Vidyavardhini's College of Engineering & Technology Department of Computer Engineering

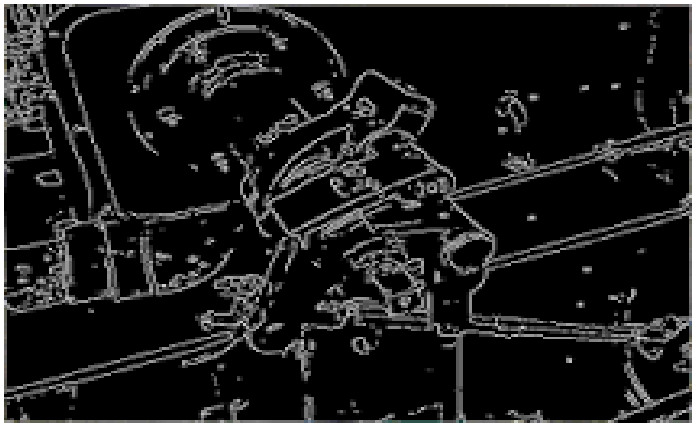
---

Aim: To study Edge detection with Canny

Objective : Perform Canny Edge detector using Noise reduction using Gaussian filter ,Gradient calculation along the horizontal and vertical axis Non-Maximum suppression of false edges ,Double thresholding for segregating strong and weak edges ,Edge tracking by hysteresis

Theory :

The Canny edge detector is an edge detection operator that uses a multi-stage algorithm to detect a wide range of edges in images. It was developed by John F. Canny in 1986. Canny also produced a computational theory of edge detection explaining why the technique works.



What are the three stages of the Canny edge detector
To fulfill these objectives, the edge detection process included the following stages.

- Stage One - Image Smoothing.
- Stage Two - Differentiation.
- Stage Three - Non-maximum Suppression.

The basic steps involved in this algorithm are:

- Noise reduction using Gaussian filter
- Gradient calculation along the horizontal and vertical axis
- Non-Maximum suppression of false edges
- Double thresholding for segregating strong and weak edges
- Edge tracking by hysteresis

Now let us understand these concepts in detail:

1. Noise reduction using Gaussian filter

This step is of utmost importance in the Canny edge detection. It uses a Gaussian filter for the removal of noise from the image, it is because this noise can be assumed as edges due to sudden intensity change by the edge detector. The sum of the elements in the Gaussian kernel is 1, so the kernel should be normalized before applying convolution to the image. In this

Experiment, we will use a kernel of size 5 X 5 and sigma = 1.4, which will blur the image and remove the noise from it. The equation for Gaussian filter kernel is

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

$$K_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, K_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}.$$

after applying these kernel we can use the gradient magnitudes and the angle to further process this step. The magnitude and angle can be calculated as

$$|G| = \sqrt{I_x^2 + I_y^2},$$
$$\theta(x, y) = arctan\left(\frac{I_y}{I_x}\right)$$

Non-Maximum Suppression

This step aims at reducing the duplicate merging pixels along the edges to make them uneven. For each pixel find two neighbors in the positive and negative gradient directions, supposing that each neighbor occupies

the angle of pi /4, and 0 is the direction straight to the right. If the magnitude of the current pixel is greater than the magnitude of the neighbors, nothing changes, otherwise, the magnitude of the current pixel is set to zero.

4. Double Thresholding

The gradient magnitudes are compared with two specified threshold values, the first one is lower than the second. The gradients that are smaller than the low threshold value are suppressed, the gradients higher than the high threshold value are marked as strong ones and the corresponding pixels are included in the final edge map. All the rest gradients are marked as weak ones and pixels corresponding to these gradients are considered in the next step.

5. Edge Tracking using Hysteresis

Since a weak edge pixel caused by true edges will be connected to a strong edge pixel, pixel W with weak gradient is marked as edge and included in the final edge map if and only if it is involved in the same connected component as some pixel S with strong gradient. In other words, there should be a chain of neighbor weak pixels connecting W and S (the neighbors are 8 pixels around the considered one). We will make up and implement an algorithm that finds all the connected components of the gradient map considering each pixel only once. After that, you can decide which pixels will be included in the final edge map. Below is the implementation.

```
import numpy as np
```

In [2]:

```
##Guasain Filter
```

In [3]:

```
def gausian_kernel(size,sigma=1):
    size = int(size)//2
    x,y = np.mgrid[-size:size+1,-size:size+1]
    norm = 1/(2.0*np.pi*sigma**2)
    g = np.exp(-(x**2+y**2)/(2*sigma**2))*norm
    return g
```

In [4]:

```
##Sobel Filter
```

In [5]:

```
from scipy import ndimage

def sobel_filters(img):
    Kx = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], np.float32)
    Ky = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]], np.float32)

    Ix = ndimage.filters.convolve(img, Kx)
    Iy = ndimage.filters.convolve(img, Ky)

    G = np.hypot(Ix, Iy)
    G = G / G.max() * 255
    theta = np.arctan2(Iy, Ix)

    return (G, theta)
```

In [6]:

```
##Non Maximum Supression
```

In [7]:

```
def non_max_suppression(img, D):
    M, N = img.shape
```

```
    Z = np.zeros((M,N), dtype=np.int32)
    angle = D * 180. / np.pi
    angle[angle < 0] += 180


    for i in range(1,M-1):
        for j in range(1,N-1):
            try:
                q = 255
                r = 255

               #angle 0
                if (0 <= angle[i,j] < 22.5) or (157.5 <= angle[i,j] <= 180):
                    q = img[i, j+1]
                    r = img[i, j-1]
                #angle 45
                elif (22.5 <= angle[i,j] < 67.5):
                    q = img[i+1, j-1]
                    r = img[i-1, j+1]
                #angle 90
                elif (67.5 <= angle[i,j] < 112.5):
                    q = img[i+1, j]
                    r = img[i-1, j]
                #angle 135
                elif (112.5 <= angle[i,j] < 157.5):
                    q = img[i-1, j-1]
                    r = img[i+1, j+1]

                if (img[i,j] >= q) and (img[i,j] >= r):
                    Z[i,j] = img[i,j]
                else:
                    Z[i,j] = 0

            except IndexError as e:
                pass

    return Z
```

In [8]:

```
##Double thresholding
```

In [9]:

```
def threshold(img, lowThresholdRatio=0.05, highThresholdRatio=0.09):
```

```
    highThreshold = img.max() * highThresholdRatio;
    lowThreshold = highThreshold * lowThresholdRatio;

    M, N = img.shape
    res = np.zeros((M,N), dtype=np.int32)

    weak = np.int32(25)
    strong = np.int32(255)

    strong_i, strong_j = np.where(img >= highThreshold)
    zeros_i, zeros_j = np.where(img < lowThreshold)

    weak_i, weak_j = np.where((img <= highThreshold) & (img >= lowThreshold))

    res[strong_i, strong_j] = strong
    res[weak_i, weak_j] = weak

    return (res, weak, strong)
```

In [10]:

```
##Hysterisis
```

In [11]:

```
def hysteresis(img, weak, strong=255):
    M, N = img.shape
    for i in range(1, M-1):
        for j in range(1, N-1):
            if (img[i,j] == weak):
                try:
                    if ((img[i+1, j-1] == strong) or (img[i+1, j] == strong) or (img[i+1, j+1] == strong)
                        or (img[i, j-1] == strong) or (img[i, j+1] == strong)
                        or (img[i-1, j-1] == strong) or (img[i-1, j] == strong) or (img[i-1, j+1] == strong)):
                        img[i, j] = strong
                    else:
                        img[i, j] = 0
                except IndexError as e:
                    pass
    return img
```

In [12]:

```
import cv2
import matplotlib.pyplot as plt
```

```
bgrimg = cv2.imread("input_image-1.jpg")
plt.imshow(bgrimg)
```

Output :

Input Image

Output image

Conclusion:

In conclusion, the Canny edge detection technique is a sophisticated multi-stage algorithm for identifying edges in images. It comprises three main stages: image smoothing, gradient calculation, and non-maximum suppression. The process begins with noise reduction using a Gaussian filter to eliminate unwanted artifacts that might be mistaken for edges. Gradient calculation then reveals the direction and strength of edges within the image. Non-maximum suppression further refines the detected edges by preserving only the local maxima in gradient magnitude.