kernel: np.array = None,
        sigma_int: float = None, norm_fac: float = None):
  k_h, k_w = kernel.shape
  pad = k_h//2
  img_h,img_w = image.shape
  new_image = np.zeros_like(image)
  for r in range(img_h):
      for c in range(img_w):
          norm = 0
          for i in range(r-pad, r+pad+1):
              for j in range(c-pad, c + pad + 1):
                  if i>=img_w or j>=img_h or i<0 or j<0:
                    continue #out of bounds
                  else:
                    intensity_diff = np.abs((image[r][c]-image[i][j]))
                    gauss_inten= ((1/(np.sqrt(2*np.pi)*sigma_int))*
                            np.exp(-(intensity_diff**2)/(2*sigma_int**2)))
                    gauss_spatial = kernel[i-(r - pad)][j-(c - pad)]
                    new_image[r][c]=new_image[r][c]+
                            gauss_spatial*gauss_inten*image[i][j]
                    norm += gauss_spatial * gauss_inten
          new_image[r][c]/=norm
  new_image *= norm_fac
  return new_image
```

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

Combining a bilateral filter with a median filter can achieve this effect. The bilateral filter with a low-intensity sigma smoothens out similarly colored regions, while the median filter helps to reduce extreme values in the image, creating a more cartoonish appearance. Although an example is provided below, further adjustments and fine-tuning can enhance the cartoonish quality of the image. In the example below, due to complexities and needing to apply the bilateral filter thrice for every color and then merge the results together, the image was converted to grayscale to avoid these:

```
new_image = Image.open(os.path.join(path, 'Singles/img.png'))
grayscale_image = new_image.convert('L')
grayscale_data = np.asarray(grayscale_image)
grayscale_data = grayscale_data / 255.0
post_bil = filtering_2(grayscale_data, gauss_filt, sigma_int=0.1, norm_fac=1)
median_filt = median_filtering(post_bil, 11)
plt.imshow(median_filt, cmap='gray')
plt.axis('off')
plt.show()
```

This is a crude implementation of the code above, converting the original square image to grayscale, and then applying the filtering techniques described above.

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.

Median filtering is a process that replaces each pixel's value with the median value of the intensities in the neighborhood given a filter window. For a 9×9 median filter, the median is calculated from the 81 values (intensities) inside the filter window centered at each pixel. When this filter is applied to an image with a 9×9 square of intensity 255 in the center, the following would happen: First Iteration: The boundary pixels of the square will be replaced with the median of a window that contains more zeros than 255s. Since the window is 9×9 has 81 values, and the square is completely surrounded by zeros, the median value for the boundary pixels will be 0 after the first iteration. This will erode the square, reducing its size by one pixel on all sides. With each subsequent iteration, the square will continue to erode because the boundary pixels will always encounter more zero values than 255s within their 9×9 window, and thus the median will always be 0. The process will continue until the square is completely eroded away. Thus, if we run the median filter an infinite number of times, the square will eventually disappear, leaving a uniform image with all pixels having an intensity of 0.