

<b>S.No.</b>	<b><i>OpenCV Index Function's</i></b>	
<b>1.</b>	<b><i>Importing OpenCV &amp; Reading an Image</i></b>	
<b>2.</b>	<b>Demonstrating Gray image intensities</b>	
<b>3.</b>	<b>Image Resizing</b>	
<b>4.</b>	<b>Changing Color Spaces</b>	
<b>5.</b>	<b>Averaging</b>	
<b>6.</b>	<b>Blurring</b>	
<b>7.</b>	<b>Image Threshold</b>	
<b>8.</b>	<b>Adaptive Threshold</b>	
<b>9.</b>	<b>Image Rotation</b>	
<b>10.</b>	<b>Drawing Function</b>	
<b>11.</b>	<b>Edge Detection</b>	
<b>12.</b>	<b>Histogram Equalization</b>	

## Importing OpenCV & Reading an Image

Code:-

```
import cv2
import numpy as np
from google.colab.patches import cv2_imshow

img1 = cv2.imread('/content/gdrive/MyDrive/def.jpg',0)
cv2_imshow(img1)

print(img1.shape)
```

Output:-



## Demonstrating Gray image intensities

Code:-

```
img = np.ones((256,256), np.uint8) #Return a new array of given shape and
type,filled with 1.

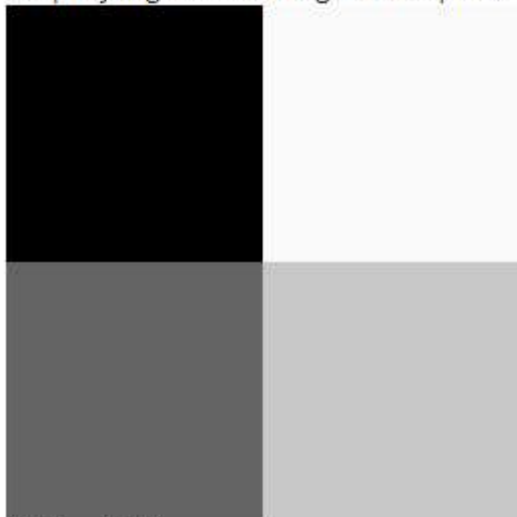
#Creating 4 Matrics of dimension (128,128)
black=np.zeros((128,128),np.uint8)
white=255*np.ones((128,128),np.uint8)
gray_1=100*np.ones((128,128),np.uint8)
gray_2=200*np.ones((128,128),np.uint8)

#Filling created(256,256)-Matrics with 4-(128,128) Martics
img [0:128,0:128]= black #1st Quadrant
img [0:128,128:256]= white #2nd Quadrant
img [128:256,0:128]= gray_1 #3rd Quadrant
img [128:256,128:256]= gray_2 #4th Quadrant

print("Displaying above image & respective quadrants")
cv2_imshow(img)
print(img.shape)
```

Output:-

Displaying above image & respective quadrants



(256, 256)

## Image Resizing

Image resizing refers to the scaling of images. Scaling comes in handy in many image processing as well as machine learning applications. It helps in reducing the number of pixels from an image and that has several advantages e.g. It can reduce the time of training of a neural network as more is the number of pixels in an image more is the number of input nodes that in turn increases the complexity of the model.

It also helps in zooming in images. Many times we need to resize the image i.e. either shrink it or scale up to meet the size requirements. OpenCV provides us several interpolation methods for resizing an image.

**We can express the remap for every pixel location**

**(x,y) As:  $g(x,y)=f(h(x,y))$**

we have an image and, say, we want to do a remap such that:

$$h(x,y)=(l.cols-x,y)$$

## Choice of Interpolation Method for Resizing –

- `cv2.INTER_AREA`: This is used when we need to shrink an image.
- `cv2.INTER_CUBIC`: This is slow but more efficient.
- `cv2.INTER_LINEAR`: This is primarily used when zooming is required.

This is the default interpolation technique in OpenCV.

**Code:-**

[illegible]

```

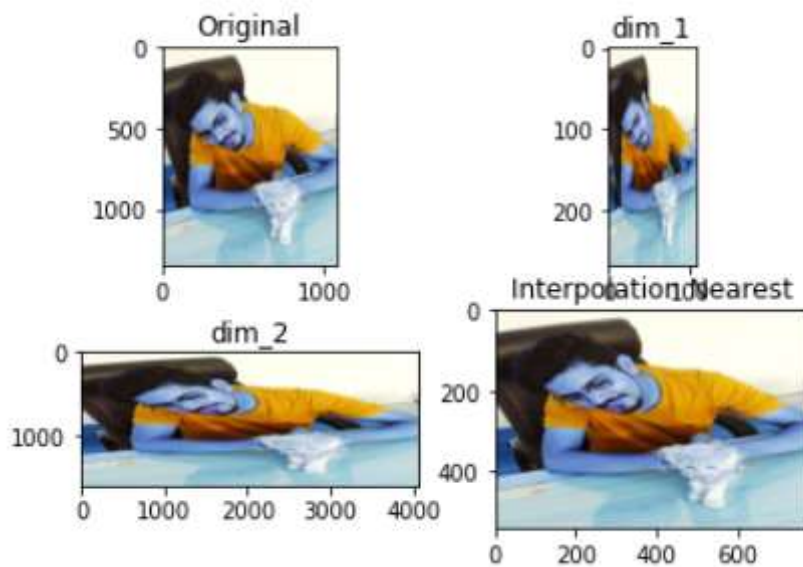
Titles=["Original", "dim_1", "dim_2", "Interpolation Nearest"]
images=[image, dim_1, dim_2, stretch_near]
count = 4

for i in range(count):
    plt.subplot(2, 2, i + 1)
    plt.title(Titles[i])
    plt.imshow(images[i])

plt.show()

```

**Output:-**



## Changing Color Spaces

Color spaces are a way to represent the color channels present in the image that gives the image that particular hue. There are several different color spaces and each has its own significance.

Some of the popular color spaces are *RGB* (Red, Green, Blue), *CMYK* (Cyan, Magenta, Yellow, Black), *HSV* (Hue, Saturation, Value), etc.

**BGR color space:** OpenCV's default color space is RGB. However, it actually stores color in the BGR format. It is an additive color model where the different intensities of Blue, Green and Red give different shades of color.

### 1. Color to Gray\_Scale

```
img3 = cv2.imread("/content/gdrive/MyDrive/xyz1.jpg",1)
gray = cv2.imread("/content/gdrive/MyDrive/xyz1.jpg",0)
cv2_imshow(gray)
```



**HSV color space:** It stores color information in a cylindrical representation of RGB color points. It attempts to depict the colors as perceived by the human eye.

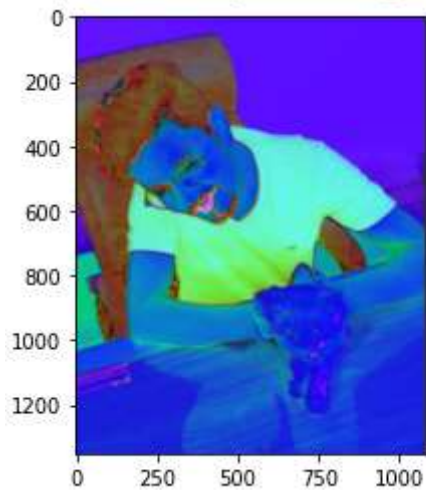
Hue value varies from 0-179, Saturation value varies from 0-255 and Value value varies from 0-255. It is mostly used for color segmentation purposes.

## 2. Converting image to HSV

```
hsv_image = cv2.cvtColor(img3, cv2.COLOR_BGR2HSV)
```

```
#plotting the HSV image
```

```
plt.imshow(hsv_image)
```



## Averaging

This is done by convolving an image with a normalized box filter. It simply takes the average of all the pixels under the kernel area and replaces the central element. This is done by the function `cv2.blur()` or `cv2.boxFilter()`. Check the docs for more details about the kernel. We should specify the width and height of the kernel. A 3x3 normalized box filter would look like this:

$$K = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

### Code:-

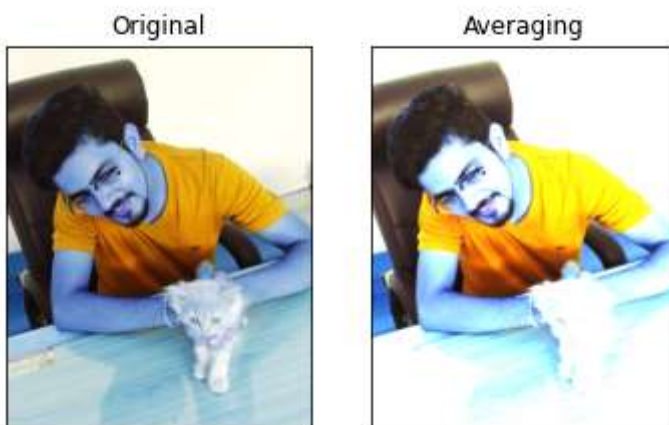
```
img4 = cv2.imread('/content/gdrive/MyDrive/xyz1.jpg')

kernel = np.ones((5,8),np.float32)/25
dst = cv2.filter2D(img4,-1,kernel)

plt.subplot(121),plt.imshow(img4),plt.title('Original')
plt.xticks([], plt.yticks([]))

plt.subplot(122),plt.imshow(dst),plt.title('Averaging')
plt.xticks([], plt.yticks([]))

plt.show()
```





## **Blurring**

Blurring refers to making the image less clear or distinct. It is done with the help of various low pass filter kernels.

Advantages of blurring:

- It helps in Noise removal. As noise is considered as high pass signal so by the application of a low pass filter kernel we restrict noise.
- It helps in smoothing the image.
- Low intensity edges are removed.
- It helps in hiding the details when necessary. For e.g. in many cases police deliberately want to hide the face of the victim, in such cases blurring is required.

**Normalized Box Filter :** This filter is the simplest of all! Each output pixel is the *mean* of its kernel neighbors ( all of them contribute with equal weights)

$$K = \frac{1}{K_{width} \cdot K_{height}} \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & 1 & 1 & \dots & 1 \\ \cdot & \cdot & \cdot & \dots & 1 \\ \cdot & \cdot & \cdot & \dots & 1 \\ 1 & 1 & 1 & \dots & 1 \end{bmatrix}$$

### Code :

```
img = cv2.imread('/content/gdrive/MyDrive/xyz1.jpg')  
# You can change the kernel size as you want  
blurImg = cv2.blur(img, (10,10))  
cv2_imshow(blurImg)
```

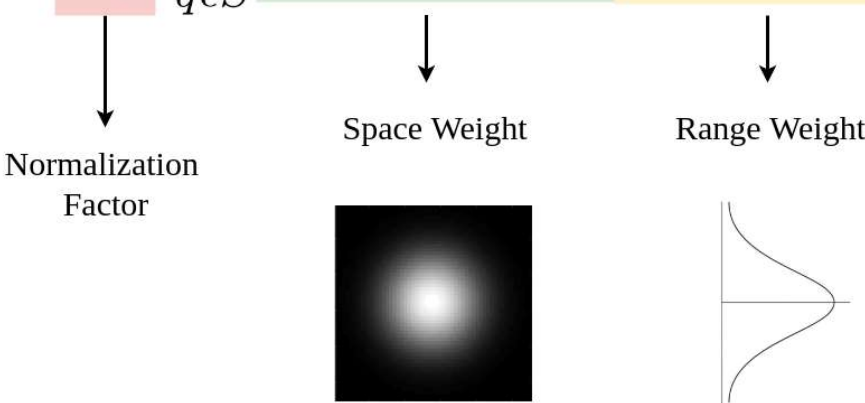


## Bilateral Filtering

A bilateral filter is used for smoothening images and reducing noise, while preserving edges.

However, these convolutions often result in a loss of important edge information, since they blur out everything, irrespective of it being noise or an edge. To counter this problem, the non-linear bilateral filter was introduced.

$$BF[I]_p = \frac{1}{W_p} \sum_{q \in S} G_{\sigma_s}(\|p - q\|) G_{\sigma_r}(|I_p - I_q|) I_q$$



Normalization Factor      Space Weight      Range Weight

**bilateralFilter()** have following arguments:

1. **d**: Diameter of each pixel neighborhood.
2. **sigmaColor**: Value of 'sigma' in the color space. The greater the value, the colors farther to each other will start to get mixed.
3. **sigmaColor**: Value of 'sigma' in the coordinate space. The greater its value, the more further pixels will mix together, given that their colors lie within the sigmaColor range.

OpenCV provides a function **cv.filter2D()** to convolve a kernel with an image. As an example, we will try an averaging filter on an image. A 5x5 averaging filter kernel will look like the below:

$$K = \frac{1}{25} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

### Code:

```
import cv2

# Read the image.
img = cv2.imread('/content/gdrive/MyDrive/xyz1.jpg')

# Apply bilateral filter with d = 15,
# sigmaColor = sigmaSpace = 75.
bilateral = cv2.bilateralFilter(img, 15, 75, 75)

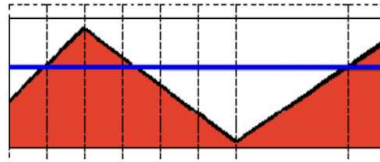
cv2_imshow(bilateral)
```

### Output:



## Image Threshold

Thresholding is a technique in OpenCV, which is the assignment of pixel values in relation to the threshold value provided. In thresholding, each pixel value is compared with the threshold value. If the pixel value is smaller than the threshold, it is set to 0, otherwise, it is set to a maximum value (generally 255). Thresholding is a very popular segmentation technique, used for separating an object considered as a foreground from its background. A threshold is a value which has two regions on its either side i.e. below the threshold or above the threshold.

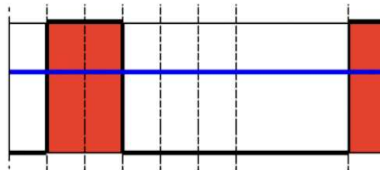


### Threshold Binary

- This thresholding operation can be expressed as:

$$dst(x, y) = \begin{cases} \text{maxVal} & \text{if } src(x, y) > \text{thresh} \\ 0 & \text{otherwise} \end{cases}$$

- So, if the intensity of the pixel  $src(x, y)$  is higher than  $thresh$ , then the new pixel intensity is set to a  $MaxVal$ . Otherwise, the pixels are set to 0.

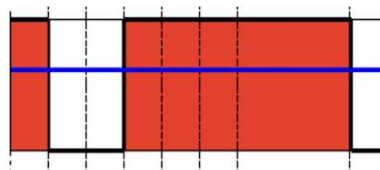


### Threshold Binary, Inverted

- This thresholding operation can be expressed as:

$$dst(x, y) = \begin{cases} 0 & \text{if } src(x, y) > \text{thresh} \\ \text{maxVal} & \text{otherwise} \end{cases}$$

- If the intensity of the pixel  $src(x, y)$  is higher than  $thresh$ , then the new pixel intensity is set to a 0. Otherwise, it is set to  $MaxVal$ .



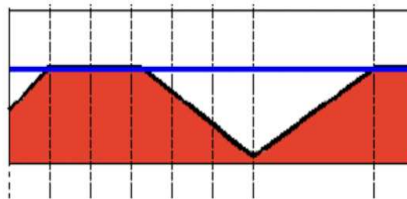
---

## Truncate

- This thresholding operation can be expressed as:

$$\text{dst}(x,y) = \begin{cases} \text{threshold} & \text{if } \text{src}(x,y) > \text{thresh} \\ \text{src}(x,y) & \text{otherwise} \end{cases}$$

- The maximum intensity value for the pixels is *thresh*, if *src*(*x*, *y*) is greater, then its value is *truncated*. See figure below:

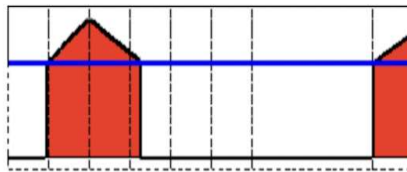


## Threshold to Zero

- This operation can be expressed as:

$$\text{dst}(x,y) = \begin{cases} \text{src}(x,y) & \text{if } \text{src}(x,y) > \text{thresh} \\ 0 & \text{otherwise} \end{cases}$$

- If *src*(*x*, *y*) is lower than *thresh*, the new pixel value will be set to 0.

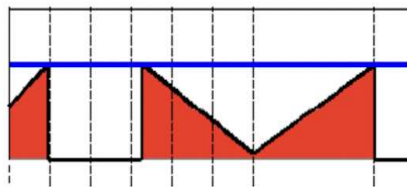


## Threshold to Zero, Inverted

- This operation can be expressed as:

$$\text{dst}(x,y) = \begin{cases} 0 & \text{if } \text{src}(x,y) > \text{thresh} \\ \text{src}(x,y) & \text{otherwise} \end{cases}$$

- If *src*(*x*, *y*) is greater than *thresh*, the new pixel value will be set to 0.



## Code:-

```
#importing the required libraries
```

```

import numpy as np

import cv2

import matplotlib.pyplot as plt

%matplotlib inline

#here 0 means that the image is loaded in gray scale format
gray_image = cv2.imread('/content/gdrive/MyDrive/xyz1.jpg',0)


ret,thresh_binary = cv2.threshold(gray_image,127,255,cv2.THRESH_BINARY)
ret,thresh_binary_inv =
cv2.threshold(gray_image,127,255,cv2.THRESH_BINARY_INV)
ret,thresh_trunc = cv2.threshold(gray_image,127,255,cv2.THRESH_TRUNC)
ret,thresh_tozero = cv2.threshold(gray_image,127,255,cv2.THRESH_TOZERO)
ret,thresh_tozero_inv =
cv2.threshold(gray_image,127,255,cv2.THRESH_TOZERO_INV)


#DISPLAYING THE DIFFERENT THRESHOLDING STYLES

names = ['Original
Image', 'BINARY', 'THRESH_BINARY_INV', 'THRESH_TRUNC', 'THRESH_TOZERO', 'THRESH
_TOZERO_INV']

images =
gray_image,thresh_binary,thresh_binary_inv,thresh_trunc,thresh_tozero,thresh_tozero_inv


for i in range(6):
    plt.subplot(2,3,i+1),plt.imshow(images[i], 'gray')
    plt.title(names[i])
    plt.xticks([],plt.yticks([]))


plt.show()

```

**Output:-**

Original Image



BINARY



THRESH\_BINARY\_INV



THRESH\_TRUNC



THRESH\_TOZERO



THRESH\_TOZERO\_INV





## Adaptive Thresholding

In case of adaptive thresholding, different threshold values are used for different parts of the image. This function gives better results for images with varying lighting conditions – hence the term “adaptive”.

Otsu’s binarization method finds an optimal threshold value for the whole image. It works well for bimodal images (images with 2 peaks in their histogram).

**Code:-**

```
#ADAPTIVE THRESHOLDING
```

```
gray_image = cv2.imread('/content/gdrive/MyDrive/xyz1.jpg',0)
```

```
ret,thresh_global = cv2.threshold(gray_image,127,255,cv2.THRESH_BINARY)
```

```
#here 11 is the pixel neighbourhood that is used to calculate the  
threshold value
```

```
thresh_mean =
```

```
cv2.adaptiveThreshold(gray_image,255,cv2.ADAPTIVE_THRESH_MEAN_C,cv2.THRESH  
_BINARY,11,2)
```

```
thresh_gaussian =
```

```
cv2.adaptiveThreshold(gray_image,255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C,cv2.TH  
RESH_BINARY,11,2)
```

```
names = ['Original Image','Global Thresholding','Adaptive Mean  
Threshold','Adaptive Gaussian Thresholding']
```

```
images = [gray_image,thresh_global,thresh_mean,thresh_gaussian]
```

```
for i in range(4):
```

```
    plt.subplot(2,2,i+1),plt.imshow(images[i], 'gray')
```

```
    plt.title(names[i])
```

```
    plt.xticks([],plt.yticks([]))
```

```
plt.show()
```

**Output:-**

Original Image



Global Thresholding



Adaptive Mean Threshold



Adaptive Gaussian Thresholding



# Image Rotation

## 1.

Rotation of an image for an angle  $\theta$  is achieved by the transformation matrix of the form

$$M = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

But OpenCV provides scaled rotation with adjustable center of rotation so that you can rotate at any location you prefer. The modified transformation matrix is given by

$$\begin{bmatrix} \alpha & \beta & (1-\alpha) \cdot \text{center.x} - \beta \cdot \text{center.y} \\ -\beta & \alpha & \beta \cdot \text{center.x} + (1-\alpha) \cdot \text{center.y} \end{bmatrix}$$

where:

$$\alpha = \text{scale} \cdot \cos\theta,$$

$$\beta = \text{scale} \cdot \sin\theta$$

```
rows,cols = image1.shape[:2]

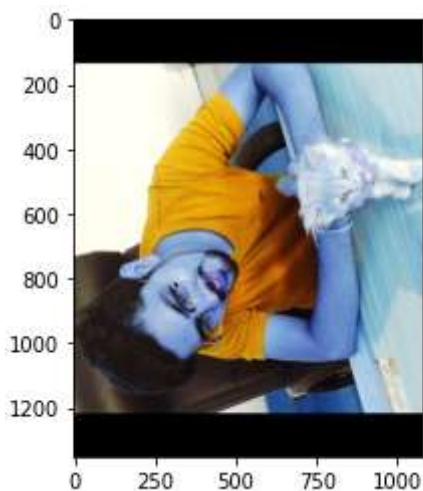
#(col/2,rows/2) is the center of rotation for the image
# M is the coordinates of the center

M = cv2.getRotationMatrix2D((cols/2,rows/2),90,1)

dst = cv2.warpAffine(image1,M,(cols,rows))

plt.imshow(dst)
```

**Output:**



2.

We can express the remap for every pixel location

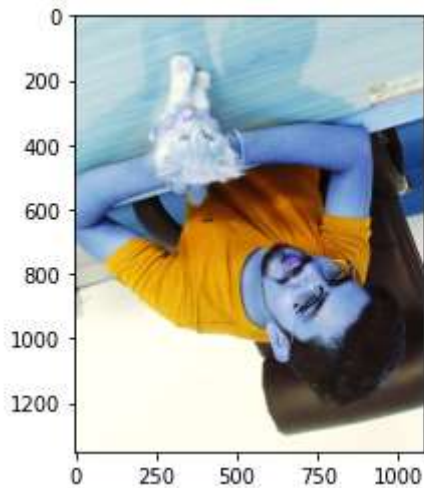
$$(x,y) \text{ As: } g(x,y)=f(h(x,y))$$

we have an image and, say, we want to do a remap such that:

$$h(x,y)=(l.cols-x,y)$$

```
rows,cols = image1.shape[:2]
#(col/2,rows/2) is the center of rotation for the image
# M is the coordinates of the center
M = cv2.getRotationMatrix2D((cols/2,rows/2),180,1)
dst = cv2.warpAffine(image1,M,(cols,rows))
plt.imshow(dst)
```

Output:



## Drawing Functions

[OpenCV](#) provides many drawing functions to draw geometric shapes and write text on images. Let's see some of the drawing functions and draw geometric shapes on images using OpenCV.

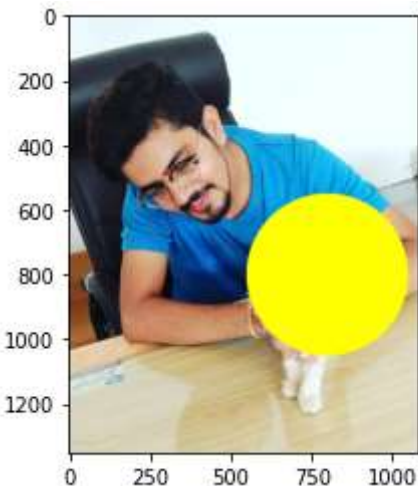
Some of the drawing functions are :

- *cv2.line()* : Used to draw lines on an image.
- *cv2.rectangle()* : Used to draw a rectangle on an image.
- *cv2.circle()* : Used to draw a circle on an image.
- *cv2.putText()* : Used to write text on image.

### 1. #circle

```
circle=cv2.circle(cv2.cvtColor(cv2.imread('/content/gdrive/MyDrive/xyz1.jpg'),cv2.COLOR_BGR2RGB),(800,800), 250, (255,255,0), -1)
```

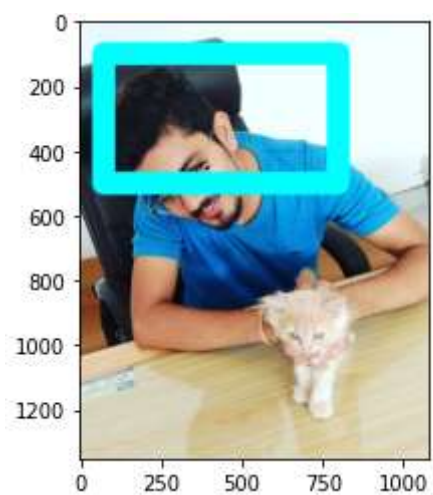
```
plt.imshow(circle)
```



### 2. #rectangle

```
rectangle=cv2.rectangle(cv2.cvtColor(cv2.imread('/content/gdrive/MyDrive/xyz1.jpg'),cv2.COLOR_BGR2RGB),(75,100),(800,500),(0,255,255),70)
```

```
plt.imshow(rectangle)
```



## Edge Detection

Edges are the points in an image where the image brightness changes sharply or has discontinuities. Such discontinuities generally correspond to:

- Discontinuities in depth
- Discontinuities in surface orientation
- Changes in material properties
- Variations in scene illumination

Edges are very useful features of an image that can be used for different applications like classification of objects in the image and localization. Even deep learning models calculate edge features to extract information about the objects present in image.

### Steps

1. Filter out any noise. The Gaussian filter is used for this purpose. An example of a Gaussian kernel of  $size = 5$  that might be used is shown below:

$$K = \frac{1}{159} \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

2. Find the intensity gradient of the image. For this, we follow a procedure analogous to Sobel:
  - a. Apply a pair of convolution masks (in  $x$  and  $y$  directions):

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

- b. Find the gradient strength and direction with:

$$G = \sqrt{G_x^2 + G_y^2}$$
$$\theta = \arctan\left(\frac{G_y}{G_x}\right)$$

The direction is rounded to one of four possible angles (namely 0, 45, 90 or 135)

3. *Non-maximum* suppression is applied. This removes pixels that are not considered to be part of an edge. Hence, only thin lines (candidate edges) will remain.
4. *Hysteresis*: The final step. Canny does use two thresholds (upper and lower):
  - a. If a pixel gradient is higher than the *upper* threshold, the pixel is accepted as an edge
  - b. If a pixel gradient value is below the *lower* threshold, then it is rejected.
  - c. If the pixel gradient is between the two thresholds, then it will be accepted only if it is connected to a pixel that is above the *upper* threshold.Canny recommended a *upper:lower* ratio between 2:1 and 3:1.

### Code:

```
#import the required libraries
import numpy as np
import cv2
import matplotlib.pyplot as plt

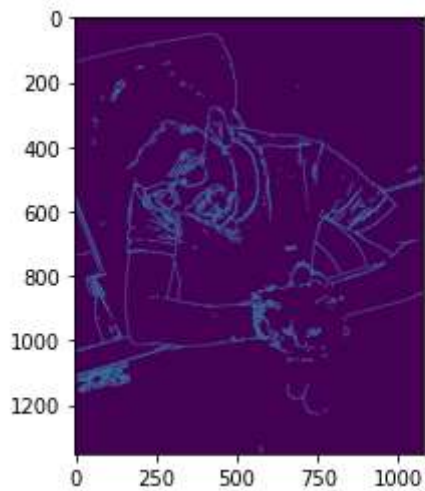
%matplotlib inline

#read the image
image = cv2.imread('/content/gdrive/MyDrive/xyz1.jpg')

#calculate the edges using Canny edge algorithm
edges = cv2.Canny(image,100,200)

#plot the edges
plt.imshow(edges)
```

### Output:

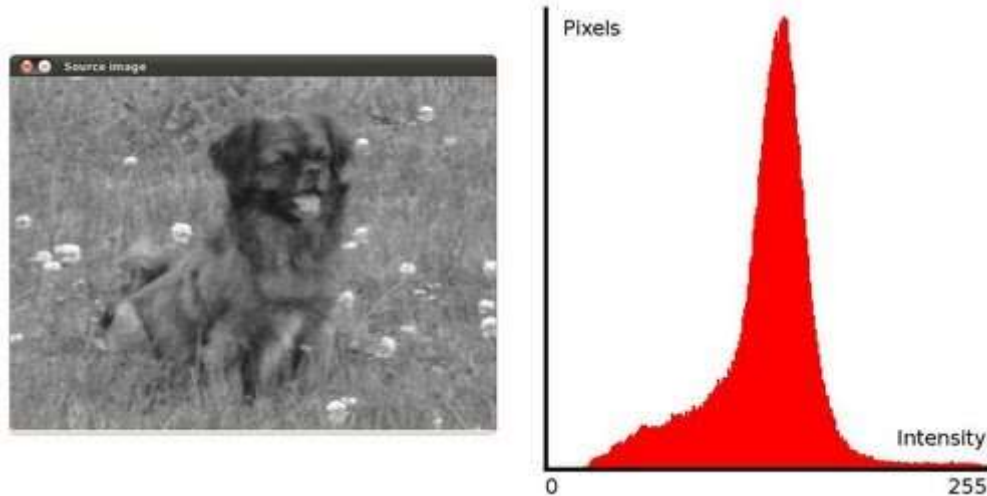




# Histogram Equalization

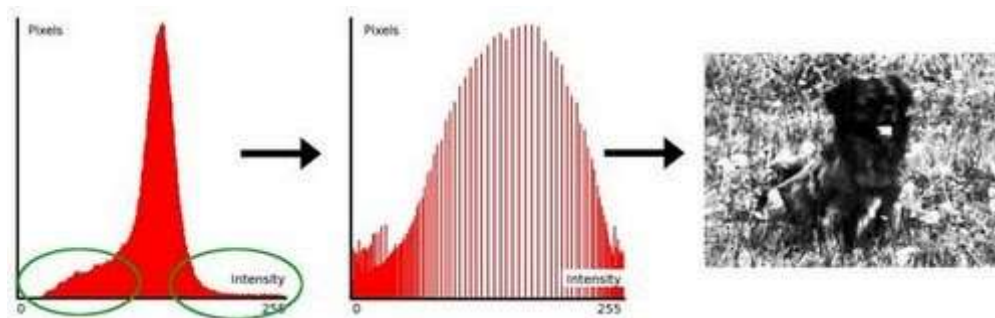
## What is an Image Histogram?

- It is a graphical representation of the intensity distribution of an image.
- It quantifies the number of pixels for each intensity value considered.



## What is Histogram Equalization?

- It is a method that improves the contrast in an image, in order to stretch out the intensity range (see also the corresponding [Wikipedia entry](#)).
- 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. Take a look at the figure below: The green circles indicate the *underpopulated* intensities. After applying the equalization, we get an histogram like the figure in the center. The resulting image is shown in the picture at right.

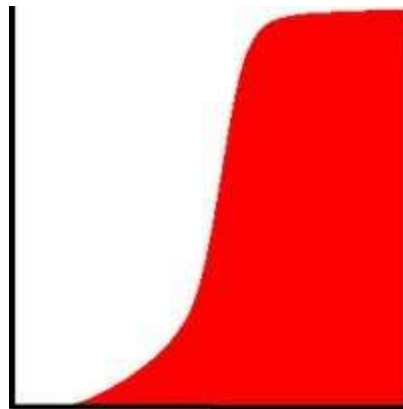


## How does it work?

- 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 spread 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:

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

To use this as a remapping function, we have to normalize such that the maximum value is 255 ( or the maximum value for the intensity of the image ). From the example above, the cumulative function is:



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

$$\text{equalized}(x,y) = H(\text{src}(x,y))$$

## Code:-

Load the source image:

```
CommandLineParser parser( argc, argv, "{@input | lena.jpg | input image}"
);
Mat src = imread( samples::findFile( parser.get<String>( "@input" ) ),
IMREAD_COLOR );
if( src.empty() )
```

```
{
    cout << "Could not open or find the image!\n" << endl;
    cout << "Usage: " << argv[0] << " <Input image>" << endl;
    return -1;
}
```

- Convert it to grayscale:

```
cvtColor( src, src, COLOR_BGR2GRAY );
```

- Apply histogram equalization with the function **cv::equalizeHist** :

```
Mat dst;
equalizeHist( src, dst );
```

As it can be easily seen, the only arguments are the original image and the output (equalized) image.

- Display both images (original and equalized):

```
imshow( "Source image", src );
imshow( "Equalized Image", dst );
```

- Wait until user exists the program

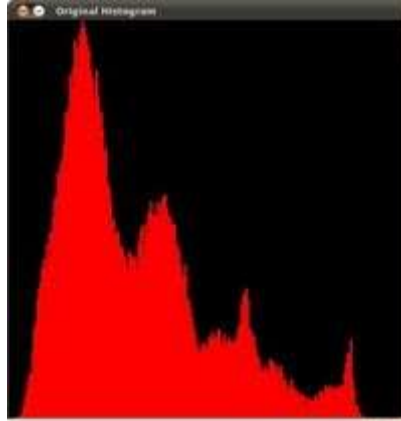
```
waitKey();
```

## Results

1. To appreciate better the results of equalization, let's introduce an image with not much contrast, such as:



which, by the way, has this histogram:

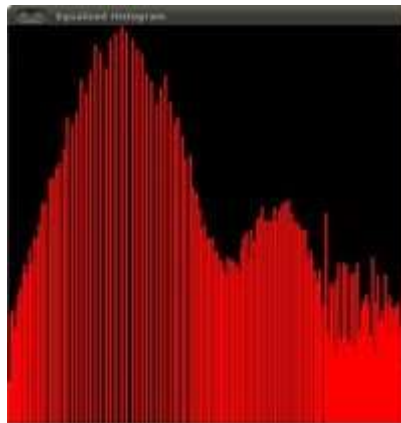


notice that the pixels are clustered around the center of the histogram.

2. After applying the equalization with our program, we get this result:



this image has certainly more contrast. Check out its new histogram like this:



Notice how the number of pixels is more distributed through the intensity range.