**Sobel Filter:**

Sobel filtering involves applying two 3 x 3 convolutional kernels (also called filters) to an image. The kernels are usually called *Gx* and *Gy*. These two kernels detect the edges in the image in the horizontal and vertical directions. They are applied separately and then combined to produce a pixel value in the output image at each position in the input image. Sobel filtering is usually used for edge detection in images.

The Sobel operator gives you the gradient in x or y direction. For Sobel-based edge detection, you have to make two sobel operations and blend them. Also, be sure you are working on a gray-scaled image.

Example code:

gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)  
grad\_x = cv2.Sobel(gray, cv2.CV\_64F, 1, 0, ksize=3)   
grad\_y = cv2.Sobel(gray, cv2.CV\_64F, 0, 1, ksize=3)   
abs\_grad\_x = cv2.convertScaleAbs(grad\_x)   
abs\_grad\_y = cv2.convertScaleAbs(grad\_y)

grad = cv2.addWeighted(abs\_grad\_x, 0.5, abs\_grad\_y, 0.5, 0)

plt.imshow(grad,cmap=’gray’)

**Laplacian Filter:**

we use a Laplacian filter to compute the second-order derivative of an image to detect edges. We need a Laplacian filter so that we can extract the features of the image in a better way.

In first-order derivatives, We detect the horizontal and vertical edges and combine them. On the other hand, the second-order derivative allows us to detect all the edges of an image at once.

The zero-crossing property of the Laplacian filter can be used for edge location. Often these derivative filters are applied to a smoothed function to avoid problems with image noise amplification.   
  
Example Code:  
gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)  
laplacian = cv2.Laplacian(gray, cv2.CV\_64F)

**Canny Edge Detector:**

The Canny operator works in a multi-stage process. First of all the image is smoothed by Gaussian convolution. Then a simple 2-D first derivative operator is applied to the smoothed image to highlight regions of the image with high first spatial derivatives. Edges give rise to ridges in the gradient magnitude image. The algorithm then tracks along the top of these ridges and sets to zero all pixels that are not actually on the ridge top so as to give a thin line in the output, a process known as non-maximal suppression. The tracking process exhibits hysteresis controlled by two thresholds: T1 and T2, with T1 > T2. This hysteresis helps to ensure that noisy edges are not broken up into multiple edge fragments.

Example Code:

**import** cv2

img **=** cv2.imread("test.jpeg") # Read image

# Setting parameter values

t\_lower **=** 50 # Lower Threshold

t\_upper **=** 150 # Upper threshold

# Applying the Canny Edge filter

edge **=** cv2.Canny(img, t\_lower, t\_upper)

cv2.imshow('original', img)

cv2.imshow('edge', edge)

**Contours in Image Processing:**

In computer vision, a contour is like a digital representation of that outline. It can be described as the series of connected points that define the boundary of an object, separating and/or highlighting it from the background. These points tend to share similar color or intensity values, making them distinct from their surroundings

Contour detection involves identifying and extracting contours from images using various techniques, such as edge detection algorithms (e.g., Canny edge detection) or thresholding methods. Once detected, contours can be represented as a sequence of points or as hierarchies of curves, capturing the shape and structure of objects within the image.

Example Code:

Edge-Based Preprocessing

Here, edge detection is implicitly performed as part of the contour detection process using the cv2.findContours() function with the cv2.RETR\_TREE mode. Detected contours are drawn on a copy of the original image (detected\_contours) using cv2.drawContours().  
# Finding contours with RETR\_TREE mode

contours, \_ = cv2.findContours(thresh, cv2.RETR\_TREE, cv2.CHAIN\_APPROX\_SIMPLE)

detected\_contours = img.copy()

cv2.drawContours(detected\_contours, contours, -1, (0, 255, 0), -1)