CANNY-DERICHE IMPLEMENTATION

Atanda Abdullahi Adewale, MSc. Computer Vision

INTRODUCTION

Canny-Deriche edge detector based on Canny's edge detector and his criteria for optimal edge detection:. Its developed by Rachid Deriche in 1987. It is a multi-step algorithm used to obtain an optimal result of edge detection in a discrete two-dimensional image.

Like Canny, the Deriche detector consists of Smoothing, Calculation of magnitude and gradient direction, Non-maximum suppression, Hysteresis thresholding (using two thresholds)

The Deriche edge detector uses the IIR filter $f(x) = \frac{S}{\omega} e^{-\alpha|x|} \sin \omega x$
and
The filter optimizes the Canny criteria when the value of ωapproaches 0. then $f(x) = Sxe^{-\alpha|x|}$

$$h_{n,j}(x) = \mathrm{e}^{-s_j|x|} \sum_{i=0}^{n} \frac{s_j^{n-i}}{(n-i)!} |x|^{n-i}.$$

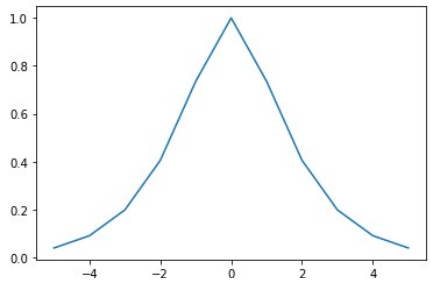
Generally, for n=1, we obtain the Canny-Deriche smoothing filter. The corresponding equation for h1. Let h1 become h, sj becomes s, the scale factor fixing the width of h.

$e^{-|x|} \cdot |x| + 1$ in Numpy, **np.exp(-np.abs(i))*(np.abs(i)+1)**

METHODS

1 - Normalize h in the discrete domain: N x sum(h)=1. Plot h in the discrete domain (for example h(-10:1:10) with s=1). Truncate the support of h according to s (check by plotting). You can define a criterion to automatically choose the support of h (for example, you represent h while > h(0)/100.

Taking range of -5 to 5 the deriche filter (5x5) :



[0.39970926 0.29408964 0.1622843 0.07960141 0.03660465
0.01615932 0.00693546 0.0029159 0.00120679 0.00049328]

2 - Apply h (in 1D) by convolution on the rows/columns of the image to obtain a smoothed image. Note it is not necessary to define h on rows and on columns (use math. transpose operator).

```
# Compute Deriche Smoothing Filter
k = length // 2
derichex = np.zeros(length)
derichey = np.zeros(length)
for i in range(length) :
        derichex[i] = np.exp(-np.abs(i))*(np.abs(i)+1)
#normlization
derichex =derichex / np.sum(derichex)
print('window for x direction', derichex)
for j in range(length) :
        derichey[j] = np.exp(-np.abs(j))*(np.abs(j)+1)
derichey =derichey / np.sum(derichey)
```

```
# Use Filter
W, H = image.shape
new_imagex = np.zeros([W , H - k * 2])
new_imagey = np.zeros([W - k * 2, H - k * 2])
for i in range(W ) :
    for j in range(H - 2 * k) :
        # 卷积运算
        new_imagex[i, j] = np.sum(image[i , j :j + length] * derichex)


for j in range(H - 2 * k) :
    for i in range(W - 2 * k) :
        # 卷积运算
        new_imagey[i, j] = np.sum(new_imagex[i:i+ length , j ] * derichey)
new_image = np.uint8(new_imagey)
return new_image
```

Original Image

Smoothed image

3 - The gradient image is a representation of the edges in an image and is calculated using the gradient magnitude and direction.

We apply a simple first derivative [1 -1] and $[1\ -1]^T$ to the image components components Gi, Gj) or apply Sobel filter as derivative.

Then compute the direction: arctan2 the changes in y with respect to x

direction[i, j] = np.arctan(dy / dx)

```
Gx = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
Gy = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])

W, H = image.shape
gradients = np.zeros([W - 2, H - 2])
direction = np.zeros([W - 2, H - 2])

for i in range(W - 2) :
    for j in range(H - 2) :
        dx = np.sum(image[i :i + 3, j :j + 3] * Gx)
        dy = np.sum(image[i :i + 3, j :j + 3] * Gy)
        gradients[i, j] = np.sqrt(dx ** 2 + dy ** 2)
        if dx == 0 :
            direction[i, j] = np.pi / 2
        else :
            direction[i, j] = np.arctan(dy / dx)

return gradients, direction
```

4 - Extract through a simplified (directions simplified to the four sectors) non-local maximum procedure the edge candidates.

```
def NMS(gradients, direction) :
    W, H = gradients.shape
    nms = np.copy(gradients[1 :-1, 1 :-1])

    for i in range(1, W - 1) :
        for j in range(1, H - 1) :
            theta = direction[i, j]
            weight = np.tan(theta)
            if theta > np.pi / 4 :
                d1 = [0, 1]
                d2 = [1, 1]
                weight = 1 / weight
            elif theta >= 0 :
                d1 = [1, 0]
                d2 = [1, 1]
            elif theta >= - np.pi / 4 :
                d1 = [1, 0]
                d2 = [1, -1]
                weight *= -1
            else :
                d1 = [0, -1]
                d2 = [1, -1]
                weight = -1 / weight

            g1 = gradients[i + d1[0], j + d1[1]]
            g2 = gradients[i + d2[0], j + d2[1]]
            g3 = gradients[i - d1[0], j - d1[1]]
            g4 = gradients[i - d2[0], j - d2[1]]

            grade_count1 = g1 * weight + g2 * (1 - weight)
            grade_count2 = g3 * weight + g4 * (1 - weight)

            if grade_count1 > gradients[i, j] or grade_count2 > gradients[i, j] :
                nms[i - 1, j - 1] = 0
    return nms
```

Gradients after Local Maximum Suppression

The non-local maximum suppression is a process of removing any pixels that are not local maxima in their neighborhood. The code calculates the weight of each gradient based on its direction and then checks if the gradient values of the neighboring pixels are greater than the current gradient value.
If the gradient values of the neighboring pixels are greater, then the gradient value at the current pixel is set to 0, implying that the current pixel is not a local maximum. The final result of the local maximum suppression is returned as a new gradient image with only local maxima preserved.

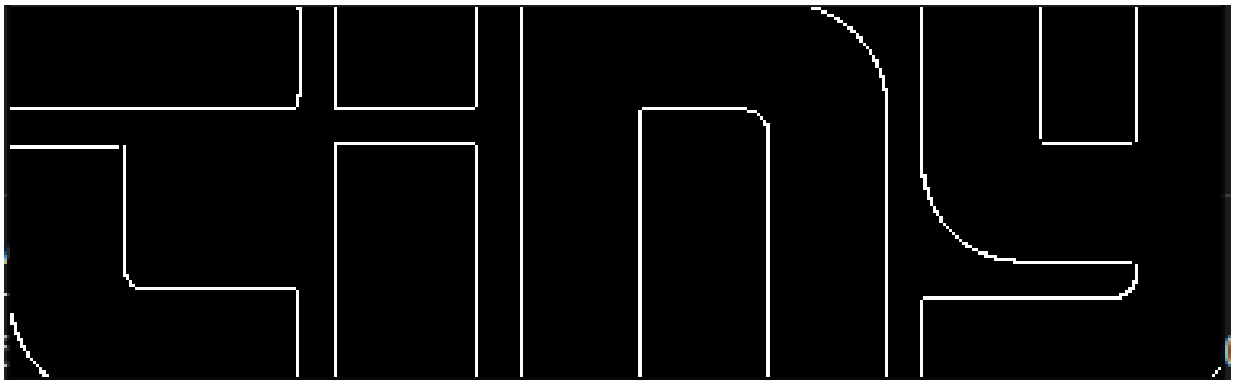
5 - Refine the segmentation with an hysteresis algorithm : two thresholds are fixed, a high threshold sh and a low threshold sb. Start by selecting the points which exceed the high threshold and then apply the low threshold by keeping only the connected components which contain a point above sh.

After the Non maximum suppression, there may still be some pixels that are not true edge pixels. These pixels can be removed using a double threshold method.

Pixels with gradient values greater than the high threshold are considered as strong edges and are kept in the edge map. Pixels with gradient values less than the low threshold are considered as non-edges and are removed from the edge map. Pixels with gradient values between the two thresholds are considered as weak edges and are kept only if they are connected to strong edges. This helps to reduce the number of false edges in the final edge map.

RESULT:

Applied a double threshold: low at 40 and high of 100 intensities



Output Image.

Appendix:

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt

def smooth(image, sigma = 1.4, length = 10) :
    """ Smooth the image
    """
    # Compute deriche filter
```

```
    k = length // 2
    derichex = np.zeros(length)
    derichey = np.zeros(length)
    for i in range(length) :
            derichex[i] = np.exp(-np.abs(i))*(np.abs(i)+1)
    #normlization
    derichex =derichex / np.sum(derichex)
    print('window for x-row direction', derichex)
    for j in range(length) :
            derichey[j] = np.exp(-np.abs(j))*(np.abs(j)+1)
    derichey =derichey / np.sum(derichey)
    print('\n')
    print('window for y-row direction', derichey)
    # Use the Filter
    W, H = image.shape
    new_imagex = np.zeros([W , H - k * 2])
    new_imagey = np.zeros([W - k * 2, H - k * 2])
    for i in range(W ) :
        for j in range(H - 2 * k) :
            new_imagex[i, j] = np.sum(image[i , j :j + length] * derichex)

    for j in range(H - 2 * k) :
        for i in range(W - 2 * k) :
            new_imagey[i, j] = np.sum(new_imagex[i:i+ length , j ] * derichey)
    new_image = np.uint8(new_imagey)
    return new_image

def get_gradient_and_direction(image) :
    """ Compute gradients and its direction using Sobel
    """
    Gx = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
    Gy = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])

    W, H = image.shape
    gradients = np.zeros([W - 2, H - 2])
    direction = np.zeros([W - 2, H - 2])

    for i in range(W - 2) :
        for j in range(H - 2) :
            dx = np.sum(image[i :i + 3, j :j + 3] * Gx)
            dy = np.sum(image[i :i + 3, j :j + 3] * Gy)
            gradients[i, j] = np.sqrt(dx ** 2 + dy ** 2)
            if dx == 0 :
                direction[i, j] = np.pi / 2
            else :
                direction[i, j] = np.arctan(dy / dx)

    return gradients, direction

def NMS(gradients, direction) :
    """ Non-maxima suppression
    """
```

```
    W, H = gradients.shape
    nms = np.copy(gradients[1 :-1, 1 :-1])

    for i in range(1, W - 1) :
        for j in range(1, H - 1) :
            theta = direction[i, j]
            weight = np.tan(theta)
            if theta > np.pi / 4 :
                d1 = [0, 1]
                d2 = [1, 1]
                weight = 1 / weight
            elif theta >= 0 :
                d1 = [1, 0]
                d2 = [1, 1]
            elif theta >= - np.pi / 4 :
                d1 = [1, 0]
                d2 = [1, -1]
                weight *= -1
            else :
                d1 = [0, -1]
                d2 = [1, -1]
                weight = -1 / weight

            g1 = gradients[i + d1[0], j + d1[1]]
            g2 = gradients[i + d2[0], j + d2[1]]
            g3 = gradients[i - d1[0], j - d1[1]]
            g4 = gradients[i - d2[0], j - d2[1]]

            grade_count1 = g1 * weight + g2 * (1 - weight)
            grade_count2 = g3 * weight + g4 * (1 - weight)

            if grade_count1 > gradients[i, j] or grade_count2 > gradients[i, j] :
                nms[i - 1, j - 1] = 0
    return nms


def double_threshold(nms, threshold1, threshold2) :
    """ Double Threshold Returns:The binary image.
    """
    visited = np.zeros_like(nms)
    output_image = nms.copy()
    W, H = output_image.shape

    def dfs(i, j) :
        if i >= W or i < 0 or j >= H or j < 0 or visited[i, j] == 1 :
            return
        visited[i, j] = 1
        if output_image[i, j] > threshold1 :
            output_image[i, j] = 255
            dfs(i - 1, j - 1)
            dfs(i - 1, j)
            dfs(i - 1, j + 1)
```

```
            dfs(i, j - 1)
            dfs(i, j + 1)
            dfs(i + 1, j - 1)
            dfs(i + 1, j)
            dfs(i + 1, j + 1)
        else :
            output_image[i, j] = 0

    for w in range(W) :
        for h in range(H) :
            if visited[w, h] == 1 :
                continue
            if output_image[w, h] >= threshold2 :
                dfs(w, h)
            elif output_image[w, h] <= threshold1 :
                output_image[w, h] = 0
                visited[w, h] = 1

    for w in range(W) :
        for h in range(H) :
            if visited[w, h] == 0 :
                output_image[w, h] = 0
    return output_image

if __name__ == "__main__" :
    # code to read image
    i = cv.imread('test.png')
    image = cv.imread('test.png', 0)
    cv.imshow("Original", image)
    smoothed_image = smooth(image)
    cv.imshow("Canny-Deriche smoothing filter (5*5)", smoothed_image)
    gradients, direction = get_gradient_and_direction(smoothed_image)
    # print(gradients)
    # print(direction)
    nms = NMS(gradients, direction)
    output_image = double_threshold(nms, 40, 100)
    cv.imshow("outputImage", output_image)
    cv.waitKey(0)
```