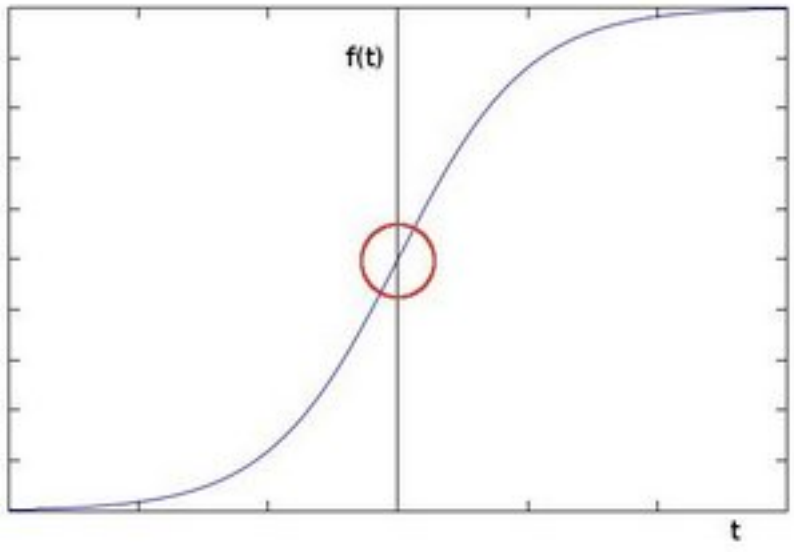
Goal

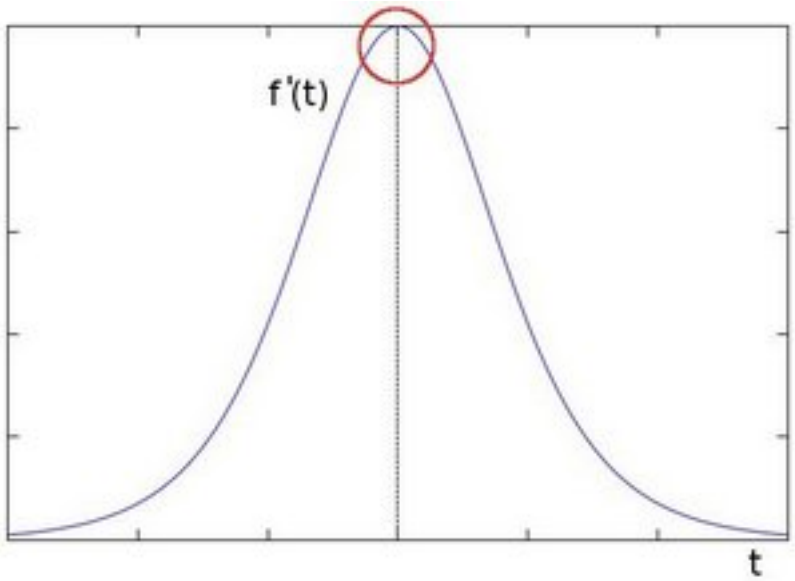
In this chapter we will learn:
* How to use various gradient operators to detect the edges.
* We will see : **cv2.Sobel()**, **cv2.Scharr()**, **cv2.Laplacian()**

Theory

We have seen that LPF(low pass filters) tend to blur the edges. You can easily notice that in an edge, the pixel intensity changes in a notorious way. A good way to express changes is by using derivatives. A high change in gradient indicates a major change in the image.
To be more graphical, let's assume we have a 1D-image. An edge is shown by the "jump" in intensity in the plot below:



However if we take the first derivative we can see the edge peak more clearly as shown below:



Hence we can detect edges in an image by locating pixel locations where the gradient is higher than its neighbors (or to generalize, higher than a threshold). We're going to look into two commonly used edge detection schemes - the gradient (**Sobel** - first order derivatives) based edge detector and the **Laplacian** (2nd order derivative, so it is extremely sensitive to noise) based edge detector. Both of them work with convolutions and achieve the same end goal - *Edge Detection*.

Sobel Edge Detection

The Sobel operator, sometimes called Sobel Filter is a discrete differentiation operator, computing an approximation of the gradient of the image intensity function.
Sobel edge detector is a gradient based method based on the first order derivatives. It calculates the first derivatives of the image separately for the X and Y axes. The operator uses two 3X3 kernels which are convolved with the original image to calculate approximations of the derivatives - one for horizontal changes, and one for vertical. The picture below shows Sobel Kernels in x-dir and y-dir:

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

Scharr

Working Principle

Scharr operator is used to find image gradients(or edge detection). This operator uses two kernels to convolve the image and calculate derivatives in two directions. The derivatives track changes in horizontal as well as vertical directions. Scharr operator tries to overcome Sobel operator's drawback of not having perfect rotational symmetry. The commonly used filter kernels are -

$$\begin{bmatrix} +3 & +10 & +3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix} \qquad \begin{bmatrix} +3 & 0 & -3 \\ +10 & 0 & -10 \\ +3 & 0 & -3 \end{bmatrix}$$

Figure 1: Scharr kernel

The main function used for Scharr operator is-

```
cv2.Scharr(src, ddepth, dx, dy[, dst[, scale[, delta[, borderType]]]])
```

where * **src** : Input image * **ddepth** : Output image depth. Following are the compatible depths-

src.depth	ddepth
CV_8U	1/CV_16S/CV_32F/CV_64F
CV_16U/CV_16S	-1/ CV_32F/CV_64F
CV_32F	-1/ CV_32F/CV_64F
CV_64F	-1/CV_64F

- **dst** : Output image of the same size and same number of channels as src.
- **scale** : Optional scale factor for computed derivative values. No scaling is applied by default.

- **delta** : Optional delta value that is added to results prior to storing them in dst.
- **borderType**: Pixel extrapolation method

Example

Consider the following image-

Figure 2: Example house

On applying Scharr operator on above image, we get the following-

Figure 3: Scharr image

Laplacian Edge Detection

The Laplacian edge detector uses only one kernel. It calculates second order derivatives in a single pass. The Laplacian function is given below:

$$\mathtt{dst}(x,y) = \frac{d^2\mathtt{src}}{dx^2} + \frac{d^2\mathtt{src}}{dy^2}$$

A kernel used in this Laplacian detection looks like this:

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

Applications

- These operators can be used to extract features from an image.
- They used for edge detection, which is the basis for most image processing applications.

Code

Sobel and Laplacian

```
#Import modules
import cv2
import numpy as np
import matplotlib.pyplot as plt

#Defining variables
scale = 1
delta = 0
ddepth = cv2.CV_64F

#Read the image
img = cv2.imread('lamp.jpg')

'''
OpenCV represents RGB images as multi-dimensional NumPy arrays...but in reverse
order!This means that images are actually represented in BGR order rather than
```

```
RGB!
'''
#Convert to RGB
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

#Convert to grayscale
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

#Applying threshold
ret, thresh = cv2.threshold(gray,127,255, cv2.THRESH_BINARY)

# Gradient-X
grad_x = cv2.Sobel(thresh,ddepth,1,0,ksize = 3, scale = scale,
delta = delta,borderType = cv2.BORDER_DEFAULT)

# Gradient-Y
grad_y = cv2.Sobel(thresh,ddepth,0,1,ksize = 3, scale = scale,
delta = delta, borderType = cv2.BORDER_DEFAULT)

#Laplacian
lap = cv2.Laplacian(thresh,ddepth)

# converting back to uint8
abs_grad_x = cv2.convertScaleAbs(grad_x)
abs_grad_y = cv2.convertScaleAbs(grad_y)

#Finding the weighted mean
Sobel = cv2.addWeighted(grad_x,0.5,grad_y,0.5,0)

#Plotting the images
plt.subplot(3,2,1),plt.imshow(img,cmap = 'gray')
plt.title('Original'), plt.xticks([]), plt.yticks([])
plt.subplot(3,2,2),plt.imshow(grad_x,cmap = 'gray')
plt.title('Sobel x'), plt.xticks([]), plt.yticks([])
plt.subplot(3,2,3),plt.imshow(grad_y,cmap = 'gray')
plt.title('Sobel y'), plt.xticks([]), plt.yticks([])
plt.subplot(3,2,4),plt.imshow(Sobel,cmap = 'gray')
plt.title('Sobel'), plt.xticks([]), plt.yticks([])
plt.subplot(3,2,5),plt.imshow(lap,cmap = 'gray')
plt.title('Laplacian'), plt.xticks([]), plt.yticks([])

#Display the window
plt.show()
```

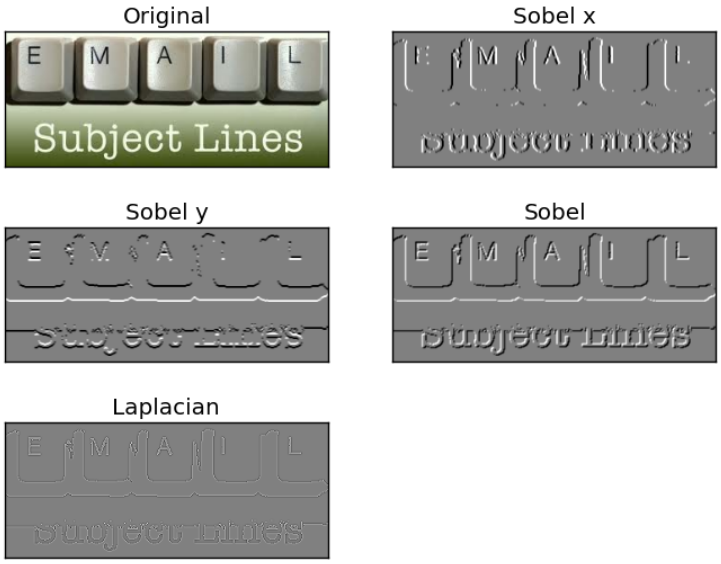


Figure 4: Output Image

Scharr

- Let us first import opencv package and read our image.

```
import cv2
img = cv2.imread('example.jpg')
```

- Now, we remove noise from image using Gaussian Blur and then convert it to grayscale.

```
img = cv2.GaussianBlur(img, (3,3), 0)
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

- The next step is to find gradients in x- and y directions.

```
grad_x = cv2.Scharr(gray, cv2.CV_16S, 1, 0) #Gradient X
grad_y = cv2.Scharr(gray, cv2.CV_16S, 0, 1) #Gradient Y
```

- The gradients should now be converted back to unsigned 8-bit integer form.

```
abs_grad_x = cv2.convertScaleAbs(grad_x)
abs_grad_y = cv2.convertScaleAbs(grad_y)
```

- We can now obtain our Scharr derivative by adding these two gradients.

```
scharr = cv2.add(abs_grad_x, abs_grad_y)
```

- Finally, we display our result.

```
cv2.imshow('Scharr Derivative', scharr)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

References

1. Gradients - opencv and python tutorials
2. Gradients - More info
3. Sobel Operator
4. OpenCv docs

Excercises