Assignment 1

Computer Vision

DR. Ahmed M. Badawi
T.A. Laila Abbas
T.A. Peter Salah

STUDENT NAME	SEC	BN.
GHOFRAN MOHAMMED	2	8
KAREMAN YASER	2	9
MAYAR FAYEZ	2	42
NADA AHMED	2	46
NAIRA YOUSSEF	2	48

Description:

A small web application based app developed with python and streamlit, to apply different image processing techniques.

Requirements:

- Python 3.
- Streamlit 1.13.0
- Numpy 1.23.4
- Scipy 1.9.2
- Matplotlib 3.6.2
- Seaborn 3.6.2

Running command:

Streamlit run server.py

- The UI contains three main tabs filtering, histogram and hybrid images.

<u>Tab1:</u>

- Noise
- Filters
- Edge masks

❖ Noise:

1. Uniform noise

<u>Algorithm</u>

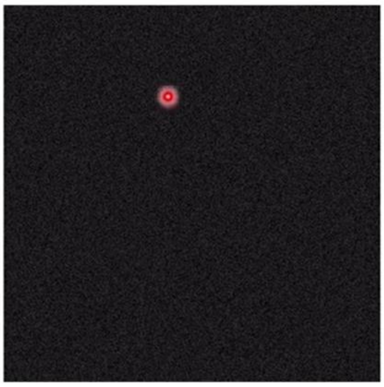
```
def Uniform_noise(image):
    img=image
    img=img/255          #normalization
    # uniform noise
    x,y=img.shape
    a=0

    b=0.5
    noise=np.random.uniform(size = (x,y), low = a, high = b)
    # add noise to image
    noise_img = img + noise
    return noise_img
```

Using uniform distribution

Random number that follows a uniform distribution

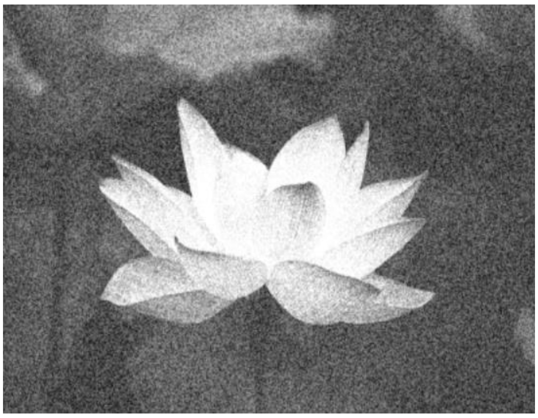
$$p(z) = \begin{cases} \dfrac{1}{b-a} & a \le z \le b \\ 0 & \text{otherwise} \end{cases}$$

Uniform Noise (a=0, b=0.2)

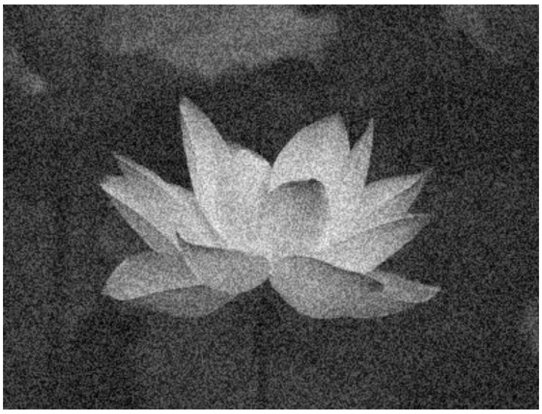


Result

Opencv



Implemented



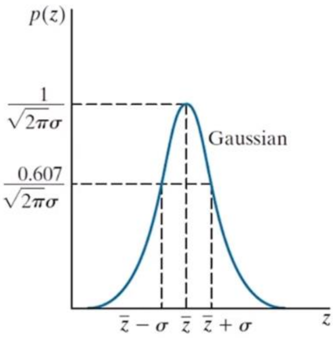
2. Gaussian noise

Algorithm

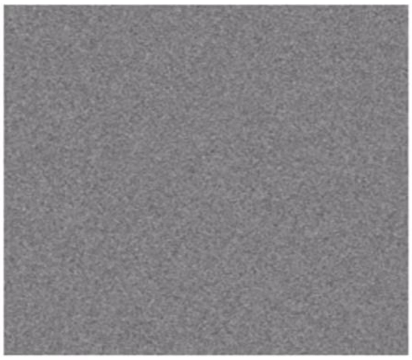
```
def Gaussian_noise(image):
    img=image
    img=img/255          #normalization
    # create gaussion noise
    x,y=img.shape
    mean = 0
    var = 0.01
    sigma = np.sqrt(var)
    noise= np.random.normal(loc=mean,scale=sigma,size=(x,y))
    # add a gaussian noise
    noise_img = img + noise
    return noise_img
```

Using Gaussian distribution

$$p(z) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(z-\bar{z})^2}{2\sigma^2}} \qquad -\infty < z < \infty$$

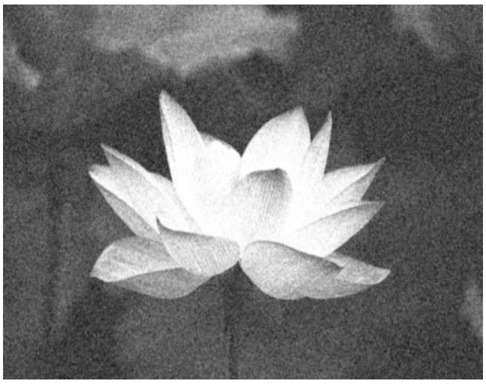


$n(x, y)$

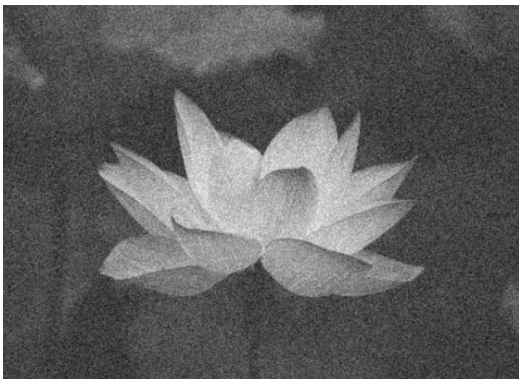


Result

Opencv



Implemented



3. Salt and pepper noise

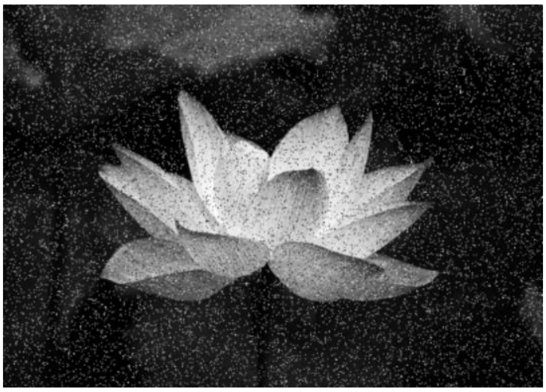
Algorithm

```
def salt_pepper_noise(image):
    # salt=white=1   pepper=black=0
    img=image
    img=img/255           #normalization
    # black image
    x,y = img.shape
    noise=np.zeros((x,y), dtype=np.float64)
    # salt and pepper amount
```

```python
    pepper = 0.1
    salt = 1 - pepper
    #create salt and pepper noise image
    for i in range(x):
        for j in range(y):
            rdn = np.random.random()
            if rdn < pepper:
                noise[i][j] = 0
            elif rdn > salt:
                noise[i][j] = 1
            else:
                noise[i][j] = img[i][j]

    return noise
```

Result

Opencv

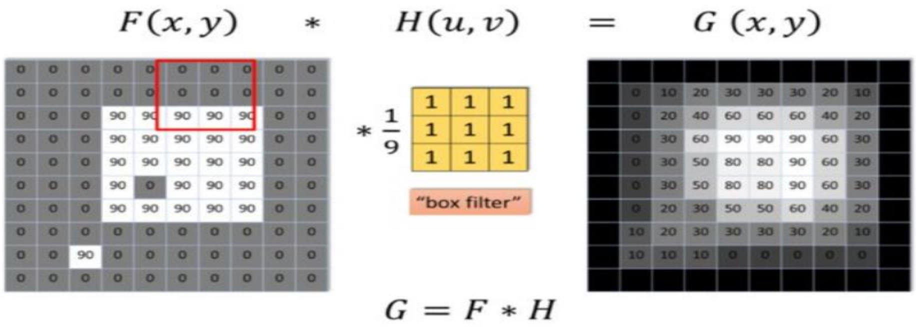


Implemented

❖ Filters:

1. Average filter

. It uses 3 x 3 kernel which is convolved with the input image to calculate output.



Algorithm

```
def average_filter(image,kernel_size):
    img=image
    # Obtain number of rows and columns of the image
    x, y = img.shape
    # Develop Averaging filter(3, 3) mask
    kernel = np.ones([kernel_size , kernel_size ], dtype = int)
    kernel = kernel / (kernel_size**2)
    Xnew=x-kernel_size +1
    Ynew=y-kernel_size +1
    # Convolve the 3X3 mask over the image
    new_image = np.zeros([Xnew , Ynew])
    for i in range(Xnew):
        for j in range(Ynew):
            value=np.multiply(img[i:i+kernel_size,j:j+kernel_size], kernel)
            new_image[i][j]=np.sum(value)
    return new_image
```

Result

Opencv

Implemented

Parameters: Kernel_size

2. Gaussion filter

The formula to design gaussian kernel.

$$\frac{1}{2\pi\sigma^2}.e^{\frac{-(x^2+y^2)}{2\sigma^2}}$$

Example:

i.e $X = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$ and $Y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$

$$\frac{-(x^2+y^2)}{2\sigma^2} = \begin{bmatrix} -2.7778 & -1.3889 & -2.7778 \\ -1.3889 & 0 & -1.3889 \\ -2.7778 & -1.3889 & -2.7778 \end{bmatrix}$$

. It uses gaussian kernel which is convolved with the input image to calculate output.

Algorithm

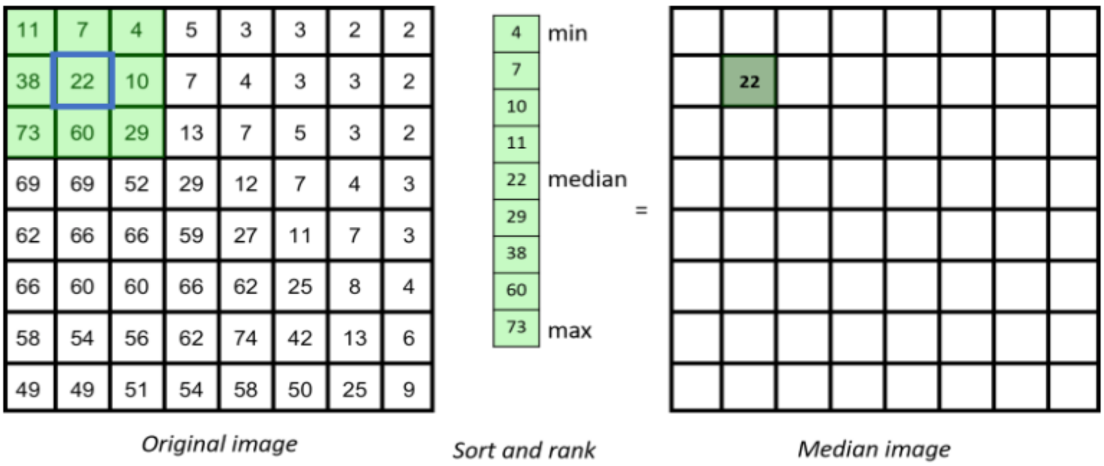
```
def Gaussian_filter(image,kernel_size,sigma):
    img=image
    # Obtain number of rows and columns of the image
    m, n = img.shape
    # Develop gaussian filter(3, 3) mask
    x, y = np.meshgrid(np.linspace(-1,1,kernel_size ), np.linspace(-1,1,kernel_size ))
    d = -(x*x+y*y)
    kernel= np.exp(-( (d)**2 / ( 2.0 * sigma**2 ) ) )/(2*np.pi*sigma**2)
    Xnew=m-kernel_size +1
    Ynew=n-kernel_size +1

    # Convolve the 3X3 mask over the image
    img_new = np.zeros([Xnew , Ynew ])
    for i in range(Xnew):
        for j in range(Ynew):
            value=np.multiply(img[i:i+kernel_size,j:j+kernel_size], kernel)
            img_new[i][j]=np.sum(value)
    return img_new
```

Result

Opencv Implemented

Parameters: Kernel size,sigma

3. Median filter



Algorithm

```
def median_filter(image,size):
    img=image
    # Obtain number of rows and columns of the image
    x, y = img.shape
    Xnew=x-size +1
    Ynew=y-size +1
    # Traverse the image. For every 3X3 area,
    # find the median of the pixels and
    # replace the center pixel by the median
    img_new = np.zeros([Xnew , Ynew])

    for i in range(Xnew ):
        for j in range(Ynew):
```

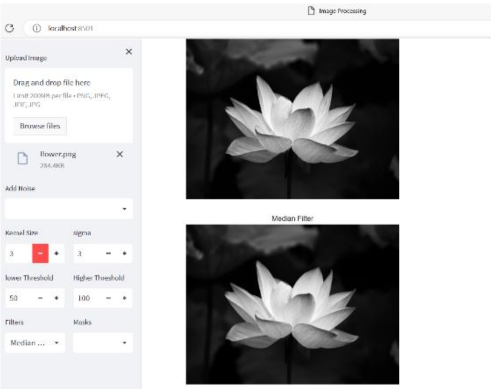
```python
        value = img[i:i+size,j:j+size]
        value=value.reshape(-1)
        value = sorted(value)
        img_new[i, j]= np.median(value )
return  img_new
```

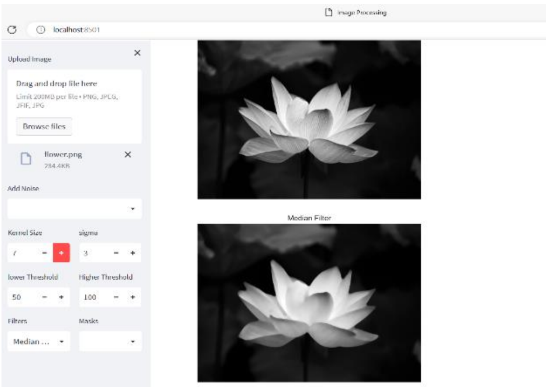
Result

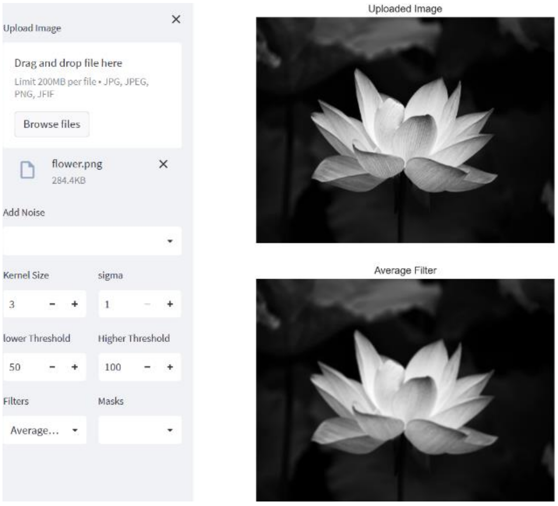
Opencv

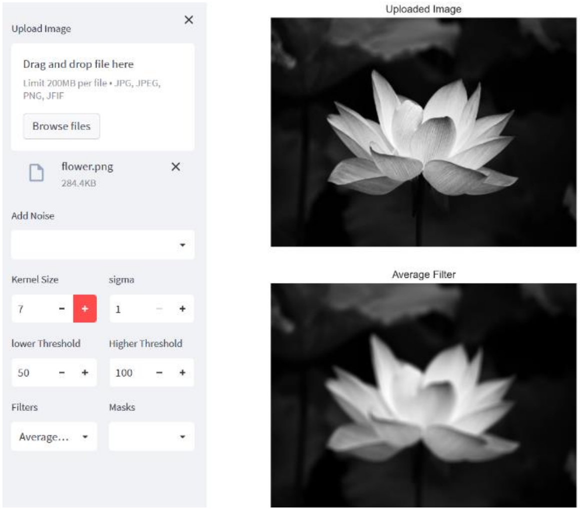
Implemented

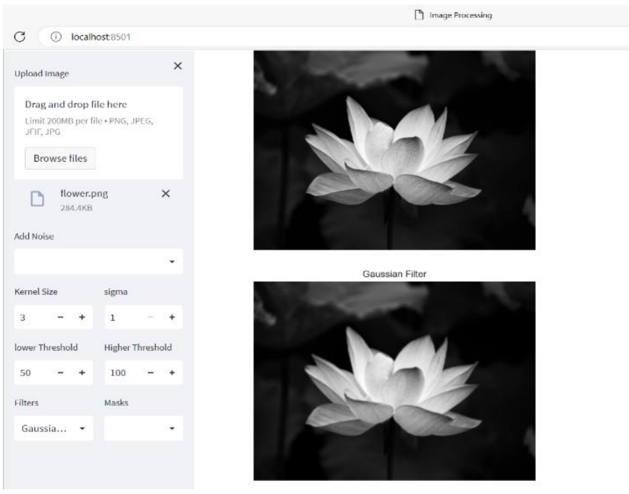
Parameters: Kernel size

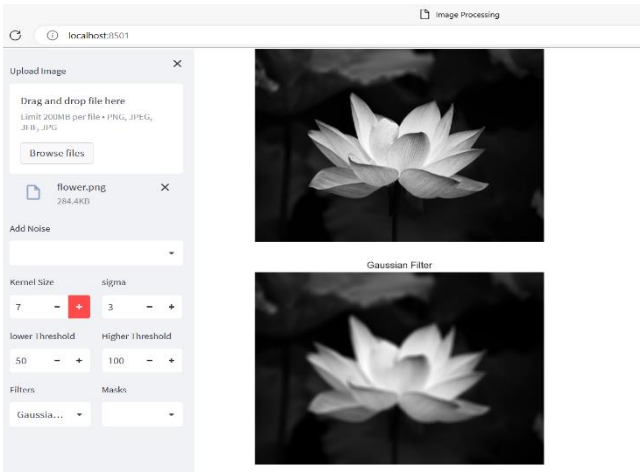






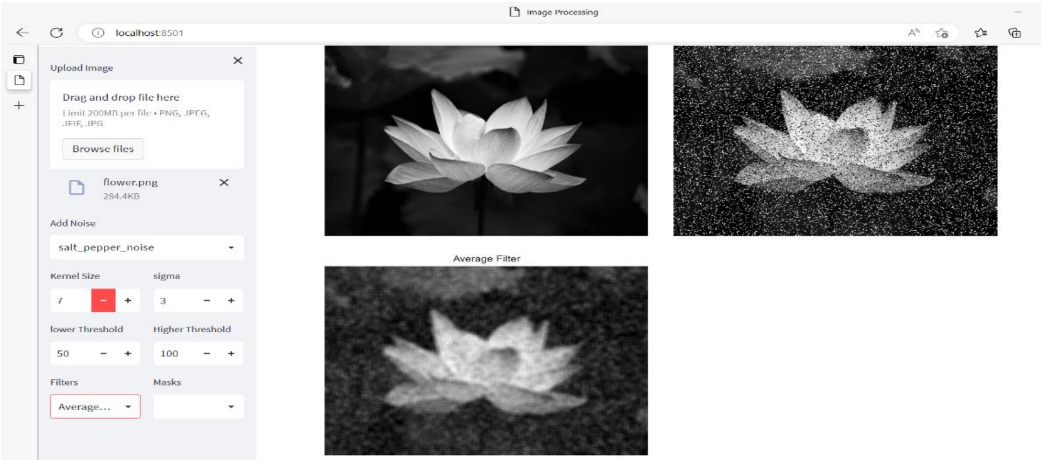


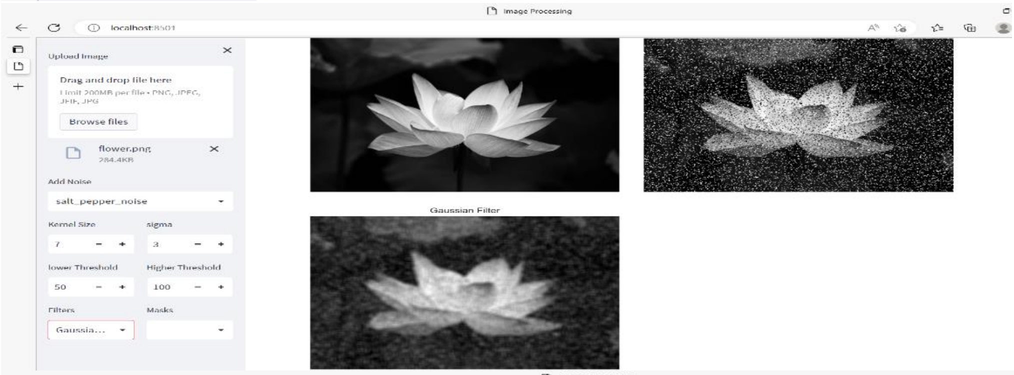




Note

The larger the kernel size, the greater the smoothing.





Note

Median filter is the best filter for salt and pepper noise.

❖ Edges:

Edges are significant changes in intensity of pixels of digital image. There are 3types of edges: horizontal, vertical and diagonal. Edge detection is segmentation into regions of discontinuity, there are 2 types of operators:

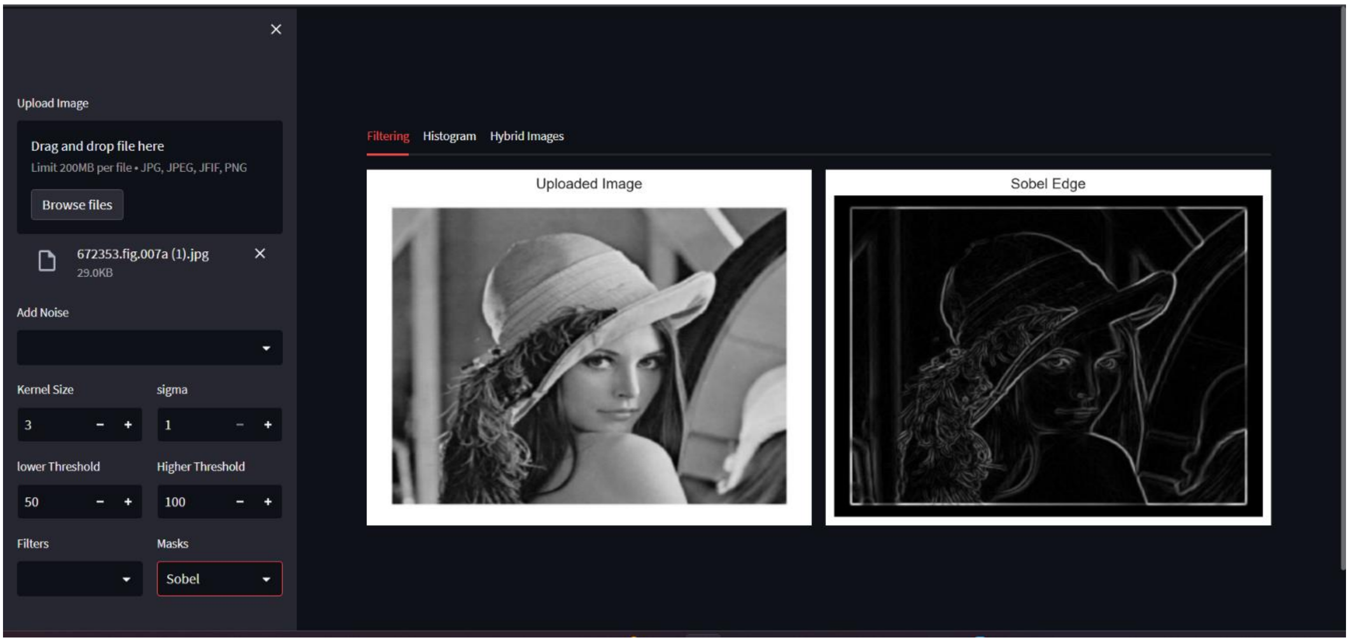
Gradient: compute 1^{st} order derivative like **Sobel**, **Prewitt** and **Roberts.**

Gaussian: compute 2^{nd} order derivative like **Canny.**

1. **Sobel:**

discrete differentiation operator. It computes the gradient approximation of image intensity. It uses two 3 x 3 kernels or masks which are convolved with the input image to calculate the vertical and horizontal derivative approximations.

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \qquad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$



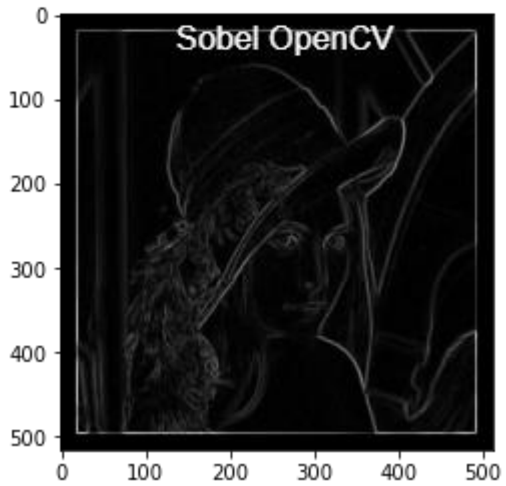
Algorithm:

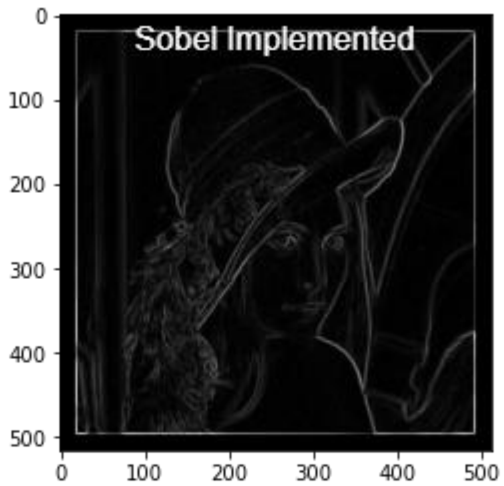
```
_def Sobel(image):

    img=image
    kernelx = np.array([[1.0, 0.0, -1.0], [2.0, 0.0, -2.0], [1.0, 0.0, -1.0]])
    kernely = np.array([[1.0, 2.0, 1.0], [0.0, 0.0, 0.0], [-1.0, -2.0, -1.0]])
    m, n = img.shape
    img_sobelx = np.zeros([m, n])
```

```
    img_sobely = np.zeros([m, n])
    for i in range(1, m-1):
        for j in range(1, n-1):
            value = img[i-1, j-1]*kernelx[0, 0]+img[i-1, j]*kernelx[0, 1]+img[i-1, j +
1]*kernelx[0, 2]+img[i, j-1]*kernelx[1, 0]+ img[i, j]*kernelx[1, 1]+img[i, j +
1]*kernelx[1, 2]+img[i + 1, j-1]*kernelx[2, 0]+img[i + 1, j]*kernelx[2, 1]+img[i + 1, j
+ 1]*kernelx[2, 2]

            img_sobelx[i, j]= value
    for i in range(1, m-1):
        for j in range(1, n-1):
            value = img[i-1, j-1]*kernely[0, 0]+img[i-1, j]*kernely[0, 1]+img[i-1, j +
1]*kernely[0, 2]+img[i, j-1]*kernely[1, 0]+ img[i, j]*kernely[1, 1]+img[i, j +
1]*kernely[1, 2]+img[i + 1, j-1]*kernely[2, 0]+img[i + 1, j]*kernely[2, 1]+img[i + 1, j
+ 1]*kernely[2, 2]

            img_sobely[i, j]= value
    edged_img=np.sqrt( np.square(img_sobelx) + np.square(img_sobely))
    theta= np.arctan2(img_sobely,img_sobelx)
    return edged_img ,theta
```

Result observed:

There is few differences between Sobel filter by library OpenCV and our implementation.





2. **Prewitt:**

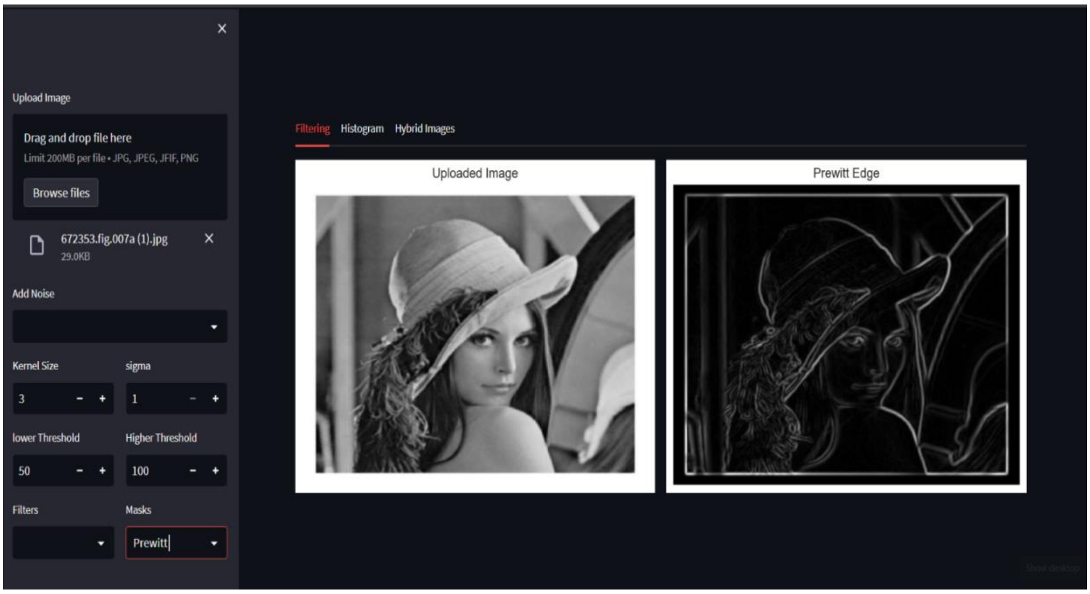
This operator is almost similar to the Sobel operator. It also detects vertical and horizontal edges of an image. It is one of the best ways to detect the orientation and magnitude of an image.

-1	0	+1
-1	0	+1
-1	0	+1

G_x

+1	+1	+1
0	0	0
-1	-1	-1

G_y



Algorithm:

```
def Prewitt(image,kernel_size,sigma):
    img_g = Gaussian_filter(image,kernel_size,sigma)
    m, n = img_g.shape
    kernely = np.array([[1,1,1],[0,0,0],[-1,-1,-1]])
    kernelx = np.array([[-1,0,1],[-1,0,1],[-1,0,1]])
    img_prewittx = np.zeros([m, n])
    img_prewitty = np.zeros([m, n])
    for i in range(1, m-1):
        for j in range(1, n-1):
            value = img_g[i-1, j-1]*kernelx[0, 0]+img_g[i-1, j]*kernelx[0, 1]+img_g[i-
1, j + 1]*kernelx[0, 2]+img_g[i, j-1]*kernelx[1, 0]+ img_g[i, j]*kernelx[1, 1]+img_g[i,
j + 1]*kernelx[1, 2]+img_g[i + 1, j-1]*kernelx[2, 0]+img_g[i + 1, j]*kernelx[2,
1]+img_g[i + 1, j + 1]*kernelx[2, 2]

            img_prewittx[i, j]= value
    for i in range(1, m-1):
        for j in range(1, n-1):
            value = img_g[i-1, j-1]*kernely[0, 0]+img_g[i-1, j]*kernely[0, 1]+img_g[i-
1, j + 1]*kernely[0, 2]+img_g[i, j-1]*kernely[1, 0]+ img_g[i, j]*kernely[1, 1]+img_g[i,
j + 1]*kernely[1, 2]+img_g[i + 1, j-1]*kernely[2, 0]+img_g[i + 1, j]*kernely[2,
1]+img_g[i + 1, j + 1]*kernely[2, 2]

            img_prewitty[i, j]= value
    edged_img=np.sqrt( np.square(img_prewittx) + np.square(img_prewitty))
    return edged_img
```

3. Roberts:

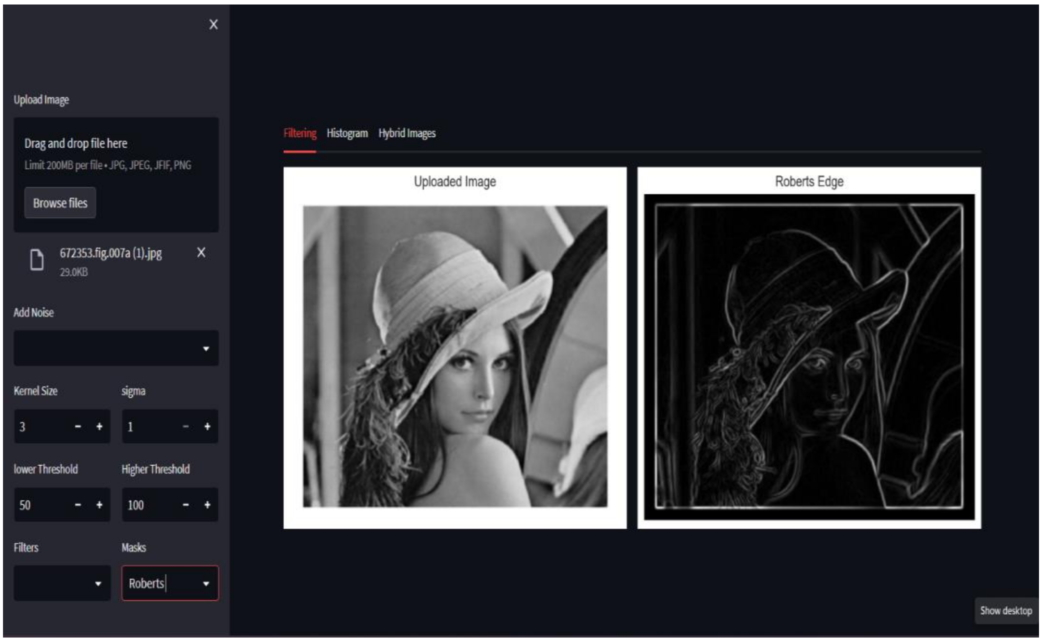
This operator computes the sum of squares of the differences between diagonally adjacent pixels in an image through discrete differentiation. Then approximation is made by convolving the image with 2 x 2 kernels.

+1	0
0	-1

Gx

0	+1
-1	0

Gy



Algorithm:

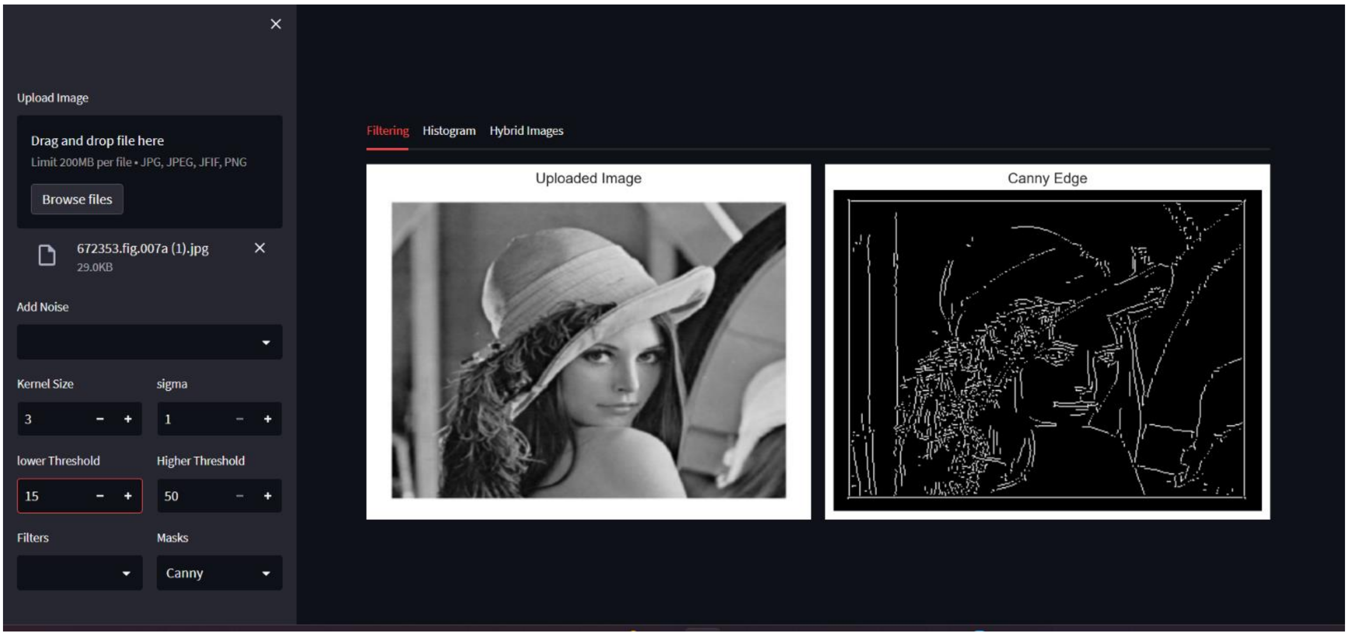
```
def Roberts(image,kernel_size,sigma):

    img_g = Gaussian_filter(image,kernel_size,sigma)
    kernelx = np.array( [[1, 0 ],
                         [0,-1 ]] )

    kernely = np.array( [[ 0, 1 ],
                         [ -1, 0 ]] )
    vertical = ndimage.convolve( img_g, kernelx )
    horizontal = ndimage.convolve( img_g, kernely)

    edged_img = np.sqrt( np.square(horizontal) + np.square(vertical))
    return edged_img
```

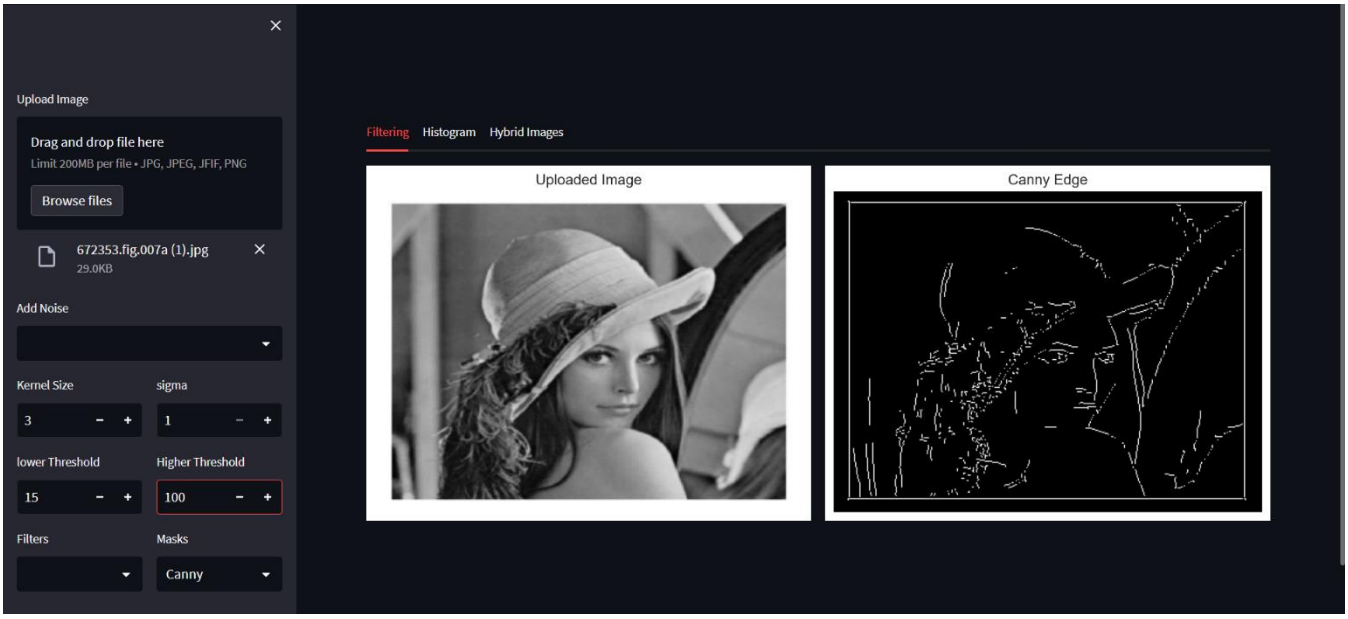
4. Canny:

This operator is not susceptible to noise. It extracts image features without affecting the feature. Canny edge detector have advanced algorithm on 5 steps:

- Noise reduction by Gaussian filter.
- Gradient calculation by sobel mask.
- Non-maximum suppression: thin out the edges.
- Double threshold: identifying 3 kinds of pixels: strong, weak, and non-relevant
- Edge Tracking by Hysteresis:
  transforming weak pixels into strong ones, if and only if at least one of the pixels around the one being processed is a strong one.

By changing the input lower and higher thresholds the edges changes.





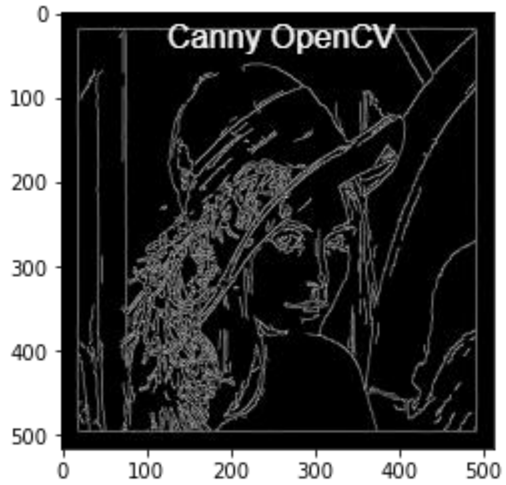
Algorithm:

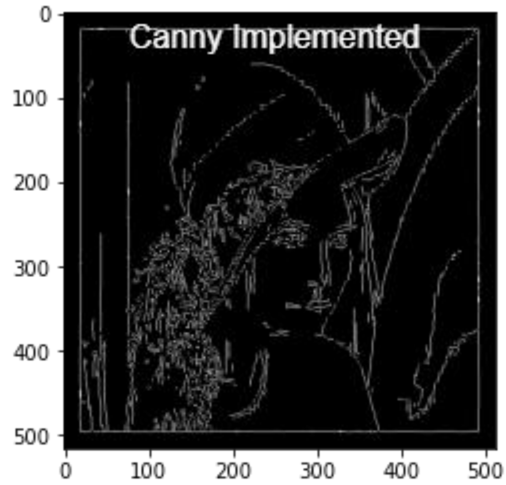
```
def Canny(image,kernal_s,sigma,lowThreshold,highThreshold):

    img_g = Gaussian_filter(image,kernal_s,sigma)
    s_img, theta= Sobel(img_g)
    M, N = image.shape
    Z = np.zeros((M,N),dtype=np.int32)
    angle = theta * 180. / np.pi
    angle[angle < 0] += 180
    for i in range(1,M-1):
            for j in range(1,N-1):
                try:
                    q = 255
                    r = 255

                #angle 0
                    if (0 <= angle[i,j] < 22.5) or (157.5 <= angle[i,j] <= 180):
                        q = s_img[i, j+1]
                        r = s_img[i, j-1]
                    #angle 45
                    elif (22.5 <= angle[i,j] < 67.5):
                        q = s_img[i+1, j-1]
                        r = s_img[i-1, j+1]
                    #angle 90
                    elif (67.5 <= angle[i,j] < 112.5):
                        q = s_img[i+1, j]
                        r = s_img[i-1, j]
                    #angle 135
                    elif (112.5 <= angle[i,j] < 157.5):
                        q = s_img[i-1, j-1]
                        r = s_img[i+1, j+1]

                    if (s_img[i,j] >= q) and (s_img[i,j] >= r):
                        Z[i,j] = s_img[i,j]
                    else:
                        Z[i,j] = 0

                except IndexError as e:
                    pass

    M, N = Z.shape
    res = np.zeros((M,N), dtype=np.int32)

    weak = np.int32(25)
    strong = np.int32(255)

    strong_i, strong_j = np.where(Z >= highThreshold)
    zeros_i, zeros_j = np.where(Z < lowThreshold)
```

```
    weak_i, weak_j = np.where((Z <= highThreshold) & (Z >= lowThreshold))

    res[strong_i, strong_j] = strong
    res[weak_i, weak_j] = weak
    strong=255
    M, N = res.shape
    for i in range(1, M-1):
        for j in range(1, N-1):
            if (res[i,j] == weak):
                try:
                    if ((res[i+1, j-1] == strong) or (res[i+1, j] == strong) or
(res[i+1, j+1] == strong)
                        or (res[i, j-1] == strong) or (res[i, j+1] == strong)
                        or (res[i-1, j-1] == strong) or (res[i-1, j] == strong) or
(res[i-1, j+1] == strong)):
                        res[i, j] = strong
                    else:
                        res[i, j] = 0
                except IndexError as e:
                    pass
    return res
```

<u>Result observed:</u>

There are noticeable differences between the library and implemented edge detection, due library handle more cases and angels.



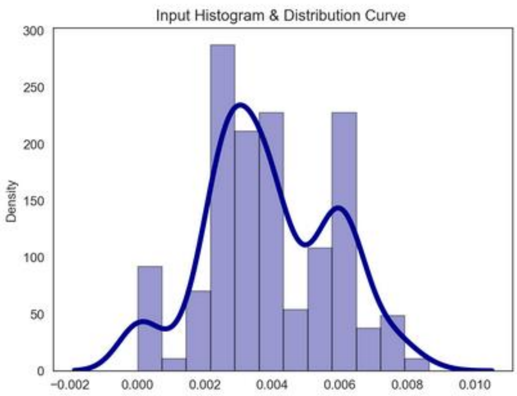


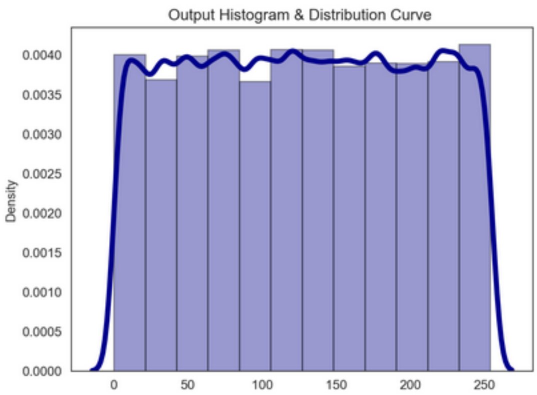
Applying noise then mask example:

Tab2:

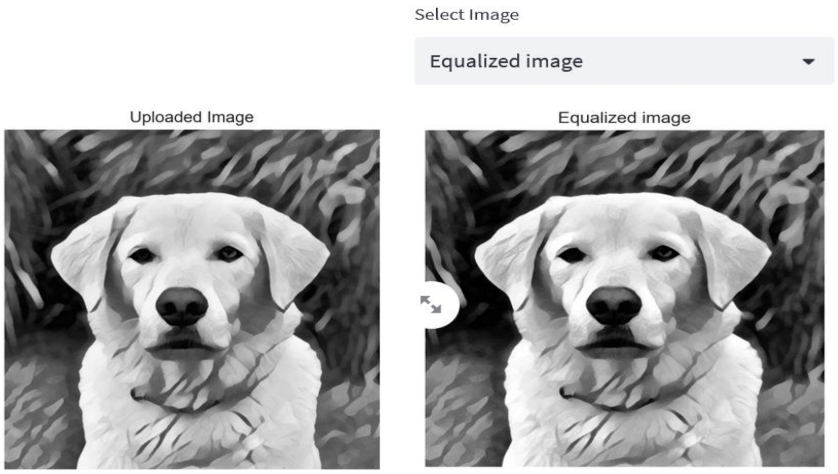
- Histogram, its distribution curve, Equalization and Normalization
- Local and Global Thresholding
- RGB Visualization

1. **Histogram and its distribution curve:**
   An image histogram is a representation of the range of tonal values that is present in an image





2. **Equalization:**
   Histogram equalization is a method in image processing of contrast adjustment using the image's histogram.



Algorithm

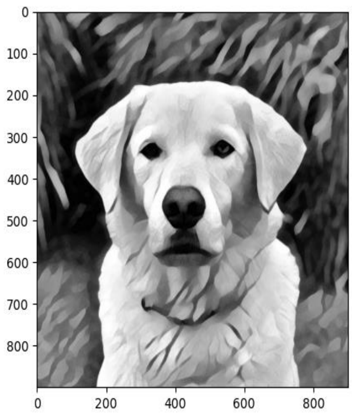
```
def histogram_fun(image):
    histogram_array = np.bincount(image.flatten(), minlength=256)
    #normalize
    num_pixels = np.sum(histogram_array)
    histogram_array = histogram_array/num_pixels
    #normalized cumulative histogram
    chistogram_array = np.cumsum(histogram_array)
    # Pixel mapping lookup table
    transform_map = np.floor(255 * chistogram_array).astype(np.uint8)
    # flatten image array into 1D list
    img_list = list(image.flatten())

    # transform pixel values to equalize
    eq_img_list = [transform_map[p] for p in img_list]

    # reshape and write back into img_array
    eq_img_array = np.reshape(np.asarray(eq_img_list), image.shape)

    return histogram_array,eq_img_array
```

we calculate the normalized histogram of the image. Normalization is performed by dividing the frequency of each bin by the total number of pixels in the image then we derive a lookup table which maps the pixel intensities to achieve an equalized histogram characteristics. Finally Transform pixel intensity of the original image with the lookup table.

Result observed:

We will notice that there's not much difference between our implementation and cv2 python library result.

Cv2 result



Our results

3. Normalization:

The process that changes the range of pixel intensity values to be from 0 to 1

<u>Algorithm:</u>

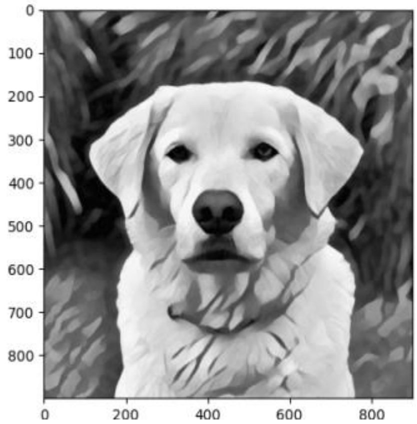
```
def normalize_images(image):

    # initial zero ndarray
    normalized_images = np.zeros_like(image.astype(float))

    # The first images index is number of images where the other indices indicates
    # hieight, width and depth of the image
    num_images = image.shape[0]

    # Computing the minimum and maximum value of the input image to do the
normalization based on them
    maximum_value, minimum_value = image.max(), image.min()

    # Normalize all the pixel values of the images to be from 0 to 1
    for img in range(num_images):
        normalized_images[img, ...] = (image[img, ...] - float(minimum_value)) /
float(maximum_value - minimum_value)

    return normalized_images
```

we normalize a color input image using min-max norm. The image pixel values are normalized to a range [0,1]

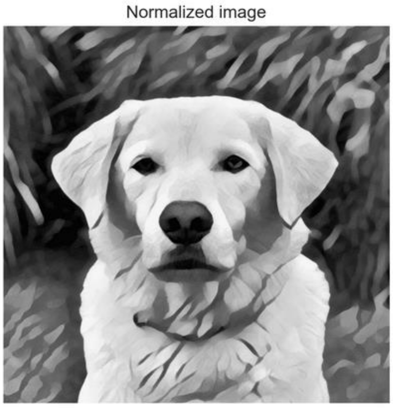
Result observed:

We will notice that there's not much difference between our implementation and cv2 python library result.

Cv2 result



Our results



4. Local and Global Thresholding:

In global thresholding, each pixel value in the image is compared with a single (global) threshold value(higher than the threshold the pixel value will be 1 otherwise it will be 0). In local thresholding the same happens however the image is divided into parts each part has its own threshold based on the pixel values in that part.

Algorithm:

```
def global_threshold(image):


    height = image.shape[0]
    width = image.shape[1]

    img_thres= np.zeros((height,width))
```

```
    thresh=np.median(image)
    # loop over the image, pixel by pixel
    for y in range(0, height):
        for x in range(0, width):
            # threshold the pixel
            pixel = image[y, x]
            img_thres[y, x] = 0 if pixel <thresh else 1
    return img_thres

def local_threshold(image):
    height = image.shape[0]
    width = image.shape[1]
    half_height = height//2
    half_width = width//2
    section1 = image[:half_height, :half_width]
    section2= image[:half_height,half_width:]
    section3= image[half_height:, :half_width]
    section4=image[half_height:,half_width:]
    img_thres= np.zeros((height,width))
    img_thres[:half_height, :half_width]=global_threshold(section1)
    img_thres[:half_height,half_width:]=global_threshold(section2)
    img_thres[half_height:, :half_width]=global_threshold(section3)
    img_thres[half_height:,half_width:]=global_threshold(section4)
    return img_thres
```

In global thresholding algorithm we create an empty array as the same size as the image and we calculate the median of an image as our threshold we loop over pixels values to decide each pixel value (either 1 or 0). In local thresholding we do the same to each part of the image.

- RGB Channels:

We read image in color mood and split it to 3 channel ,then plot them.

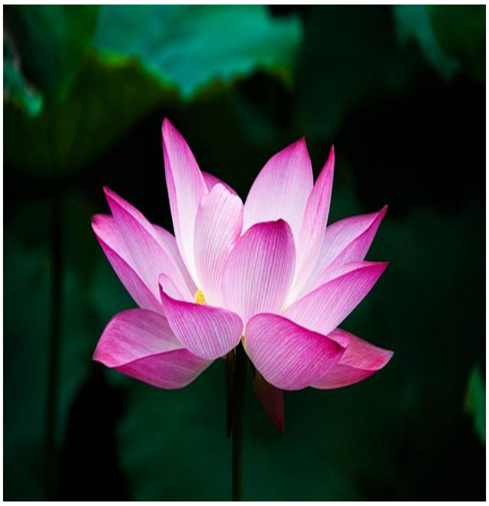
Algorithm:

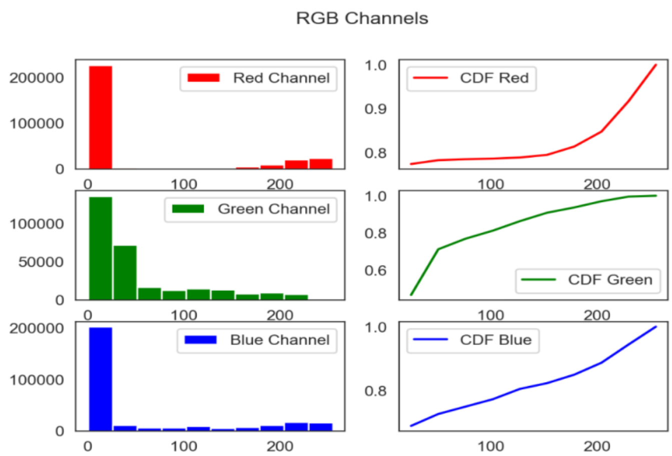
```
def CDF (data):
    count, bins = np.histogram(data, bins=10)
    pdf = count / sum(count)
    cdf = np.cumsum(pdf)
    return cdf , bins

color_img = io.imread(file_path)
red = color_img[:, :, 0]
green = color_img[:, :, 1]
blue = color_img[:, :, 2]

cdf_red, bins_red = CDF(red)
```

```
cdf_green, bins_green = CDF(green)
cdf_blue, bins_blue = CDF(blue)

figure, axs = plt.subplots(3,2)
axs[0,0].hist(red.ravel(),bins=10,color='r',label="Red Channel")
axs[1,0].hist(green.ravel(),bins=10,color='g',label="Green Channel")
axs[2,0].hist(blue.ravel(),bins=10,color='b',label="Blue Channel")
axs[0,1].plot(bins_red[1:], cdf_red,color='r', label="CDF Red")
axs[1,1].plot(bins_green[1:], cdf_green,color='g', label="CDF Green")
axs[2,1].plot(bins_blue[1:], cdf_blue,color='b', label="CDF Blue")
figure.suptitle('RGB Channels')
```

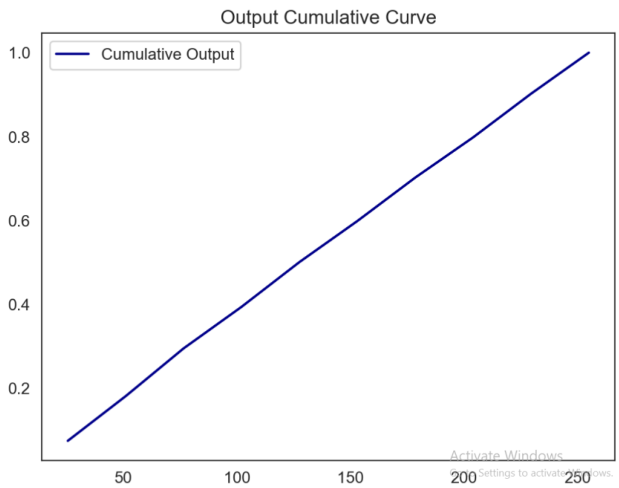
- RGB Channels & Cumulative curves for input image

Uploaded Image 1





- Cumulative Curve for Equalized image

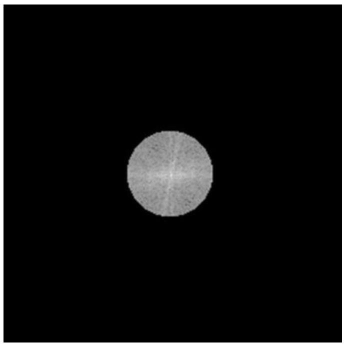


Tab3:

- Frequency Domain Filter
- Hybrid Images

❖ Frequency Domain Filter:

- First transform the image F(u,v) = F{f(u,v0} into frequency domain using fourier's transform.
- Multiply the filter H(u,v).
- Take the inverse fourier's transform g(x,y) = f^{-1}{g(u,v)}.



1. Ideal Low Pass Filter
   Low frequency components are located at corners, we shift them first to the center then apply the filter. The filter eliminates all high frequencies and the resulted image is smoothed.

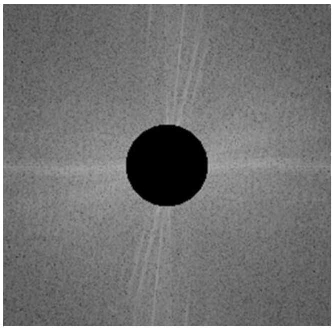
```
def lowPassFilter(image):
    fftimg = np.fft.fft2(image)
    #shifting low frequncies to the center
    fftshiftimg = np.fft.fftshift(fftimg)
    M,N = image.shape
    H = np.zeros((M,N), dtype=np.float32)
    D0 = 50
    for u in range(M):
        for v in range(N):
            D = np.sqrt((u-M/2)**2 + (v-N/2)**2)
            if D <= D0:
               H[u,v] = 1
            else:
               H[u,v] = 0
    # Ideal Low Pass Filtering
    Gshift = fftshiftimg * H
    # Inverse Fourier Transform
    ifftimg = np.fft.ifftshift(Gshift)
    lpfilterdimg = np.abs(np.fft.ifft2(ifftimg))
    return lpfilterdimg, H, fftshiftimg
```

implemented

OpenCV

2. Ideal high pass filter

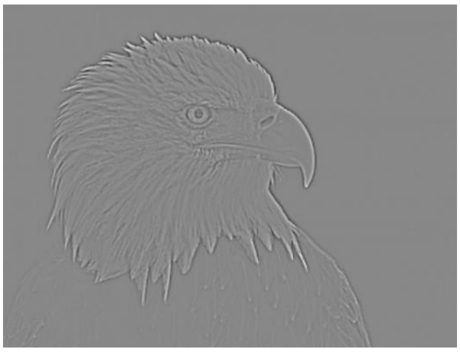
It does the opposite of LPF so simply we can create it by subtracting the LPF from 1 (HP = 1- LP).
The filter removes the constant brightness regions and leave the regions with rapid brightness transitions.



```
def highPassFilter(image) :
    lpfilterdimg, H, fftshiftimg = lowPassFilter(image)
    H = 1 - H
    Gshift= fftshiftimg * H
    ifftimg = np.fft.ifftshift(Gshift)
    hpfilterdimg = np.abs(np.fft.ifft2(ifftimg))
    return hpfilterdimg
```

implemented

OpenCV



parameters

The parameter D0 represents the radius of the cut off circle as it get larger the image will be less smoothed in LPF and more sharper in HPF.

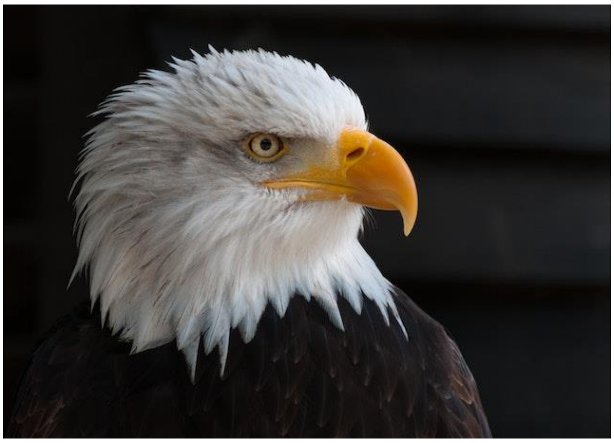
❖ Hybrid Image:

We upload 2 images apply high & low filter to them,

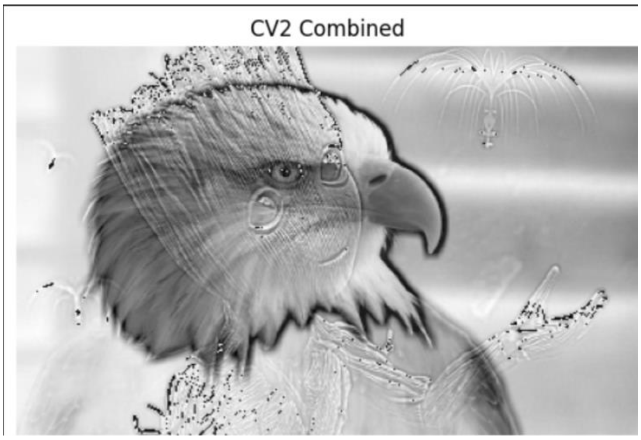
then choose option :

- Low pass image1 + High pass image2
- High pass image1 + Low pass image2

○ Uploaded Images



1) High pass image1 + Low pass image2



2) Low pass image1 + High pass image2