MACHINE PERCEPTION

Assignment 1

Submitted By:

Vivek Mehta(MT2016155)

Vijay Agarwal(MT2016152)

Swatantra Pradhan(MT2016140)

1. Choose an RGB image. Plot R,G,B separately (Write clear comments and observations)

We load a color image using cv2.imread() method of OpenCV. Color image loaded by OpenCV is in BGR mode by default. We then extract the individual channels. We then do a color conversion from BGR to RGB by using the function cv2.cvtColor(). If we don't switch BGR order, we get the RGB inverted picture.

Code:

```
# Import numpy and cv2 package
import cv2
import numpy as np

# Reading original image from the disk
img= cv2.imread('1.jpeg')
#Displaying the original image
cv2.imshow('RGB',img)

# Separate the red,blue and green channel image from the original
'img'
img_red=img[:,:,2]
img_green=img[:,:,1]
img_blue=img[:,:,0]

# Displaying the red,green and blue channel output image together
cv2.imshow('Problem1 Output.jpg',
np.concatenate((img_red,img_green,img_blue),axis=1))
# Display the images and wait till any key is pressed
cv2.waitKey(0)
# Destroy all the windows created by the imshow() function of the
OpenCV
cv2.destroyAllWindows()
```

Images :

Original/Input :

Output : (RED Channel, GREEN Channel, BLUE Channel)

2. Choose an RGB image. Plot R,G,B separately (Write clear comments and observations)

Similar to previous question, to get HSV channels we use "BGR2HSV" flag in cv2.cvtColor(input_image, flag). Then we used 3 matrices to keep the hue , saturation and variance images separate. Also we could used here an inbuilt function split() to separate three channels: Hue, Saturation, Value/Lightness (for HSL) , but the split() function is costly (in terms of time).

Expression to convert RGB to HSV/I:
The R, G, B values are divided by 255 to change the range from 0..255 to 0..1.

R' = R/ 255
G' = G/255
B' = B/255
Δ= Max(R',G',B') – Min(R',G',B')

Hue:

$$H = \begin{cases} 0^\circ & \Delta = 0 \\ 60^\circ \times \left(\frac{G'-B'}{\Delta} mod 6\right) & , C_{max} = R' \\ 60^\circ \times \left(\frac{B'-R'}{\Delta} + 2\right) & , C_{max} = G' \\ 60^\circ \times \left(\frac{R'-G'}{\Delta} + 4\right) & , C_{max} = B' \end{cases}$$

Saturation:

$$S = \begin{cases} 0 & , C_{max} = 0 \\ \frac{\Delta}{C_{max}} & , C_{max} \neq 0 \end{cases}$$

Value :

$$V = Cmax$$

Code:

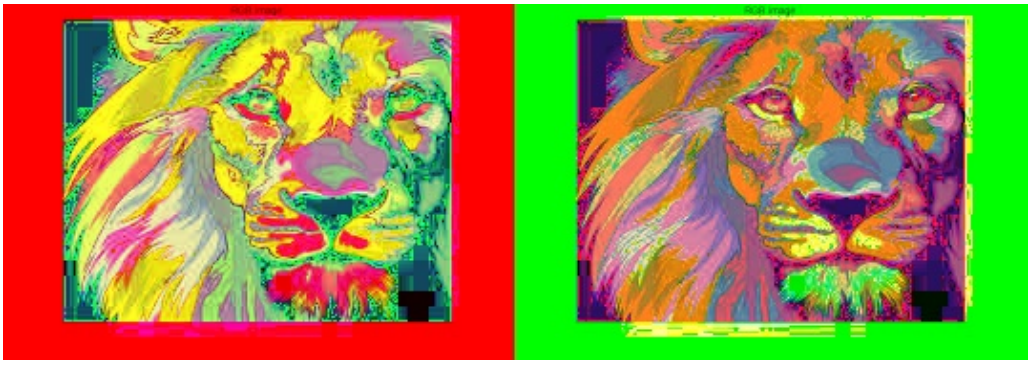
```
# Import cv2 package
import cv2


# Reading original image from the disk
img= cv2.imread('1.jpeg')
#Displaying the original image
cv2.imshow('RGB',img)
#Converting the original image from BGR to HSV and HLS using OpenCv
inbuilt function
img_hsv=cv2.cvtColor(img,cv2.COLOR_BGR2HSV)
img_hls=cv2.cvtColor(img,cv2.COLOR_BGR2HLS)

#Extracting out the variance, saturation, Hue from img_hsv image
variance=img_hsv[:,:,2]
saturation=img_hsv[:,:,1]
hue=img_hsv[:,:,0]

#Displaying the HSV and HLS image
cv2.imshow('HSV',img_hsv)
cv2.imshow('HLS',img_hls)

#Displaying separately the Hue ,Saturation and Variance Channel images
cv2.imshow('Hue',hue)
cv2.imshow('Saturation',saturation)
cv2.imshow('Variance', variance)

# Display the images and wait till any key is pressed
cv2.waitKey(0)
# Destroy all the windows created by the imshow() function of the
OpenCV
cv2.destroyAllWindows()
```

Images:

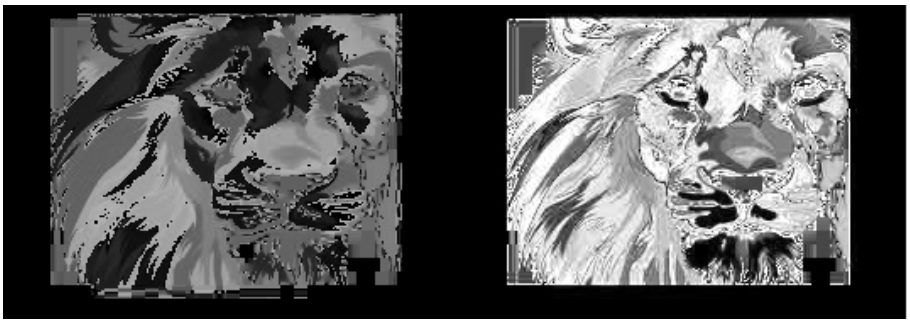
Original/Input :

Output :

(HSV and HLS)



(H channel, S channel and V channel)



3. Convert Image into L*a*b* and plot.

We load a color image using cv2.imread() method of OpenCV. And then we use the inbuilt cv2.cvtColor() to covert from BGR to LB. Cie-L*a*b is defined by lightness and the color-opponent dimensions a and b, which are based on the compressed Xyz color space coordinates. Lab is particularly notable for it's use in delta-e calculations. Color space defined by the CIE, is based on one channel for Luminance(lightness) (L) and two color channels (a and b).

Code:

```
# Import cv2 package
import cv2

# Reading original image from the disk
img= cv2.imread('1.jpeg')
#Displaying the original image
cv2.imshow('RGB',img)
#Converting the original image from BGR to L*a*b* using OpenCv inbuilt
function
img_lab=cv2.cvtColor(img,cv2.COLOR_BGR2Lab)

#Displaying the L*a*b* image using OpenCv inbuilt function
cv2.imshow('L*a*b*', img_lab)
# Display the images and wait till any key is pressed
cv2.waitKey(0)
# Destroy all the windows created by the imshow() function of the
OpenCV
cv2.destroyAllWindows()
```

Images:

Original/Input :

Output :

4. Convert an Image into Grayscale using the default OpenCv function. Write the expressions used for the conversion.

There are two most common ways to do this:

1. Using cv2.cvtColor() method with flag: "BGR2GRAY"
2. Using flag '0' while loading image via cv2.imread()

We used the 1st method, the default opencv function cv2.cvtColor() with flag : "BGR2GRAY".

Steps :

Load an image using cv2.imread().
Transform the image from BGR to Grayscale format by using cv2.cvtColor().

Save your transformed image in a file on disk (using cv2.imwrite()).

The expression used for converting RGB image to Grayscale image is :

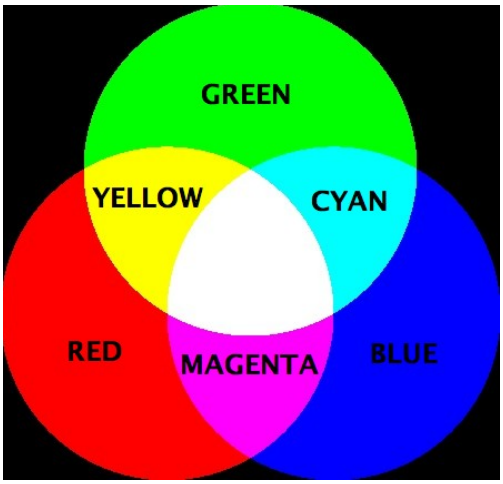
$$Value = 0.3*R + 0.59*G + 0.11*B$$

Code :

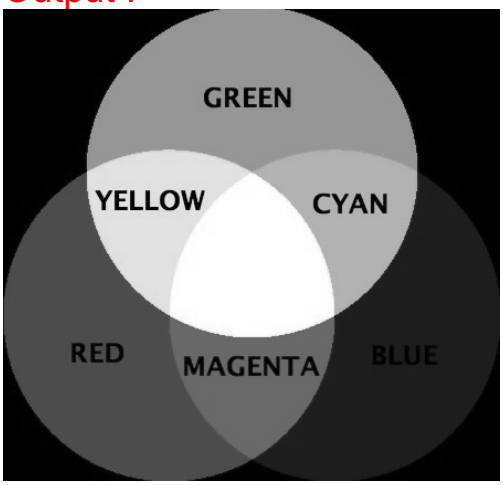
```
# Import cv2 package
import cv2

# Reading original image from the disk
image= cv2.imread('rgb.jpg')
#Displaying the original image
cv2.imshow('RGB', image)

# Converting the image from BGR to Grayscale using the OpenCv inbuilt
function
gray_img= cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
#Displaying the Grayscale image
cv2.imshow('Gray', gray_img)
# Display the images and wait till any key is pressed
cv2.waitKey(0)
# Destroy all the windows created by the imshow() function of the
OpenCV
cv2.destroyAllWindows()
```

Images:

Original/Input :



Output :



5. Take a Grayscale Image and illustrate a) Whitening b) Histogram equalization.

a) Whitening

For Whitening we are using Mean Normalization for which we require mean, standard deviation of the intensities of the pixels of the image. This process is used to brighten the image. We try to distribute the intensities of the pixels uniformly throughout the image.

$$\mu = \frac{\sum_{i=1}^{I}\sum_{j=1}^{J} p_{ij}}{IJ}$$

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

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

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

Steps:

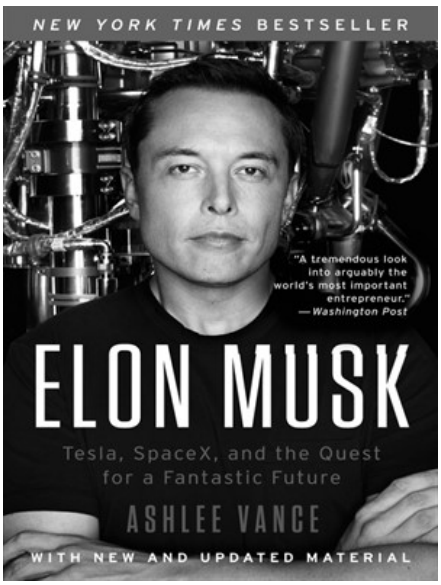
Read the image using cv2.imread().
Convert the image into GRAY SCALE using default opencv function cv2.cvtColor.
Use the variables mean, sd to keep the mean and standard deviation, calculated using function cv2.meanStdDev(), which takes the image as argument.
Img1 is a matrix used to keep each value separately.

Code: (Whitening)

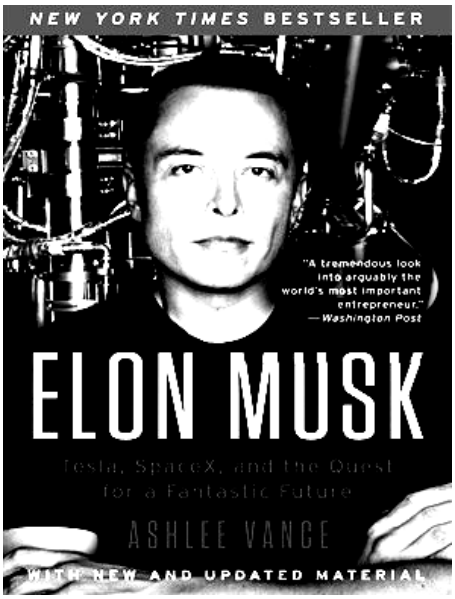
```
# Import numpy and cv2 package
import cv2
# Reading original image from the disk
image= cv2.imread('emusk.png')

# Converting the image from BGR to Grayscale
img= cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
#Displaying the Grayscale image
cv2.imshow('Gray',img)
# Calculating the mean and standard deviation of the grayscale image
mean, sd = cv2.meanStdDev(img)
# Creating the new image img1 by subtracting mean and then dividing by
standard deviation from the original image
img1= (img- mean)/sd

# # Displaying the final output image
cv2.imshow('Whitening',img1)

# Display the images and wait till any key is pressed
cv2.waitKey(0)
# Destroy all the windows created by the imshow() function of the
OpenCV
cv2.destroyAllWindows()
```

Images:

Original/Input :



Output :



b) Histogram Equalization

It is a method that improves the contrast in an image, in order to stretch out the intensity range.

To make it clearer, from the image above, you can see that the pixels seem clustered around the middle of the available range of intensities. What Histogram Equalization does is to stretch out this range.

Equalization implies *mapping* one distribution (the given histogram) to another distribution (a wider and more uniform distribution of intensity values) so the intensity values are spreaded over the whole range.

To accomplish the equalization effect, the remapping should be the cumulative distribution function (cdf) (more details, refer to Learning OpenCV). For the histogram $H(i)$, its cumulative distribution $H'(i)$ is:

$$H'(i) = \sum_{0 \le j < i} H(j)$$

To use this as a remapping function, we have to normalize $H'(i)$ such that the maximum value is 255 (or the maximum value for the intensity of the image)

Finally, we use a simple remapping procedure to obtain the intensity values of the equalized image:

$$equalized(x,y) = H'(src(x,y))$$

OpenCV library also provides inbuilt functions for Histogram Equalization.

We can apply histogram equalization on all the three channels of an RGB image and a lot improvement can be observed in the color distribution of the image.

We retrieved the B,G and R channels of the image separately and applied Histogram Equalization on the channel using the cv2.equalizeHist() function of the OpenCV library and then combined them together to get the histogramEqualized image as shown:

Code: (Histogram equalization)

```
# Import numpy and cv2 package
import cv2
import numpy as np

# Reading original image from the disk
temp=cv2.imread('valley.jpg')
# Converting the image from BGR to Grayscale using the OpenCv inbuilt
function
img= cv2.cvtColor(temp, cv2.COLOR_BGR2GRAY)
#Displaying the Grayscale image
cv2.imshow('Gray Scale Image', img)
#Getting the size of the image in row and col variables(no of rows and
no of columns)
row,col=img.shape

##Equalization by calculating PDF and CDF explicitly without using
inbuilt function

#cv2.calcHist(images, channels, mask, histSize, ranges[, hist[,
accumulate]])
hist= cv2.calcHist([img],[0],None,[256],[0,256])
hist1= hist/(row*col) # Probability of each pixel intensity(PDF of
pixel intesity)
cdf= hist1.cumsum() ## cummulative sum of probabilty distributive
function

#Normalizing the CDF calculated above
cdf_normalized = cdf * hist.max()/ cdf.max()
#Masking the cdf calculated above
cdf_m=np.ma.masked_equal(cdf,0)
cdf_m=(cdf_m - cdf_m.min())*255/(cdf_m.max()-cdf_m.min())
cdf=np.ma.filled(cdf_m,0).astype('uint8')

# Contructing histogram equalized image
hist_e=cdf[img]
#Displaying the Histogram Equalized Image
cv2.imshow('Histogram Equalized Image', hist_e)


#Equalization using CV inbuilt function for histogram equalization
hist_equalized= cv2.equalizeHist(img)
```

```
## Displaying all the output images together
img1=np.concatenate((img, hist_e,hist_equalized), axis=1)
cv2.imshow('Problem5b Output', img1)

# Display the images and wait till any key is pressed
cv2.waitKey(0)
# Destroy all the windows created by the imshow() function of the
OpenCV
cv2.destroyAllWindows()
```

Images:

Original/Input and Output :

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

Steps :
Here we first read a grayscale image image from the disk.
Then we used the inbuilt function cv2.GaussianBlur() provided in the OpenCV library to filter the noise.
The prototype of the function is as follows:
cv2.GaussianBlur(src,ksize,sigmaX,sigmaY)
where;

- src : is the source image,
- ksize : is the size of the kernel ,
- sigmaX : is the sigma for X direction and
- sigmaY : is the sigma value for Y direction.

By varying the last three parameters, we can change the amount of blurring.
Then we used the concatenate() function, it concatenates the image and display together.
Then the imshow() function to show the image on window.

Code:

```
# Import numpy and cv2 package
import cv2
import numpy as np

# Reading original image from the disk as grayscale image
img= cv2.imread('low_illum.jpg',0)

#Smoothening the image using different kernel size and sigmaX & sigmaY
G_smooth1= cv2.GaussianBlur(src=img, ksize=(3,3),sigmaX=3,sigmaY=3)

G_smooth2= cv2.GaussianBlur(src=img, ksize=(3,3),sigmaX=10,sigmaY=10)

G_smooth3= cv2.GaussianBlur(src=img, ksize=(5,5),sigmaX=3,sigmaY=3)

# Displaying the original image and all the output images using
different kernel and sigmaX & sigmaY, together
img1= np.concatenate((img,G_smooth1,G_smooth2,G_smooth3), axis=1)
```

```
cv2.imshow('Problem6_Output',img1)

# Display the images and wait till any key is pressed
cv2.waitKey(0)
# Destroy all the windows created by the imshow() function of the
OpenCV
cv2.destroyAllWindows()
```

Images :

Original/Input :

Output :

7. Take an Image and add salt-and-pepper noise. Then perform median filtering to remove this noise.

Salt and pepper noise:
contains random occurrences of black and white pixels . We create an array of same size as image, assign it random values in a specific range.Then, for some threshold value since salt and pepper has just 2 values i.e. 255 and 0 we add noise to original image by changing pixel value based on threshold.

To remove salt and pepper noise we cannot use Gaussian smoothing and thus apply median filtering.
Here, 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. Interesting thing is that, in the above filters, central element is a newly calculated value which may be a pixel value in the image or a new value. But in median blurring, central element is always replaced by some pixel value in the image. It reduces the noise effectively. Its kernel size should be a positive odd integer.

Code:

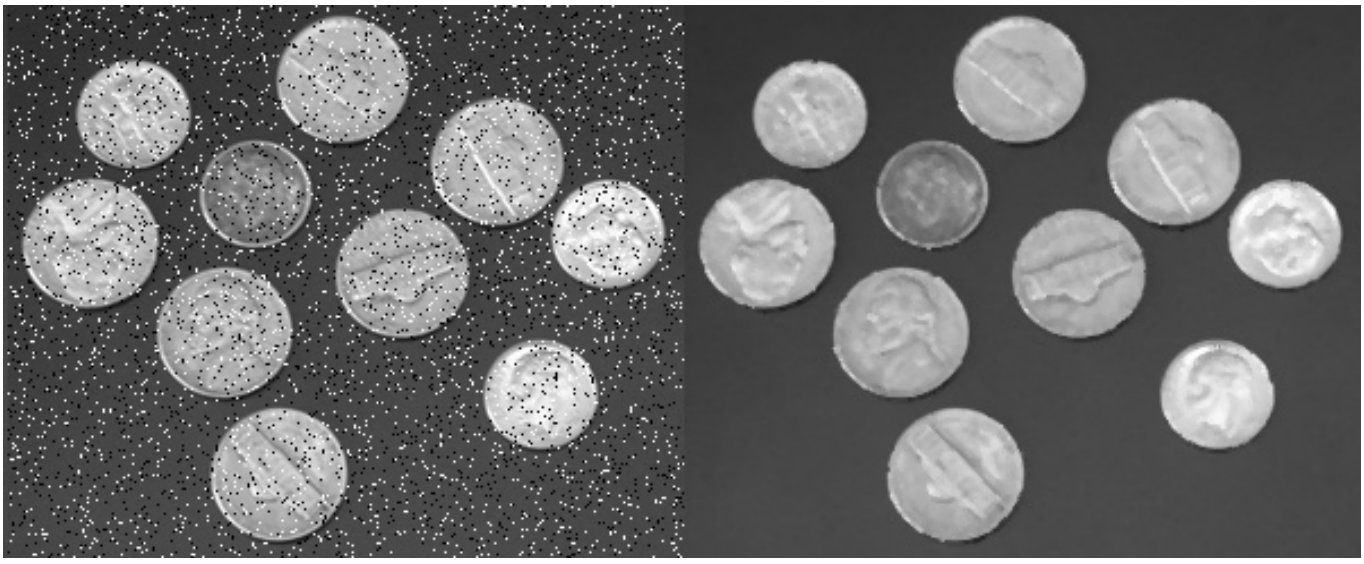
```
# Import numpy and cv2 package
import cv2
import numpy as np

# Reading original image from the disk as grayscale image
img = cv2.imread('Coins_gray.png',0)
#Displaying the image
cv2.imshow('Original',img)

# creating noise matrix
row,col= img.shape
noise= np.random.randn(row,col)*70
noise= noise.astype(dtype=np.int8)

# adding Salt and Pepper noise to the image
salt_pepper_noise= img[:,:]
salt_pepper_noise[noise>110]=255
salt_pepper_noise[noise<-110]=0
```

```
#Displaying the image with salt an pepper noise
cv2.imshow('Salt_pepper_noise', salt_pepper_noise)


# Median filtering applied to image having salt and pepper noise(it's
the most efeective one in case of salt n pepper)
median= cv2.medianBlur(salt_pepper_noise, 3)
cv2.imshow('Median_Smoothening', median)


# Display the images and wait till any key is pressed
cv2.waitKey(0)
# Destroy all the windows created by the imshow() function of the
OpenCV
cv2.destroyAllWindows()
```

Images:

Original/Input :

Output : (Salt & Paper Noise, Median Filtering)



8. Create binary synthetic images to illustrate the effect of Prewitt (both vertical and horizontal) plus sobel operators (both vertical and horizontal)? What do you observe?

The Sobel Operator is a discrete differentiation operator. It computes an approximation of the
gradient of an image intensity function.

The Sobel Operator combines Gaussian smoothing and differentiation.

a). Horizontal changes: This is computed by convolving with a kernel example for a kernel size of 3,would be computed as:

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * I$$

b). Vertical changes: This is computed by convolving with a kernel example for a kernel size of 3,would be computed as:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * I$$

At each point of the image we calculate an approximation of the gradient in that point by combining both results above:

$$G = \sqrt{G_x^2 + G_y^2}$$

Although sometimes the following simpler equation is used:

$$G = |G_x| + |G_y|$$

The Prewitt operator calculates the gradient of the image intensity at each point, giving the direction of the largest possible increase from light to dark and the rate of change in that direction. The result
therefore shows how "abruptly" or "smoothly" the image changes at that point, and therefore how likely it is that part of the image represents an edge, as well as how that edge is likely to be oriented.

$$\mathbf{G_x} = \begin{bmatrix} -1 & 0 & +1 \\ -1 & 0 & +1 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G_y} = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ +1 & +1 & +1 \end{bmatrix} * \mathbf{A}$$

Just like the Sobel operator, G x and G y give the horizontal and the vertical changes respectively.
At each point of the image we calculate an approximation of the gradient in that point by combining both results above:

$$G = \sqrt{G_x^2 + G_y^2}$$

Code:

```
# Import numpy and cv2 package
import cv2
import numpy as np

# Creating a binary image
img= np.zeros((250,250), dtype=np.uint8)
subimg= img[75:175, 75:175]
subimg[:,:]= 255
#Dispalying the image
cv2.imshow('Original',img)

## Applying the SOBEL OPERATOR

# If our Output datatype is cv2.CV_8U or np.uint8 then there is a
slight problem with that.
# Black-to-White transition is taken as Positive slope (it has a
positive value)
# while White-to-Black transition is taken as a Negative slope (It has
```

```
negative value).
# So when you convert data to np.uint8, all negative slopes are made
zero, in that case we miss that edge.

# If we want to detect both edges, better option is to keep the output
datatype to some
# higher forms, like cv2.CV_16S, cv2.CV_64F etc, take its absolute
value and then convert back to cv2.CV_8U.

# soblex8u= cv2.Sobel(img, cv2.CV_8U,1,0,ksize=5)
sobelx64f= cv2.Sobel(img, cv2.CV_64F,1,0,ksize=5)
# cv2.imshow('Sobelx8U',soblex8u)
abs_sobel64f= np.absolute(sobelx64f)
sobelx_8u = np.uint8(abs_sobel64f)

sobely64f= cv2.Sobel(img, cv2.CV_64F,0,1,ksize=5)
# cv2.imshow('Sobelx8U',soblex8u)
abs_sobel64f= np.absolute(sobely64f)
sobely_8u = np.uint8(abs_sobel64f)

img1= np.concatenate((img, sobelx64f, sobely64f), axis=1)
img2= np.concatenate((img, sobelx_8u, sobely_8u), axis=1)


cv2.imwrite('Problem8 Output Detecting Single edges.jpg', img1)
cv2.imwrite('Problem8 Output Detecting Both edges.jpg', img2)


## Applying the PREWITT filter

kernely= np.array([[1,1,1],[0,0,0],[-1,-1,-1]])
kernely= kernely.astype(dtype=np.int8)
prewitt_y= cv2.filter2D(img,-1,kernely)
cv2.imshow('prewitty',prewitt_y)

kernelx= np.array([[1,0,-1],[1,0,-1],[1,0,-1]])
kernelx= kernelx.astype(dtype=np.int8)
prewitt_x= cv2.filter2D(img,-1,kernelx)
cv2.imshow('prewittx',prewitt_x)

# Display the images and wait till any key is pressed
cv2.waitKey(0)
# Destroy all the windows created by the imshow() function of the
OpenCV
cv2.destroyAllWindows()
```

Images:

Original/Input and Output: (Detecting both Edges)

(Detecting Single Edges)

9. What filter will you use to detect a strip of 45 degrees?

We synthesized a binary image with random horizontal,vertical and slanted lines, to detect 45 degrees we use the kernel:

$$\text{Kernel} = \begin{matrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{matrix}$$

This detects +45 degrees inclined lines and ignores others.
We can use cv2.filter2d() available in the OpenCV library to perform convolution between the image and the kernel.

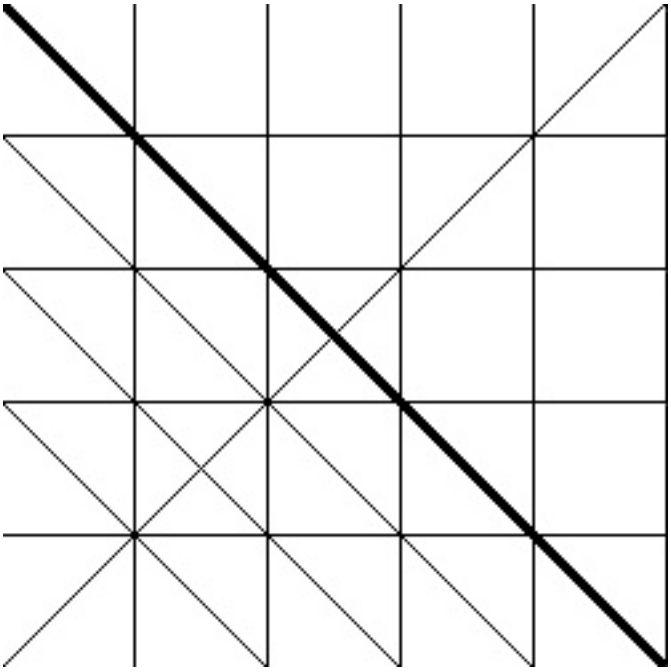
Code:

```
# Import numpy and cv2 package
import cv2
import numpy as np

#create a white image(square box) of dimension 250X250
img= np.ones((250,250), dtype=np.uint8)*255

#Make square boxes of 50X50
cv2.line(img,(0,50),(250,50),0,1)
cv2.line(img,(0,100),(250,100),0,1)
cv2.line(img,(0,150),(250,150),0,1)
cv2.line(img,(0,200),(250,200),0,1)

cv2.line(img,(50,0),(50,250),0,1)
cv2.line(img,(100,0),(100,250),0,1)
cv2.line(img,(150,0),(150,250),0,1)
cv2.line(img,(200,0),(200,250),0,1)

#draw lines of 45 degrees on lower half of square
cv2.line(img,(0,0),(250,250),0,1)
cv2.line(img,(0,50),(200,250),0,1)
cv2.line(img,(0,100),(150,250),0,1)
cv2.line(img,(0,150),(100,250),0,1)

#Display the original image
cv2.imshow('Image',img)

# kernel we are using to detect 45 degree lines
kernel = np.array([[2,-1,-1],[-1,2,-1],[-1,-1,2]])/12
```

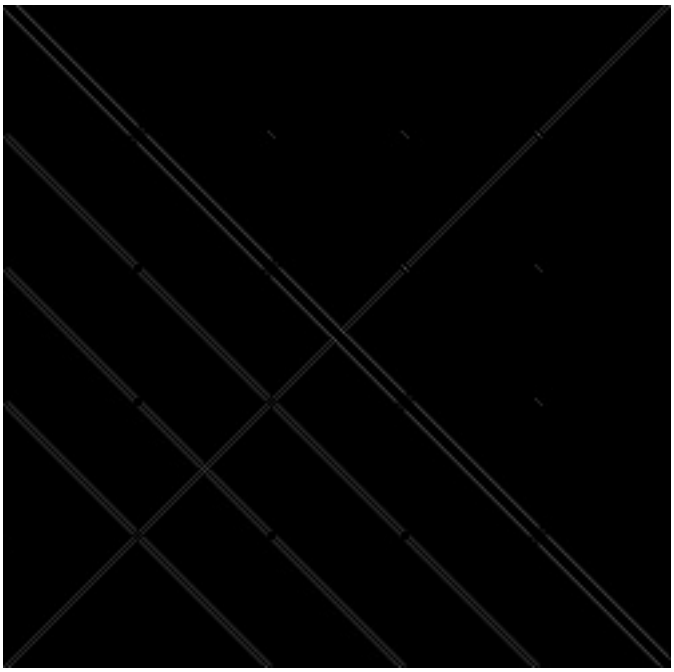
```
# Convolving the image with the kernel using the filter2d() function
of the OpenCV library
_45degreelines = cv2.filter2D(src=img,ddepth=cv2.CV_8U,kernel=kernel)

#Displaying the image in which we have detected the 45 degrees lines
only
cv2.imshow('Output Image with 45 degree lines only ',_45degreelines )

# Display the images and wait till any key is pressed
cv2.waitKey(0)
# Destroy all the windows created by the imshow() function of the
OpenCV
cv2.destroyAllWindows()
```

Images:

Original/Input :



Output :



10. Take an Image and observe the effect of Laplacian filtering? Can you show edge sharpening using Laplacian edges?

Laplacian filters are derivative filters used to find areas of rapid change (edges) in images.
we deduce that the second derivative can be used to detect edges. Since images are "2D", we would need to take the derivative in both dimensions. Here, the Laplacian operator comes handy.
The Laplacian operator is defined by:

$$\text{Laplace}(f) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

The Laplacian operator is implemented in OpenCV by the function Laplacian. In fact, since the Laplacian uses the gradient of images, it calls internally the Sobel operator to perform its computation.

The function calculates the Laplacian of the source image by adding up the second x and y derivatives calculated using the Sobel operator:

$$\texttt{dst} = \Delta \texttt{src} = \frac{\partial^2 \texttt{src}}{\partial x^2} + \frac{\partial^2 \texttt{src}}{\partial y^2}$$

This is done when ksize >1. When ksize==1, the Laplacian is computed by filtering the image with the following 3×3 aperture:

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

Since derivative filters are very sensitive to noise,we first smooth the image (e.g., using a Gaussian filter) before applying the Laplacian. This two-step process is call the Laplacian of Gaussian (LoG) operation.

Code:

```
# Import numpy and cv2 package
import cv2
import numpy as np

# Reading original image from the disk as grayscale image
img= cv2.imread('man.jpg',0)
#Smoothening the image using the Gausian Blurr
smooth= cv2.GaussianBlur(img, ksize=(5,5),sigmaX=10,sigmaY=10)
#Applying the Laplacian on the smoothen image
sharp= cv2.Laplacian(smooth,ddepth=cv2.CV_8U, ksize=3)

#Subtracting the sharp image image matrix from the original image
matrix
img1= cv2.subtract(img,sharp)

# Displaying original, sharp  and final output images together
img3= np.concatenate((img,sharp,img1),axis=1)
cv2.imshow('Problem10 Output',img3)

# Display the images and wait till any key is pressed
cv2.waitKey(0)
# Destroy all the windows created by the imshow() function of the
OpenCV
cv2.destroyAllWindows()
```

Images:

Original/Input and Output:

11. Detect Road Land Markers

The algorithm we are using to detect the road markers are as follows:
1. Read colour image from the disk.
2. Apply colour mask to retain only yellow and white colours in the image and convert rest to black.
3. Convert the masked image to grayscale.
4. Apply canny edge detection on the grayscale image.
5. Apply Hough Transform and plot lines on the original image to detect the land markings on the road.

Code:

```
# Import numpy and cv2 package
import cv2
import numpy as np

# Reading original image from the disk
original  = cv2.imread('road2.jpg')
img  = cv2.imread('road2.jpg')

#Split the image into r,g,b channel using inbuilt function of OpenCv
b,g,r = cv2.split(img)

#thresholding the red and green channel where pixel intensity  is less
than 235 make it 0.
r[r<235]=0
g[g<235]=0
#Creating a new image of same size as of original image
masked_img = np.zeros(img.shape,dtype='uint8')
#assigining the new values of r,g,b channel of original image to
masked_image
masked_img[:,:,2]=r
b_temp = b/2
masked_img[:,:,0]= b_temp.astype('uint8')
masked_img[:,:,1]=g
#Displaying the masked_image
cv2.imshow('masked img', masked_img)

#Converting the masked_img to grayscale using inbuilt function CV
gray = cv2.cvtColor(masked_img,cv2.COLOR_BGR2GRAY)
#Creating new image 'thresh'of size of gray image
thresh = np.ones(gray.shape,dtype='uint8')*255
#thresholding the new image 'thresh' where pixel intensity is less
than 200 make it zer0
```

```
thresh[gray<200]=0

# Applying the inbuilt HoughlinesP function of the OpenCV to find
coordinates of the end points of line
# in Hough space
lines =
cv2.HoughLinesP(thresh,1,np.pi/180,80,minLineLength=2,maxLineGap=15)
# We got the coordinates of each line which are passing the threshold
limit of votes set by us i.e 80
# using those coordinates we are drawing lines on the original image
which will highlight all the lanes in the road
# with green color
for line in lines[:]:
     for x1,y1,x2,y2 in line:
         cv2.line(img,(x1,y1),(x2,y2),(0,255,0),2)

# # Displaying the final output image
cv2.imshow('res',img)
# Displaying the original image and final output image together
cv2.imshow('Problem11 Output',np.concatenate((original,img),axis=1))

# Display the images and wait till any key is pressed
cv2.waitKey(0)
# Destroy all the windows created by the imshow() function of the
OpenCV
cv2.destroyAllWindows()
```

Images:

Original/Input and Output :

12. Classify modes; Night; Potrait; Landscape Design Features; Use NN.

The k-Nearest Neighbor classifier is by far the most simple machine learning/image
classification algorithm. In fact, it's so simple that it doesn't actually "learn" anything.
Inside, this algorithm simply relies on the distance between feature vectors, much like
building an image search engine — only this time, we have the labels associated with each
image so we can predict and return an actual category for the image.
Simply put, the k-NN algorithm classifies unknown data points by finding the most common
class among the k-closest examples. Each data point in the k closest examples casts a vote
and the category with the most votes wins!
In our case we have chosen k=3.

Steps followed:
1. Import required libraries (main:sklearn here)
2. Assign path of dataset and compute categories .here: night-mode, Portrait, Landscape
3. Define functions that return images,lables,compute required input to classifier.
We show 2 different methods here
a) list of raw pixel intensities after flattening images
b) Flattened 3d color histogram
4. Function to divide dataset automatically into training and testing set ratio:3:1
5. Call classifier,fit training data to it,then calculate score for test data.
Results:
We see that we get 74% accuracy from one and 67% accuracy from 2nd method.

Code:

```
#importing all the necessary packages
from sklearn.neighbors import KNeighborsClassifier
from sklearn.cross_validation import train_test_split
from sklearn.metrics import classification_report
import cv2
import numpy as np
import os
from os.path import join
import glob


#Getting the images from the designated path and store category names
train_path=os.path.abspath('dataset')
training_names=os.listdir(train_path)
print (training_names)


image_paths=[]
image_classes=[]
rawimage_pix=[]
class_id=0
nbrnames=len(training_names)
labels=[]
color_features=[]


#Fetching files
def get_imgfiles(path):
all_files=[]
all_files.extend([join(path,fname)
for fname in glob.glob(path+"/*")])
return all_files

#Storing images and their corresponding labels
for training_names,label in zip(training_names,range(nbrnames)):
class_path=join(train_path,training_names)
class_files=get_imgfiles(class_path)
image_paths+=class_files
labels+=[class_id]*len(class_files)
class_id+=1
#Method 1 for knn:resizes and flattens images and gives a list of raw
pixel intensities
def image_to_feature_vector(image, size=(32, 32)):
return cv2.resize(image, size).flatten()
```

```
#Method 2 : Forming color histogram of given images to give to
classifier
def extract_color_histogram(image,bins=(8,8,8),size=(32,32)):
image=cv2.resize(image,size)
hsv=cv2.cvtColor(image,cv2.COLOR_BGR2HSV)
hist=cv2.calcHist([hsv],[0,1,2],None,bins,[0 , 180, 0, 256, 0, 256])
#inplace normalization
cv2.normalize(hist,hist)
return hist.flatten()

for (i,image_path) in enumerate(image_paths):
image=cv2.imread(image_path)
#extract a color histogram from the image
raw=image_to_feature_vector(image)
rawimage_pix.append(raw)

hist=extract_color_histogram(image)
color_features.append(hist)

#show update every 5 image
if i>0 and i% 5==0:
print("[INFO] processed {}/{}".format(i,len(image_paths)))


features=np.array(color_features)
labels=np.array(labels)
rawImages=rawimage_pix

#Automatic grouping of images for training(75/100 ratio) and testing
(25/100 ratio)
#m1.normal rawimage pixels
(trainRI, testRI, trainRL, testRL) = train_test_split(
rawImages, labels, test_size=0.25, random_state=42)
#m2.color histogram pixels
(trainFeat, testFeat, trainLabels, testLabels) = train_test_split(
features, labels, test_size=0.25, random_state=42)



#For method 1
model=KNeighborsClassifier(3)
model.fit(trainRI, trainRL)
acc = model.score(testRI, testRL)
print (acc)
#For method 2
model1 = KNeighborsClassifier(3)
model1.fit(trainFeat, trainLabels)
acc1 = model1.score(testFeat, testLabels)
print (acc1)
```

Images:

Input images: