# MACHINE PERCEPTION

## ASSIGNMENT 1

**Submitted by:**

**Shachi Bhavsar(MT2016036)**
**Meet Patel (MT2016102)**

**Question 1: Choose an RGB image (Image1); Plot R, G, and B separately (Write clear comments and observations)**

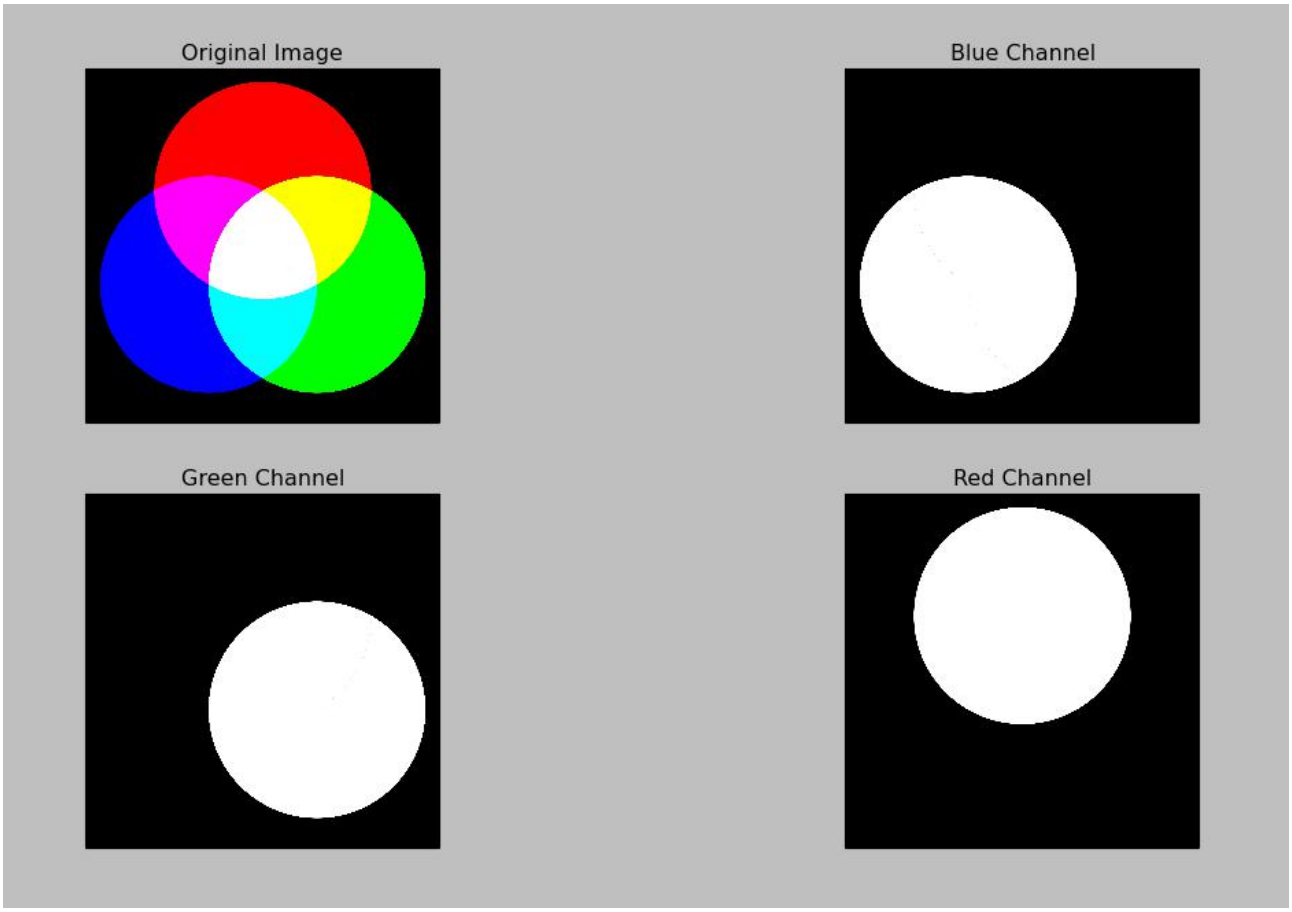
First of all load color image using **cv2.imread**(image,flag) where image is the original image which need to be plot and flag value is **cv2.IMREAD_COLOR** that reads image in color format.
After that we need to split the BGR image into single color planes by using numpy indexing instead of using cv2.split() operation because cv2.split() is a costly operation.

**Syntax:** image[:,:,index_number]

Index number 0 will be used for blue and 1 will be used for green and 2 will be used for red channel.
After extracting blue, red and green color planes, plot it using matplotlib.

Following will show output of the question 1.

**Output:**



**Question 2: Convert Image into HSL and HSV. Write the expressions for computing H, S and V/I.**

For color conversion, we use the function **cv2.cvtColor(input_image, flag)** where flag determines the type of conversion. **F**or BGR → HSV, we use the flag cv2.COLOR_BGR2HSV. So our function will become **cv2.cvtColor(image,cv2.COLOR_BGR2HSV).**HSV stands for Hue, Saturation and Value (Brightness). HSV is a rearrangement of RGB in a cylindrical shape. The HSV ranges are:

- 0 > H > 360 ⇒ OpenCV range = H/2 (0 > H > 180)
- 0 > S > 1 ⇒ OpenCV range = 255*S (0 > S > 255)
- 0 > V > 1 ⇒ OpenCV range = 255*V (0 > V > 255)

We need to split the HSV image into single channel planes by using numpy indexing.

**Syntax:** hsv_image[:,:,index_number]

Index number 0 will be used for hue channel and 1 will be used for saturation channel and 2 will be used for value channel.

**Equation for HSV:**

$$V \leftarrow \max(R, G, B)$$

$$S \leftarrow \begin{cases} \frac{V-\min(R,G,B)}{V} & \text{if } V \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$H \leftarrow \begin{cases} 60(G-B)/(V-\min(R,G,B)) & \text{if } V = R \\ 120+60(B-R)/(V-\min(R,G,B)) & \text{if } V = G \\ 240+60(R-G)/(V-\min(R,G,B)) & \text{if } V = B \end{cases}$$

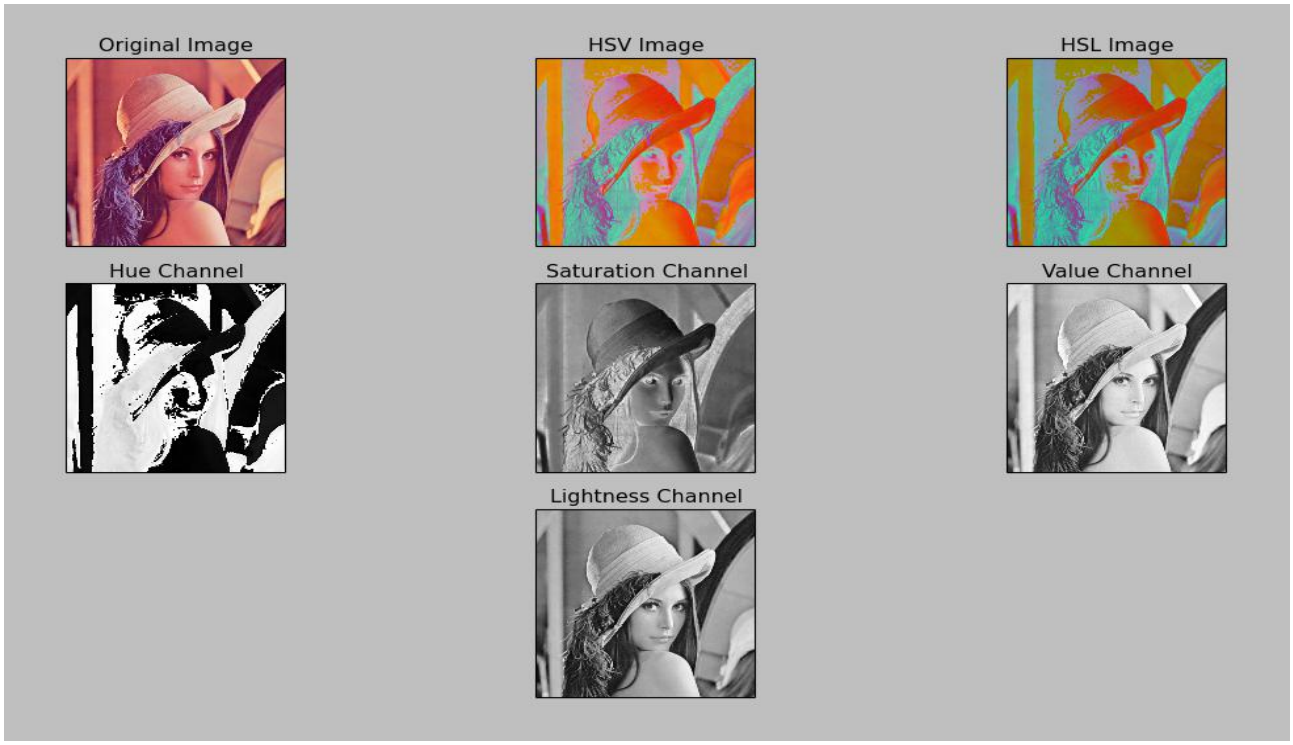
Hues are described by a number that specifies the position of the corresponding pure color on the color wheel. The *saturation* (S) of a color describes how white the color is. The *lightness(L) of a color* describes how dark the color is. A value of 0 is black, with increasing lightness moving away from black.

**Equation of HLS:**

$$V_{max} \leftarrow max(R, G, B)$$
$$V_{min} \leftarrow min(R, G, B)$$
$$L \leftarrow \frac{V_{max} + V_{min}}{2}$$
$$S \leftarrow \begin{cases} \frac{V_{max} - V_{min}}{V_{max} + V_{min}} & \text{if } L < 0.5 \\ \frac{V_{max} - V_{min}}{2 - (V_{max} + V_{min})} & \text{if } L \geq 0.5 \end{cases}$$
$$H \leftarrow \begin{cases} 60(G - B)/S & \text{if } V_{max} = R \\ 120 + 60(B - R)/S & \text{if } V_{max} = G \\ 240 + 60(R - G)/S & \text{if } V_{max} = B \end{cases}$$

For HLS conversion, use the same function which is used for HSV but only different is in flag. Here we used cv2.COLOR_BGR2HLS as a flag value. For BGR → HLS conversion, we use the function cv2.cvtColor(image,cv2.COLOR_BGR2HLS). During HSV conversion, we split hue, saturation and value channel using numpy indexing. For HSL image, only lightness channel is different. Hue and saturation channel are same. So, make copy of hsv image and replace third plane (value channel) of HSV with lightness to het HSL image.

**Output:**



**Question 3: Convert Image 1 into L*a*b* and plot**

The three coordinates of L*a*b* represent the lightness of the color:
$L^* = 0$ indicates black and $L^* = 100$ indicates diffuse white.
*If a* has* negative values that indicate green while positive values indicate magenta. If b* *has* negative values that indicate blue and positive values indicate yellow.

For L*a*b* conversion, use the function cv2.cvtColor(input_image, flag) where flag determines the type of conversion. For BGR → LAB conversion we use the flag cv2.COLOR_BGR2LAB

In this color-space, L stands for the Luminance dimension, while a and b are the color-opponent dimensions. The L*a*b ranges are:

- 0 > L > 100 ⇒ OpenCV range = L*255/100 (1 > L > 255)
- -127 > a > 127 ⇒ OpenCV range = a + 128 (1 > a > 255)
- -127 > b > 127 ⇒ OpenCV range = b + 128 (1 > b > 255)

**Equation of L*a*b*:**

$$L^\star = 116 f\left(\frac{Y}{Y_\mathrm{n}}\right) - 16$$
$$a^\star = 500\left(f\left(\frac{X}{X_\mathrm{n}}\right) - f\left(\frac{Y}{Y_\mathrm{n}}\right)\right)$$
$$b^\star = 200\left(f\left(\frac{Y}{Y_\mathrm{n}}\right) - f\left(\frac{Z}{Z_\mathrm{n}}\right)\right)$$

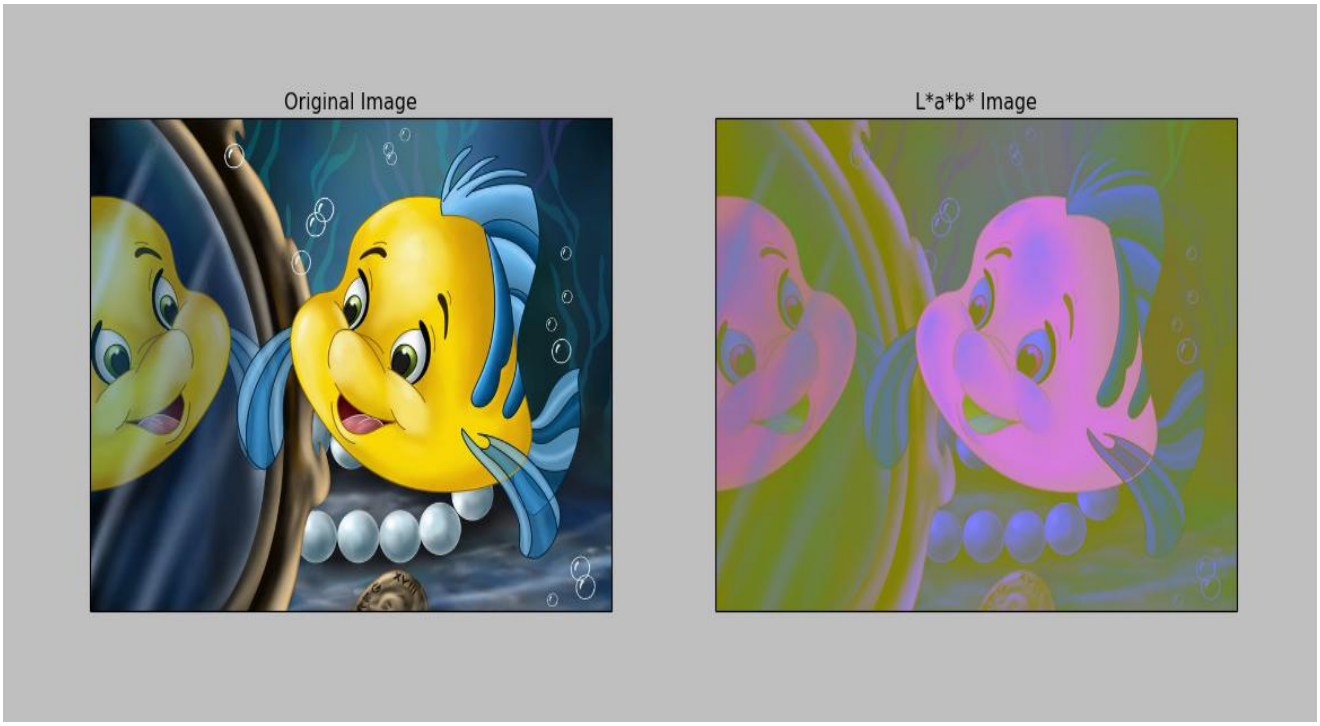
where

$$f(t) = \begin{cases} \sqrt[3]{t} & \text{if } t > \delta^3 \\ \frac{t}{3\delta^2} + \frac{4}{29} & \text{otherwise} \end{cases}$$
$$\delta = \frac{6}{29}$$

Here Xn, Yn and Zn are the CIE XYZ values of the reference white point.
The values are:

Xn = 95.047
Yn = 100.000
Zn = 108.883

## Output:



**Question 4: Convert Image into Grayscale using the default OpenCV function. Write the expressions used for the conversion.**

Read image using default function **cv2.imread ( image, cv2.IMREAD_COLOR)** then use default function **cv2.cvtColor()** with parameters as the "image" variable and flag as **"cv2.COLOR_BGR2GRAY"** to convert BGR image into RGB

**BGR to GRAY : Y<--- 0.114 * B + 0.587 * G + 0.299 * R**

**Output:**

**Question 5: Take a grayscale image and illustrate Whitening and Histogram equalization.**

The variable img contains our image. we go on and use some of the features OpenCV offers. Before we can apply a equalizer, we have to convert our original image from RGB to GRAY. Therefore we use the function cv2.cvtColor having flag value as **cv2.COLOR_RGB2GRAY** converts an image img into the given grayscale image. As we want a greyscale image we use**.**

OpenCV has a function to do this, **cv2.equilizeHist()**. Its input is just grayscale image and output is our histogram equalized image. Histogram equalization is good when histogram of the image is confined to a particular region. It will not work better when there is large intensity diffrences. In that case, histogram covers a large region, i.e. both bright and dark pixels are present.

For whitening, first compute the arithmetic mean along the specified axis using **np.mean(image)**.

It returns the average of the array elements. Compute the standard deviation along the specified axis using **np.std(image)**. Returns the standard deviation, a measure of the spread of a distribution, of the array elements.

Equation to find mean and standard deviation:

$$\mu = \frac{\sum_{i=1}^{I}\sum_{j=1}^{J} p_{ij}}{IJ}$$
$$\sigma^2 = \frac{\sum_{i=1}^{I}\sum_{j=1}^{J}(p_{ij}-\mu)^2}{IJ}.$$

These statistics are used to transform each pixel value separately so that,

$$x_{ij} = \frac{p_{ij}-\mu}{\sigma}.$$

To get whitening of image, from every pixel subtract mean and then divide by standard deviation. It will also give negative values so apply scaling to get pixel in range of 0 to 1. Here we get minimum from whole image and subtract it from whole image, so now all values is positioned. Then get a maximum from whole image and divide whole image by maximum value to restrict range from 0 to 1. After that plot image using matplotlib.

## Output:


GrayScale Image
Whitened Image
Histogram Equalized Image

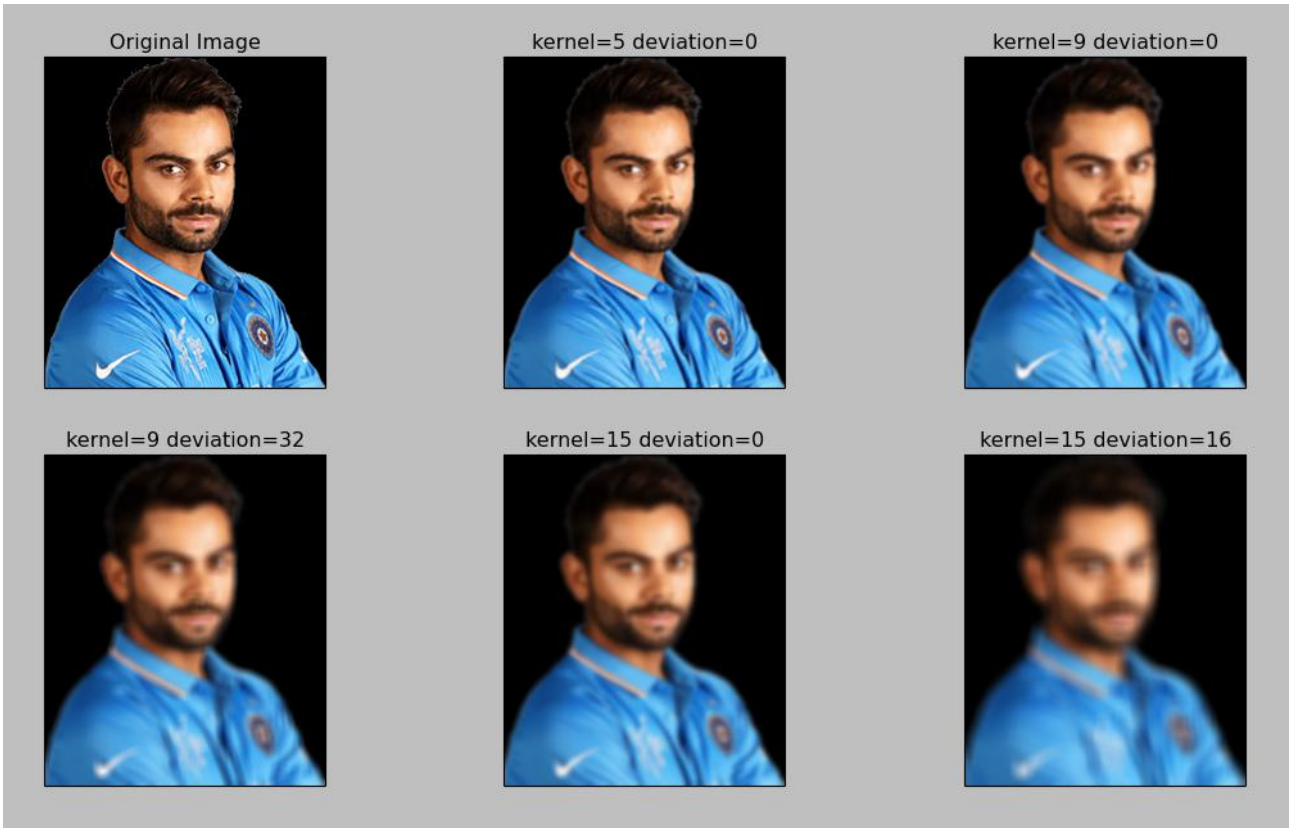
**Question 6: Take a low illumination noisy image and perform Gaussian smoothing at different scales. What do you observe w.r.t scale variation?**

First we read color image using cv2.imread() function with cv2.IMREAD_COLOR flag, then we will use cv2.GaussianBlur() built-in function to apply gaussian smoothing. This function takes image, kernel size, sigmaX and sigmaY as input parameter. We have to specify the width and height of kernel which must be positive and odd. It will apply gaussian filter on image with kernel size and signaX and sigmaY parameters. Here kernel size is used to calculate mean and other two are used to calculate standard deviation. If sigmaX and sigmaY are zeros then it will use kernel size to calculate standard deviation.

**Syntax:**
cv2.GaussianBlur(img, ksize, sigmaX, dst, sigmaY, cv2.BORDER_CONSTANT)

If you want, you can create a Gaussian kernel with the function,
**cv2.getGaussianKernel().**
As kernel size increses, bluring is also increses and same for sigmaX and sigmaY.

**Output:**



**Question 7: Take an image and add salt-and-pepper noise. Then perform median filtering to remove this noise.**

Here we will make our function to add salt and pepper noise. First we take input image and convert it to gray scale image, then we will add salt and pepper noise and then we will use median filtering to remove that noise. The function **cv2.medianBlur()** takes median of all the pixels under kernel area and central element is replaced with this median value. This is highly effective against salt-and-pepper noise in the images. So we used **cv2.medianBlur()** built-in function with kerel size 3, to get median blured image. Here if there is a pixel with salt or pepper noise in any grid of 3x3, then this pixel will be at extreme location of sorted sequence of 9 pixels, so it will no be taken as median and noise will not come in output image. Kernel size is of odd positive number.

**Salt and pepper noise function description :**

Here we take image, salt probability and pepper probability as input parameter. First we make output_image from coping input_image using **np.copy()** function. We make one matrix of size same as image named random_matrix and put random values between 0 and 1 using **np.random.rand()** function, here random values are distributed uniformally. Now to add pepper noise, whichever pixel of random_matrix have value less than pepper_probability, make them 0, so pepper noise will be added at those pixels. Similarly for salt noise, whichever pixel of random_matrix have value greater than 1-salt_probability, make them 255, so salt noise will be added at those pixels. Return output_image as noisy image.

**Output:**

**Question 8: Create binary synthetic images to illustrate the effect of Prewitt (both vertical and horizontal) plus sobel operators (both vertical and horizontal) Clue: check when you have a vertical/horizontal strip of white pixels – vary width of the strip from 1 pixel to 5 pixels. What do you observe?**

The **Sobel** Operator is a discrete differentiation operator which computes the gradient of an image intensity function.The Sobel Operator combines Gaussian smoothing and differentiation, so it is more resistant to noise. It calculates the first derivatives of the image separately for the X and Y axes. Sobel uses two 3X3 kernels to calculate approximations of the derivatives – one for horizontal change and another for vertical change.

The Prewitt operator functions similar to the Sobel operator, by computing the gradient for the image intensity function. As compared to Sobel, the Prewitt masks are simpler to implement but are very sensitive to noise. At each point in the image, the result of the Prewitt operator is the corresponding gradient vector. The Prewitt operator is based on convolving the image with a small integer valued filter in horizontal and vertical directions and is therefore relatively inexpensive in terms of computations.

The picture below shows prewitt and sobel mask in x-dir and y-dir:

| | Prewitt | Sobel |
|---|---|---|
| X derivative | $\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$ | $\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$ |
| Y derivative | $\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$ | $\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$ |

Here we have created some binary synthetic images, on which we can apply sobel and prewitt operator so that we can see effect of those operators. We use those operators to detect horizontal and vertical edges uisng their masks. Apply mask on binary image using in-built **cv2.filter2D()** function and plot images using matplotlib.

**Output:**

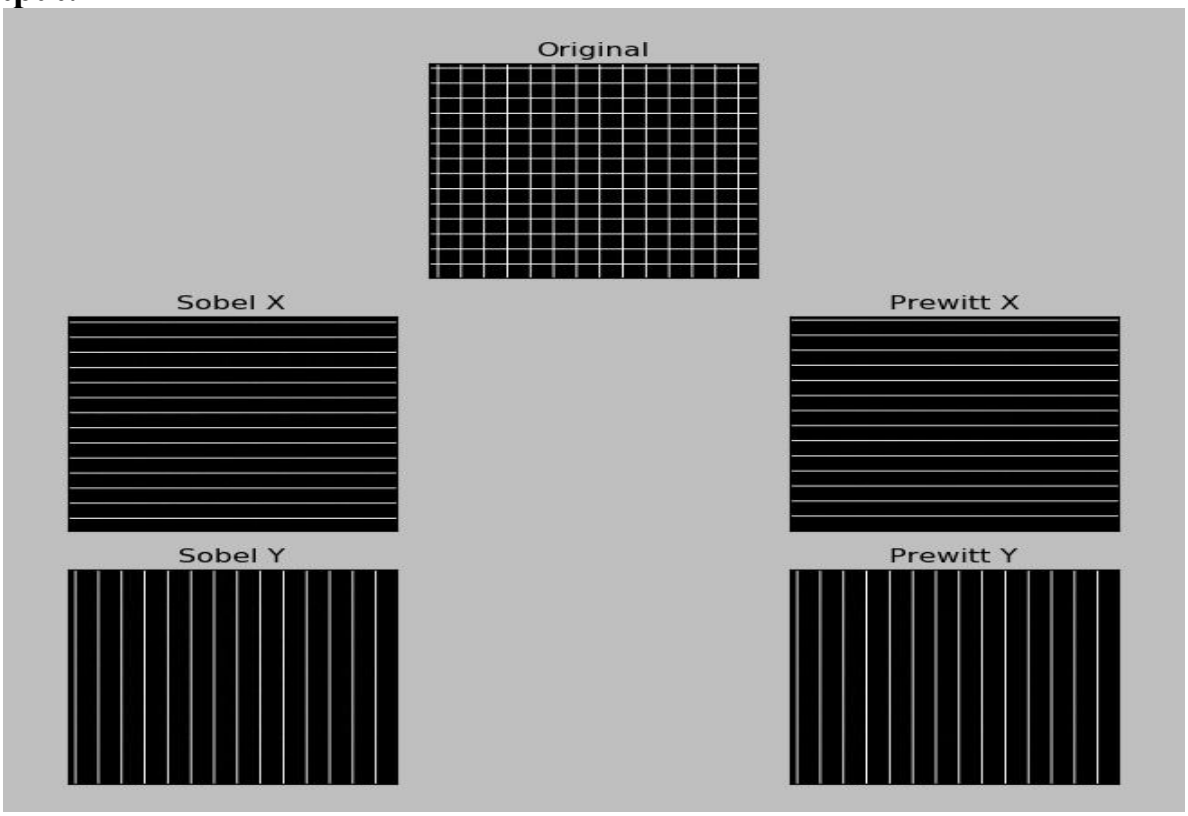



**Fig 1. Width of vertical/horizontal strip of white pixels is 1**

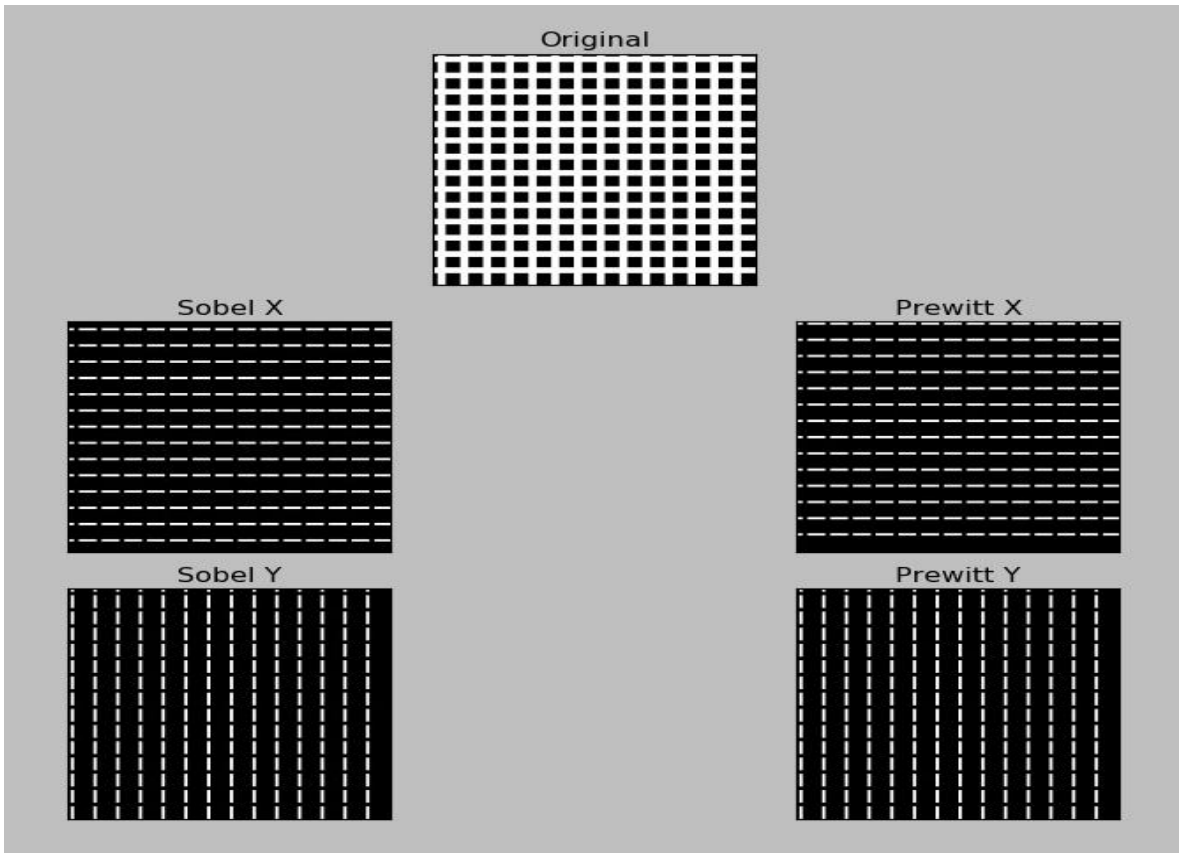



**Fig 2. Width of vertical/horizontal strip of white pixels is 3**

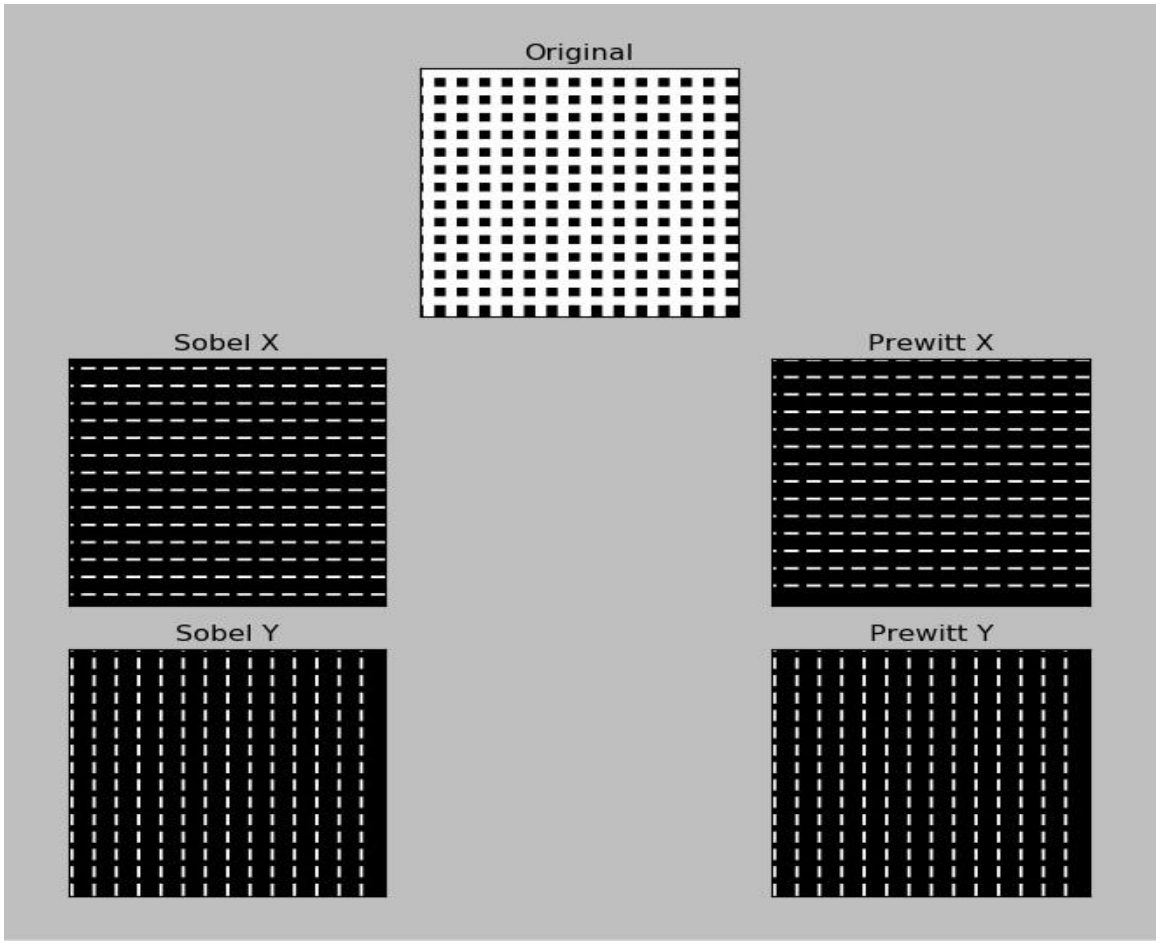



**Fig 3. Width of vertical/horizontal strip of white pixels is 5**

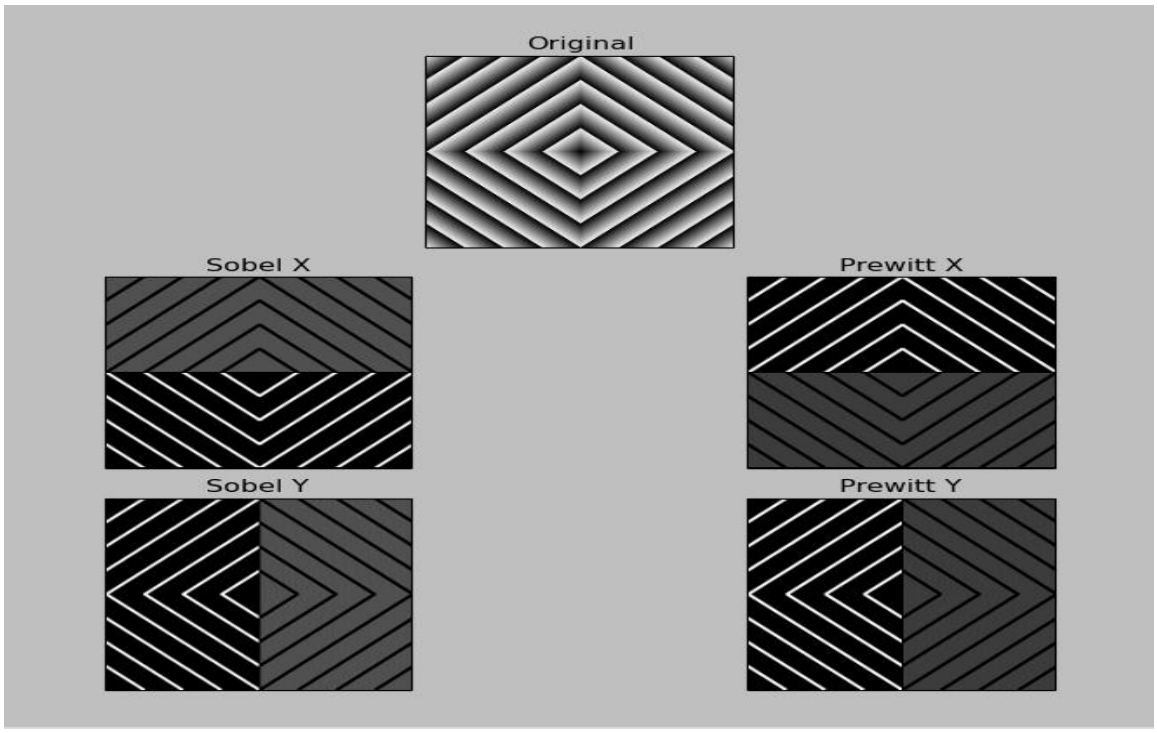



**Fig 4. binary image with prewitt and sobel operator**

**Question 9: What filter will you use to detect a strip of 45 degrees**

We are using Robert mask to detect strip of 45 degrees, as well as 135(or -45) degree. Then we will apply thresolding to get lines of this range only, because Robert mask gives lines with high intensity for nearer to 45 degree line and gradually lesser intensity as degree changes.

Define **robert mask** of size 2x2 as:

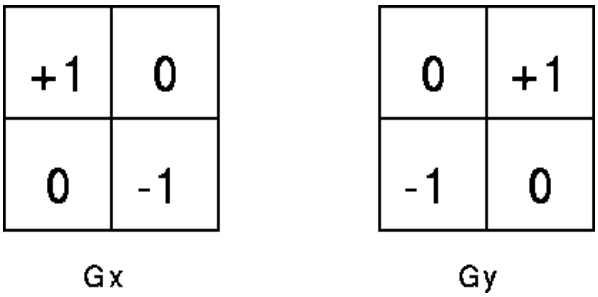



fig. Robert mask as 2x2 matrix

Apply robert mask on binary image using in-built **cv2.filter2D()** function.

Apply thresholding to get interested lines using **cv2.threshold()** function.

Thresholding is the simplest method of image segmentation. Thresholding is a non-linear operation that converts a gray-scale image into a binary image where levels are assigned based on specific threshold value to pixels. There are two levels: above threshold and below threshold. In other words, if pixel value is above threshold, it is assigned one value (may be white), else it is assigned another value (may be black). In OpenCV, we use **cv2.threshold()** function:

**Syntax:** cv2.threshold(src, threshold, maxval, type[, dst])

**src** - This is the source image, which should be a grayscale image.
**threshold** - Used to classify the pixel values.
**maxval** - If pixel value is more than the threshold value, this value will be given.
**type** – Type of threshold.
**dst** - Output array of the same size and type as src.

This function applies fixed-level threshold to a single-channel array. The function is typically used to get a binary image out of a grayscale image or for removing a noise. That is called filtering out pixels with too small or too large values. There are several types of threshold supported by the function. The function returns the computed threshold value and thresholded image.

**Output:**

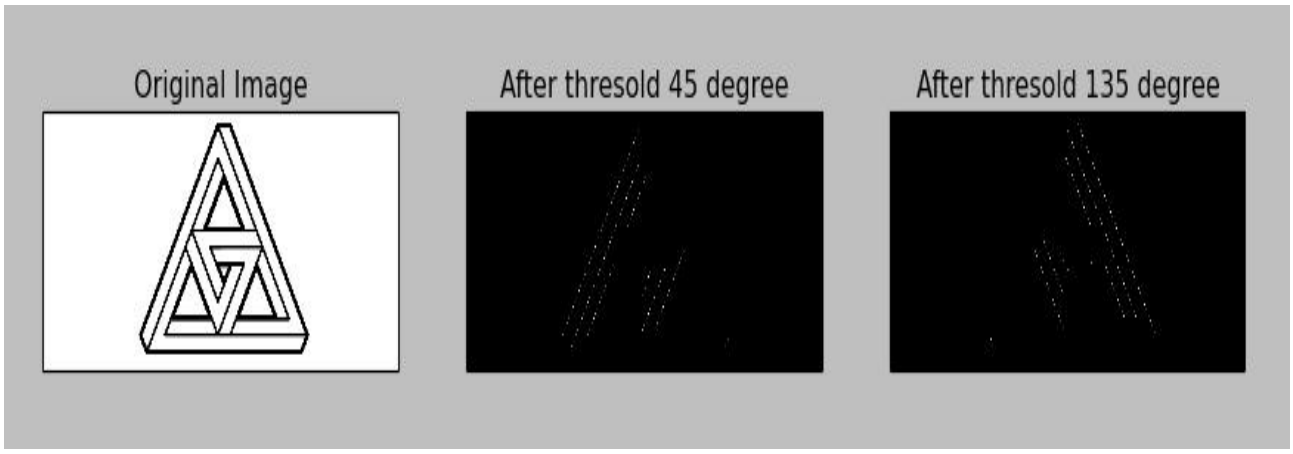



**fig 1. image with 45 degree edge detection**

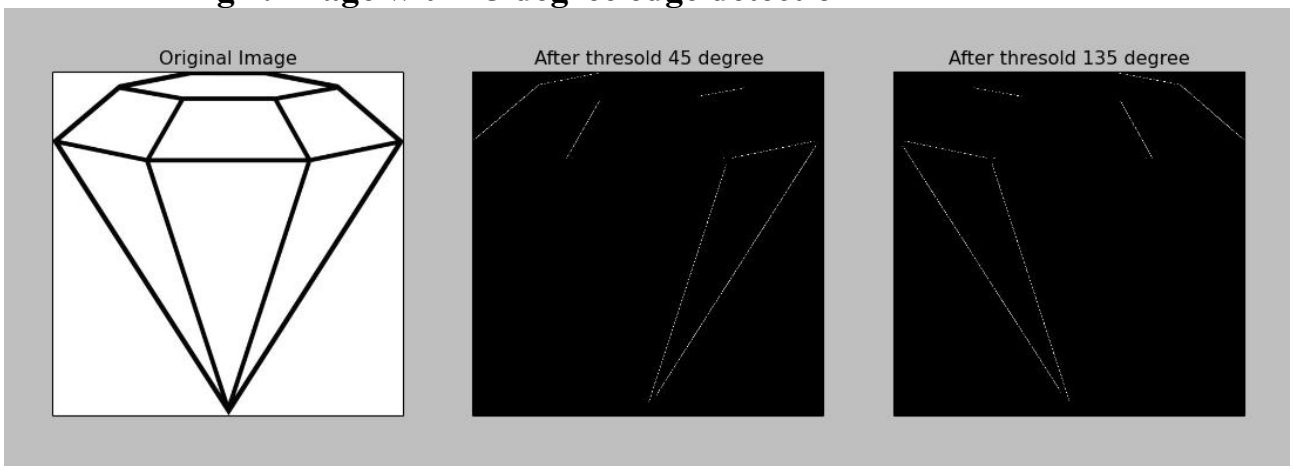



**fig 2. image with 45 degree edge detection**

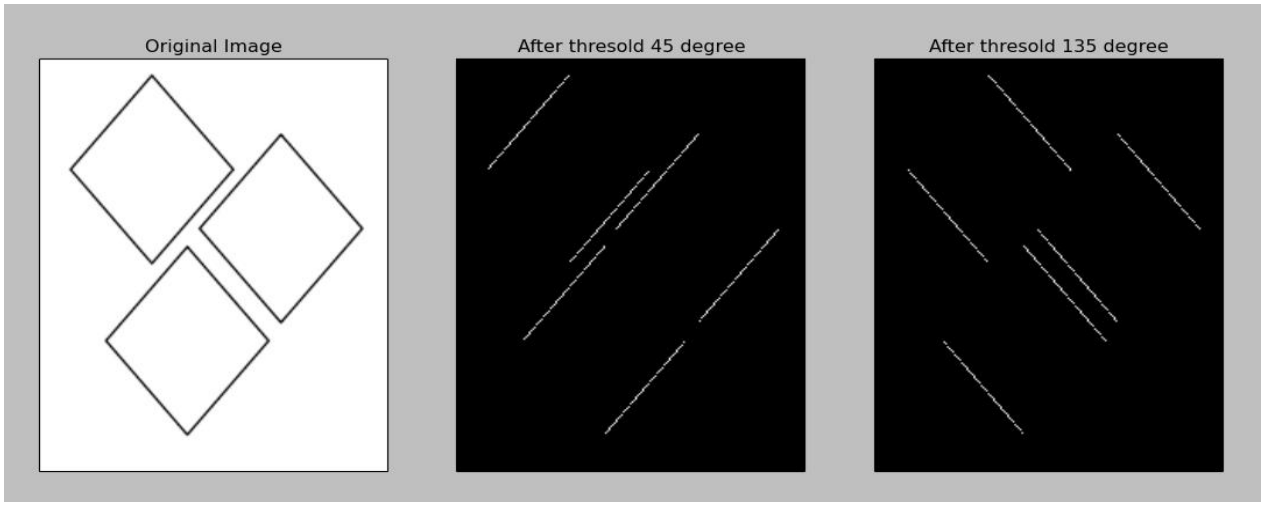



**fig 3. image with 45 degree edge detection**

**Question 10: Take an image and observe the effect of Laplacian filtering can you show edge sharpening using Laplacian edges**

Laplacian is 2nd order derivative based edge detector.It is extremely sensitive to noise. The second derivative of the intensity can be used to *detect edges*. Since images are "*2D*", we would need to take the derivative in both dimensions.

Kernel for laplacian operation is defined as follow :

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Here kernel size is 1.

We have applied laplacian mask to gray_image to get laplacian edges. Here we have applied cv2.filter2D() to do this. We have subtracted laplacian edges from gray_image to get sharp image. Now wherever we have got negative value of the pixel it is replaced with the gray value.

Laplacian image with detected edges and sharpened image is shown below.

**Output:**

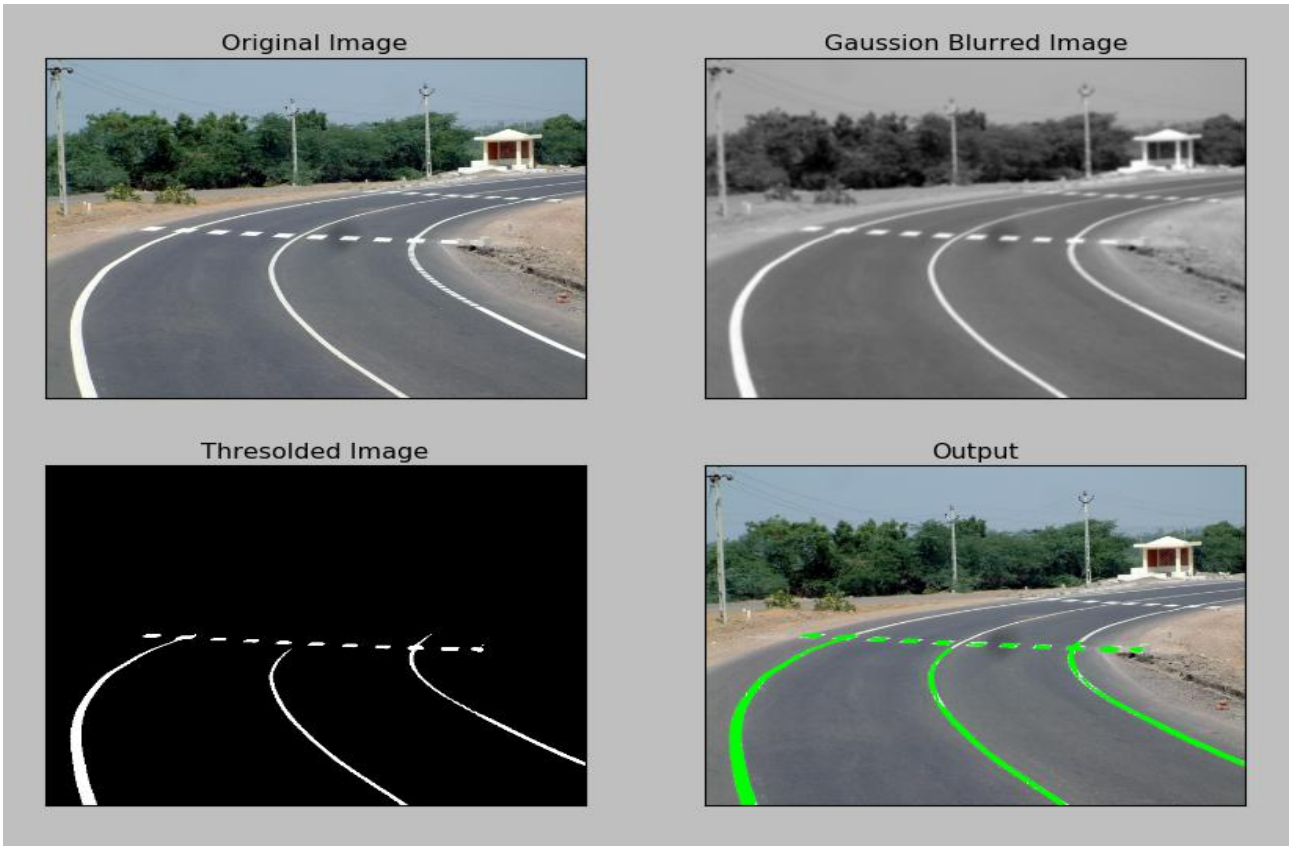
**Question 11: Detect Road land markers**

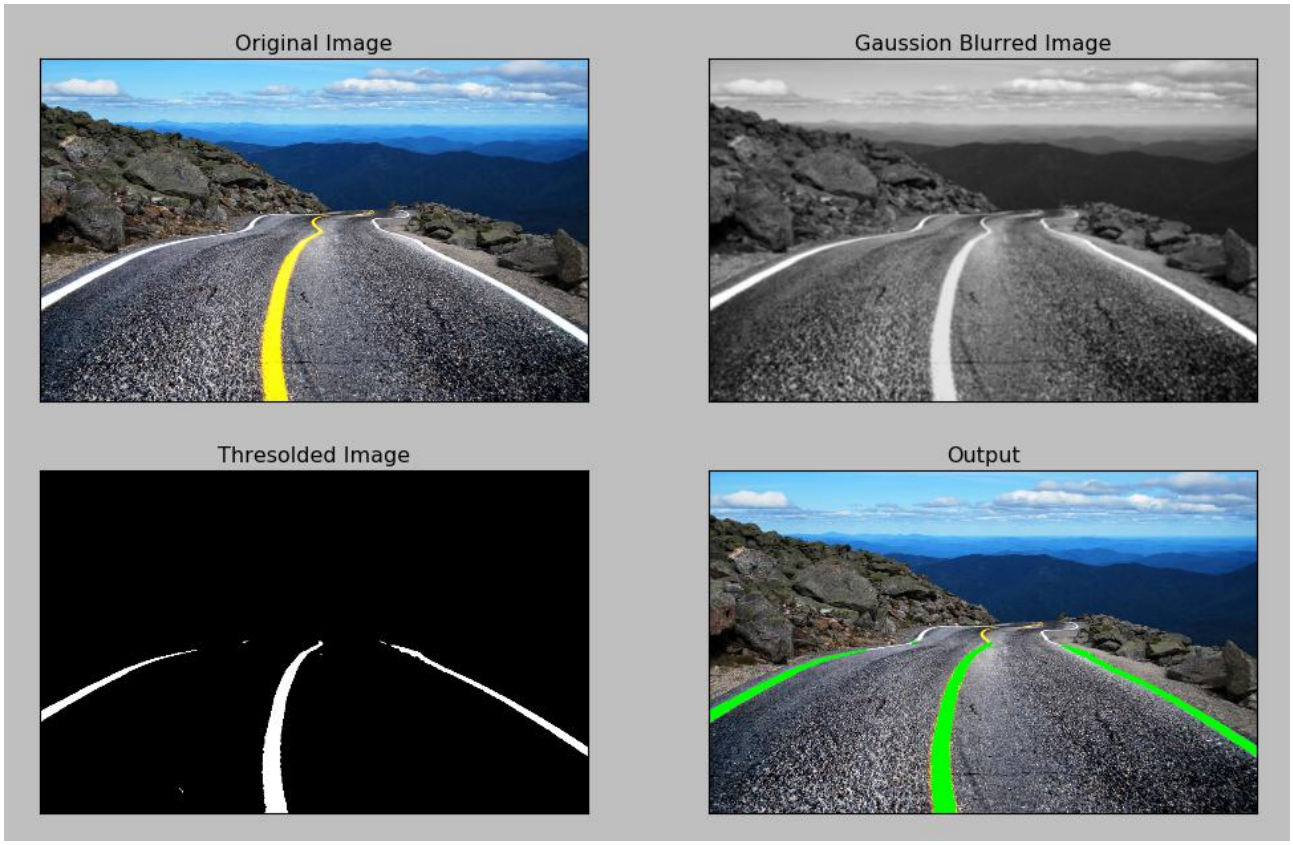
Here, we have to detect Road Lane markers in images depicting roads from first person's view. The first step we do here, is standardize the image size. We resize the image by 50% on both scale and then again check size if it is bigger than 1000x1000 resize it and so on. To resize, we use cv2.resize() function. Then we convert image to gray-scale image. We have to remove noise from image, for that we use gaussian blur of kernel size 7x7. In-built function **cv2.GaussianBlur()** was used for blurring.
In the image, only road marks are light colored(white or yellow), all other parts are dark, so if we apply binary thresholding with 180 as threshold value, we will get road marks, sky and some other noise.
Function **cv2.threshold()** was used for thresholding to remove sky, we ignore upper part of image for further image processing. We make upper third part of image black, by making all that pixels zero. Then we apply Hough Transformation to get lines. We used **cv2.HoughLinesP()** to get lines. By using **cv2.line()** we draw line on original color image which shows road marks.

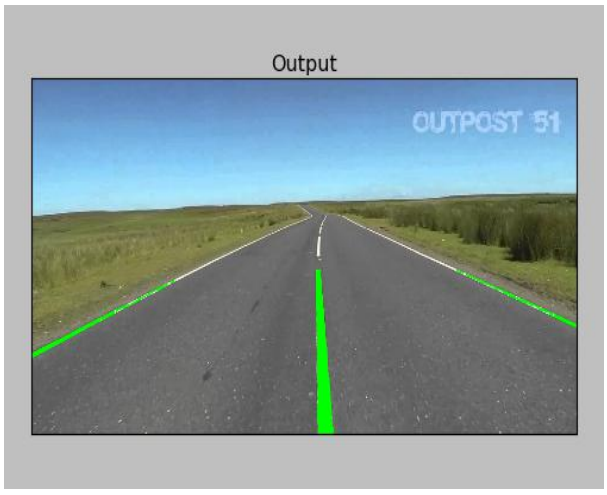
cv2.HoughLinesP has two parameters which are used for proper selection of markers.

1. **minLineLength** - Minimum length of line. Line segments shorter than this are rejected.
2. **maxLineGap** - Maximum allowed gap between line segments to treat them as single line.

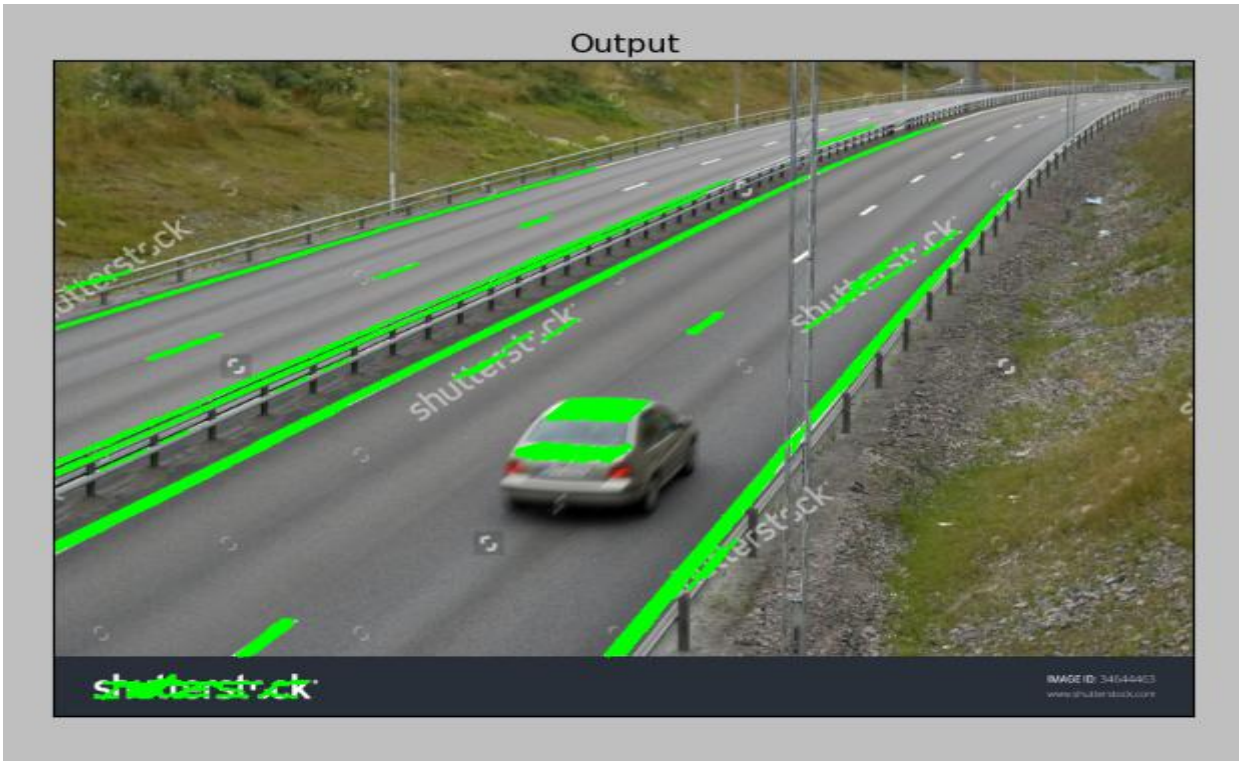
**Output:**

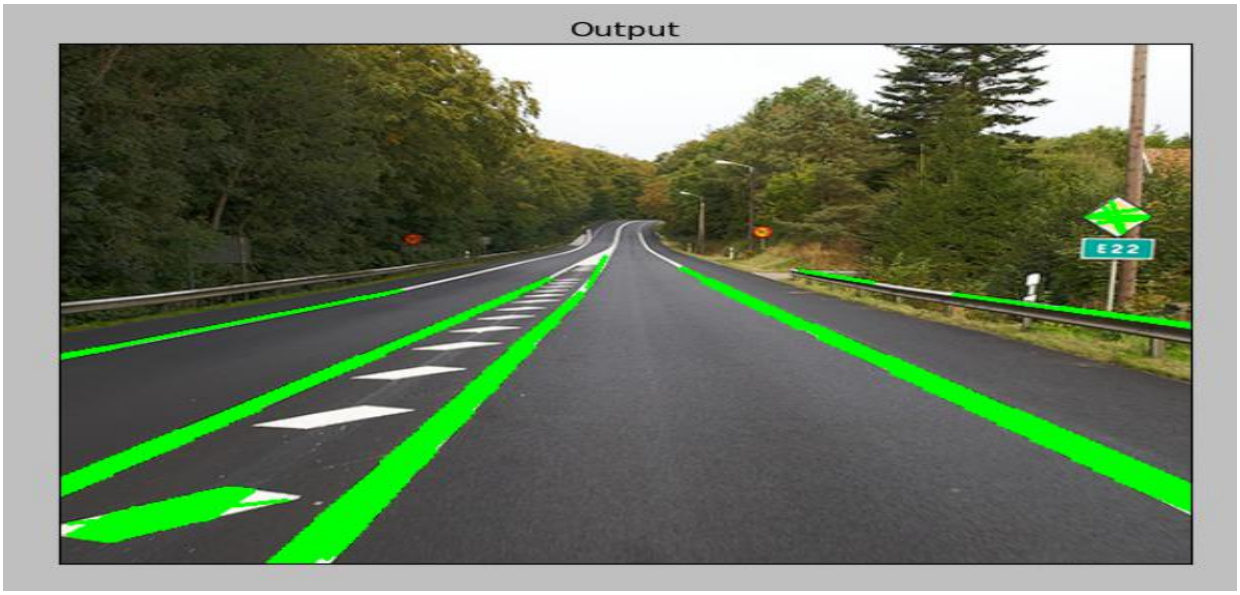


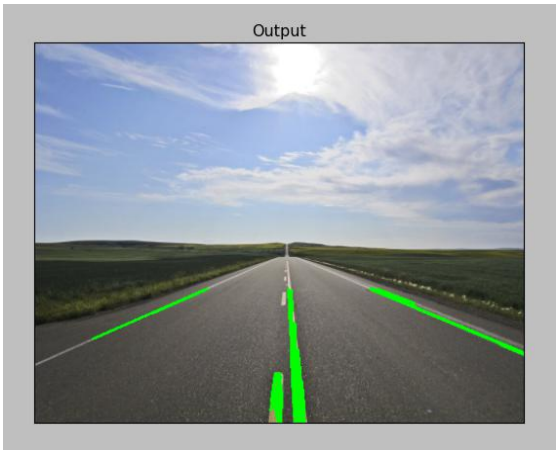

Original Image
Gaussion Blurred Image
Thresolded Image
Output

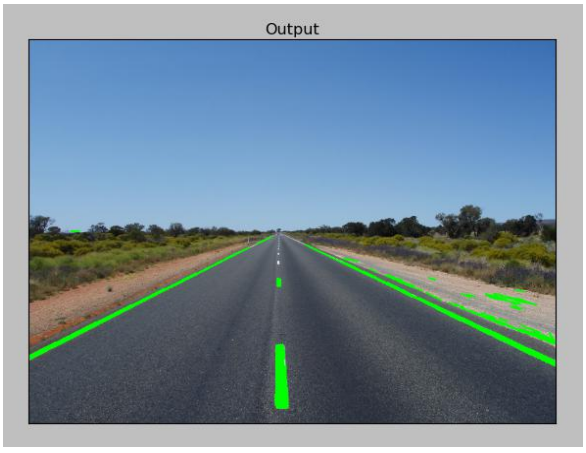

Output
OUTPOST 51


Output


Output


Output
E22


Output


Output

**Question 12: Classify modes: Night; Portrait; Landscape. Design features, use NN**

**Pseudocode:**

- Convert color image into gray image
- Define various haarcascade classifiers to detect face and eye
- Apply Harr-Cascade on the image to detect face features.
- If face or eye is detected, classify the image as a Portrait.
- If not, check if the image is a night image.We find a pixel as dark, using a threshold value 60. If more than half image is covered by dark pixels, then we classify the image as Night image.
- If not, then the image is Landscape.

First, we standardize the image resolution. If image is too big then we have to resize the image. We resize the image by 50% on both scale and then again check size if it is bigger than 700x700 resize it and so on. To resize, we use **cv2.resize()** function.

Then we convert image to gray-scale image. If image is portrait then it will contain one or more face, so if we try to detect face or eye in image and we detect something then we can say it is portrait image. To detect face or eye, we use haarcascade classifier. **cv2.CascadeClassifier()** function is used to define haarcascade classifier and **CascadeClassifier.detectMultiScale()** is used to apply it on image. If any one of nine classifier detects a feature, then it is portrait image, otherwise it is landscape or night image.

Now to check whether image is night or not, we will get count of very dark pixels, if number of dark pixels are greater than half of total pixels then it is night picture. For dark night pixels, values for Red,Green and Blue are less, and value channel of HSV gives maximum from Red,Green and Blue, so for any pixel if value is low then it is dark pixel. So to get count of dark pixels, we extract value plane from HSV image of original image and apply binary threshold with threshold = 60 using **cv2.threshold()** and then get the pixels where thresolded_image = 0 using **np.where()** and get count of these pixels.

If number of dark pixels are greater than half of total pixels then we classify it as a night picture, otherwise it is classified as landscape picture.

**Output:**

**Actual output: Ladscape**
**Expected output: Landscape**

**Actual output:Ladscape**
**Expected output: Landscape**

**Actual output: Ladscape**
**Expected output: Landscape**

**Actual output: Ladscape**
**Expected output: Landscape**

**Actual output: Portrait**
**Expected output: Portrait**

**Actual output: Portrait**
**Expected output: Portrait**

**Actual output: Portrait**
**Expected output: Portrait**

**Actual output: Portrait**
**Expected output: Portrait**

**Actual output: Night**
**Expected output: Night**

**Actual output: Night**
**Expected output: Night**

**Actual output: Night**
**Expected output: Night**

**Actual output:Landscape**
**Expected output: Night**