# GNR 602: Project no: 20

Implementing Canny edge detector

April, 2023

## 1 Group members

- Patil Vipul Sudhir(210050115)
- P.Hari Prakash Reddy(210050119)
- Yabaji Pratik Sanjay(210070094)

## 2 Source code

### 2.1 Guassian kernel

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

Code Listing 1: Guassian kernel

The above code gives out a Gaussian operator which can be used to smoothen the image and reduce the noise based on the sigma and size given as input.

### 2.2 Apply convolution

```
    def apply_convolution(img, kernel):
    M, N = img.shape
    m, n = kernel.shape
    padding = np.zeros((M+m-1, N+n-1))
    padding[m//2:M+m//2, n//2:N+n//2] = img

    output = np.zeros_like(img)
    for i in range(M):
        for j in range(N):
            output[i, j] =np.sum(padding[i:i+m, j:j+n] * kernel)

    return output
```

Code Listing 2: Convolution

This applies any given filter(matrix) to convolute on the whole image and gives out the resultant image as output.

## 2.3 Double Thresholding and Hysteresis

```
    def hysteresis_thresholding(img, t1, t2):

    weak = np.zeros_like(img)
    strong = np.zeros_like(img)
    strong_threshold = np.max(img) * t2
    weak_threshold = np.max(img) * t1

    strong[img >= strong_threshold] = 255
    weak[(img >= weak_threshold) & (img < strong_threshold)] = 128

    # perform connectivity analysis to determine strong edges
    M, N = img.shape
    edge_map = np.uint8(strong)
    for i in range(1, M-1):
        for j in range(1, N-1):
            if weak[i,j] == 128:
                if (strong[i-1:i+2, j-1:j+2] == 255).any():
                    edge_map[i,j] = 255
                else:
                    edge_map[i,j] = 0

    return edge_map
```

Code Listing 3: Double Thresholding and Hysteresis

This part of the code does the double thresholding part of the canny edge detector, first finds the strong edges (pixels with intensity greater than the higher threshold), and checks if the pixel with greater than the lower threshold has any pixels with a strong edge in its neighbors and if present makes it also a strong edge.

## 2.4 Sobel

```
def sobel_op(img):
  dx_kernel = np.array([[-1, 0, 1],
                        [-2, 0, 2],
                        [-1, 0, 1]], dtype=np.float32)

  dy_kernel = np.array([[-1, -2, -1],
                        [0, 0, 0],
                        [1, 2, 1]], dtype=np.float32)

  dx = np.zeros_like(img, dtype=np.float32)
  dy = np.zeros_like(img, dtype=np.float32)

  height, width = img.shape


  for i in range(1, height - 1):
```

```
    for j in range(1, width - 1):
      dx[i, j] = np.sum(img[i-1:i+2, j-1:j+2] * dx_kernel)
      dy[i, j] = np.sum(img[i-1:i+2, j-1:j+2] * dy_kernel)
  return dx,dy
```

Code Listing 4: Sobel operator

This part of the code uses the Sobel operator and applies it to the image given as input using for loops and gives out the resulting image which has gradients as outputs.

## 2.5 Non maximum suppression

```
def non_maximum_suppression(G, theta):
    # Finding dimensions of the image
    N, M = G.shape
    # Parsing through all the pixels
    for i_x in range(M):
        for i_y in range(N):

            grad_ang = theta[i_y, i_x]
            grad_ang = abs(grad_ang-180) if abs(grad_ang)>180 else
    abs(grad_ang)

            # selecting the neighbours of the target pixel
            # according to the gradient direction
            # In the x axis direction
            if grad_ang<= 22.5:
                neighb_1_x, neighb_1_y = i_x-1, i_y
                neighb_2_x, neighb_2_y = i_x + 1, i_y

            # top right (diagonal-1) direction
            elif grad_ang>22.5 and grad_ang<=(22.5 + 45):
                neighb_1_x, neighb_1_y = i_x-1, i_y-1
                neighb_2_x, neighb_2_y = i_x + 1, i_y + 1

            # In y-axis direction
            elif grad_ang>(22.5 + 45) and grad_ang<=(22.5 + 90):
                neighb_1_x, neighb_1_y = i_x, i_y-1
                neighb_2_x, neighb_2_y = i_x, i_y + 1

            # top left (diagonal-2) direction
            elif grad_ang>(22.5 + 90) and grad_ang<=(22.5 + 135):
                neighb_1_x, neighb_1_y = i_x-1, i_y + 1
                neighb_2_x, neighb_2_y = i_x + 1, i_y-1

            # Now it restarts the cycle
            elif grad_ang>(22.5 + 135) and grad_ang<=(22.5 + 180):
                neighb_1_x, neighb_1_y = i_x-1, i_y
                neighb_2_x, neighb_2_y = i_x + 1, i_y

            # Non-maximum suppression step
            if M>neighb_1_x>= 0 and N>neighb_1_y>= 0:
                if  G[i_y, i_x]< G[neighb_1_y, neighb_1_x]:
                    G[i_y, i_x]= 0
                    continue
```

```
43
44              if M>neighb_2_x>= 0 and N>neighb_2_y>= 0:
45                  if  G[i_y, i_x]< G[neighb_2_y, neighb_2_x]:
46                      G[i_y, i_x]= 0
47
48      return G
```

Code Listing 5: Non maximum suppression

This part of the code does the Nonmaximum suppression, which means it removes all pixels which are having pixels with greater intensity that themselves in the direction of its gradient.

## 2.6 Canny edge detector

```
1       def Canny_detector(img, sigma, t1, t2):
2       # Step 1: Convert given image to grayscale image
3       img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
4       img = img_gray
5
6       # Step 2: Apply Gaussian filter to smooth the image
7       kernel_size = 7
8       gaussian_kernel = generate_gaussian_kernel(kernel_size, sigma)
9       img_smooth = apply_convolution(img, gaussian_kernel)
10      smooth_img = img_smooth
11
12      # Step 3: Compute gradient magnitude and direction using Sobel
        operators
13      gx, gy = sobel_op(img)
14      G_mag, G_dir = cv2.cartToPolar(gx, gy, angleInDegrees = True)
15
16      # Step 4: Perform non-maximum suppression to thin the edges
17      G_suppressed = non_maximum_suppression(G_mag, G_dir)
18
19      # Step 5: Perform hysteresis thresholding to detect strong and
        weak edges
20      edge_image= hysteresis_thresholding(G_suppressed, t1, t2)
21
22    # returns a grayscale image, smoothened image, edge image
23      return img_gray, smooth_img, edge_image
```

Code Listing 6: Canny$_e dge_d etector$

This brings all the functions into one place and does them one by one as specified by canny edge detection, in the following order

- Converting to grayscale
- Gaussian smoothening
- Finding Gradient using Sobel
- Non-Maximum supression
- Double thresholding and hysteresis