About OpenCV

OpenCV (Open Source Computer Vision Library) is an open-source library for computer vision and machine learning. It was initiated by Intel and has been primarily supported by the company since its inception. The purpose of OpenCV is to offer a standardized infrastructure for computer vision applications, thereby promoting the integration of machine perception into commercial products. Written in the C++ programming language, OpenCV also has wrappers available in other languages such as Python, Java, and Matlab.

In [ ]:
```
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

In [ ]:
```
# reading and displaying images
# cv2.imread(path, flag)
img_path = "data/01_raw_data/RGBY.jpg"
img = cv2.imread(img_path)
img
```

Out[ ]:
```
array([[[ 38,   3,   0],
        [ 70,  12,   7],
        [ 96,   0,   0],
        ...,
        [  0,  20,  21],
        [  0,  20,  21],
        [  0,  20,  21]],

       [[ 65,   6,   4],
        [112,  29,  27],
        [152,  22,  22],
        ...,
        [118, 238, 237],
        [118, 239, 235],
        [118, 238, 237]],

       [[101,   0,   0],
        [160,  30,  30],
        [208,  29,  30],
        ...,
        [ 54, 246, 239],
        [ 54, 247, 237],
        [ 54, 246, 239]],

       ...,

       [[  0,  55,   0],
        [ 57, 163,  56],
        [ 35, 193,  27],
        ...,
        [ 16,   6, 250],
        [ 16,   6, 250],
        [ 16,   6, 250]],

       [[  0,  55,   0],
        [ 57, 163,  56],
        [ 35, 193,  27],
        ...,
        [ 16,   6, 250],
        [ 16,   6, 250],
        [ 16,   6, 250]],

       [[  0,  55,   0],
        [ 57, 163,  56],
        [ 35, 193,  27],
        ...,
        [ 16,   6, 250],
        [ 16,   6, 250],
        [ 16,   6, 250]]], dtype=uint8)
```

An image is nothing more than a collection of pixels and their intensities. Generally, we access images in the RGB color format, as every color perceivable by the human eye is a combination of these three colors: Red, Green, and Blue. However, the initial developers at OpenCV chose the BGR color format instead of RGB. This decision was influenced by the popularity of the BGR format among software providers and camera manufacturers at the time.

What is Digital Image?

An image may be defined as a two-dimensional function, f (x,y), where x and y are spatial (plane) coordinates, and the amplitude of f at any pair of coordinates (x, y) is called the intensity or gray level of the image at that point.

Here, the intensity of each point in (x, y) coordinates

$$f(x,y)=\begin{bmatrix} f(0,0) & f(0,1) & \cdots & f(0,N-1) \\ f(1,0) & f(1,1) & \cdots & f(1,N-1) \\ \vdots & \vdots & & \vdots \\ f(M-1,0) & f(M-1,1) & \cdots & f(M-1,N-1) \end{bmatrix}$$

Image shown as a 2-D numerical array. (The numbers 0, .5, and 1 represent black, gray, and white respectively)

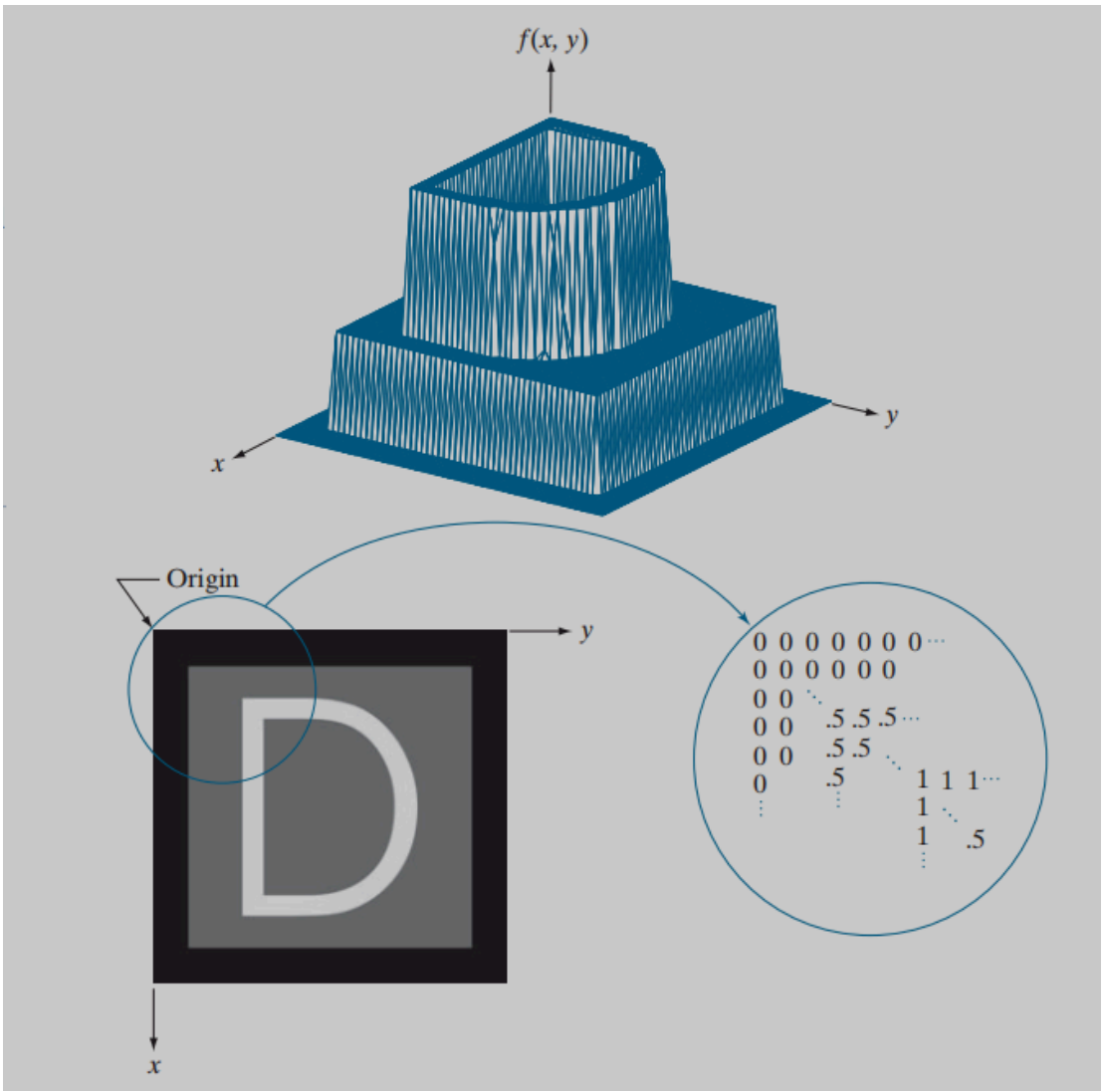


Image Properties

In [ ]:
```
# Resolution of image, it depicts the no. of rows, the no. of columns, and the n
img.shape
```

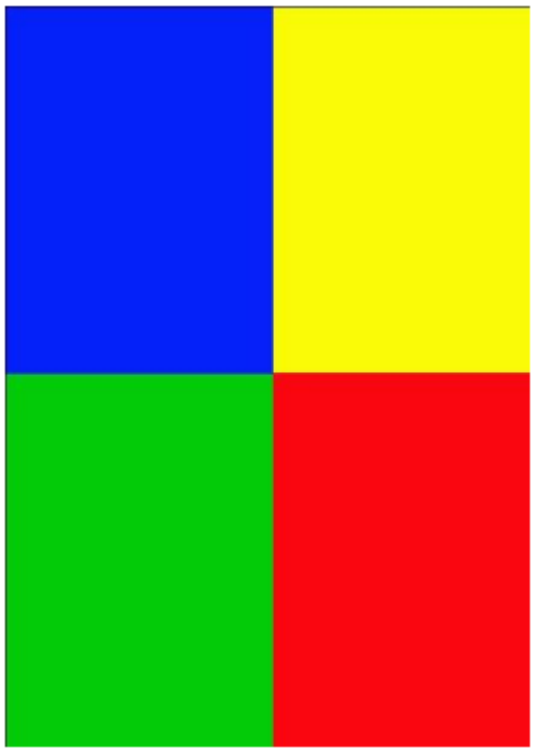
Out[ ]:
```
(586, 415, 3)
```

In [ ]:
```
img.size
```

Out[ ]:
```
729570
```

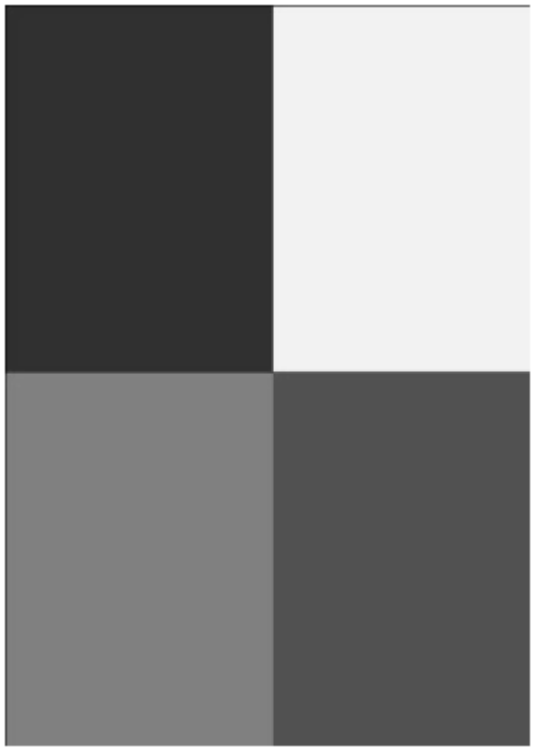
In [ ]:
```
# displaying image using cv2
cv2.imshow("Image Display",img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

In [ ]:
```
# displaying image using matplotlib
rgb_img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
plt.imshow(rgb_img)
plt.axis('off')
plt.show()
```



In [ ]:
```
# Saving an image
# cv2.imwrite(<filename>, <image>)
```

In [ ]:
```
# Changing color space

gray_img =  cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

plt.imshow(gray_img, cmap =  'gray')
plt.axis('off')
plt.show()
```



Converting an RGB image into a grayscale image involves a weighted sum of the RGB color channels. The typical formula for converting an RGB image to grayscale is:

$$Gray = 0.2989 \times Red + 0.5870 \times Green + 0.1140 \times Blue$$

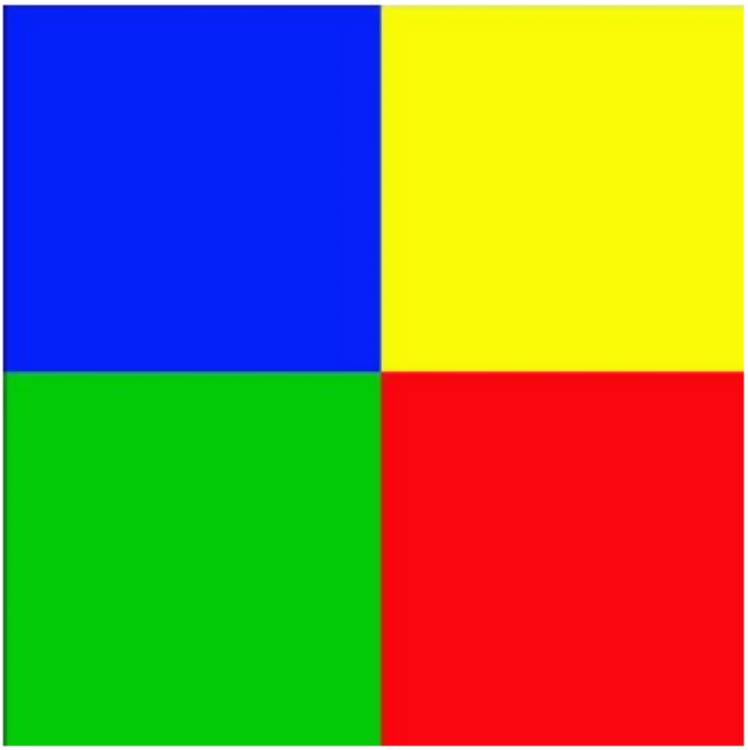
This formula represents the luminance of each pixel in the grayscale image, where the coefficients (0.2989, 0.5870, and 0.1140) represent the perceived brightness of each color channel to the human eye. These coefficients are based on the luminosity formula used in television systems.

So, for each pixel in the RGB image, you take the corresponding Red, Green, and Blue values, multiply them by their respective coefficients, and sum the results to obtain the grayscale intensity value for that pixel. This process is applied to every pixel in the image to generate the grayscale representation.

In [ ]:
```python
def rgb_to_gray(rgb_image):
    # Extract the dimensions of the input image
    height, width, _ = rgb_image.shape

    # Create a grayscale image with the same dimensions as the input image
    gray_image = np.zeros((height, width), dtype=np.uint8)

    # Iterate over each pixel in the image
    for y in range(height):
        for x in range(width):
            # Extract the RGB values for the current pixel
            red, green, blue = rgb_image[y, x]

            # Convert RGB to grayscale using the formula
            gray_value = int(0.2989 * red + 0.5870 * green + 0.1140 * blue)

            # Set the grayscale value for the current pixel in the output image
            gray_image[y, x] = gray_value
```

```
    return gray_image

# Load the RGB image
rgb_image = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# Convert RGB image to grayscale
gray_image = rgb_to_gray(rgb_image)

# Display the original RGB image and the grayscale image
cv2.imshow('RGB Image', rgb_image)
cv2.imshow('Grayscale Image', gray_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

In [ ]:
```
# resizing the image
resized_img = cv2.resize(rgb_img, (200, 200))

plt.imshow(resized_img)
plt.axis('off')
plt.show()
```



In [ ]:
```
# Displaying text on the image
# img_with_txt = cv2.putText(img, 'Text', (10, 10), cv2.FONT_HERSHEY_COMPLEX, 4,

# plt.imshow(img_with_txt)
# plt.axis('off')
# plt.show()
```

Splitting and Merging Channels

In [ ]:
```
# grey_img = cv2.imread("images/RGBY.jpg", 0)
# img = cv2.imread("images/RGBY.jpg", 1)   #Color is BGR not RGB
```

```
print(img.shape)     #(586, 415, 3)
print("Top left", img[0,0])    #Top left pixel
print("Top right", img[0, 400])  # Top right
print("Bottom Left", img[580, 0]) # Bottom left
print("Bottom right", img[580, 400])  # Bottom right

cv2.imshow("color pic", img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

```
(586, 415, 3)
Top left [38  3  0]
Top right [ 0 20 21]
Bottom Left [ 0 55  0]
Bottom right [ 16   6 250]
```

In [ ]:
```
#Split and merging channels
#Show individual color channels in the image
blue = img[:, :, 0]   #Show only blue pic. (BGR so B=0)
green = img[:, :, 1]  #Show only green pixels
red = img[:, :, 2]  #red only

cv2.imshow("blue pic", blue)
cv2.imshow("green pic", green)
cv2.imshow("red pic", red)
cv2.imshow("original image", img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

In [ ]:
```
#Or split all channels at once

b,g,r = cv2.split(img)

cv2.imshow("blue pic", b)
cv2.imshow("green pic", g)
cv2.imshow("red pic", r)
cv2.imshow("original image", img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

In [ ]:
```
#to merge each image into bgr

img_merged = cv2.merge((b,g,r))

cv2.imshow("merged pic", img_merged)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Image Smoothing/ Blurring

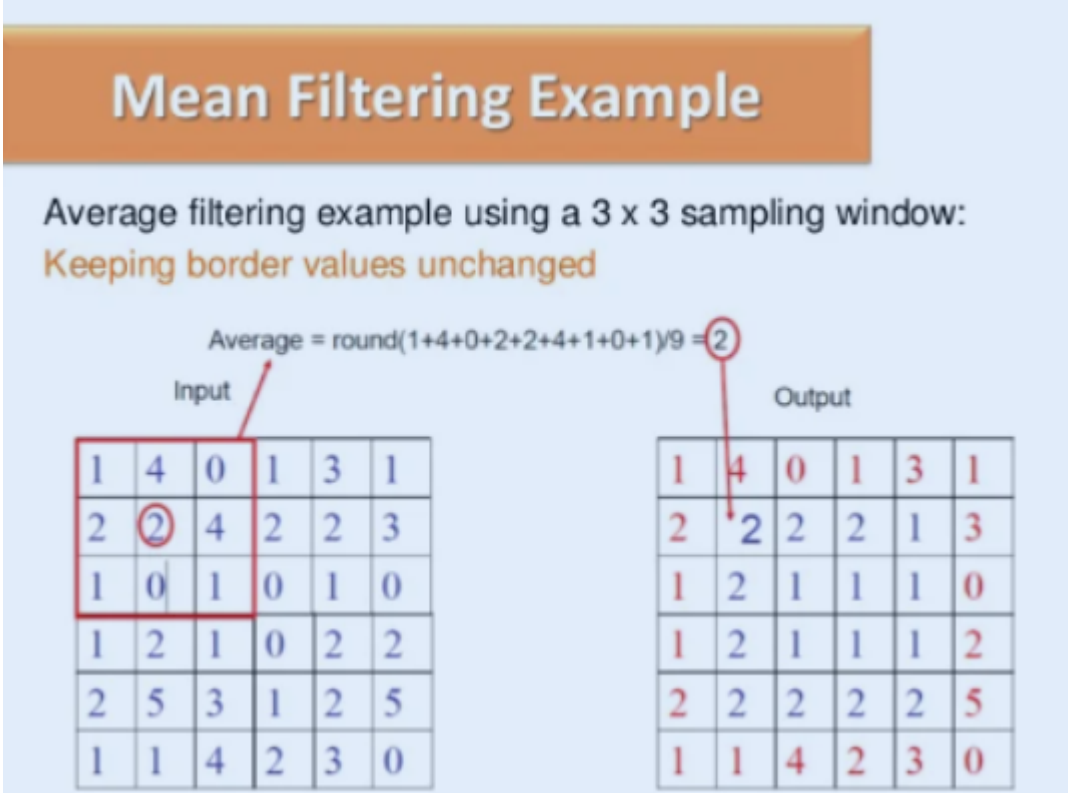
In image processing, blurring is the process of making an image less sharp and reducing its detail. It does this by making color transitions between edges smooth instead of sudden, and by averaging out rapid pixel intensity changes.

It is mainly used to reduce down the noise of the image. There are four ways we can perform smoothing operations,

- Mean Blur
- Median Blur
- Gaussian Blur
- Bilateral Filter

Mean Blur

It is the simplest case of smoothing operation. This operation takes the average of pixels and replace the central pixel with this average.



In [ ]:
```
img = cv2.imread("data\\01_raw_data\\sample_image1.jpg")
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

plt.figure()
plt.imshow(img)
plt.axis('off')
plt.show()
```

In [ ]:
```
# creating mean filter kernel
def meankernel(size):
    mk = np.ones((size, size), dtype = np.float32)/(size**2)
    return mk

# performing mean blur
for size in range(3, 14, 2):
    blur_img = cv2.filter2D(img , -1, meankernel(size))
    plt.figure()
    plt.imshow(blur_img)
    plt.title(f"Mean Blured Image after {size}X{size} kernel operation")
    plt.axis('off')
    plt.show()
```

Mean Blured Image after 3X3 kernel operation

Mean Blured Image after 5X5 kernel operation

Mean Blured Image after 7X7 kernel operation

Mean Blured Image after 9X9 kernel operation

Mean Blured Image after 11X11 kernel operation

Mean Blured Image after 13X13 kernel operation

In [ ]:
```
for size in range(3, 14, 2):
    blur_img = cv2.blur(img, (size, size))
    plt.figure()
    plt.imshow(blur_img)
    plt.title(f"Mean Blured Image after {size}X{size} kernel operation")
    plt.axis('off')
    plt.show()
```

Mean Blured Image after 3X3 kernel operation

Mean Blured Image after 5X5 kernel operation

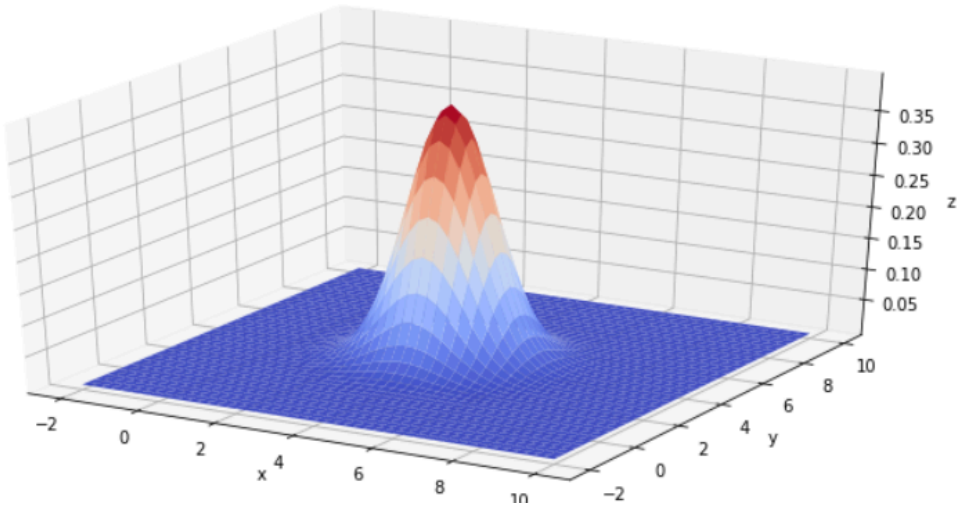
Mean Blured Image after 7X7 kernel operation

Mean Blured Image after 9X9 kernel operation

Mean Blured Image after 11X11 kernel operation

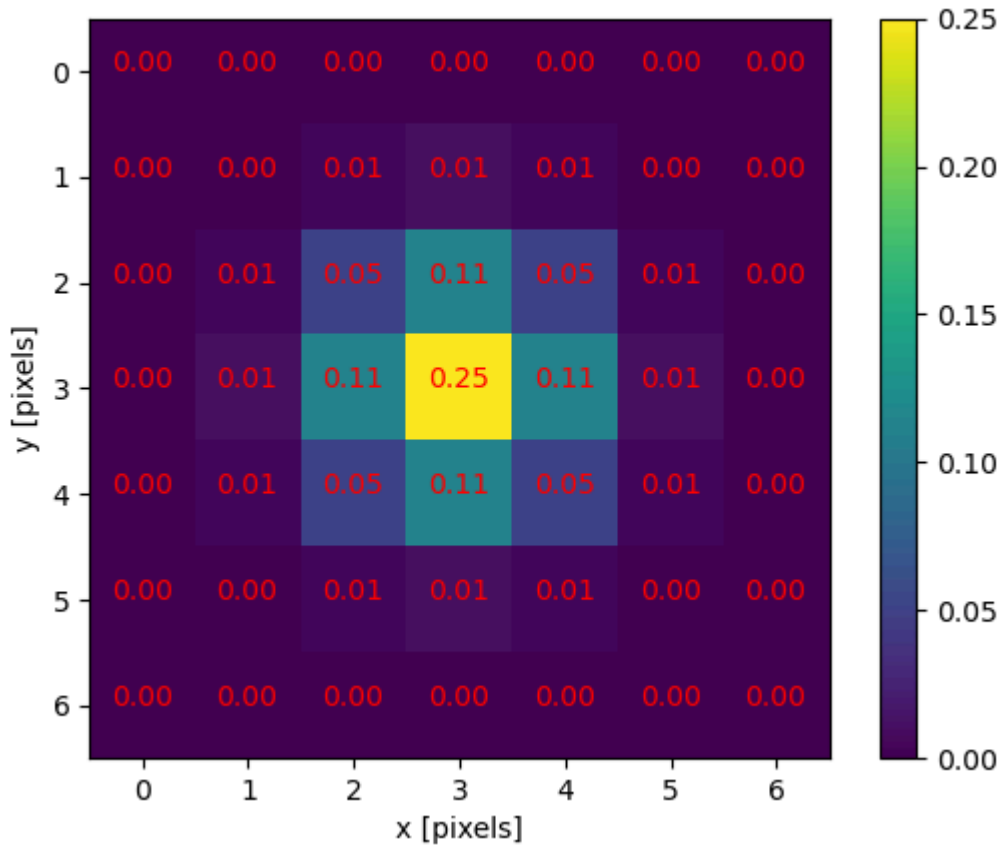
Gaussian Blur

It is a method of blurring the image through the use of Gaussian Function.



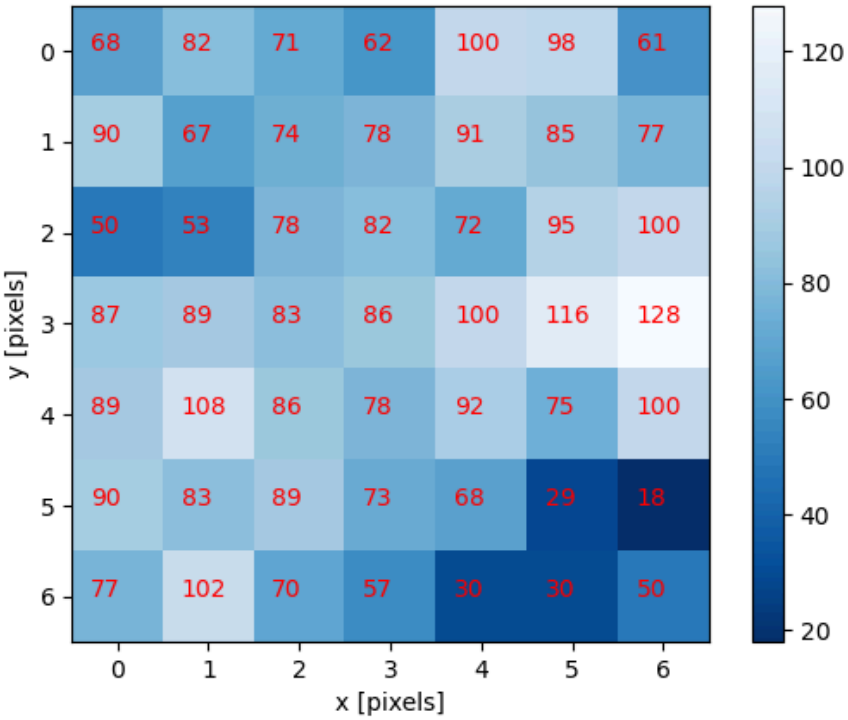
Imagine that this distribution is superimposed over a group of pixels in an image. It should be apparent looking at this graph, that if we took a weighted average of a pixel's values and the height of the curve at that point, the pixels in the center of the group would contribute most significantly to the resulting value.

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

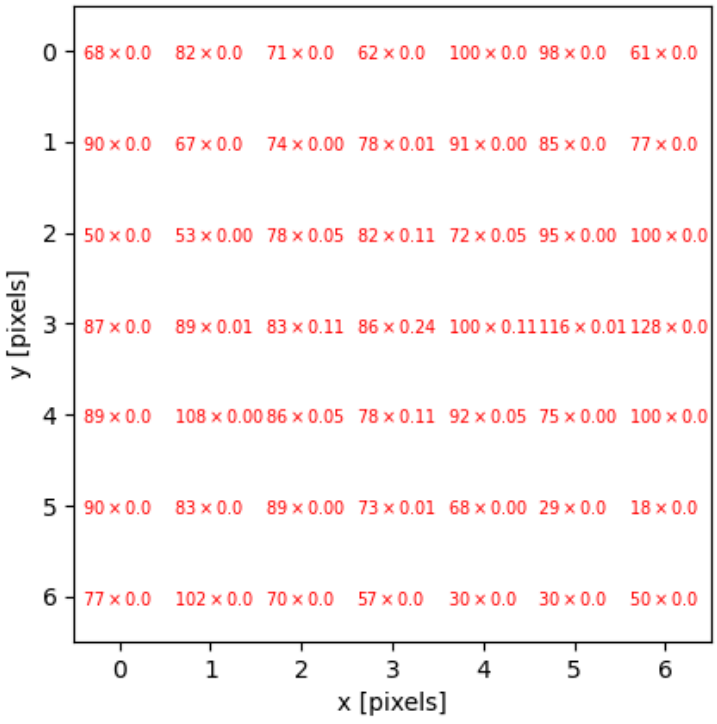


Imagine that plot laid over the kernel for the Gaussian blur filter. The height of the plot corresponds to the weight given to the underlying pixel in the kernel. I.e., the pixels close to the centre become more important to the filtered pixel colour than the pixels close to the outer limits of the kernel. The shape of the Gaussian function is controlled via its standard deviation, or sigma. A large sigma value results in a flatter shape, while a smaller sigma value results in a more pronounced peak.

To illustrate the blurring process, consider the blue channel colour values from the seven-by-seven region of an image :



The filter is going to determine the new blue channel value for the centre pixel – the one that currently has the value 86. The filter calculates a weighted average of all the blue channel values in the kernel giving higher weight to the pixels near the centre of the kernel.



This weighted average, the sum of the multiplications, becomes the new value for the centre pixel (3, 3). The same process would be used to determine the green and red

channel values, and then the kernel would be moved over to apply the filter to the next pixel in the image.

In [ ]:
```python
import numpy as np

def gaussian_kernel(sigma):
    kernel_size = 3
    kernel = np.zeros((kernel_size, kernel_size))
    center = kernel_size // 2
    normalization_factor = 1 / (2 * np.pi * sigma**2)
    # print(center)

    for i in range(kernel_size):
        for j in range(kernel_size):
            # x and y represent the distance of the current pixel from
            # the center of the kernel in the horizontal and vertical directions
            x = i - center
            y = j - center
            kernel[i, j] = normalization_factor*np.exp(-(x**2 + y**2) / (2 * sig

    # Normalize the kernel
    kernel /= np.sum(kernel)

    return kernel

# Example usage
sigma = 0.8
gaussian_kernel_3x3 = gaussian_kernel(sigma)
print("3x3 Gaussian Kernel:")
print(gaussian_kernel_3x3)
```

```
3x3 Gaussian Kernel:
[[0.05711826 0.12475775 0.05711826]
 [0.12475775 0.27249597 0.12475775]
 [0.05711826 0.12475775 0.05711826]]
```

In [ ]:
```python
for size in range(3, 14, 2):
    blur_img = cv2.GaussianBlur(img, (size, size), 0)
    plt.figure()
    plt.imshow(blur_img)
    plt.title(f"Gaussian Blured Image after {size}X{size} kernel operation")
    plt.axis('off')
    plt.show()
```

Gaussian Blured Image after 3X3 kernel operation

Gaussian Blured Image after 5X5 kernel operation

Gaussian Blured Image after 7X7 kernel operation

Gaussian Blured Image after 9X9 kernel operation

Gaussian Blured Image after 11X11 kernel operation

Gaussian Blured Image after 13X13 kernel operation

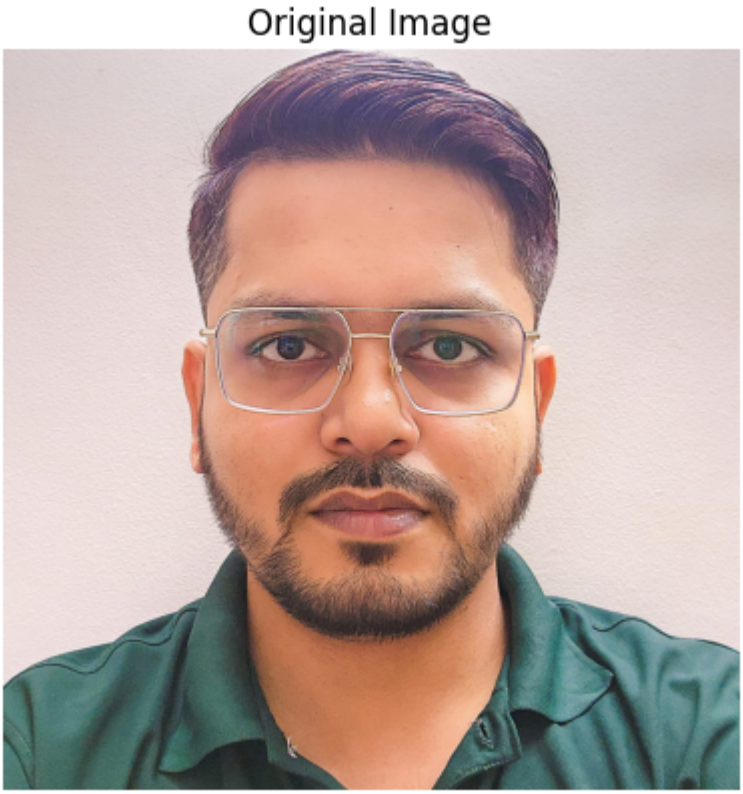
In [ ]:
```
# plt.plot(img.ravel())
```

Histograms

You can consider histogram as a graph or plot, which gives you an overall idea about the intensity distribution of an image. It is a plot with pixel values(ranging from 0 to 255) in X-axis and corresponding number of pixels in the image on Y-axis.

In [ ]:
```
face_img = cv2.imread("data\\01_raw_data\\bibek_face.jpg")
img = cv2.cvtColor(face_img, cv2.COLOR_BGR2RGB)
plt.figure()
plt.imshow(img)
plt.title("Original Image")
```

```
plt.axis("off")
plt.show()
```



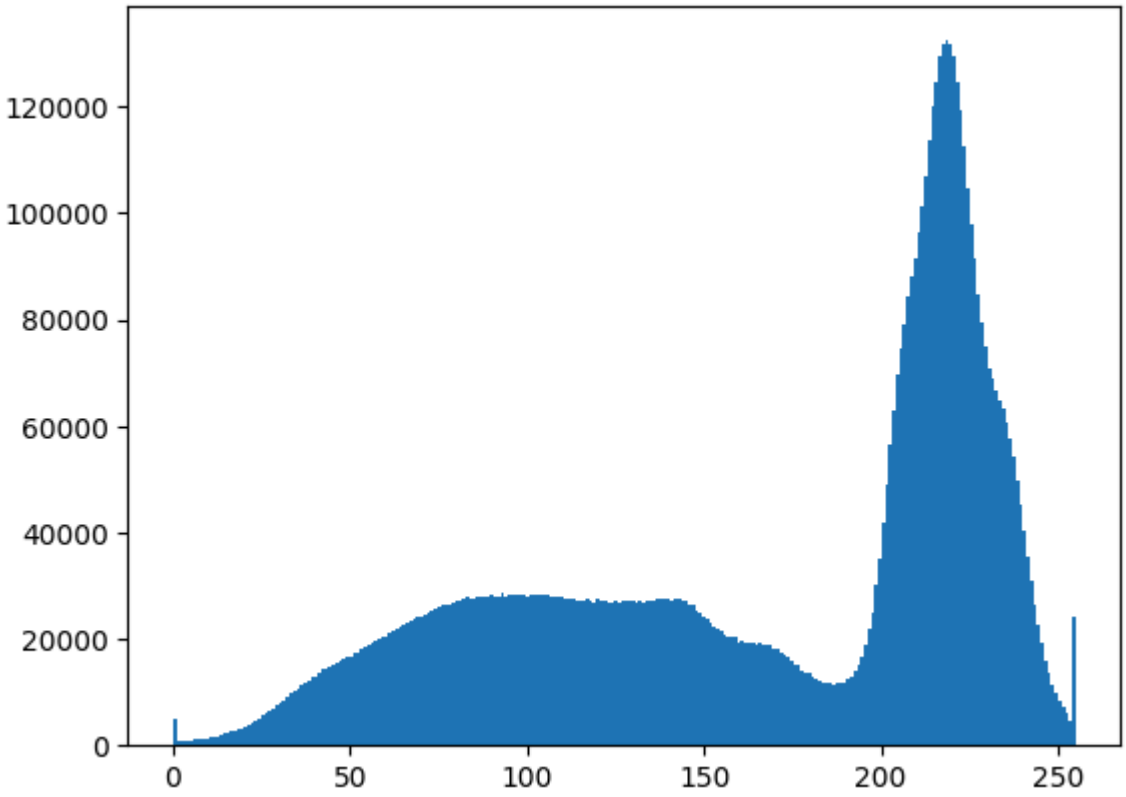
In [ ]:
```
img.shape, img.size
```

Out[ ]:
```
((1600, 1589, 3), 7627200)
```

In [ ]:
```
# shape after flattening the image
img.ravel().shape
```

Out[ ]:
```
(7627200,)
```

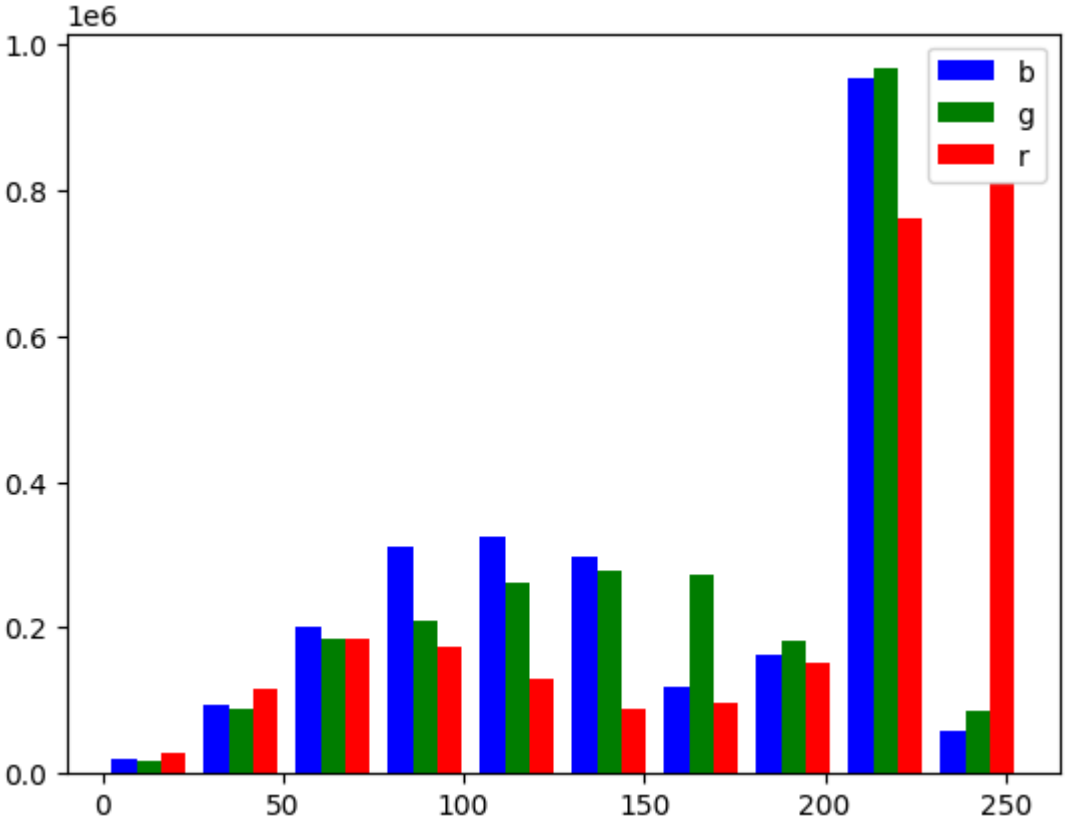
In [ ]:
```
# Histograms
plt.hist(img.ravel(), bins =  256, range=[0, 255])
plt.show()
```



In [ ]:
```
colors = ('b', 'g', 'r')

bgr_img_ravel = [bgr_img[:, :, 0].ravel(), bgr_img[:, :, 1].ravel(), bgr_img[:,:
type(bgr_img_ravel[0])
```

Out[ ]:
```
numpy.ndarray
```

In [ ]:
```
plt.hist(bgr_img_ravel, color=colors, label=colors)
plt.legend()
plt.show()
```



We can use histograms to define the threshold for image segmentation to isolate the background from an object.

Thresholding

Thresholding is a technique in image processing that segments an image into two regions: a foreground region and a background region. It uses pixel values to do this by replacing each pixel with a black pixel if the image intensity is less than a fixed value called the threshold, and with a white pixel if the pixel intensity is greater than the threshold. The threshold can be defined by the user or automatically, and any pixel value outside the range becomes 0, and any pixel value inside the range becomes 1, or a user-defined value. The result is a binary image

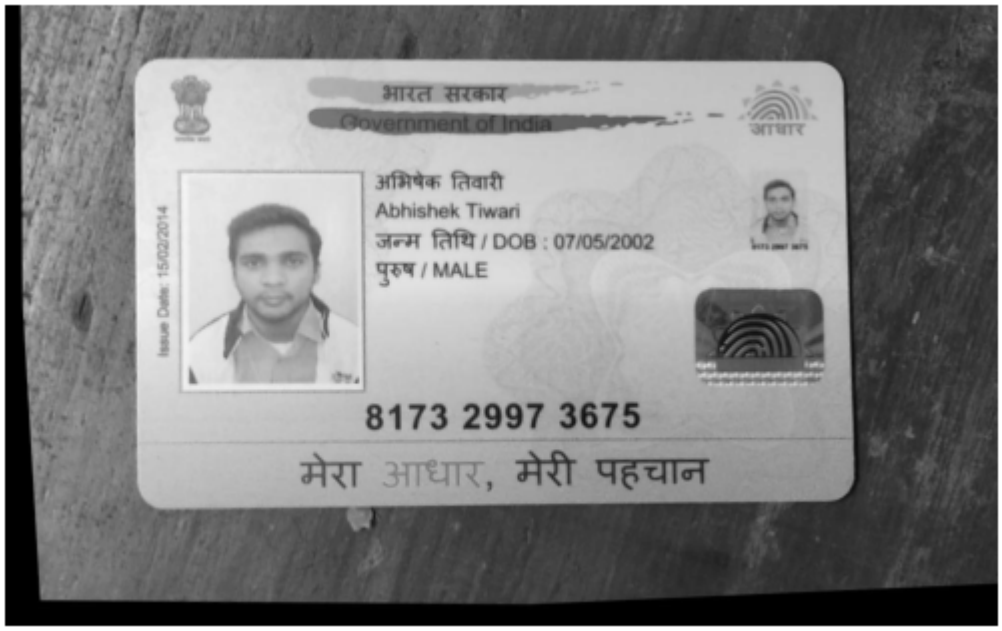
Simple Thresholding

- Where you have to manually supply threshold value.
- Applying simple thresholding methods requires human intervention. We must specify a threshold value T. All pixel intensities below T are set to 255. And all pixel intensities greater than T are set to 0.

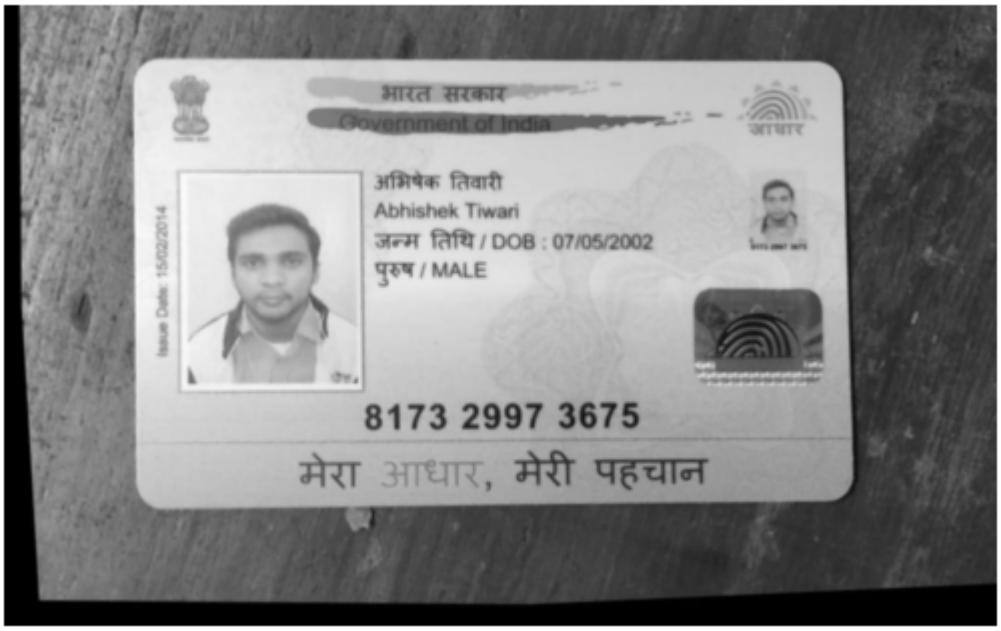
In [ ]:
```
# Reading image
image = cv2.imread("data\\01_raw_data\\sample_image1.jpg")
rgb_image =  cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

plt.imshow(rgb_image)
```

```
plt.axis("off")
plt.show()
```

In [ ]:
```
# Gray Scale Conversion

gray =  cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

plt.imshow(gray, cmap='gray')
plt.axis("off")
plt.show()
```



In [ ]:
```
blurred =  cv2.GaussianBlur(gray, (3, 3), 0)

plt.imshow(blurred, cmap='gray')
plt.axis("off")
plt.show()
```



- After the image is blurred, we compute the thresholded image using cv2.threshold function. This method requires four arguments.
- The first is the grayscale image that we wish to threshold. We supply our blurred image as the first.
- Then, we manually supply our T threshold value. We use a value of T=200.
- Our third argument is the output value applied during thresholding. Any pixel intensity p that is greater than T is set to zero and any p that is less than T is set to the output value:

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

In our example, any pixel value that is greater than 200 is set to 0. Any value that is less than 200 is set to 255.

Finally, we must provide a thresholding method. We use the `cv2.THRESH_BINARY_INV` method, which indicates that pixel values p less than T are set to the output value (the third argument).

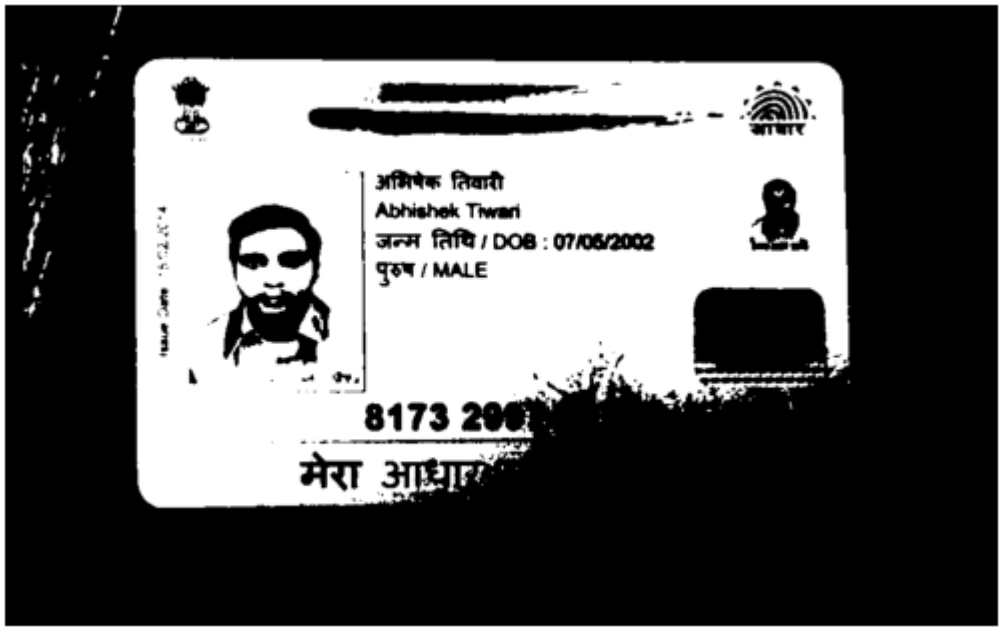
But what if we wanted to perform the reverse operation, like this:

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

Using `cv2.THRESH_BINARY` we can achieve this.

In most cases you are normally seeking your segmented objects to appear as white on a black background, hence using `cv2.THRESH_BINARY_INV`. But in the case that you want your objects to appear as black on a white background, be sure to supply the `cv2.THRESH_BINARY` flag.

In [ ]:
```
T, thresh =  cv2.threshold(blurred, 150, 255, cv2.THRESH_BINARY)
plt.imshow(thresh, cmap='gray')
plt.axis("off")
plt.show()
```



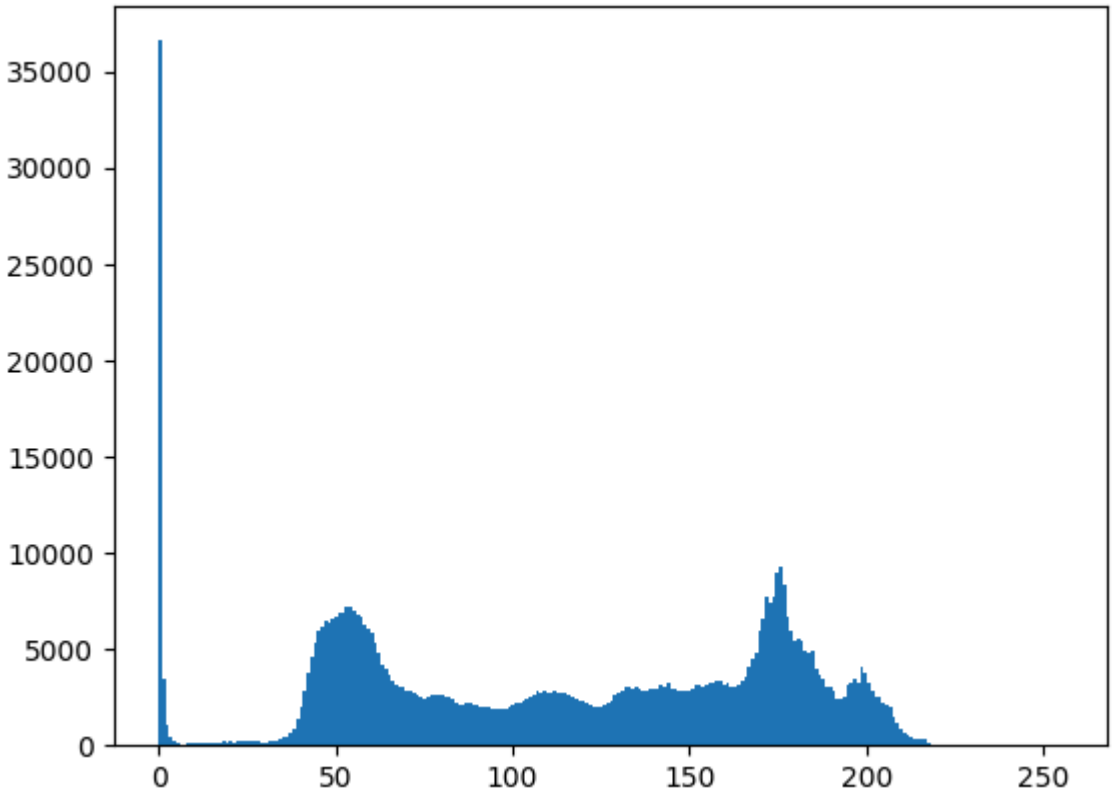
In [ ]:
```
T, threshInv =  cv2.threshold(blurred, 150, 255, cv2.THRESH_BINARY_INV)
plt.imshow(threshInv, cmap='gray')
plt.axis("off")
plt.show()
```

Otsu Thresholding

In the previous section on simple thresholding we needed to manually supply a threshold value of T. For simple images in controlled lighting conditions, it might be feasible for us to hardcode this value.

But in real-world conditions where we do not have any a priori knowledge of the lighting conditions, we actually automatically compute an optimal value of T using Otsu's method.

In [ ]:
```
# Histograms
plt.hist(gray.ravel(), bins =  256, range=[0, 255])
plt.show()
```



Otsu's method makes the assumption that the grayscale histogram of our pixel intensities of our image is bi-modal, which simply means that the histogram is two peaks.

Notice how the histogram clearly has two peaks — the first sharp peak corresponds to the uniform background color of the image, while the second peak corresponds to the card region itself

In [ ]:
```
(T, otsu_threshInv) = cv2.threshold(blurred, 0, 255,
        cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU)

print("[INFO] otsu's thresholding value: {}".format(T))

plt.imshow(otsu_threshInv, cmap='Greys_r')
plt.axis("off")
plt.show()
```

```
[INFO] otsu's thresholding value: 110.0
```

one of the downsides of using simple thresholding methods is that we need to manually supply our threshold value, T. Furthermore, finding a good value of T may require many manual experiments and parameter tunings, which is simply not practical in most situations.

To aid us in automatically determining the value of T, we leveraged Otsu's method. And while Otsu's method can save us a lot of time playing the "guess and checking" game, we are left with only a single value of T to threshold the entire image.

For simple images with controlled lighting conditions, this usually isn't a problem. But for situations when the lighting is non-uniform across the image, having only a single value of T can seriously hurt our thresholding performance.

Adaptive Thresholding

Goal in adaptive thresholding is to statistically examine local regions of our image and determine an optimal value of T for each region — which begs the question: Which statistic do we use to compute the threshold value T for each region?

It is common practice to use either the `arithmetic mean` or the `Gaussian mean` of the pixel intensities in each region (other methods do exist, but the arithmetic mean and the Gaussian mean are by far the most popular).

In the arithmetic mean, each pixel in the neighborhood contributes equally to computing T. And in the Gaussian mean, pixel values farther away from the (x, y)-coordinate center of the region contribute less to the overall calculation of T.

In [ ]:
```
adp_thresh = cv2.adaptiveThreshold(blurred, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2

plt.imshow(adp_thresh, cmap='Greys_r')
plt.axis("off")
plt.show()
```

The third argument is the adaptive thresholding method. Here we supply a value of `cv2.ADAPTIVE_THRESH_MEAN_C` to indicate that we are using the arithmetic mean of the local pixel neighborhood to compute our threshold value of T.

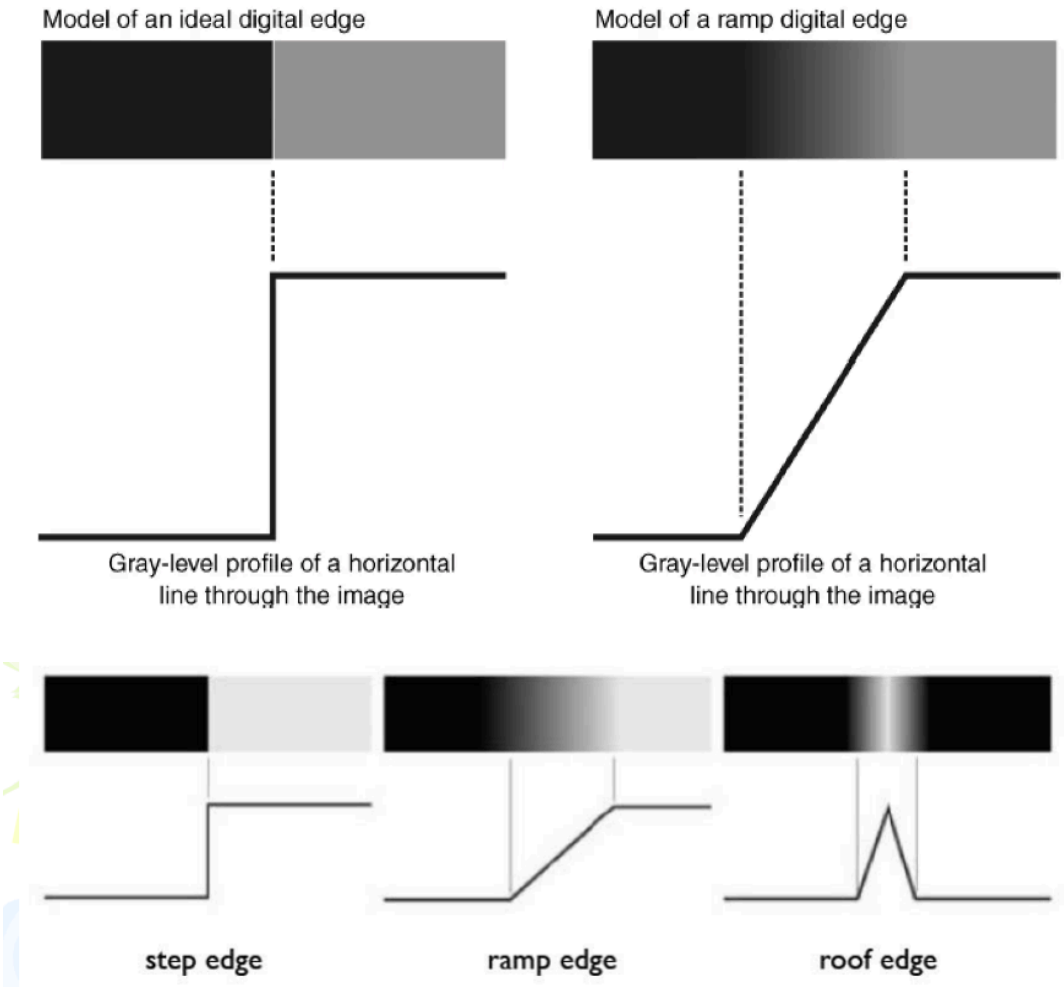
We could also supply a value of `cv2.ADAPTIVE_THRESH_GAUSSIAN_C` (which we'll do next) to indicate we want to use the Gaussian average — which method you choose is entirely dependent on your application and situation, so you'll want to play around with both methods.

In [ ]:
```
thresh = cv2.adaptiveThreshold(blurred, 255,
        cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY_INV, 21, 4)
plt.imshow(adp_thresh, cmap='Greys_r')
plt.axis("off")
plt.show()
```

Edge Detection

- An edge is a location in an image where there is a rapid change in intensity.
- We convert a 2D image(here it's a binary image) in a set of points where image intensity changes rapidly.
- Edges are very much important in terms of conveying or capturing visual information of the image.

Below are the examples for types of edges,

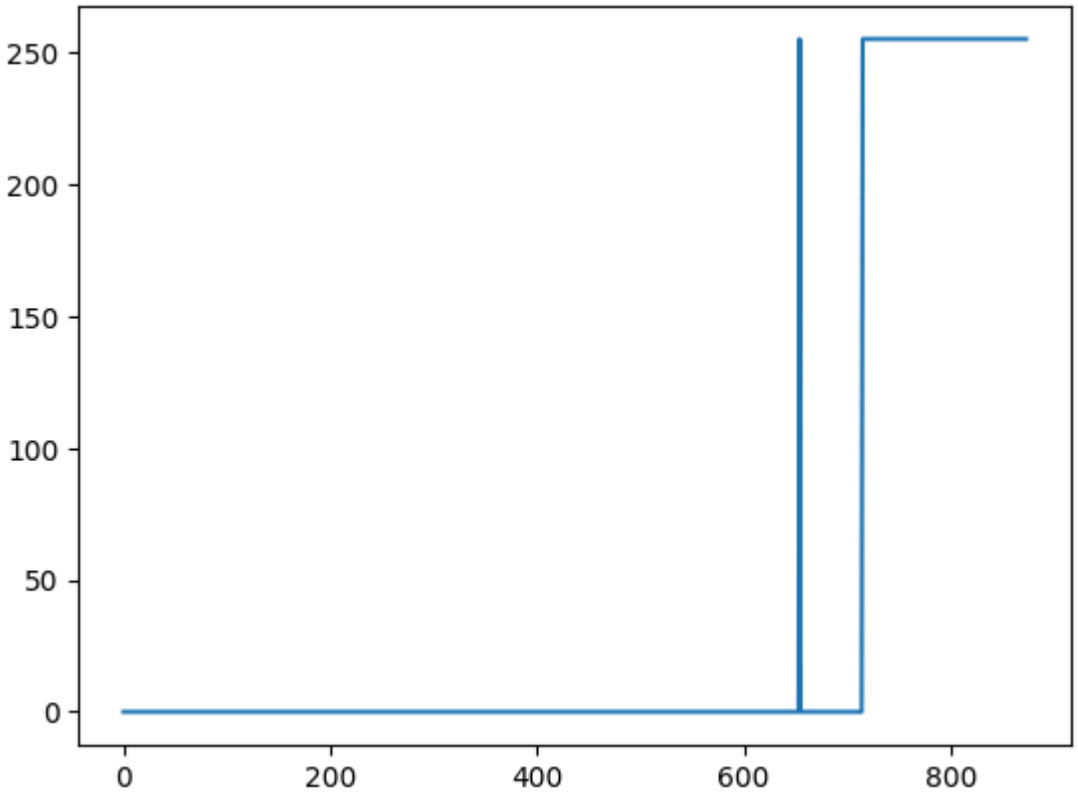


In [ ]:
```
otsu_threshInv.shape
```

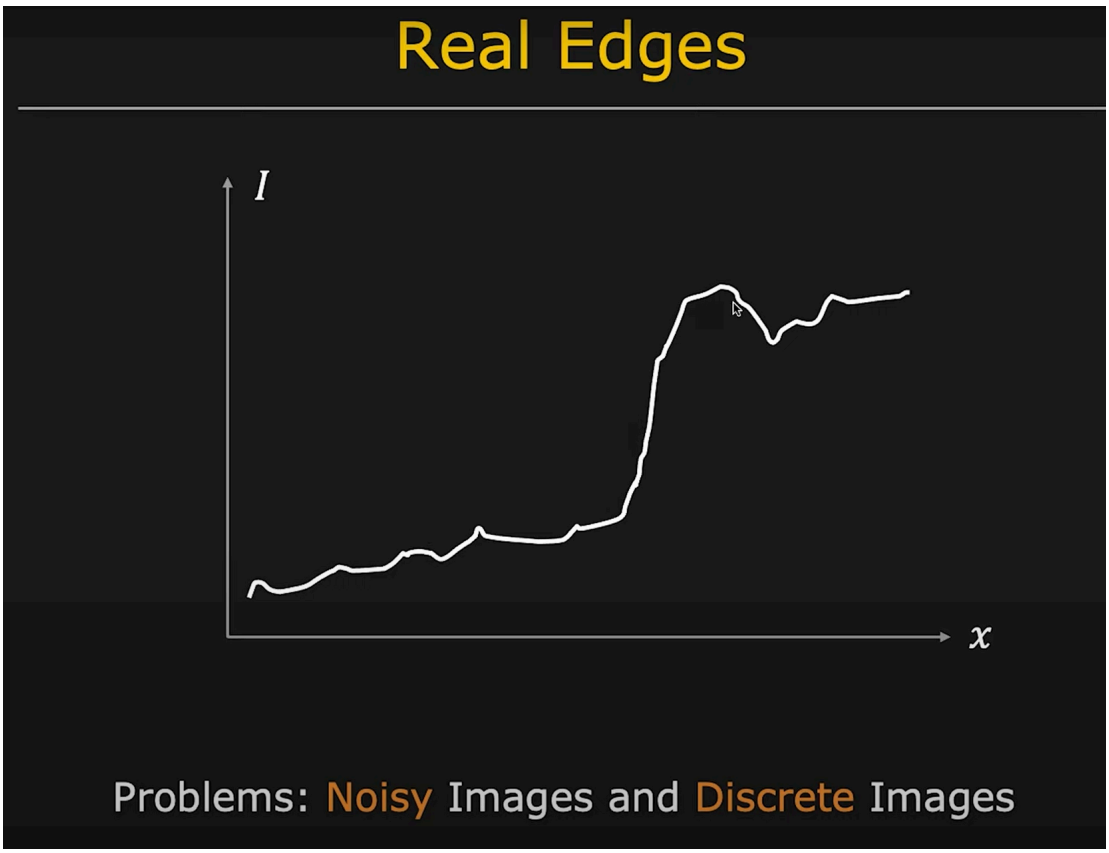
Out[ ]:
```
(640, 1024)
```

In [ ]:
```
plt.imshow(otsu_threshInv, cmap = 'gray')
plt.axis('off')
plt.show()
```

In [ ]:
```
horiz_pixels = otsu_threshInv[100, 150:]

plt.plot(horiz_pixels)
plt.show()
```



In real time due to noise in the image, edges look like this



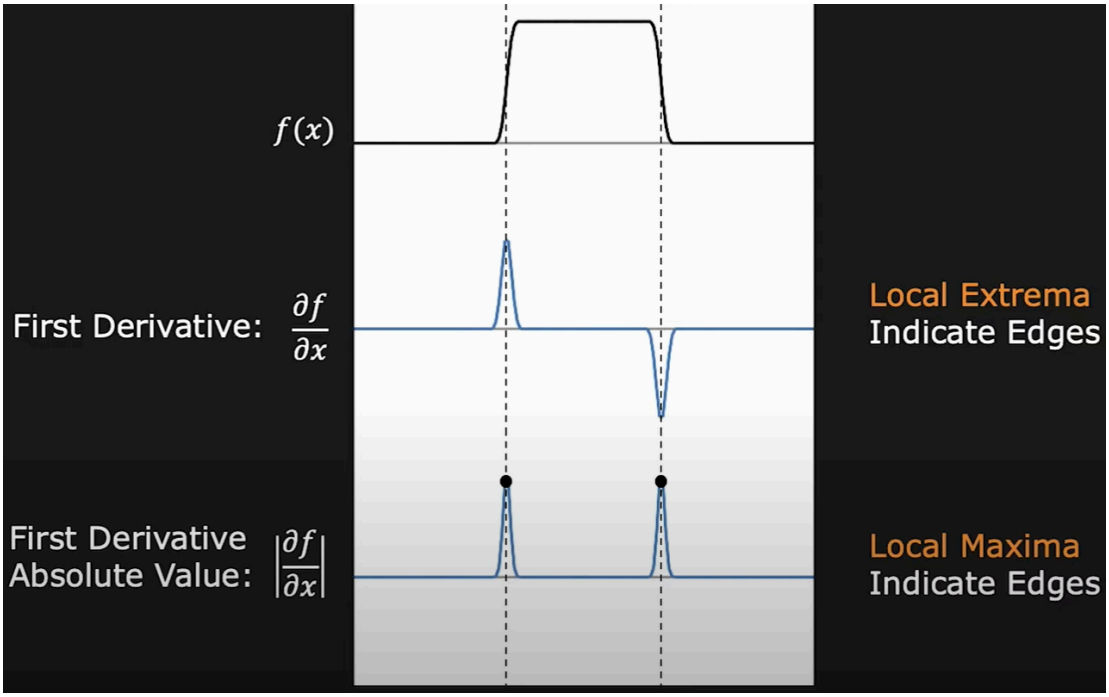
We want our `Edge Detector` to provide below informations:

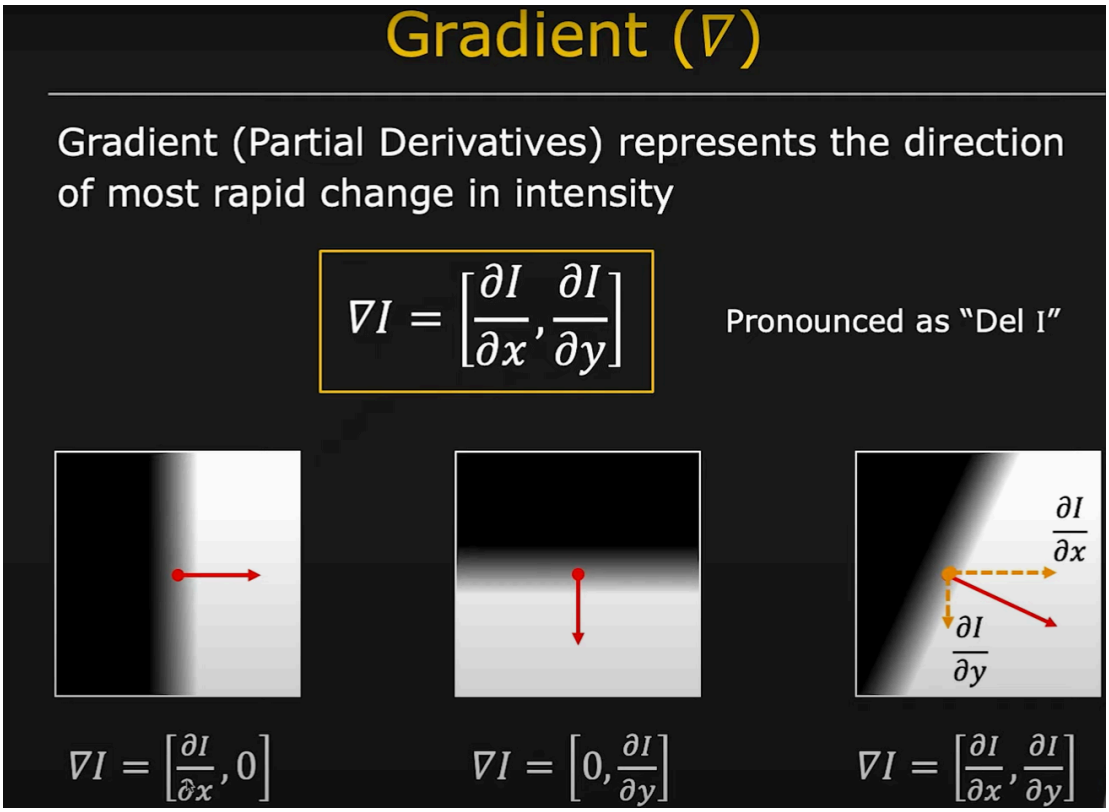
- Edge Position
- Edge Magnitude
- Edge Orientation

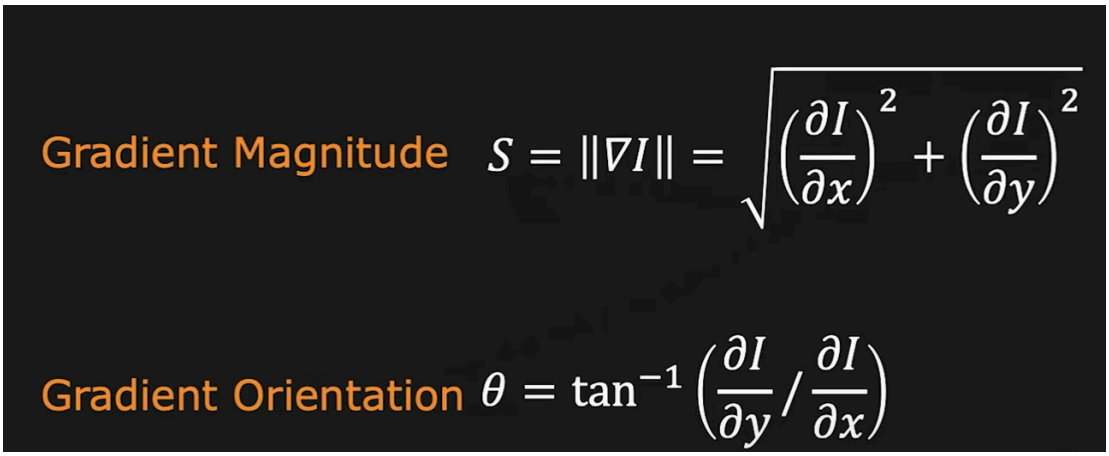
Edge Detection using Gradient

`1D Edge Detection`

We understand that an edge represents a rapid change in image intensity within a small region. To understand the changes in pixel intensity, we can use the first-order derivative, which indicates the rate of change in a function.

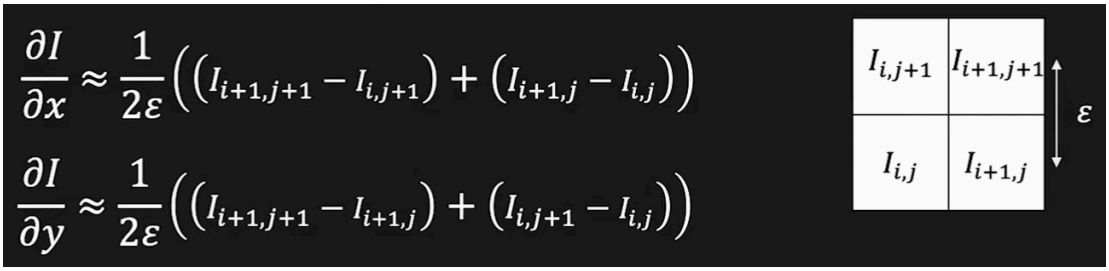

$f(x)$
First Derivative: $\frac{\partial f}{\partial x}$
Local Extrema
Indicate Edges
First Derivative
Absolute Value: $\left|\frac{\partial f}{\partial x}\right|$
Local Maxima
Indicate Edges

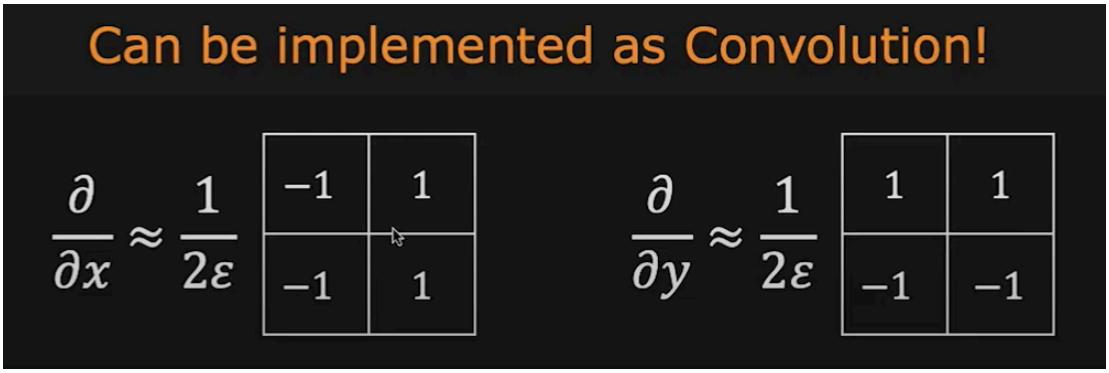




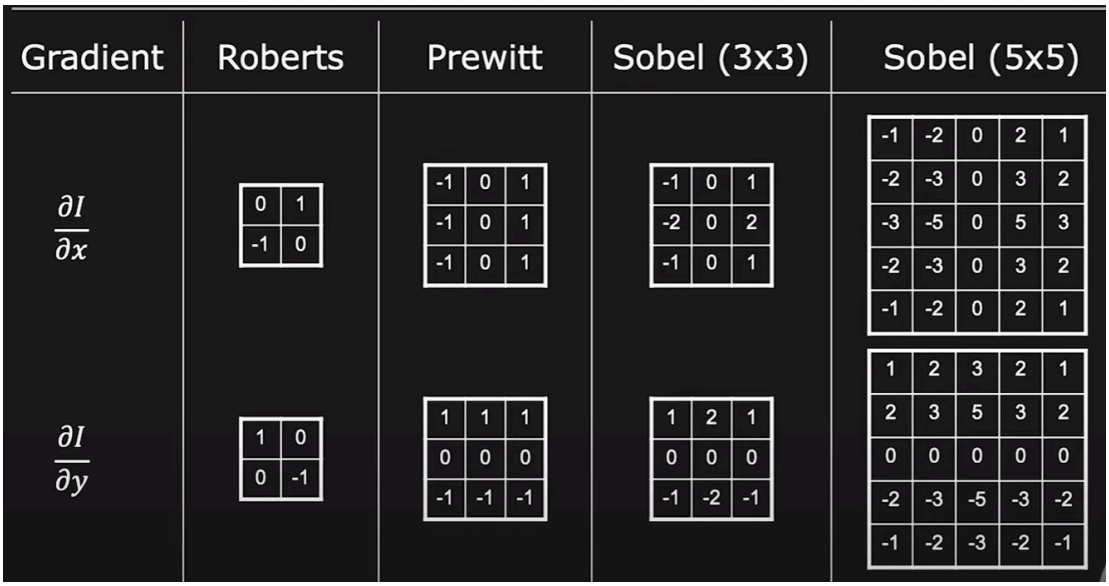
`Discrete Gradient Operator`

epsilon is physical distance between 2 pixels





When we convolve the image with these 2 kernels it gives us the strength and orientation of edge along x and y axis.



When these kernels are convolved with the original image, you get a 'Edge Image'.

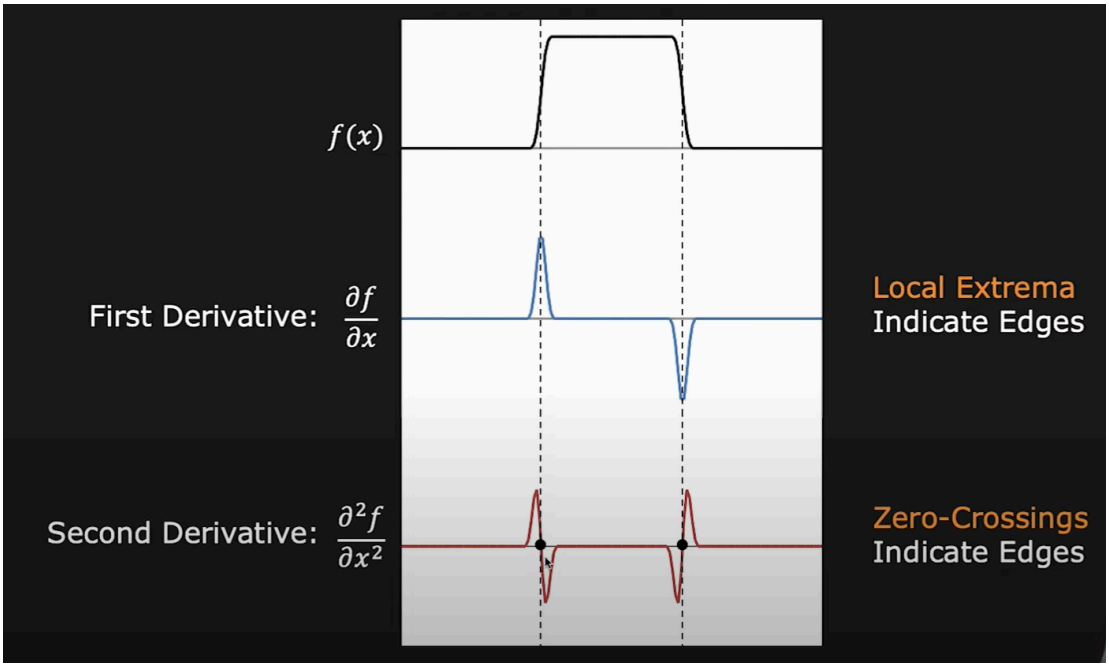
If we use only the Vertical Kernel, the convolution yields a Sobel image, with edges enhanced in the X-direction Using the Horizontal Kernel yields a Sobel image, with edges enhanced in the Y-direction.

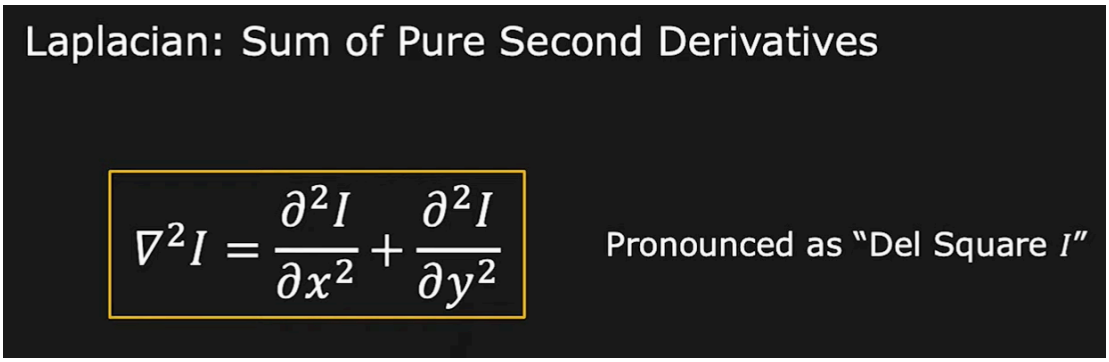
In [ ]:
```
#  Sobel Edge Detection
sobelx = cv2.Sobel(src=otsu_threshInv, ddepth=cv2.CV_64F, dx=1, dy=0, ksize=5) #
sobely = cv2.Sobel(src=otsu_threshInv, ddepth=cv2.CV_64F, dx=0, dy=1, ksize=5) #
sobelxy = cv2.Sobel(src=otsu_threshInv, ddepth=cv2.CV_64F, dx=1, dy=1, ksize=5)

# Display Sobel Edge Detection Images
cv2.imshow('Sobel X', sobelx)
cv2.waitKey(0)


cv2.imshow('Sobel Y', sobely)
cv2.waitKey(0)

cv2.imshow('Sobel X Y using Sobel() function', sobelxy)
cv2.waitKey(0)
cv2.destroyAllWindows()

# plt.imshow(sobelx, cmap = 'gray')
# plt.axis('off')
```

```
# plt.title('Sobel X')
# plt.show()
```

Edge Detection using Laplacian

Like 1st order derivative here we will not get peaks at the edges. Instead we have zeros, wherever we have zero-crossing we have an edge.





- Edges are "zero-crossings" in Lapacian of image.
- Laplacian does not provide direction of edges.

Canny Edge Detection

This algorithm follows a three-stage process for extracting edges from an image. Add to it image blurring, a necessary preprocessing step to reduce noise. This makes it a four-stage process, which includes:

```
- Noise Reduction
- Calculating the Intensity Gradient of the Image
- Suppression of False Edges
- Hysteresis Thresholding
```

`Suppression of False Edges:`

After reducing noise and calculating the intensity gradient, the algorithm in this step uses a technique called non-maximum suppression of edges to filter out unwanted pixels (which may not actually constitute an edge). To accomplish this, each pixel is compared to its neighboring pixels. If the gradient magnitude of the current pixel is greater than its neighboring pixels, it is left unchanged. Otherwise, the magnitude of the current pixel is set to zero.

`Hysteresis Thresholding`

In this final step of Canny Edge Detection, the gradient magnitudes are compared with two threshold values, one smaller than the other.

- If the gradient magnitude value is higher than the larger threshold value, those pixels are associated with solid edges and are included in the final edge map.
- If the gradient magnitude values are lower than the smaller threshold value, the pixels are suppressed and excluded from the final edge map.
- All the other pixels, whose gradient magnitudes fall between these two thresholds, are marked as 'weak' edges (i.e. they become candidates for being included in the final edge map).
- If the 'weak' pixels are connected to those associated with solid edges, they are also included in the final edge map.

In [ ]:
```
# Canny Edge Detection
edges = cv2.Canny(image=blurred, threshold1=100, threshold2=200)

# Display Canny Edge Detection Image
cv2.imshow('Canny Edge Detection', edges)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Contour Detection

- Contour is a boundary around something that has well defined edges, so the machine is able to calculate difference in gradient, and form a recognisable shape through continuing chnage and draw a boundary around it.
- When we join all the points on the boundary of an object, we get a contour. Typically, a specific contour refers to boundary pixels that have the same color and intensity

`cv2.findContours()`

- image: The binary input image.
- mode: Contor-retrieval mode.
- method: Contour-approximation method.

Retrieval Modes

Function `cv2.findContours()` does not only returns the contours found in an image but also returns valuable information about the hierarchy of the contours in the image. The hierarchy encodes how the contours may be arranged in the image, e.g, they may be nested within another contour. Often we are more interested in some contours than others. For example, you may only want to retrieve the external contour of an object.

Using the Retrieval Modes specified, the `cv2.findContours()` function can determine how the contours are to be returned or arranged in a hierarchy. For more information on Retrieval modes and contour hierarchy Read here.

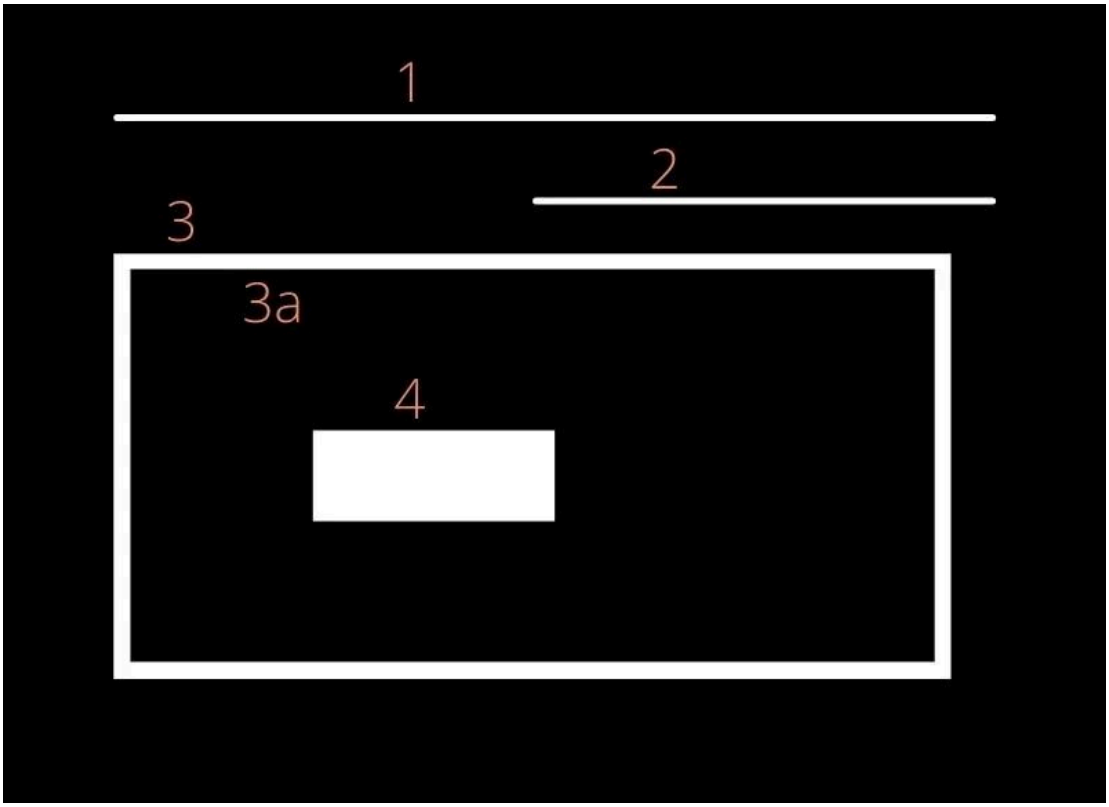
Some of the important retrieval modes are:

- `cv2.RETR_EXTERNAL` - retrieves only the extreme outer contours.
- `cv2.RETR_LIST` - retrieves all of the contours without establishing any hierarchical relationships.
- `cv2.RETR_TREE` - retrieves all of the contours and reconstructs a full hierarchy of nested contours.
- `cv2.RETR_CCOMP` - retrieves all of the contours and organizes them into a two-level hierarchy. At the top level, there are external boundaries of the components. At the second level, there are boundaries of the holes. If there is another contour inside a hole of a connected component, it is still put at the top level.

Hierarchies denote the parent-child relationship between contours.

Now see below figure, where the contours associated with each shape in Figure 10 have been identified. Each of the numbers in Figure 11 have a significance.

- All the individual numbers, i.e., 1, 2, 3, and 4 are separate objects, according to the contour hierarchy and parent-child relationship.
- We can say that the 3a is a child of 3. Note that 3a represents the interior portion of contour 3.
- Contours 1, 2, and 4 are all parent shapes, without any associated child, and their numbering is thus arbitrary. In other words, contour 2 could have been labeled as 1 and vice-versa.



Compression Methods

Detected contours can be compressed to reduce the number of points. In this sense, OpenCV provides sevaral methods to reduce the number of points. This can be set with the parameter method.

- `CHAIN_APPROX_SIMPLE` : This is faster, approximates the contour and compresses horizontal, vertical, and diagonal segments, keeping only their end points.
- `CHAIN_APPROX_NONE` : This retains all the original contour points, making it suitable when precise contour representation is required, but it might be slower and consume more memory compared to CHAIN_APPROX_SIMPLE.

In [ ]:
```
contours, _ = cv2.findContours(adp_thresh, mode = cv2.RETR_EXTERNAL, method = cv

# Select the largest contour (assuming the ID card is the largest object)
largest_contour = None
largest_area = 0
for cnt in contours:
    area = cv2.contourArea(cnt)
    if area > largest_area:
        largest_contour = cnt
        largest_area = area

x, y, w, h = cv2.boundingRect(largest_contour)

print("contours", (x, y, w, h))
print("Area", largest_area)
contour_id = rgb_image[y:y+h, x:x+w]

plt.imshow(contour_id)
```

```
plt.axis('off')
plt.show()
```

```
contours (133, 55, 743, 465)
Area 335355.0
```

In [ ]:
```
contours_h, hr = cv2.findContours(adp_thresh, mode = cv2.RETR_TREE, method = cv2

hr[0]
```

Out[ ]:
```
array([[   1,   -1,   -1,   -1],
       [   2,    0,   -1,   -1],
       [   3,    1,   -1,   -1],
       ...,
       [2345, 2343,   -1, 1349],
       [  -1, 2344,   -1, 1349],
       [  -1, 1349,   -1,   -1]], dtype=int32)
```