2019702002
Nivedita Rufus

Digital Image Processing
Assignment 2

—

INTRODUCTION

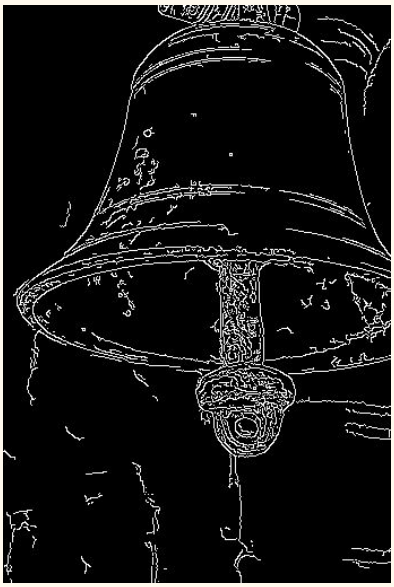
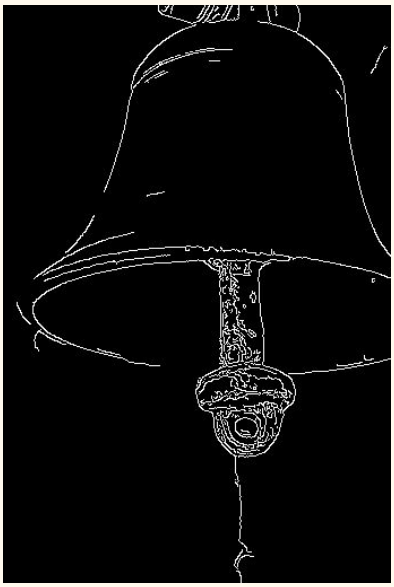
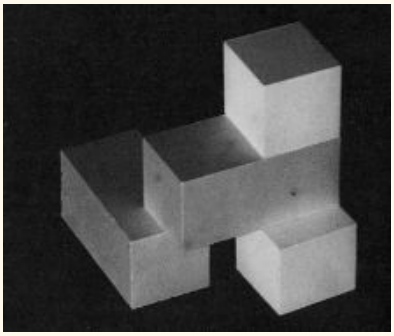
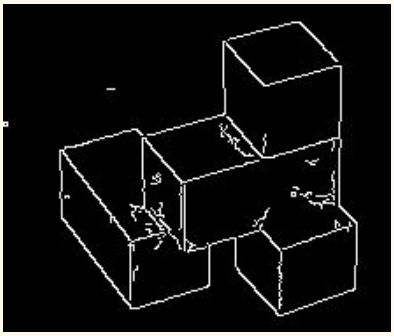
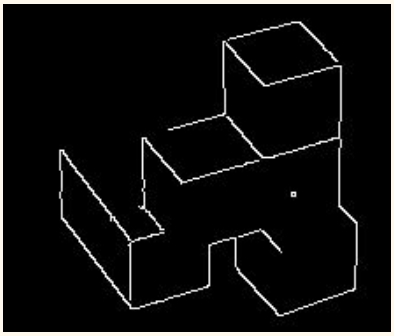
Q1) Edge Detection

1) **Apply the Canny edge detector(use cv2.Canny or MATLAB's edge function) to bell.jpg and cubes.png. Tweak the values of the arguments(minVal and maxVal), and report the values that give the best results for each image. Hint: For the bell image, try to detect as many edges in the bell as possible, while avoiding edges in the background.**

Steps:

- Apply Canny edge detector on the images.
- Varry the arguments of the fir the Canny edge detector function to get different results

```
import cv2

im = cv2.imread("cubes.png")
edges = cv2.Canny(im,50,100)


cv2.imwrite('cube50_100.jpg',edges)
```

Input	Output: min=50,max=100	Output: min=100,max=200
		
		

Q1)

2) Consider Roberts, Prewitt, and Sobel filters and Laplacian filters. Apply these filters on barbara.jpg and make observations upon comparing their outputs. Compare these with the output of Canny edge detector on the same image.

Steps:

- Slide each of the edge detector mask all over the image
- Multiply element-wise all the elements with the mask and the image window followed by computing the sum iteratively.

- Replace the middle pixel with the computed value

```
def sliding_window(image, mask,window = 3):
      a = np.zeros(image.shape)
      for x in range(1, image.shape[0]-1):
            for y in range(1, image.shape[1]-1):
                  win_im = image[x-1:x+window-1, y-1:y+window-1]
                  win_im = np.multiply(win_im, mask)
                  a[x][y] = np.sum(win_im)

      return a
```

Input	Canny edge detector
	

Sobel X

Sobel Y

Prewitt X

Prewitt Y

robert X

Robert Y

Laplacian 1

Lapalacian 2

Observations :

The Robert, Sobel, Prewitt masks are 1-D masks, they have to be applied twice to get the edges in the X and the Y direction respectively. But the Canny edge detector gives edges in both

directions simultaneously in both directions. The canny edge detector is most accurate in delivering all the edges as gives weak and strong edges. The Laplacian filter detects zero crossings as edges, so only strong edges are returned.

Q1)

3) **Add noise to the input image above using Gaussian sampling. Study the effect of applying the filters on noise-affected inputs.**

Input	Canny edge detector
	

Sobel X

Sobel Y

Prewitt X

Prewitt Y

robert X

Robert Y

Laplacian 1

Lapalacian 2

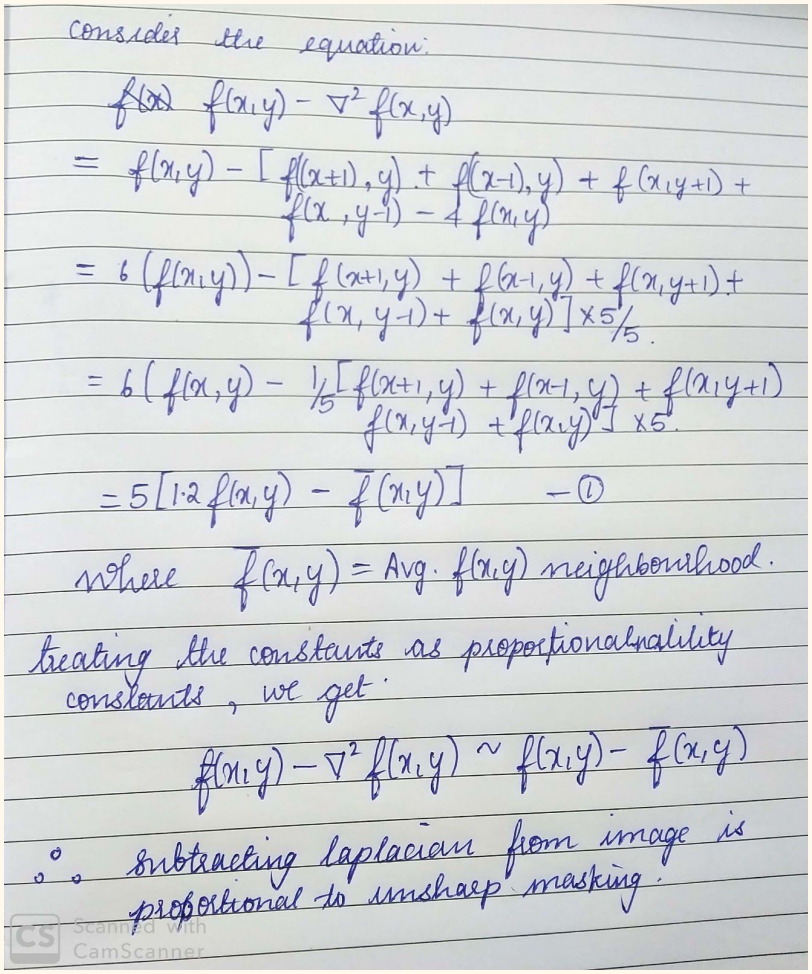
Observations :

The Laplacian mask seems to be most affected by the noise and has the maximum amount of distortion. The Canny edge detector' parameters can be modified to give proper edges in the

noisy version. But using it with same values yields noisy edges. The 1D masks also yields noisy edges.

Q2)

1) **Prove that subtracting the Laplacian from an image is proportional to unsharp masking.**



consider the equation:

$f(x,y) - \nabla^2 f(x,y)$

$= f(x,y) - [f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1) - 4f(x,y)]$

$= 6(f(x,y)) - [f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1) + f(x,y)] \times 5/5.$

$= 6(f(x,y) - 1/5[f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1) + f(x,y)] \times 5.$

$= 5[1.2f(x,y) - \bar{f}(x,y)] \quad -①$

where $\bar{f}(x,y)$ = Avg. $f(x,y)$ neighbourhood.

treating the constants as proportionality constants, we get.

$f(x,y) - \nabla^2 f(x,y) \sim f(x,y) - \bar{f}(x,y)$

∴ subtracting laplacian from image is proportional to unsharp masking.

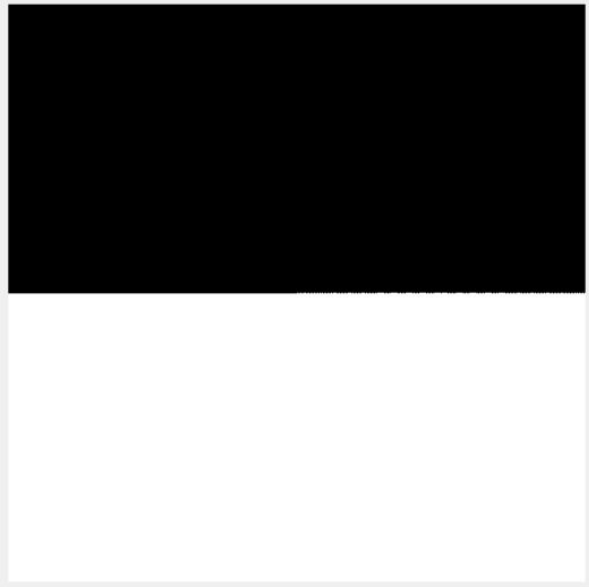
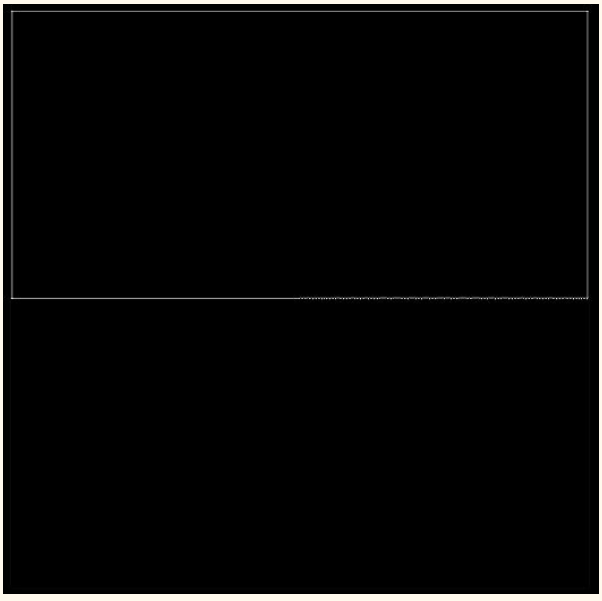
Q3)

1) **Create a matrix of size 3x3 which when convolved with box.png results in a white line where the white meets the black.**

Steps:

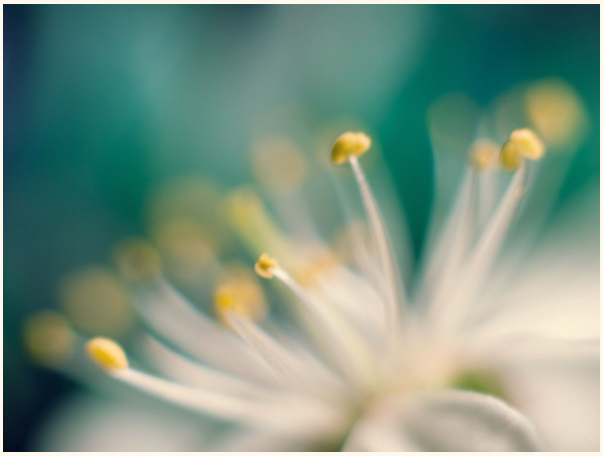
- To achieve the required potpu we have to perform edge detection on the figure
- This can be achieved by using a laplacian filter which marks the abrupt change in intensity when moving from black region to white.
- The filter used:

1	1	1
1	−8	1
1	1	1

Input:	Output:
	

Q3)

2) **Convolve blur.jpg with the above matrix and the transpose of the above matrix. Report your observations.**

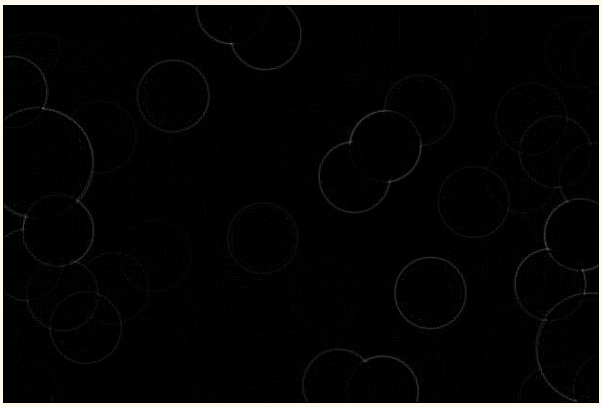
Input:	Output:
	

Observations :

The same Laplacian mask used in the previous question was used for *blur.jpg*. The mask could capture very blur.jpg few edges as the image is very blurry and there is no sudden jumps in intensity to have zero crossings.

Q3)

3) **Follow the above steps for suitable images of your choice.**

Input:	Output:
	

Q4) High-Boost Filtering

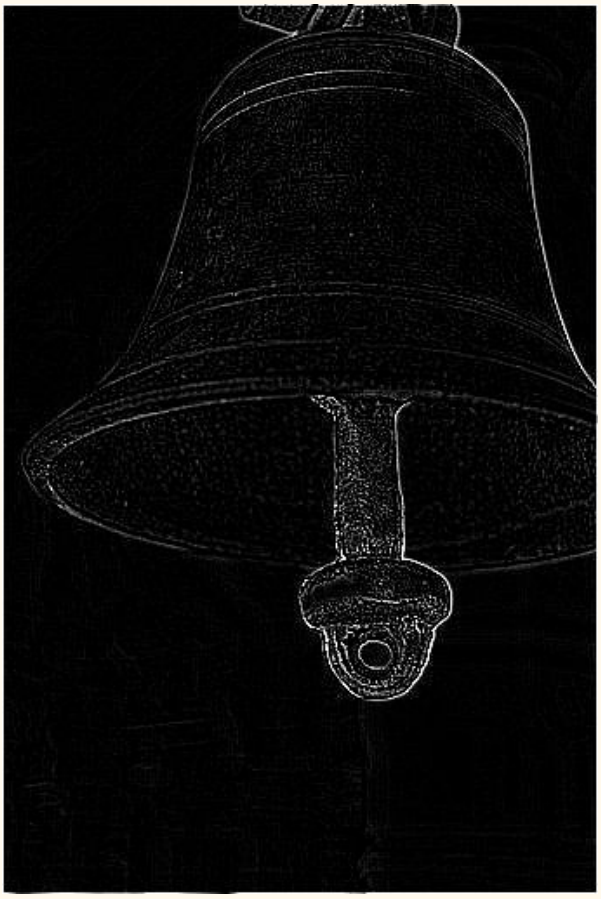
1) Implement high-boost filtering on the image bell.jpg varying the window size and the weight factor and report your observations.

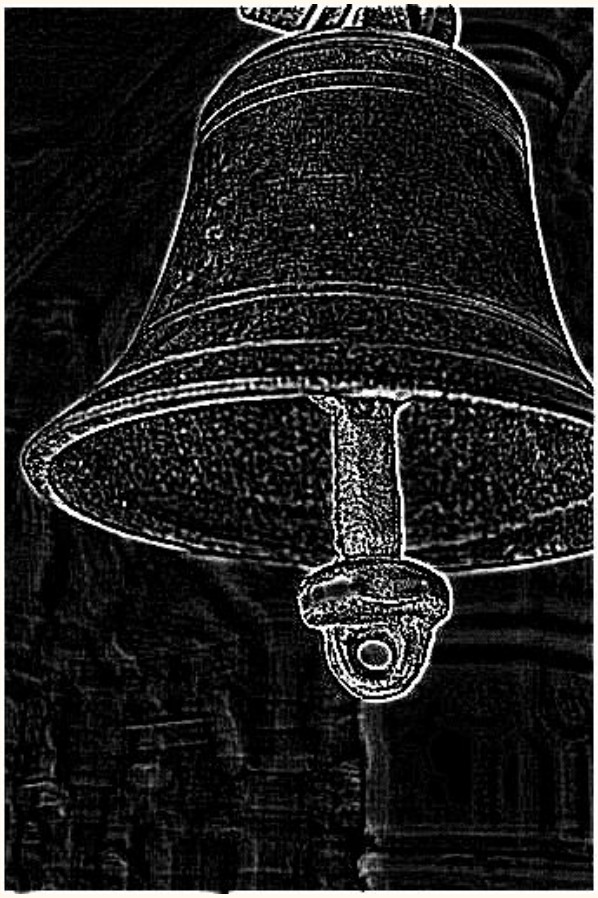
Steps:

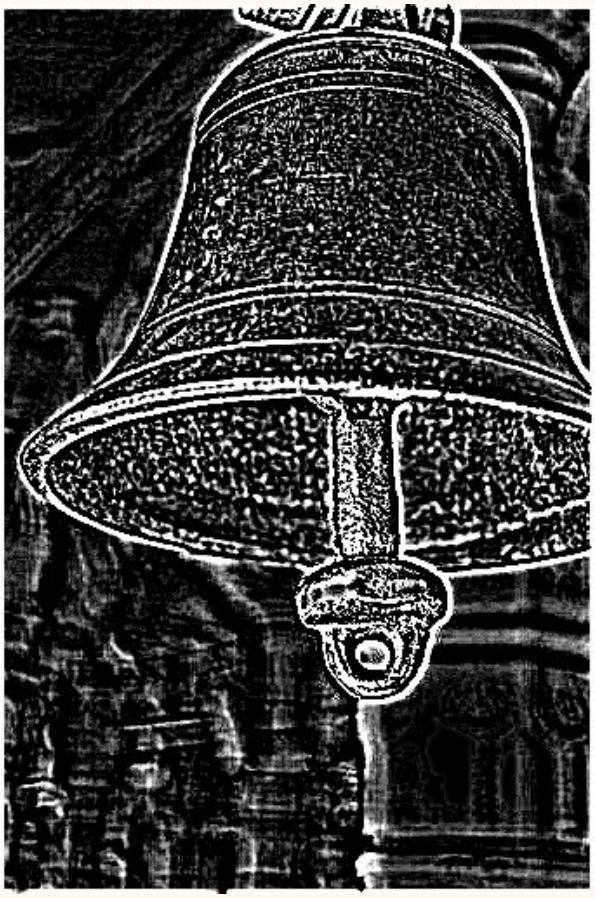
- Generate the appropriate mask for the high boost filter in accordance with the weight and window size
- Multiply this the generated mask with the image to obtain the sharpened image.
- Vary window size and weights to get different results

$$Highboost = A\ Original - Lowpass$$
$$= (A-1)\ Original + Original - Lowpass$$
$$= (A-1)\ Original + Highpass$$

```
def sliding_window(image,padded_image, mask,window = 3):
      a = np.zeros(image.shape)
      for x in range(image.shape[0]):
            for y in range(image.shape[1]):
                  win_im = padded_image[x:x+window, y:y+window]
                  win_im = np.multiply(win_im, mask)
                  a[x][y] = np.sum(win_im)
      return a

def padding(im,mask):
      image_row, image_col = im.shape
      kernel_row, kernel_col = mask.shape

      pad_height = int((kernel_row - 1) / 2)
      pad_width = int((kernel_col - 1) / 2)

      padded_image = np.zeros((image_row + (2 * pad_height), image_col + (2
* pad_width)))

      padded_image[pad_height:padded_image.shape[0] - pad_height,
pad_width:padded_image.shape[1] - pad_width] = im
      return padded_image


def generate_mask(weight,size = 3):
      mask = -1*(np.ones([size,size]))
      x = int(size/2)
      mask[x][x] = (size**2*weight) - 1
      print(mask)
      return mask

im = cv2.imread('ice.jpg')
im = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)

mask = generate_mask (1,5)
padded_image = padding(im,mask)
output = sliding_window(im,padded_image,mask,5)
cv2.imwrite('ice_filter32.jpg',output)
```

Input:	Output: window=(3x3), weight = 1
	
Output: window=(3x3), weight = 1.1	Output: window=(5x5), weight = 1

	
Output: window=(5x5), weight = 1.056	**Output: window=(7x7), weight = 1**



Observations :

It can be observed that for a lower filter size only darker and more prominent edges are filtered as the output, this is because the derivative is taken with respect to a larger neighbourhood.

It can also be observed that on increasing weight > 1, part of the original image is added to the filtered image and this image has more sharp edges than the original, but very high weights causes image degradation as well.

Q4)

2) **Repeat the above exercise on suitable images of your choice.**

Input:	Output: window=(3x3), weight = 1
	
Output: window=(3x3), weight = 1.1	Output: window=(5x5), weight = 1
	
Output: window=(5x5), weight = 1.056	Output: window=(7x7), weight = 1

Observations :

It can be observed that for a lower filter size only darker and more prominent edges are filtered as the output, this is because the derivative is taken with respect to a larger neighbourhood.

It can also be observed that on increasing weight > 1, part of the original image is added to the filtered image and this image has more sharp edges than the original, but very high weights causes image degradation as well.

Q4)

3) How is bilateral filtering different from high-boost filtering?

High boost filtering is a method of image sharpening wherein edges get sharpened and more pronounced. But if the image is noisy, it does not produce any satisfactory results. Bilateral filtering on the other hand does smoothing to remove noise from the image but also preserves the edges.

Q5)

1) Implement an algorithm for low-pass filtering a grayscale image by moving a k × k averaging filter of the form ones(k)/(k 2)

Steps :

- The mask for an LPF for image is given by $\mathbf{k/k^2}$, where k = window size.
- Perform appropriate padding for the image.
- Convolve the image with the mask to obtain the low-pass filtered image.

```
def sliding_window(image,padded_image, mask,window = 3):
      a = np.zeros(image.shape)
      for x in range(image.shape[0]):
            for y in range(image.shape[1]):
                  win_im = padded_image[x:x+window, y:y+window]
                  win_im = np.multiply(win_im, mask)
                  a[x][y] = np.sum(win_im)/(window**2)
      return a

def padding(im,mask):
      image_row, image_col = im.shape
      kernel_row, kernel_col = mask.shape

      pad_height = int((kernel_row - 1) / 2)
      pad_width = int((kernel_col - 1) / 2)

      padded_image = np.zeros((image_row + (2 * pad_height), image_col + (2
* pad_width)))

      padded_image[pad_height:padded_image.shape[0] - pad_height,
pad_width:padded_image.shape[1] - pad_width] = im
      return padded_image

def generate_mask(weight,size = 3):
      mask = (np.ones([size,size]))
      print(mask)
      return mask
```

Input:	Output: 3x3 filter
	

Q5)

2) As the filter is moved from one spatial location to the next one, the filter window shares many common pixels in adjacent neighborhoods. Exploit this observation and implement a more efficient version of averaging filter. To appreciate the benefits of doing so, generate a plot of k vs run-time for various sized images. The plot diagram should contain a line plot for each image size you pick. Use different marker types to distinguish the default implementation and improved implementation.

Idea :

Avoid redundant computations over consecutive iterations.

Steps :

- The low pass filter is basically the sum of all elements in the window divided square of the filter size.

- To make this procedure more efficient would be to compute the cumulative sum of all the rows and columns in the image array and store them.
- Once we have the array of the cumulative sum, by querying the corner points we can obtain the sum without actually computing it.

Input:	Output: 3x3 filter
	

Graph :

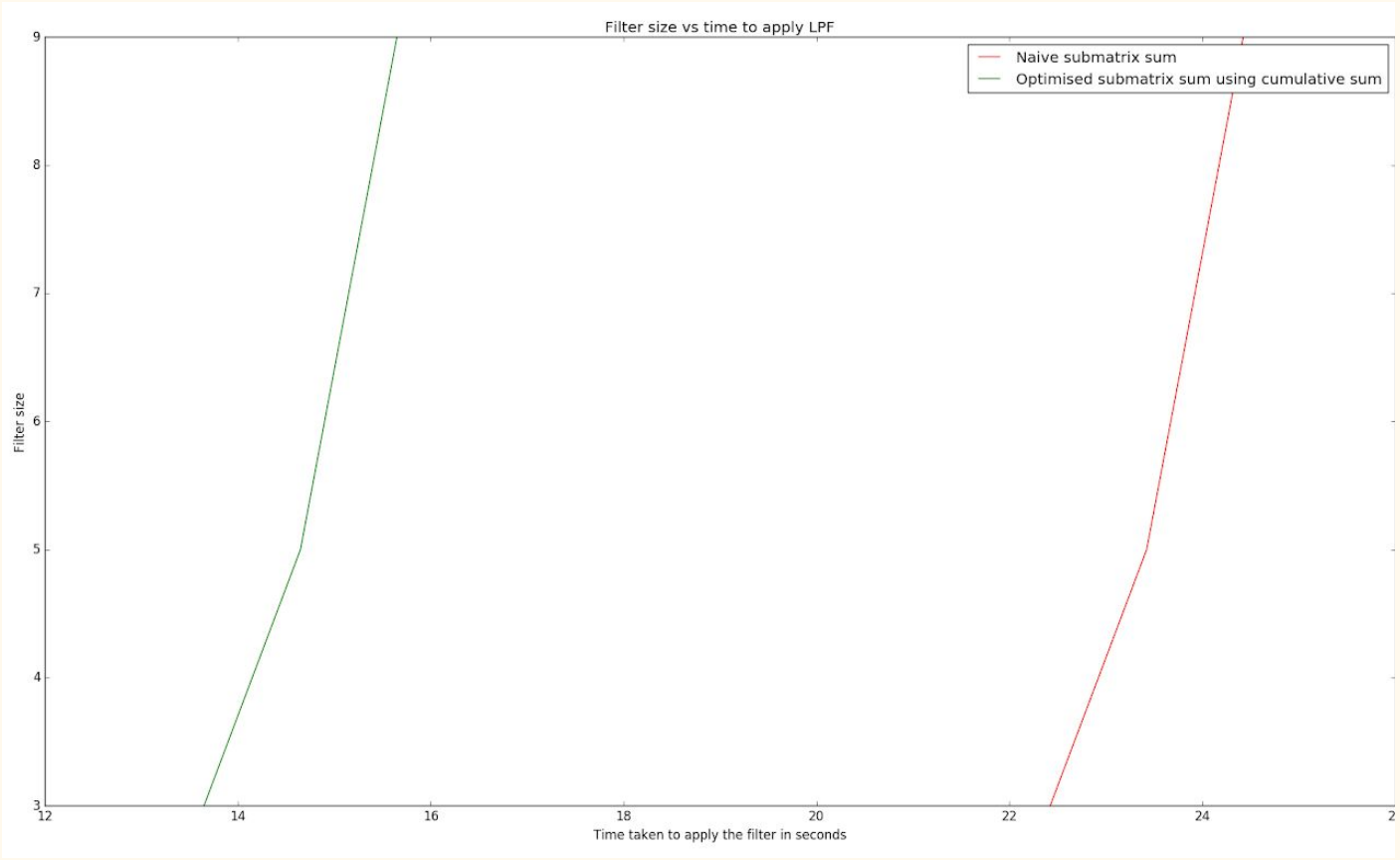
Complexity of Naive submatrix sum method : $\mathbf{O(N_{pixels} * n^2)}$

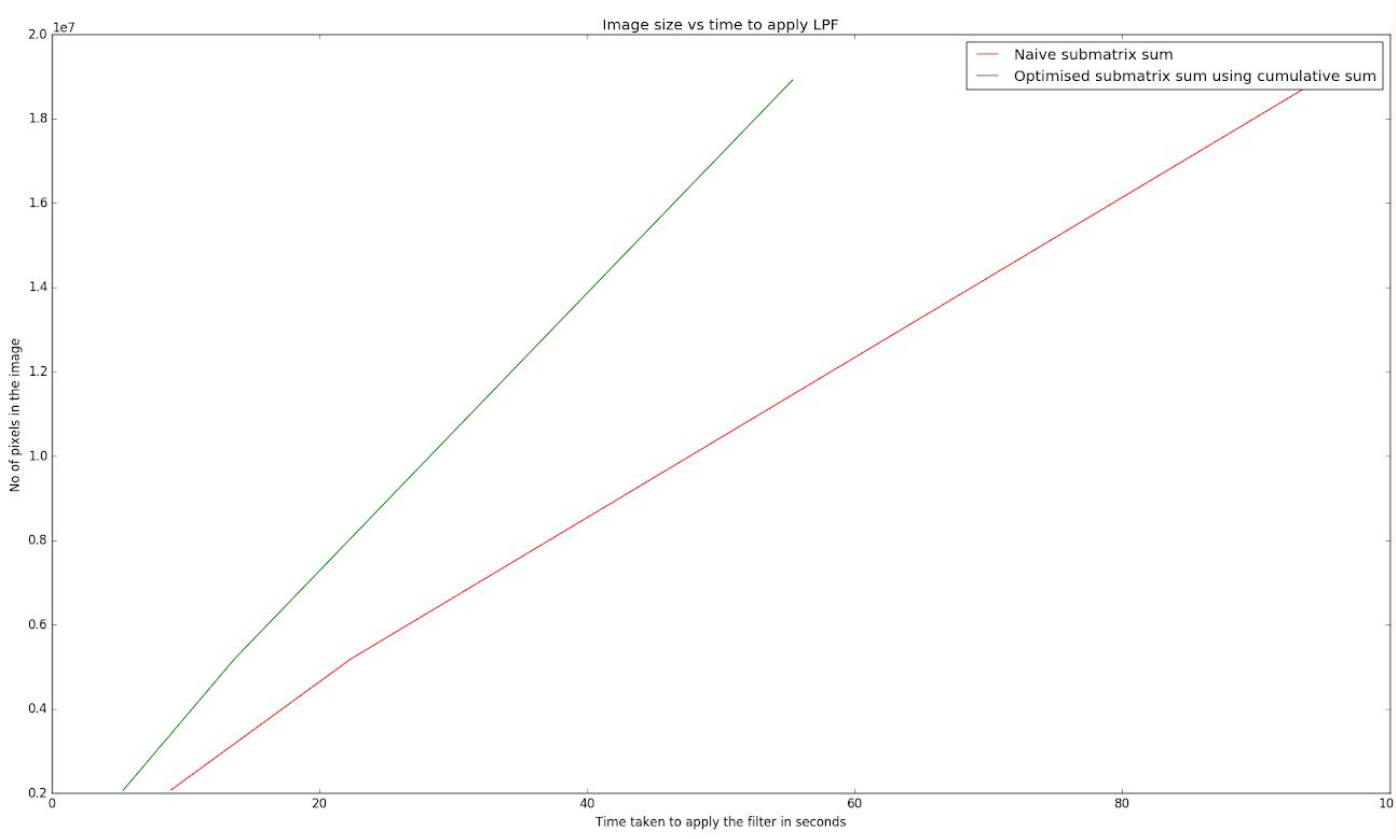
Complexity of optimised submatrix sum using cumulative sum : $\mathbf{O(N_{pixels})}$

Where, N_{pixels} = no. of pixels and n = filter size

```
def preProcess(mat, aux):
    N=aux.shape[1]
    M=aux.shape[0]
    for i in range(0, N, 1):
        aux[0][i] = mat[0][i]
    for i in range(1, M, 1):
        for j in range(0, N, 1):
            aux[i][j] = mat[i][j] + aux[i - 1][j]
    for i in range(0, M, 1):
        for j in range(1, N, 1):
            aux[i][j] += aux[i][j - 1]
```

```
def sumQuery(aux, tli, tlj, rbi, rbj):
    res = aux[rbi][rbj]
    if (tli > 0):
        res = res - aux[tli - 1][rbj]
    if (tlj > 0):
        res = res - aux[rbi][tlj - 1]
    if (tli > 0 and tlj > 0):
        res = res + aux[tli - 1][tlj - 1]
    return res
```





Observations :

The optimised Low-pass filter is always faster for increasing filter size and for the image size. But for very small images the optimised Low pass filter is slower than the naive submatrix sum method.

3) **Utilize the observation similar to above to implement an efficient version of a k × k median filter.**

Idea :

Avoid redundant computations over consecutive iterations.

Conventional median filter (sort and find):

Steps :

- Perform the appropriate padding on the image depending on the size of the filter.
- For each window in the image, sort the pixel values and replace the centre pixel with median value.

- Iterate over the entire image to get median filtered image.

```
def sliding_window(image,padded_image,window = 3):
      a = np.zeros(image.shape)
      for x in range(image.shape[0]):
            for y in range(image.shape[1]):
                  win_im = padded_image[x:x+window, y:y+window]
                  # print(image[x,y])

                  a[x][y][0] = remove(win_im[:,:,0])
                  a[x][y][1] = remove(win_im[:,:,1])
                  a[x][y][2] = remove(win_im[:,:,2])
      print("median done")

      return a

def padding(im,kernel_row=3,kernel_col=3):
      image_row, image_col,ch = im.shape

      pad_height = int((kernel_row - 1) / 2)
      pad_width = int((kernel_col - 1) / 2)

      padded_image = np.zeros((image_row + (2 * pad_height), image_col + (2
* pad_width),ch))
      print(padded_image.shape)

      padded_image[pad_height:padded_image.shape[0] - pad_height,
pad_width:padded_image.shape[1] - pad_width] = im
      return padded_image

def filter(win_im):
      x = np.sort(win_im.ravel())
      idx = x.shape[0]
      return x[int(idx/2)]
```

Optimised median filter (find k^{th} smallest element):

Steps :

- To optimise the conventional sort and find method, the algorithm of finding the kth smallest element in a BST.

- As the the filter iterates over the image the the new elements are pushed into the tree and old elements are pushed out and the k^{th} smallest i.e. the middle element element is found.

```
def kthSmallest(input):
    result = input[0]
    heapq.heapify(list(result))
    for row in input[1:]:
         for ele in row:
              heapq.heappush(result,ele)
    kSmallest = heapq.nsmallest(k,result)
    return kSmallest[-1]
```

Graph :

Complexity of sort and find : **O(n^2 logn)**

Complexity of optimised median filter: **O(nlogn)**

Where, n = filter size

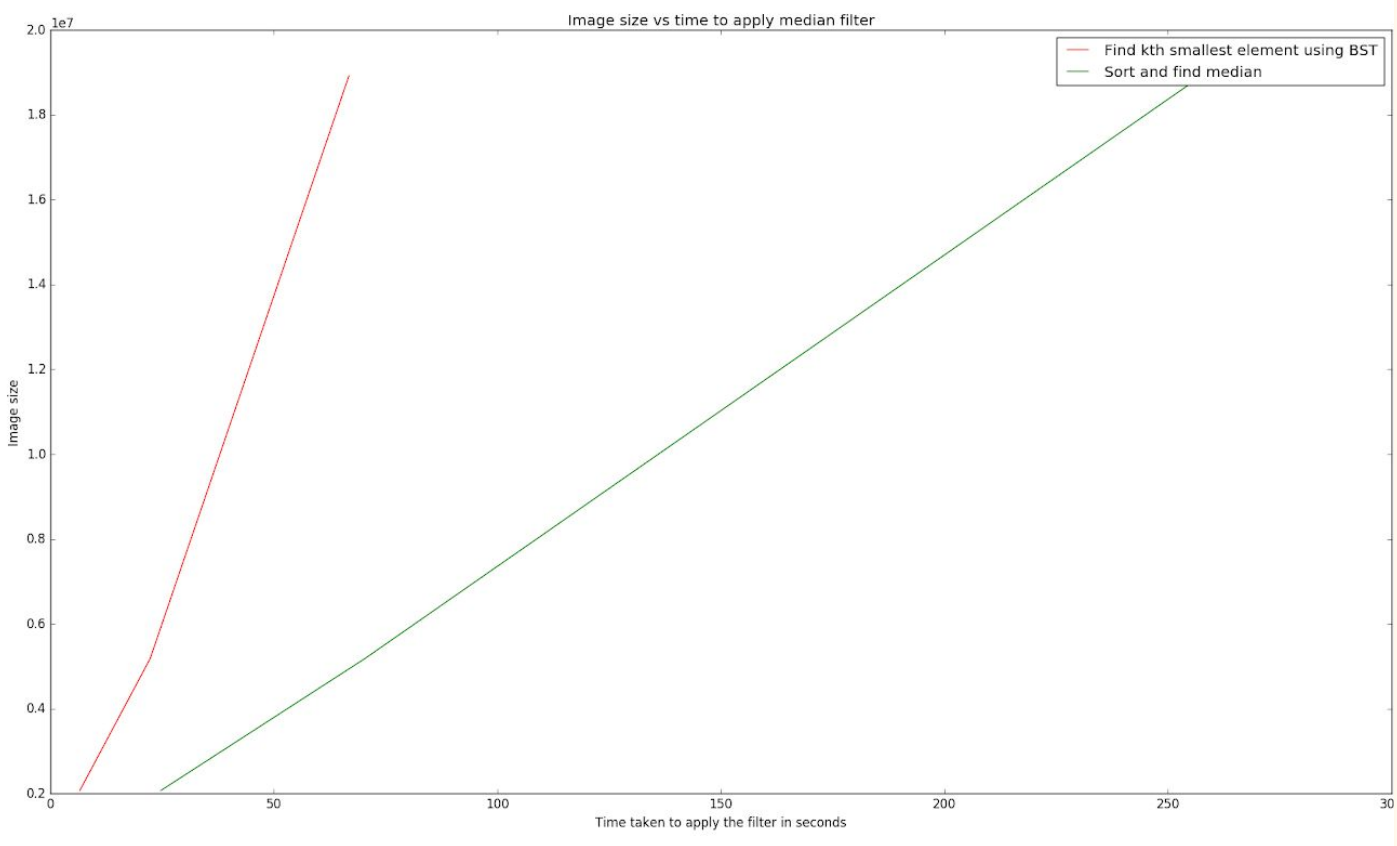
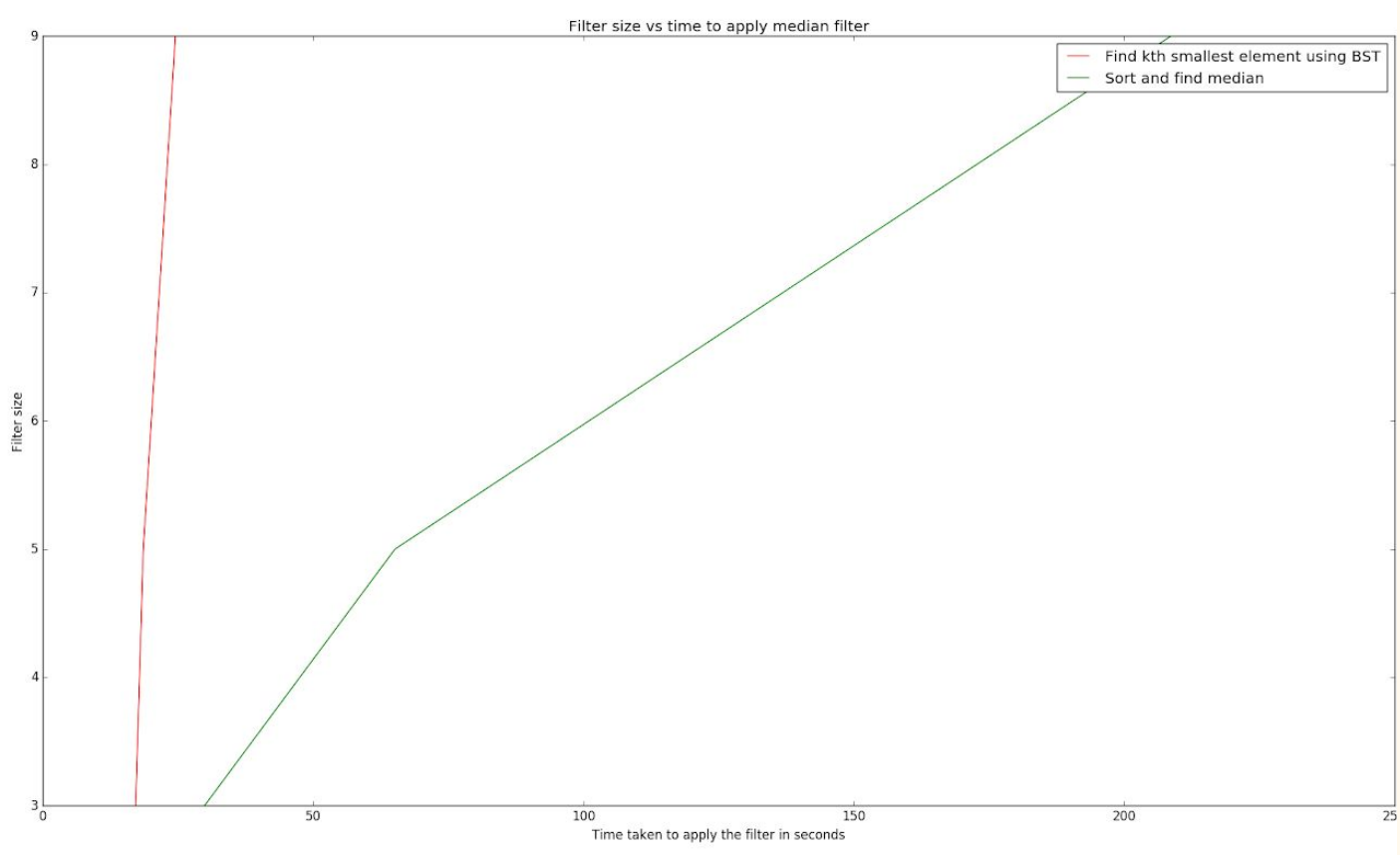


Image size vs time to apply median filter
1e7
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
Image size
0
50
100
150
200
250
30
Time taken to apply the filter in seconds
Find kth smallest element using BST
Sort and find median



Observations :

The optimised median filter is always faster for increasing filter size and for the image size.

Input:

Output: 5x5 filter

Q6)

1) Implement bilateral filter and apply it to sky.png and noir.png.

Steps :

- The bilateral filter is a range dependent mask.
- The weights of each of the masks are computed depending on the intensity values on that particular window, i.e. multiplication of the domain kernel and the range kernel.
- The computed maske is then convolved with the image to obtain the bilaterally filtered image.
-

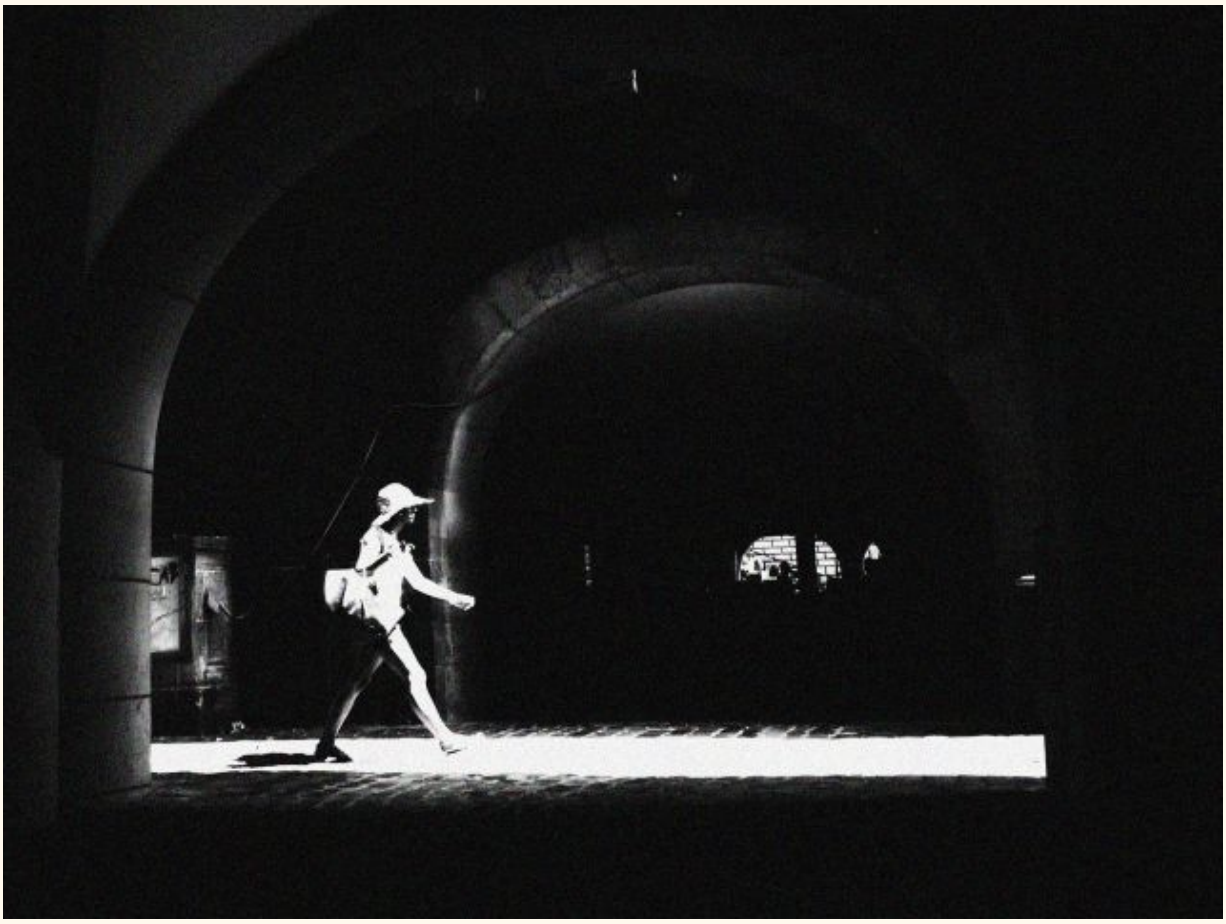
```
def bilateral_filter(win_im,sd=1,sr=1):
      w = 0
      gk = 0
      i= int(win_im.shape[0]/2)
```

```
    # print(i)
    for k in range(win_im.shape[0]):
          for l in range(win_im.shape[1]):

                d = weighting((i-k),sd) * weighting((i-l),sd)
                # print(d)
                v = abs(win_im[i][i] - win_im[k][l])
                r = weighting(v,sr)
                gk= gk + (win_im[k][l]*(r*d))
                w = w + (r*d)

    return (gk/w)
```

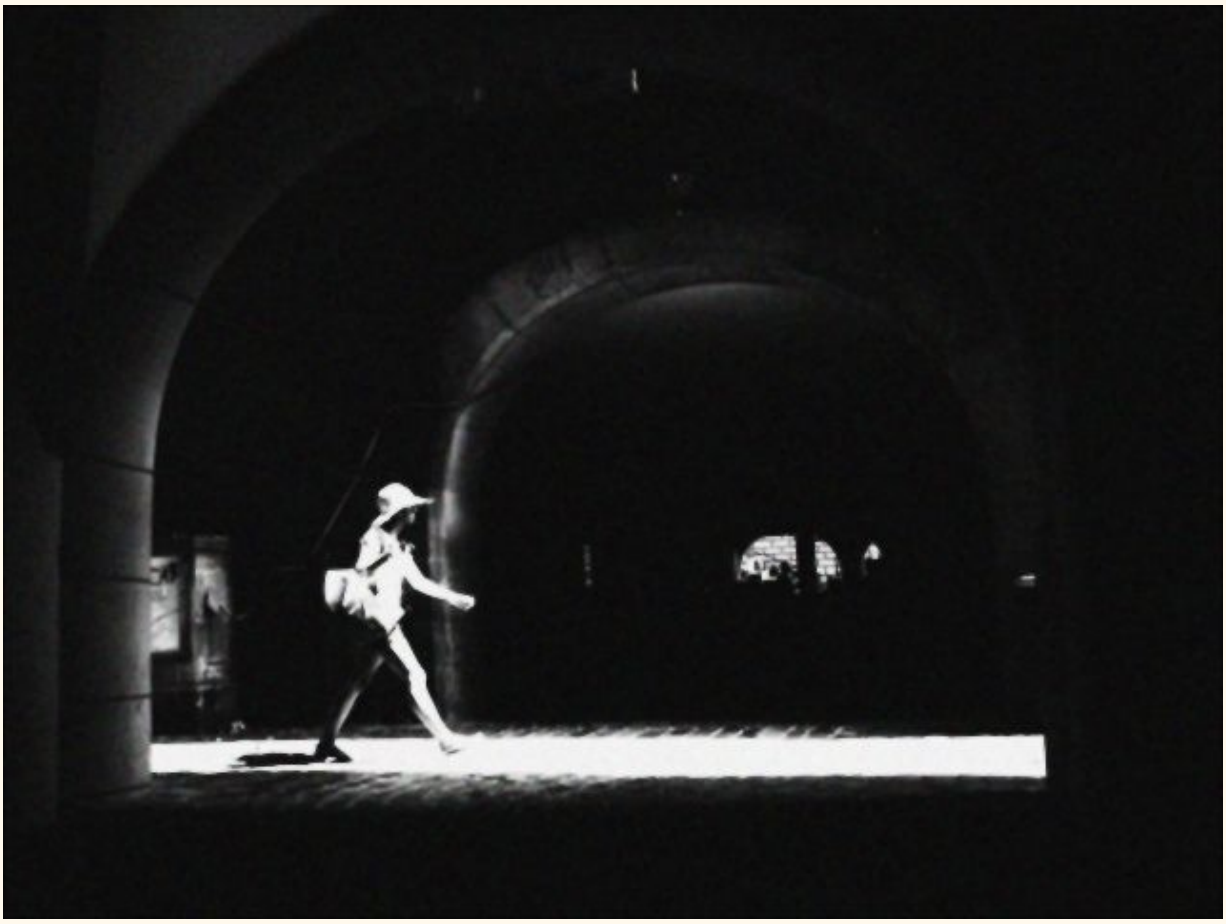
Input:

Output :

Input:



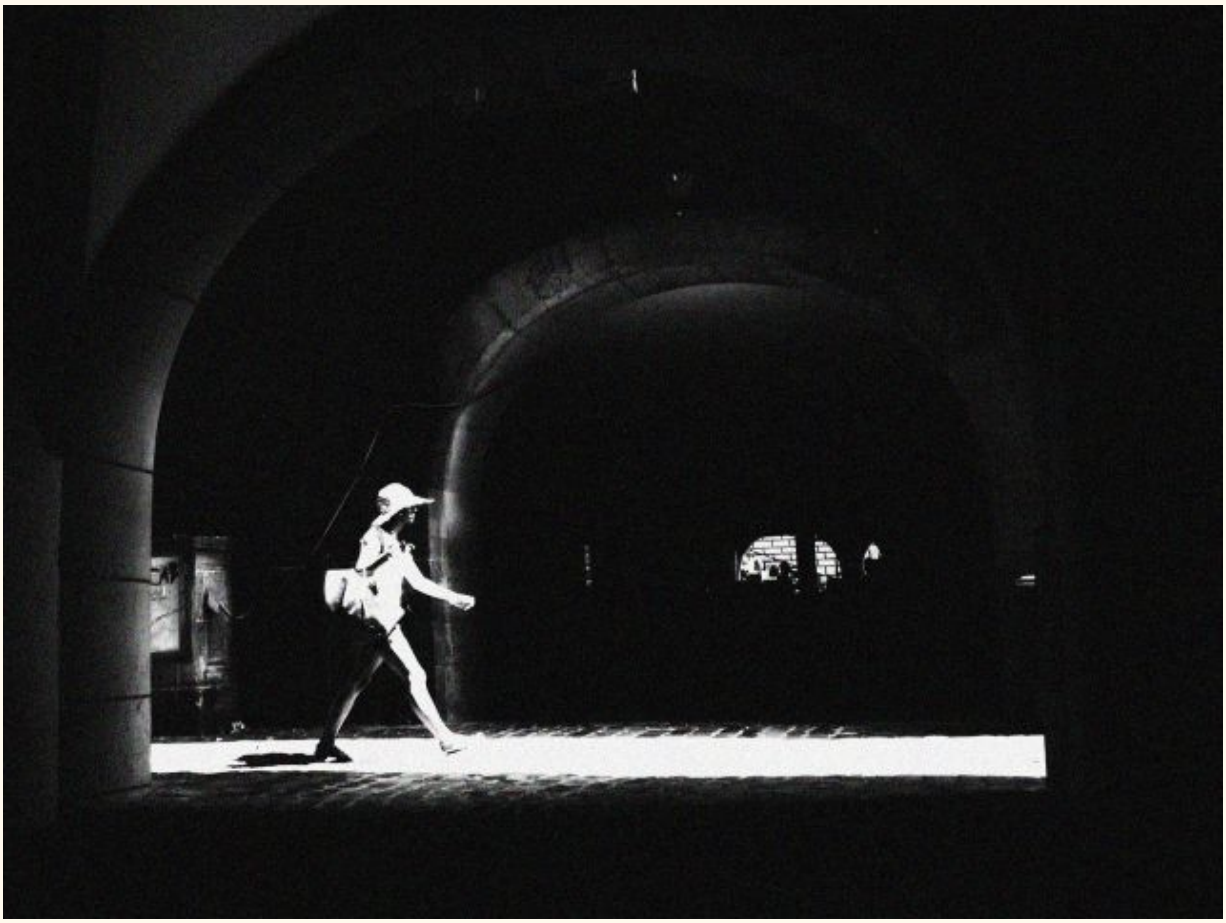
Output :



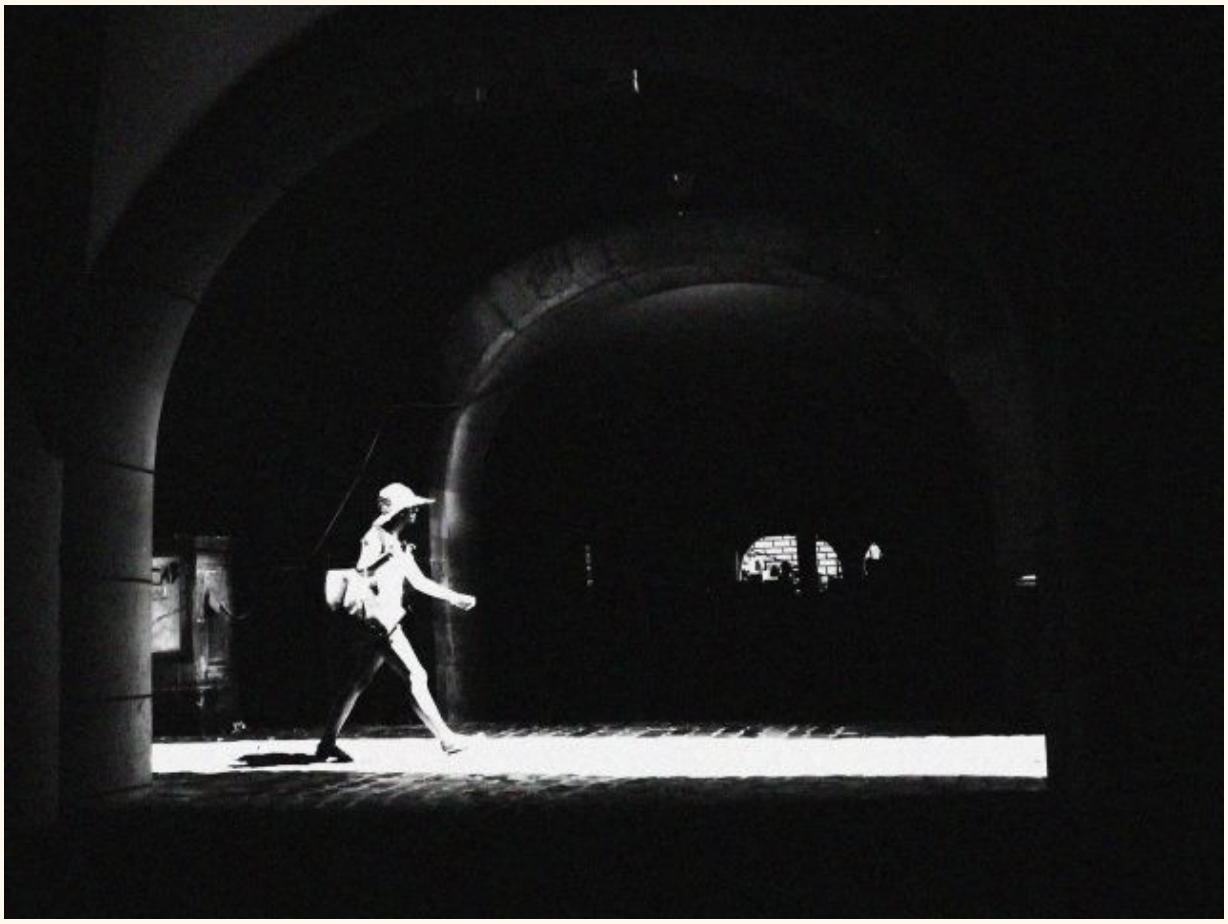
Q6)

1) **Vary the effect of domain and range components of the bilateral filter and try to minimize the L2 distance from the ground truth images. Display the images and the approximate values for the sigma.**

Input:



Output : sd = 0.5, sr = 100



Output : sd = 2, sr = 100

Output : sd = 50, sr = 2

Output : sd = 50, sr = 50

Q7)

1) **Cross Bilateral Filter : In low light imaging, images tend to be noisy and loose sharp edges. However, with flash, there is an unpleasing direct-lighting effect. Thus a cross bilateral filter can be used with the range and spatial filters acting on two different images. Use a cross bilateral filter on the images pots flash.jpg and pots no flash.jpg to produce the output and report the parameters?**

The cross bilateral filter smoothens the image while preserving the edges of the given auxiliary image. The noisy image is a low light image and the auxiliary image is an image with flash.

Steps :

- Implement the cross bilateral filter following the same procedure as the bilateral filter.
- The difference is that the weights will be multiplied with pixel value of auxiliary image.
- The computed maske is then convolved with the image to obtain the cross-bilaterally filtered image.

Input image	Auxiliary image	Output image
		

Q7)

2) **Inverse Bilateral Filter : Does it makes sense to develop an inverse bilateral filter, which blurs an image at edges and preserves the homogeneous regions. If it makes sense, design it and suggest its applications**

Inverse bilateral filter can be used to extract the saliency map of an image, i.e. it helps us to identify the areas of prominence as usually the foreground is more important as it is in focus and has more edges.

Steps:

- The inverse bilateral filter follows the procedure as in the case of the bilateral filter.
- The difference is that for the range kernel in inverse bilateral filter is the inverse of that of the bilateral filter.

```
def inverse_bilateral_filter(win_im,sd=3,sr=50):
      w = 0
      gk = 0
```

```
    i= int(win_im.shape[0]/2)
    # print(i)
    for k in range(win_im.shape[0]):
          for l in range(win_im.shape[1]):

                d = weighting((i-k),sd) * weighting((i-l),sd)
                # print(d)
                v= 1 - abs(win_im[i][i] - win_im[k][l])
                r = weighting(v,sr)
                gk= gk + (win_im[k][l]*(r*d))
                w = w + (r*d)

    return (gk/w)
```

Input	Output
	

Q8)

1) **The image Degraded.jpg has been degraded in some way. Try to find out what kind of degradation has been applied on the image and try to restore it. The original image before degradation is Clean.jpg. Clearly explain what you did to restore the image.of degradation has been applied on the image and try to restore it. The original image before degradation is Clean.jpg. Clearly explain what you did to restore the image.**

Method implemented to de-noise the image:

On looking at corrupted image, it is evident that it has been corrupted by salt and pepper noise. The method to remove the noise from the image is to adopt a median filter.

The median filter considers each pixel in the image in turn and looks at its nearby neighbors to decide whether or not it is representative of its surroundings. Instead of simply replacing the pixel value with the mean of neighboring pixel values, it replaces it with the median of those values. The median is calculated by first sorting all the pixel values from the surrounding neighborhood into numerical order and then replacing the pixel being considered with the middle pixel value.

Steps :

- Perform the appropriate padding on the image depending on the size of the filter.
- For each window in the image, sort the pixel values and replace the centre pixel with median value.
- Iterate over the entire image to get median filtered image.

```
def sliding_window(image,padded_image,window = 3):
      a = np.zeros(image.shape)
      for x in range(image.shape[0]):
            for y in range(image.shape[1]):
                  win_im = padded_image[x:x+window, y:y+window]
                  # print(image[x,y])

                  a[x][y][0] = remove(win_im[:,:,0])
                  a[x][y][1] = remove(win_im[:,:,1])
                  a[x][y][2] = remove(win_im[:,:,2])
      print("median done")

      return a

def padding(im,kernel_row=3,kernel_col=3):
      image_row, image_col,ch = im.shape

      pad_height = int((kernel_row - 1) / 2)
      pad_width = int((kernel_col - 1) / 2)
```

```
      padded_image = np.zeros((image_row + (2 * pad_height), image_col + (2
* pad_width),ch))
      print(padded_image.shape)

      padded_image[pad_height:padded_image.shape[0] - pad_height,
pad_width:padded_image.shape[1] - pad_width] = im
      return padded_image

def remove(win_im):
      x = np.sort(win_im.ravel())
      idx = x.shape[0]
      return x[int(idx/2)]
```

Input:	Output: