DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING, UCLA
EC ENGR 188: INTRODUCTION TO COMPUTER VISION

INSTRUCTOR: Prof. Achuta Kadambi, Prof. Stefano Soatto
TA: Zhen Wang

NAME: Sidarth Srinivasan
UID: 005629203

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. 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 Jupyter notebook (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

You will need to make two submissions: (1) Gradescope: You will submit the Overleaf PDF with all the answers on Gradescope. (2) CCLE: You will submit your Jupyter notebook (.ipynb file) with all the cells executed on CCLE.

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.

The Periodicity depends on the $\omega_1 and \omega_2$.

Since $k_1 * \omega_1 and$

$k_2 * \omega_2 = 2 * \pi * c, where c = constant.$

The 2D complex exponential is periodic if $2*\pi * c/\omega$

is a rational function.

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) $T[x_4] = 5a+4b+3c$ - To get the first dimension of $T[x_4]$, 5 is multiplied with $T[x_1]$, similarly to get the second dimension of $T[x_4]$, 4 is multiplied with $T[x_2]$, and third dimension $T[x_4]$, 3 is multiplied with $T[x_3]$. And they are then added together to get the entire 3D vector.
(ii) No, The reason being we need a non zero element in the last dimension of the vector to form $T[x_4]$. And since $T[x_3]$ is the only basis vector with non zero element in the last dimension, hence it is needed to form $T[x_4]$.
(iii) $T[x_5] = 8a + 1b - 3d$. Yes using $T[x_5]$ it is possible to find the value of $T[x_4]$ as the last dimension in $T[x_5]$ is non zero, 8 is multiplied with $T[x_1]$, 1 is multiplied with $T[x_2]$, -3 is multiplied with $T[x_5]$. And they are then added together to get the entire 3D vector.

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) This is a space invariant system. On shifting the input by k $T_1[x(n_1)-k]$, it will result in $2x(n_1-k)$, which is the same as shifting the output by k $2x(n_1-k)$.
(ii) This is not a space invariant system. On shifting the input by k $T_2[x(n_1)-k]$, it will result in $x(2(n_1-k))$. On shifting the output by k it results in $x(2n_1-k)$.

1.4 Convolutions (4.0 points)

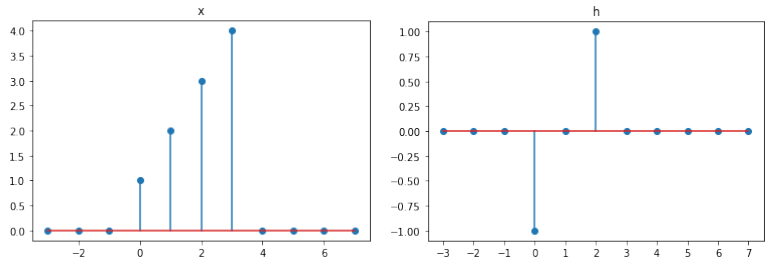


Figure 1: (a) Graphical representation of x (b) Graphical representation of h

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) Discrete convolution - $h * x$
i= 0 $->$ -1
i= 1 $->$ -2
i= 2 $->$ -2
i= 3 $->$ -2
i= 4 $->$ 3
i= 5 $->$ 4

(ii)

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

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

(2)

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):
    #determine the padding factor
    paddingFactor = kernel[0].size//2
    #detemine the size of the image, given 256
    inputImageSize = image[0].size
    paddedImage = np.zeros((256+(2*paddingFactor),(256+(2*paddingFactor))))

    #Creating a new image that includes the padding factor as well
    for i in range(0,inputImageSize):
        for j in range(0,inputImageSize):
            paddedImage[i+paddingFactor,j+paddingFactor]=image[i,j]

    convolvedImage = np.zeros((256,256))

    # Array of size same as the input image to store the Convolved output
    for i in range(256):
        for j in range(256):
            convolvedImage[i,j] =


 (paddedImage[i:i+kernel[0].size,j:j+kernel[0].size]*kernel)
                        .sum()

    return convolvedImage
```

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.)

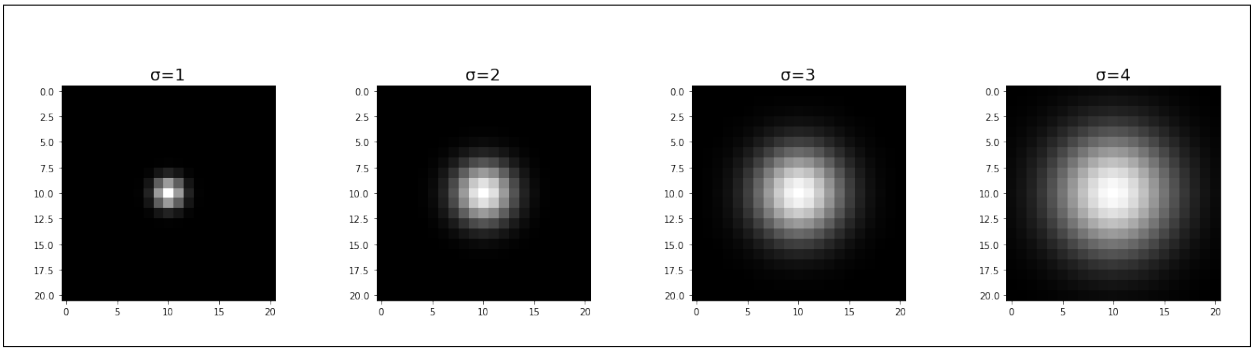
Make sure that your code is within the bounding box.

```
def gaussian_filter(size: int, sigma: float):
    x = np.arange(-size//2+1,size//2+1)
    y = np.arange(-size//2+1,size//2+1)
    xx,yy = np.meshgrid(x, y, sparse=True)
    z = (1.0 / ( 2 * np.pi * (sigma**2)))
    * np.exp (- (xx**2+yy**2) / (2 * sigma ** 2))
    z = z/np.sum(z)
    return z
```

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.

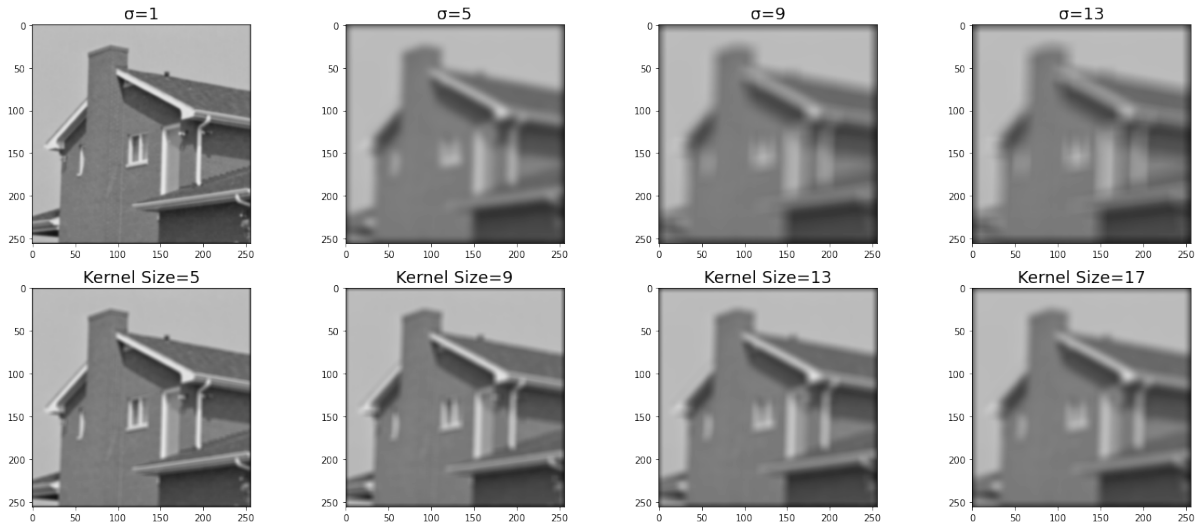
Copy the saved image from the Jupyter notebook here.



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?

As we increase σ keeping the kernel size constant, we are flattening the Gaussian curve thereby assigning more weightage to the neighbouring pixels. More weightage to the surrounding pixels impacts the value of the pixel under consideration and causes a blur. With increasing kernel size, we allow more pixels in the window the effect the pixel under consideration thereby diluting the image information. This also can induce a blur in the image. Hence it can be inferred that an increase in their ability to smooth noise corresponds to an increase in the blurring effect.

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.

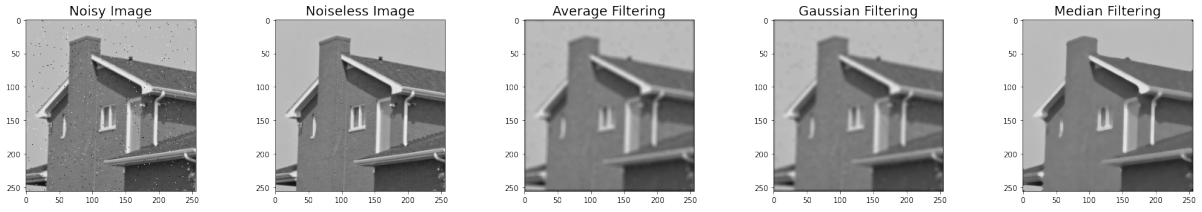
```
def median_filtering(image: np.array, kernel_size: int = None):
    paddingFactor  = kernel_size//2
    #Determine the padding factor
    inputImageSize = image[0].size
    #Detemine the size of the image
    paddedImage = np.zeros((256+(2*paddingFactor),(256+(2*paddingFactor))))
```

```
    for i in range(inputImageSize):
        for j in range(inputImageSize):
            paddedImage[i+paddingFactor,j+paddingFactor]=image[i,j]

    convolvedImage = np.zeros((256,256)) #Convolved Image
    for i in range(inputImageSize):
        for j in range(inputImageSize):
            medianArray =
            np.sort(paddedImage[i:i+kernel_size,j:j+kernel_size]
            .flatten())
            convolvedImage[i,j] = medianArray[len(medianArray)//2]

    return convolvedImage
```

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?

For the given image, median filter performs the best among the three. Since it replaces each pixel by the median of its neighbouring pixels considered in the kernel window, it is less sensitive to outliers and better removes the salt and pepper noise that is present in the image without losing the details.

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?

For the given image, Median Filter preserves the edges best among the three. Since median pixel replaces the value of the pixel by the median of its neighbour, during high intensity transitions, the same information is maintained as the neighbour values are considered. Hence it performs comparatively better than the other 2 filters.

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]])
```

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.)

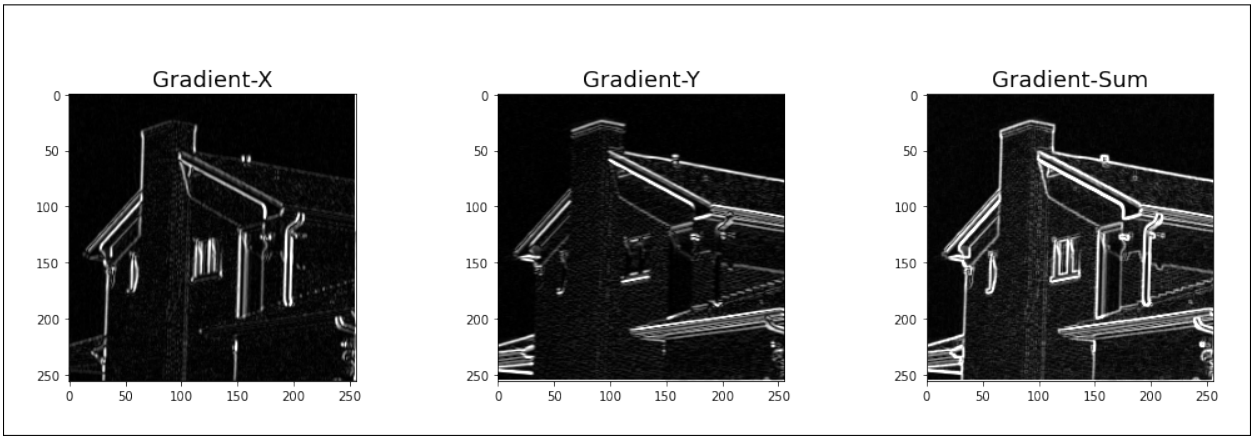
Make sure that your code is within the bounding box.

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

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?

$$\theta = tan^{-1}[gradient_y / gradient_x] \tag{3}$$

4.5 Separable filter (1.0 points)

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

Yes, Gradient Filters are separable as they can be expressed as a product of two 1D filters.

$$GradientX = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \tag{4}$$

$$GradientY = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \tag{5}$$

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 can be thought as a low pass filter. Edges, are nothing but sudden intensity transitions and they constitute to high spatial frequencies. Since, Gaussian filter removes the higher frequencies, it also smoothens the edges and hence does not preserve the edge information.

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?

- Spatial location of point q with respect to point p. This is done to account for the relative distance from the centre pixel p. As we move away from P, the weightage for the points q must decrease.
- Intensity differences between the point q and surrounding pixels q. To preserved edge information, we must account for intensity differences between p and surrounding pixels. Weightage to p and q must be given in such a way that we still preserve the intensity transitions.

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 need to have a function that penalises the larger change in intensities. In other assign lesser weightage to larger change in intensities. This also means, no penality is given for intensity that are similar to p. Hence, we need to have a function that behaves like a low pass filter.

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.

Yes, 1D-Gaussian function satisfies the above mentioned properties. Hence the new filter can be modelled as follows:

$$BF[I_p] = \sum_{q \in S} G_{\sigma_s}(\|p-q\|) G_{\sigma_r}(\|I_p - I_q\|) I_q. \tag{6}$$

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 gaussian_intensity(x: np.array, sigma, mean_pixel, kernel_size):
    ki = np.zeros((kernel_size,kernel_size))
    for i in range(kernel_size):
        for j in range(kernel_size):
            ki[i,j] = (1.0 / ((2 * np.pi)**(0.5)*(sigma)))
            * np.exp( (-1) * (((x[i,j] - mean_pixel)**2)/(2 * sigma ** 2)))

    return ki

def filtering_2(image: np.array, kernel: np.array = None,
sigma_int: float = None, norm_fac: float = None):
```

```
    pad = kernel[0].size//2 #determine the padding factor
    imgsize = image[0].size #detemine the size of the image, given 256
    p_image = np.zeros((256+(2*pad),(256+(2*pad))))
    #Creating a new image of bigger size

    for i in range(imgsize):
        for j in range(imgsize):
            p_image[i+pad,j+pad]=image[i,j]

    f_image = np.zeros((256,256)) #final image
    for i in range(256):
        for j in range(256):
            mean_pixel = p_image
            [i+(kernel[0].size//2),j+(kernel[0].size//2)]
            intensity_kernel = gaussian_intensity
          (p_image[i:i+kernel[0].size,j:j+kernel[0].size],
          sigma_int, mean_pixel, kernel[0].size)
            f_image[i,j]  =
            (p_image[i:i+kernel[0].size,j:j+kernel[0].size]
            *intensity_kernel*kernel).sum()*1/norm_fac
    return f_image/np.max(f_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

We can apply a combination of Median filtering and Bilateral filtering to obtain the cartoonized version of the image.

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.

The square will vanish by the third iteration. This can be thought of trying to understand the majority of the pixels i.e, the proportion of pixels that have intensity 255 and proportion of pixels that have intensity 0. The pixel under consideration will flip to 0 if the proportion of intensity 0 values are more than intensity 255 values. During the first iteration, starting from the corner pixel, we need to count the number of pixel zero values and pixel 255 values. Since at a time, we will consider 81 pixel values (9x9 kernel) when we iterate over the 9x9 square, the pixels will flip till the overlap of zero values dominates the intensity 255 values i.e Till intensity 255 values are present in 41 of those pixels and rest are 40. Hence at the end of the first iteration, we have lesser number of 255 intensity values. The same continues till all the bright pixels are converted. Intuitively, we can think at the end of first iteration, 25 percent of the 9x9 square would get converted and the next iteration, the 25 percent of the remaining bright squares gets converted.