A Simple Canny Edge Detector Implementation

Development of the Canny algorithm

Canny edge detection is a technique to extract useful structural information from different vision objects and dramatically reduce the amount of data to be processed. It has been widely applied in various computer vision systems. Canny has found that the requirements for the application of edge detection on diverse vision systems are relatively similar. Thus, an edge detection solution to address these requirements can be implemented in a wide range of situations. The general criteria for edge detection include:

1. Detection of edge with low error rate, which means that the detection should accurately catch as many edges shown in the image as possible
2. The edge point detected from the operator should accurately localize on the center of the edge.
3. A given edge in the image should only be marked once, and where possible, image noise should not create false edges.

Process of Canny edge detection algorithm

The Process of Canny edge detection algorithm can be broken down to **5 different steps:**

1. Apply Gaussian filter to smooth the image in order to remove the noise
2. Find the intensity gradients of the image
3. Apply non-maximum suppression to get rid of spurious response to edge detection
4. Apply double threshold to determine potential edges
5. Track edge by hysteresis: Finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges.

```
class CannyEdgeDetector(object):
    """
    A naive canny edge detector's implementations:

    Canny Edge Detect Algorithms Procedure:
    Step1: Apply gauss filter to smooth image in order to denoise
    Step2: Generate gradient intensity and phase map for smoothed image
    Step3: Apply non-maximum suppression(NMS) to get rid of spurious response to edge detection(Slim edge)
    Step4: Apply double threshold to determine potentials edges: (strong and weak)
    step5: Track edge by hysteresis:
    Finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong
edges.
```

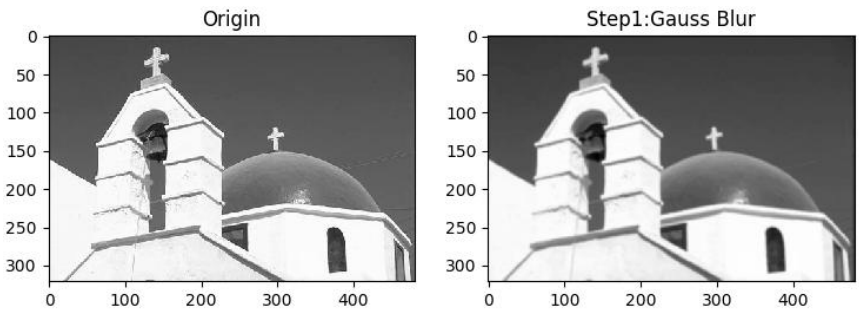
```
    see this paper for more details :
    "A Computational Approach to Edge Detection" ,IEEE Transactions on Pattern Analysis and Machine Intelligence
( Volume: PAMI-8 , Issue: 6 , Nov. 1986 )
    """
    def __init__(self,img_path,kernel_size,sigma,threshold):
        """
        :param img_path: /path/to/image
        :param kernel_size: gauss'kernel's size
        :param sigma: gauss kernel's variance
        :param threshold: [low_threshold, high_threshold] for step4 ,eg:[0.2,0.6]
        """
        self.kernel_size = kernel_size
        self.sigma = sigma
        # define a set of kernel
        self.kernel_dic = {
                           'sobel_x': np.array([[-1,0,1],[-2,0,2],[-1,0,1]]),
                           'sobel_y': np.array([[1,2,1],[0,0,0],[-1,-2,-1]]),
                         }
        self.threshold = threshold
        # read image with gray mode
        self.img = cv2.imread(img_path,0)
        self.height = self.img.shape[0]
        self.width = self.img.shape[1]
```

Step1: Gaussian filter

Since all edge detection results are easily affected by image noise, it is essential to filter out the noise to prevent false detection caused by noise. To smooth the image, a Gaussian filter is applied to convolve with the image. This step will slightly smooth the image to reduce the effects of obvious noise on the edge detector. The equation for a Gaussian filter kernel of size (2k+1)×(2k+1) is given by:

$$H_{ij} = \frac{1}{2\pi\sigma^2}\exp\left(-\frac{(i-(k+1))^2+(j-(k+1))^2}{2\sigma^2}\right); 1 \le i,j \le (2k+1)$$

```
def gauss_filter(self):
    """
    Step1: Apply gauss filter to smooth image based on given gauss_kernel
    :return: gauss-blur img
    """
    img_blur = cv2.GaussianBlur(self.img,self.kernel_size,self.sigma)
    return img_blur
```



Step2: Finding the intensity gradient of the image

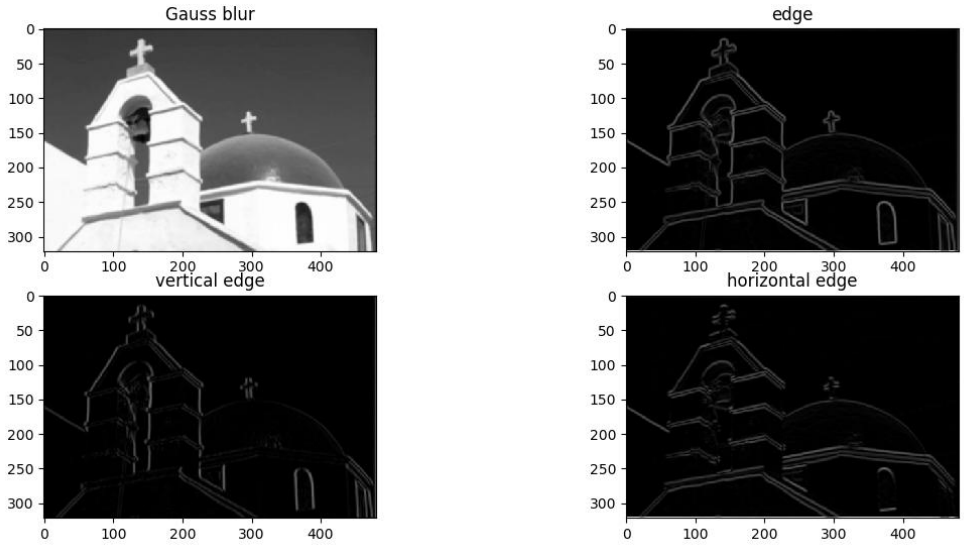
An edge in an image may point in a variety of directions, so the Canny algorithm uses four filters to detect horizontal, vertical and diagonal edges in the blurred image. The edge detection operator(such as Roberts,Prewitt, or Sobel) returns a value for the first derivative in the horizontal direction (Gx) and the vertical direction (Gy). From this the edge gradient and direction can be determined:

$$G_{intensity} = \sqrt{G_X{}^2 + G_Y{}^2}$$
$$G_{phase} = arctan2(G_y / G_x)$$

```
def conv(self,img,filter_name):
    """
    implement a simple 2D-Conv operations
    :param filter_name: filter's name
    :return: feature map
    """
    mask = self.kernel_dic[filter_name]
    res = np.zeros((self.height,self.width))

    h_size = mask.shape[0]
    w_size = mask.shape[1]

    for h in range(self.height):
        for w in range(self.width):
            field = img[h:h+h_size, w:w+w_size]
            res[h][w] = np.sum(field*mask)

    return res
```

```
def Gradients_Compute(self,img):
    """
    Step2: Generate gradient intensity and phase map for smoothed image
    sobel_x:   [-1 0 1;
                -2 0 2;
                -1 0 1];
```

```
    sobel_y:   [1 2 1;
                0 0 0;
               -1 -2 -1];
    :return:
    """
    h_size = 3
    w_size = 3
    # pad img to keep same size
    p = int((h_size-1)/2)
    q = int((w_size-1)/2)
    img_pad = np.pad(img,((p,p),(q,q)),'constant')

    grad_x = np.abs(self.conv(img_pad, 'sobel_x'))
    grad_y = np.abs(self.conv(img_pad, 'sobel_y'))

    grad_abs = np.sqrt(grad_x**2+grad_y**2)
    grad_phase = np.arctan2(grad_y,grad_x)*180/np.pi

    return grad_abs, grad_x, grad_y, grad_phase
```



Step3: Non-maximum suppression

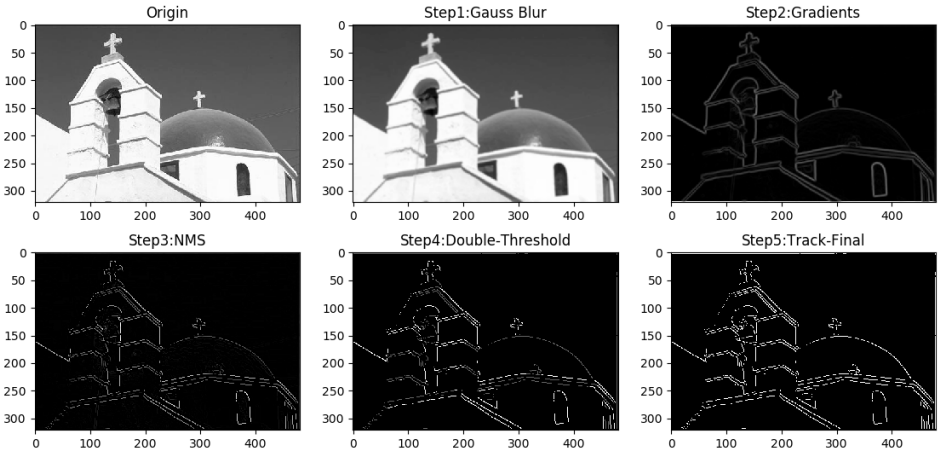
The algorithm for each pixel in the gradient image is:

1、Compare the edge strength of the current pixel with the edge strength of the pixel in the positive and negative gradient directions.

2、If the edge strength of the current pixel is the largest compared to the other pixels in the mask with the same direction (i.e., the pixel that is pointing in the y-direction, it will be compared to the pixel above and below it in the vertical axis), the value will be preserved. Otherwise, the

value will be suppressed.

```
def NMS(self, det, phase):
    """
    apply NMS to every pixel
    :param det: grad_abs
    :param phase: grad_phase
    :return: pixel after NMS
    """
    gmax = np.zeros(det.shape)
    for i in range(gmax.shape[0]):
        for j in range(gmax.shape[1]):
            if phase[i][j] < 0:
                phase[i][j] += 360
            # In order to beyond boarder
            if ((j + 1) < gmax.shape[1]) and ((j - 1) >= 0) and ((i + 1) < gmax.shape[0]) and ((i - 1) >= 0):
                # 0 degrees
                if (phase[i][j] >= 337.5 or phase[i][j] < 22.5) or (phase[i][j] >= 157.5 and phase[i][j] < 202.5):
                    # compare with respond two pixels
                    if det[i][j] >= det[i][j + 1] and det[i][j] >= det[i][j - 1]:
                        gmax[i][j] = det[i][j]
                # 45 degrees
                if (phase[i][j] >= 22.5 and phase[i][j] < 67.5) or (phase[i][j] >= 202.5 and phase[i][j] < 247.5):
                    if det[i][j] >= det[i - 1][j + 1] and det[i][j] >= det[i + 1][j - 1]:
                        gmax[i][j] = det[i][j]
                # 90 degrees
                if (phase[i][j] >= 67.5 and phase[i][j] < 112.5) or (phase[i][j] >= 247.5 and phase[i][j] < 292.5):
                    if det[i][j] >= det[i - 1][j] and det[i][j] >= det[i + 1][j]:
                        gmax[i][j] = det[i][j]
                # 135 degrees
                if (phase[i][j] >= 112.5 and phase[i][j] < 157.5) or (phase[i][j] >= 292.5 and phase[i][j] < 337.5):
                    if det[i][j] >= det[i - 1][j - 1] and det[i][j] >= det[i + 1][j + 1]:
                        gmax[i][j] = det[i][j]
    return gmax
```



Step4: Double threshold

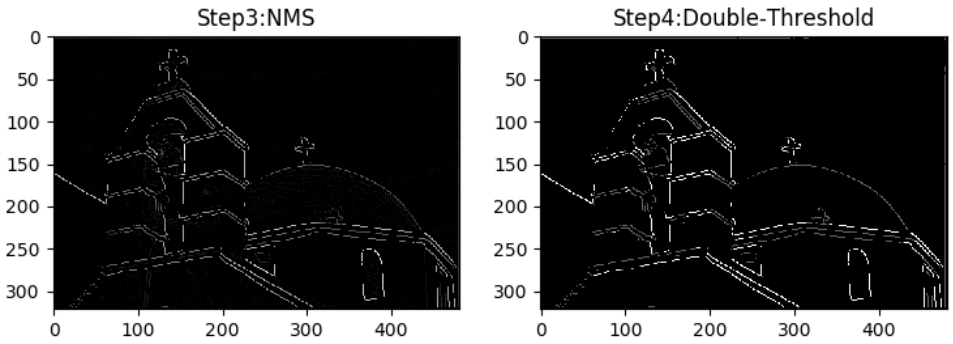
After application of non-maximum suppression, remaining edge pixels provide a more accurate representation of real edges in an image. However, some edge pixels remain that are caused by noise and color variation. In order to account for these spurious responses, it is essential to filter out edge pixels with a weak gradient value and preserve edge pixels with a high gradient value. This is accomplished by selecting high and low threshold values.

If an edge pixel's gradient value is higher than the high threshold value, it is marked as a strong edge pixel. If an edge pixel's gradient value is smaller than the high threshold value and larger than the low threshold value, it is marked as a weak edge pixel. If an edge pixel's value is smaller than the low threshold value, it will be suppressed.

The two threshold values are empirically determined and their definition will depend on the content of a given input image.

```
def double_threshold(self,grad_nms):
    """
    procedure:
    if grad_intensity[i][j]>high_threshold:
        grad_intensity[i][j] = high_threshold
    elif grad_intensity[i][j]>low_threshold:
        grad_intensity[i][j] = low_threshold
    else:
        grad_intensity[i][j] = 0

    :param grad_nms: grad_intensity after nms
    :return: a grad_intensity which contains [weak,strong]
             a flag array record edge's locations

    """
    # a flag
    is_edge = np.zeros(grad_nms.shape)
    grad_th = np.zeros(grad_nms.shape)
    strong = 1.0
    weak = 0.4
    mmax = np.max(grad_nms)
    print(grad_nms)

    low_threshold = self.threshold[0]*mmax
    high_threshold = self.threshold[1]*mmax

    for i in range(grad_nms.shape[0]):
        for j in range(grad_nms.shape[1]):
            if grad_nms[i][j]>=high_threshold:
                grad_th[i][j] = strong
                is_edge[i][j] = 1.0
```

```
            elif grad_nms[i][j] >= low_threshold:
                grad_th[i][j] = weak
            else :
                grad_th[i][j] = 0

    return grad_th, is_edge
```

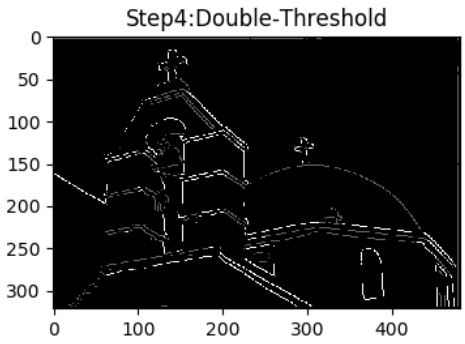


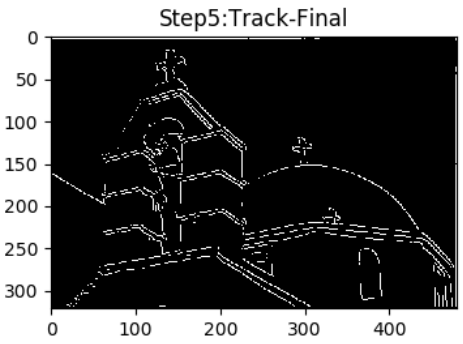
Step5: Edge tracking by hysteresis

So far, the strong edge pixels should certainly be involved in the final edge image, as they are extracted from the true edges in the image. However, there will be some debate on the weak edge pixels, as these pixels can either be extracted from the true edge, or the noise/color variations. To achieve an accurate result, the weak edges caused by the latter reasons should be removed. Usually a weak edge pixel caused from true edges will be connected to a strong edge pixel while noise responses are unconnected. To track the edge connection, blob analysis is applied by looking at a weak edge pixel and its 8-connected neighborhood pixels. As long as there is one strong edge pixel that is involved in the blob, that weak edge point can be identified as one that should be preserved.

```
def track(self, gradmCopy, is_edge):
    """
    :param gradmCopy: grad_intensity after double threshold
    :param is_edge: A flag matrix record edge's locations
    :return: A flag matrix record edge's locations
    """
    strong = 1.0
    weak = 0.4
    height = gradmCopy.shape[0]
    width = gradmCopy.shape[1]

    for ii in range(1,height-2):
        for jj in range(1,width-2):

            flag = True
```

```
        if gradmCopy[ii][jj] == weak:

            neighbors = [
                        [gradmCopy[ii - 1][jj - 1], gradmCopy[ii - 1][jj], gradmCopy[ii - 1][jj + 1] ],
                        [gradmCopy[ii][jj - 1], 0, gradmCopy[ii][jj + 1] ],
                        [gradmCopy[ii + 1][jj - 1], gradmCopy[ii + 1][jj], gradmCopy[ii + 1][jj + 1]]
                        ]
            for i in range(3):
                for j in range(3):
                    if neighbors[i][j] == strong:
                        flag = True
                        break
            if flag==True:
                is_edge[ii][jj] = 1.0
    return is_edge
```

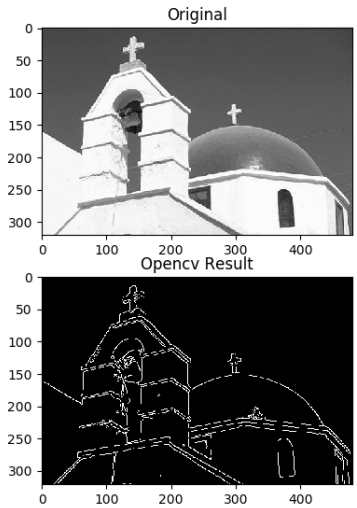


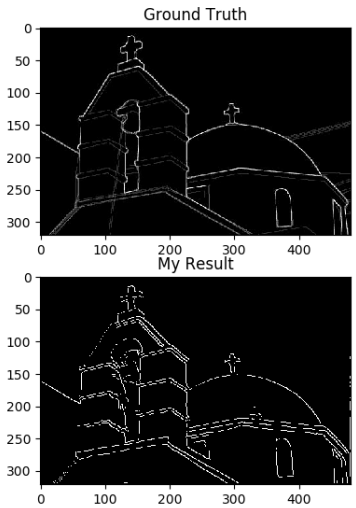


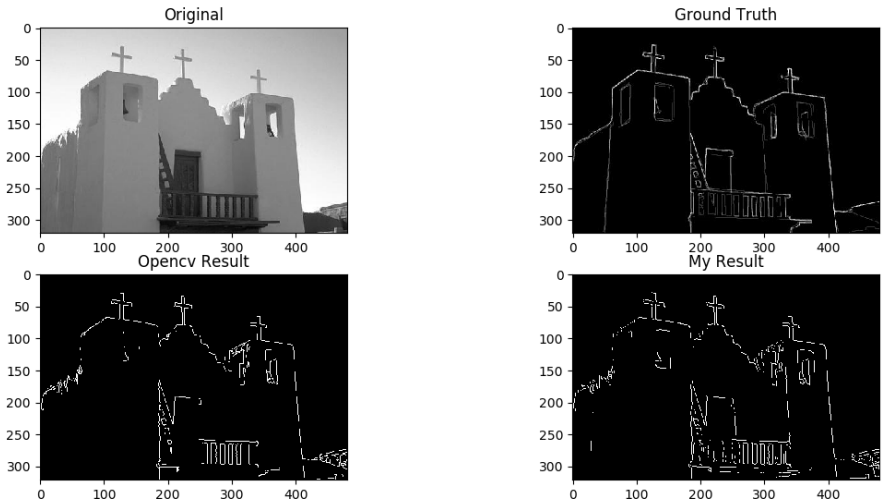
Compare With Ground Truth And OpenCV Bulit-in Algorithm

The Berkeley Segmentation Dataset and Benchmark

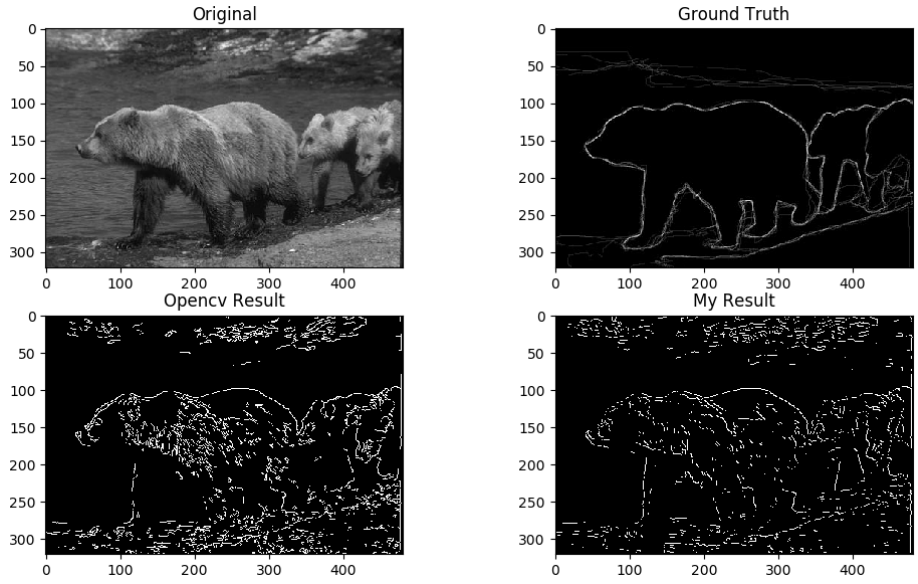
Good Results:







Bad Results:



Thinking

It is not easy to achieve edge detection due to various situations.

Low-level computer vision task still have a long way to go !

Reference

[1]、https://en.wikipedia.org/wiki/Canny_edge_detector

[2]、Canny, J.,A Computational Approach To Edge Detection, IEEE Trans. Pattern Analysis and Machine Intelligence, 8(6):679–698, 1986.

[3]、https://github.com/sebasvega95/Canny-edge-detector