Individual topical study report

Name: 林穎志 Student ID: 411551007

Introduction / Objectives:

The purpose of this report is to survey and implement papers about improved Canny Edge Detection algorithm, which is a method to find the particular edge points in an image. One can observe the difference of the original results from paper **and the results implemented by me**.

Method:

Noise reduction: Gaussian filter, Median filter, Morphology filter

Canny edge detection

Adaptation Threshold of Canny Edge Detection

Review of the methods I have implemented:

	Original	Paper1	Paper2	Paper3
Denoising	Gaussian filter	**Median filter**	Gaussian filter	**Morphology filter**
Gradient calculating	Sobel filter with 2 directions	Sobel filter with 2 directions	Sobel filter with **4** directions	Sobel filter with **4** directions
Non maximum suppression	NMS	NMS	NMS	NMS
Threshold decision	Manual	**Adaptive thresholding**	**Adaptive thresholding**	**Otsu's thresholding**
Double thresholding and Hysteresis	Double thresholding and hysteresis	**Formation and Filtering Generalized Chains**	Double thresholding and hysteresis	Double thresholding and hysteresis

I. Noise reduction

1. Gaussian filter: A Gaussian has the advantage of being smooth in both the spatial and frequency domains.

2. Median filter: An order-statistic filter that change the pixel value to its neighborhood median value .

II. Sobel filter: A filter for calculating image gradients in horizontal and vertical direction to find the details of edge.

III. Non maximum suppression: Keep only the "strongest" edge points along the gradient direction.

IV. Double thresholding: Retain all the "strong" edge points as well as the "weak"

edge points that are connected to the “strong” edge points.

3. Adaptive Thresholding of Canny Edge Detection

The two adaptive threshold method in paper1 and paper2 is different, which is going to be introduced in the following section respectively.

4. Otsu’s Thresholding

Automatic global thresholding algorithms usually have following steps:

- Process the input image
- Obtain image histogram (distribution of pixels)
- Compute the threshold value T
- Replace image pixels into white in those regions, where saturation is greater than T and into the black in the opposite cases.

5. Formation and Filtering Generalized Chains

Will be introduced in paper1.

Paper 1: An improved canny edge detection algorithm

I. Citation: L. Xuan and Z. Hong, "An improved canny edge detection algorithm," 2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS), 2017, pp. 275-278, doi: 10.1109/ICSESS.2017.8342913.

II. Implement Method: Canny Edge Detection, Adaptive Thresholding, Formation and Filtering Generalized Chains

A. Median Filter Algorithm

Median filter algorithm is used to replace Gaussian filtering method.

B. Adaptive Real-Time Dual Threshold Algorithm

Implemented by making a differential operation on the amplitude histogram of the image.

a. Find the maximum value of the central pixel in the same gradient direction by the non-maximum value suppression method
b. The maximum value is processed by double threshold, if it is not a maximum value, the amplitude of the pixel is set to zero, and generate the gradient amplitude histogram as G_1

The most important step is how to make the adaptive threshold. The steps are as following: doing difference operation on the adjacent gradient amplitude:

$$G_1(i+1) - G_1(i)$$

Select the first zero of the amplitude as a high threshold, and the low value take 0.4 times of this high threshold. According to this rule, the adaptive threshold setting formulas are defined as:

$$Th_H = Arg(G_1(t+1) - G_i = 0)$$

$$Th_{\mathrm{L}} = 0.4{*}Th_{\mathrm{H}}$$

Th_{H} and Th_{L} stand for high threshold and low threshold. After getting the 2 thresholds, do double thresholding. Define the strong and the weak edge points as:

First case: Strong edge point

$$G_1(\mathrm{i},\mathrm{j}) > Th_{\mathrm{H}}$$

Second Case: Weak edge point

$$Th_L < G_1(\mathrm{i},\mathrm{j}) < Th_{\mathrm{H}}$$

Third Case: Not an edge point

$$G_1(\mathrm{i},\mathrm{j}) < Th_{\mathrm{H}}$$

C. Formation and Filtering Generalized Chains

a. Find an edge starting point, which is the maximum point and link it with weak edge points respectively.
b. Remove the local maximum points, which cannot be connected. In this way, one can get the edge chains. These chains are called **generalized chains**, which contains multiple lengths. The maximum length is set to $d_{\max}$ and the

minimum length is set to $d_{\min}$. Get average modulus threshold of these chains:

$$\frac{d_{\max} + d_{\min}}{2} < d_{avrage} < d_{top}$$

c. Remove the gradient value, which smaller than d_{avrage}.

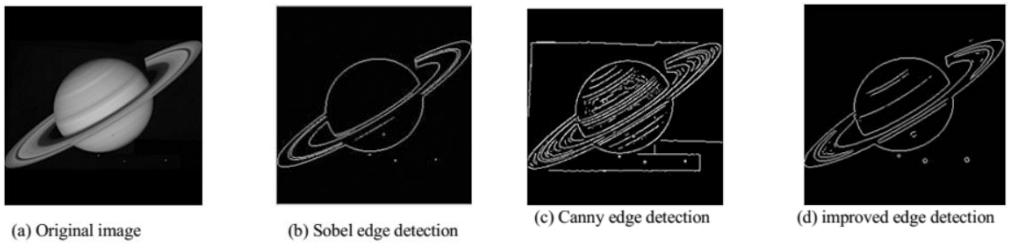
For the purpose of suppressing the pseudo-strong edge points effectively, the remaining points are the real edge points of image.

D. Take the linear fitting method to get the result of the edge image.

III. Experiments & Results:

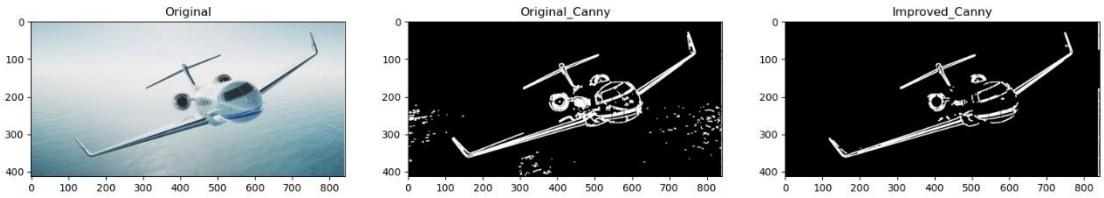
Implement and comparison

A. Result from paper



(a) Original image (b) Sobel edge detection (c) Canny edge detection (d) improved edge detection

B. My own implementation



IV. Discussions:

The traditional Canny algorithm needs to determine the threshold values by oneself and it can't separate targets from background in some situations. This paper proposed an improved method on Canny algorithm to deal with these problems. When we look upon result A, the original image is a gray scale image. The author indicated that the result of the proposed algorithm is more clear and delicate. Nonetheless, the author claimed that this approach has strong anti-noise ability because it is not like the canny algorithm to link the non-marginal pixels, which are affected by the noise to misjudge as edge points. Furthermore looking upon the result B, which is implemented by me. The original image is a RGB image, and the result shows a consistency to result A, which shows that the improved canny algorithm has a better performance.

Paper 2: Window frame obstacle edge detection based on improved Canny operator

I. Citation: Lu, H., & Yan, J. (2019, October). Window frame obstacle edge detection based on improved Canny operator. In 2019 3rd International Conference on Electronic Information Technology and Computer Engineering (EITCE) (pp. 493-496). IEEE.

II. Implement Method: Canny Edge Detection, Adaptive Thresholding

A. Improve the Calculation of Gradient Magnitude and Direction

The traditional canny algorithm uses 2×2 template to calculate the gradient amplitude and size, which is greatly interfered by noise. Therefore, the author improved canny algorithm by adding two gradient direction templates.

-1	-2	-1
0	0	0
1	2	1

（a）horizontal direction

-1	0	1
-2	0	2
-1	0	1

（b）Vertical direction

-2	-1	0
1	0	1
0	1	2

（c）45° direction

0	1	2
-1	0	1
-2	-1	0

（d）135° direction

The calculated gradients of 45° and 135° are projected to the horizontal and vertical directions, and then summed to obtain new gradient values of the x and y axes:

$$G_{x1}(x,y) = G_x(x,y) + \sqrt{2}/2 * G_{45}(x,y) + \sqrt{2}/2 * G_{135}(x,y)$$
$$G_{y1}(x,y) = G_y(x,y) + \sqrt{2}/2 * G_{45}(x,y) + \sqrt{2}/2 * G_{135}(x,y)$$

Then calculate the direction and gradient magnitude of the current pixel gray value:

$$M(x,y) = \sqrt{G_{x1}(x,y)\text{^}2 + G_{y1}(x,y)\text{^}2}$$
$$D(x,y) = \arctan(G_{y1}(x,y)/G_{x1}(x,y))$$

B. Using the iterative method to obtain the best high and low thresholds

In general, the image is generally affected by noise, and the optimal value of the threshold is generally difficult to determine. Using the iterative principle to select the optimal threshold can greatly reduce the interference of the noise pair with the threshold selection.

a. Initial threshold statistics, T is the initial threshold of the image, K is the number of iterations of the algorithm, Z_{max} is the maximum gray value, and Z_{min} is the minimum gray value:

$$T\{T_k|K = 0)$$
$$T = (Z_{max} + Z_{min})/2$$

b. The initial threshold is obtained by calculation, and the image is divided into Ho and Hr according to the initial threshold, H_0 is above the initial threshold value portion, Hr is below the initial threshold value portion.

c. To calculate the gray mean values T_H and T_L of the two parts H_0 and H_r respectively.

d. Calculate the new threshold TT:

$$TT = (T_H + T_L)/2$$

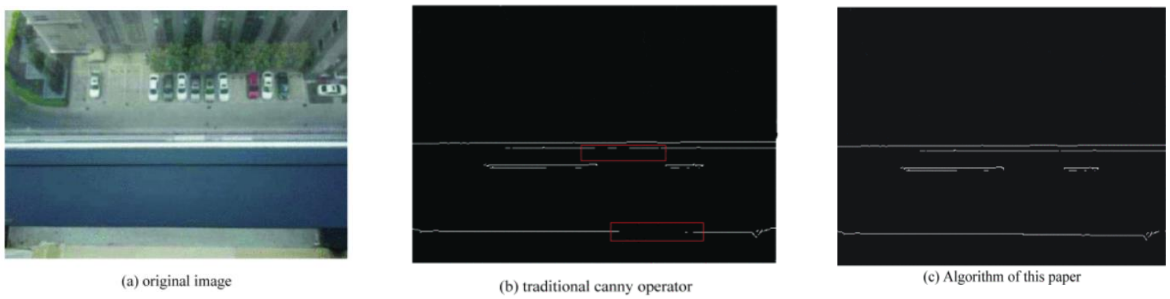
e. The iteration stops when the final iteration threshold is equal to the initial threshold or satisfies the set reasonable error range, otherwise the iteration is always running.

f. After the iterative algorithm is terminated, the finally obtained iterative thresholds T_H and T_L are segmented as optimal thresholds as double thresholds.

III. Experiments & Results:

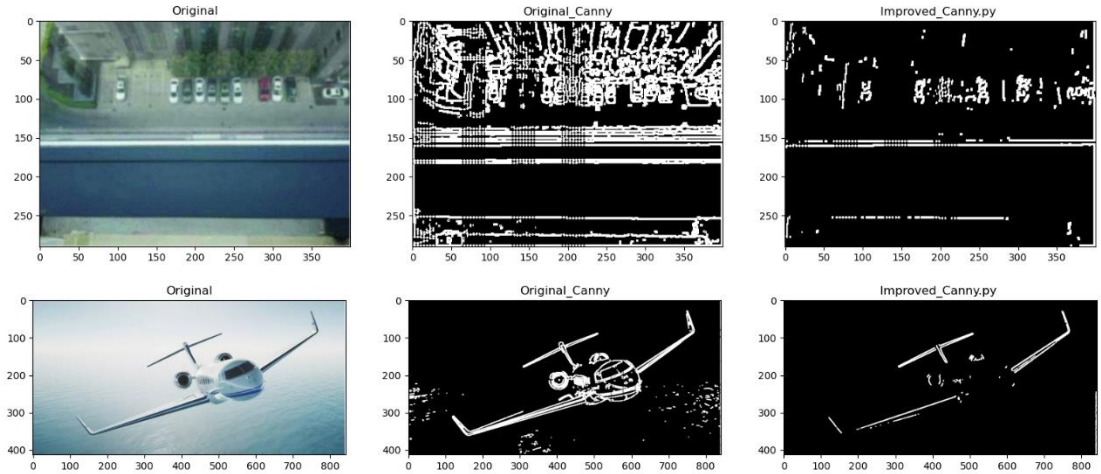
Implement and comparison

A. Result from paper



(a) original image (b) traditional canny operator (c) Algorithm of this paper

B. My own implementation



IV. Discussions:

The algorithm is designed for the wall cleaning robot. The author of the paper claimed that the lines at the edges of the resulting image appear to be significantly broken and have poor connectivity in the traditional canny algorithm which is (b) in result A. The improved canny algorithm yields a significantly more complete image with better connectivity. When looking upon result B, which implemented by me, we'll see the result of the jet picture is not ideal. Further experiment of the original example picture of the building is done by screenshot. The result shows a consistency on the algorithm eliminates some unwanted contours. The difference between the results of original canny algorithm from my implementation and the author mainly due to different thresholds and the image quality of the picture.

Paper 3: Edge detection of agricultural products based on morphologically improved Canny algorithm

I. Citation: Yu, X., Wang, Z., Wang, Y., & Zhang, C. (2021). Edge detection of agricultural products based on morphologically improved Canny algorithm. Mathematical Problems in Engineering, 2021.

II. Implement Method: Canny Edge Detection, Otsu's Thresholding, Morphology Operation

A. Compound Morphological Filtering Denoising

A morphological filter is used to replace the Gaussian filter for denoising. The composite morphological filter proposed in this study can well remove the noise interference in the process of agricultural product edge detection and can also highlight the contour and remove unnecessary false contour. Suppose F(x, y) is a grayscale image. This paper fused the two operations at a ratio of 10% and 90% to obtain a composite morphological filter, the proposed morphology operation defined as:

$$A_3 = 0.1 * \left((F \circ B_1) \cdot B_2\right) + 0.9 * \left((F \cdot B_2) \circ B_1\right)$$

While B_1, B_2 are the structural elements:

$$B_1 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \qquad B_2 = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}.$$

B. Calculation Method of Gradient Amplitude

In this study, based on the original method, gradients in the directions of 45° and 135° are added on the basis of one-step template to reduce the calculation amount.

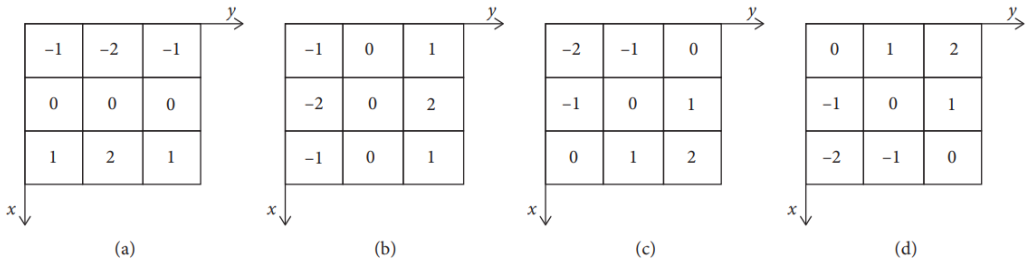


FIGURE 3: Gradient direction template. (a) x axis direction. (b) y axis direction. (c) 45°axis direction. (d) 135° axis direction.

The gradient angle and amplitude of the image defined as:

$$M(x,y) = sqrt(\, g_x(i,j)^2 + g_y(i,j)^2 + g_{45}(i,j)^2 + g_{135}(i,j)^2)$$

$$\text{theta}(x,y) = \arctan(\, g_y(i,j)^2 / g_x(i,j)^2)$$

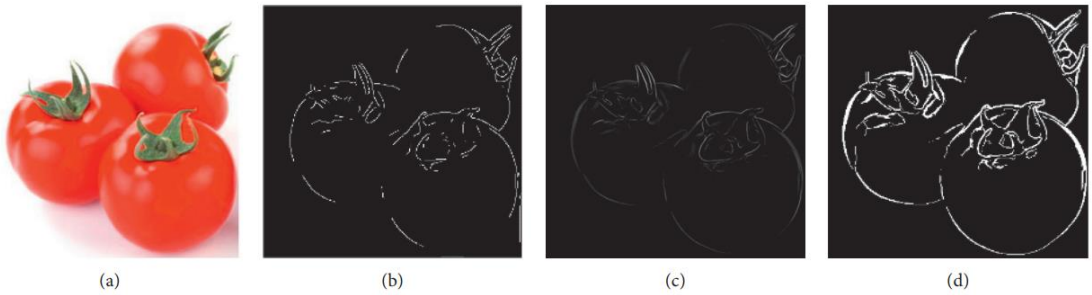
C. Improvement of Threshold Calculation Algorithm

In this study, double threshold detection and improved Otsu are combined to achieve this step.

III. Experiments & Results:

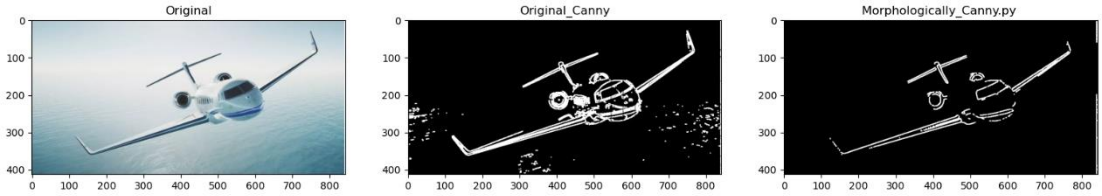
Implement and comparison

C. Result from paper



(a) Original tomato image. (b) Traditional Canny algorithm. (c)The algorithm in other reference (d) Improved algorithm in this article

D. My own implementation



IV. Discussions:

The traditional canny edge detection algorithm has its limitations in the aspect of antinoise interference, and it is susceptible to factors such as light. The author indicated that the proposed method had some advantages:

A. The composite morphological filter proposed in this study combines the two factors to remove noise from agricultural products and maintain details at the same time.
B. In the calculation method of gradient amplitude, the two directions of the traditional algorithm are changed into four directions, which can make the edge positioning more accurate, and play a better detection effect for objects such as agricultural products.
C. The adaptive threshold segmentation method was adopted for rough segmentation based on the characteristics of agricultural products.

When we look upon result A, the original image is a RGB image. The author indicated that compared to other methods, the improved algorithm can not only remove the redundant false edges but also clearly depict the real edges of each object in various agricultural products. The author also claimed that the algorithm in this study can remove the noise well and display the contour of agricultural products accurately. Furthermore looking upon the result B, which is implemented by me. The original image is a RGB image. Though the result is not as good as the result of paper 1, we can still observe the contour of the jet and the denoising ability as the result eliminates the contour of wave.

Code:

Paper1:

```
import cv2
import os
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm, trange

#Show Image
def show_all_img(result, columns = None, rows = 1):

    if columns is None:
        columns = len(result)
    fig = plt.figure(figsize=(20, 20))

    for i, img_n in enumerate(result):
        ax = fig.add_subplot(rows, columns, i+1)
        ax.title.set_text(img_n)
        # ax.set_title(i)
        img = result[img_n]
        if len(img.shape) == 2:
            plt.imshow(img, cmap='gray')
        else:
            img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
            plt.imshow(img)
    #plt.tight_layout()
    plt.show()

def distance(a, b):
    return np.sqrt((a[0] - b[0])**2 + (a[1] - b[1])**2)


def Grayscale(image):
    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    return image

def GaussianBlur(image):
    image = cv2.GaussianBlur(image, (3, 3), 0)
```

```
    return image

def MedianBlur(image):
    image = cv2.medianBlur(image, 3, 0)
    return image

def SobelFilter(image, method = 'median'):
    if method == 'median':
        image = Grayscale(MedianBlur(image))
    else:
        image = Grayscale(GaussianBlur(image))
    convolved = np.zeros(image.shape)
    G_x = np.zeros(image.shape)
    G_y = np.zeros(image.shape)
    size = image.shape
    kernel_x = np.array(([-1, 0, 1], [-2, 0, 2], [-1, 0, 1]))
    kernel_y = np.array(([-1, -2, -1], [0, 0, 0], [1, 2, 1]))

    '''
    The gradient magnitude and gradient direction of the central pixel are calculated as follows:
    Gxy = [[a0, a3, a6],
           [a1, a4, a7],
           [a2, a5, a8]]
    px=(a6+2a7+a8)−(a0+2a3+a6)
    py=(a6+2a7+a8)−(a0+a1+a2)
    '''

    for i in range(1, size[0] - 1):
        for j in range(1, size[1] - 1):
            target_image = image[i - 1 : i + 2, j - 1 : j + 2]
            G_x[i, j] = np.sum(np.multiply(target_image, kernel_x))
            G_y[i, j] = np.sum(np.multiply(target_image, kernel_y))

    convolved = np.sqrt(np.square(G_x) + np.square(G_y))
    convolved = np.multiply(convolved, 255.0 / convolved.max())

    '''
```

```
    The gradient direction is calculated as follows:
    G1(i,j)=root(p2x+p2y) 2:square
    theta(i,j) = arctan(px/py)
    '''
    angles = np.rad2deg(np.arctan2(G_y, G_x))
    angles[angles < 0] += 180
    convolved = convolved.astype('uint8')
    return convolved, angles


def non_maximum_suppression(image, angles):
    size = image.shape
    suppressed = np.zeros(size)
    for i in range(1, size[0] - 1):
        for j in range(1, size[1] - 1):
            if (0 <= angles[i, j] < 22.5) or (157.5 <= angles[i, j] <= 180):
                value_to_compare = max(image[i, j - 1], image[i, j + 1])
            elif (22.5 <= angles[i, j] < 67.5):
                value_to_compare = max(image[i - 1, j - 1], image[i + 1, j + 1])
            elif (67.5 <= angles[i, j] < 112.5):
                value_to_compare = max(image[i - 1, j], image[i + 1, j])
            else:
                value_to_compare = max(image[i + 1, j - 1], image[i - 1, j + 1])

            if image[i, j] >= value_to_compare:
                suppressed[i, j] = image[i, j]
    suppressed = np.multiply(suppressed, 255.0 / suppressed.max())
    return suppressed

def adaptive_threshold(image):
    '''
    generate the gradient amplitude histogram as G1
    '''
    print('Adaptive Choosing Thresholds...')
    # create the histogram
    histogram, bin_edges = np.histogram(image, bins=256, range=(0, 255))
    G1 = histogram
```

```
    '''
    Select the first zero of the amplitude as a high threshold, and the low value take 0.4 times of this high threshold
    '''
    i = 0
    while i <= 255 :
        #print(i)
        i +=1
        if G1[i+1] - G1[i] == 0:
            break

    h_threshold = G1[i]
    l_threshold = h_threshold*0.4

    return h_threshold, l_threshold


def double_threshold_hysteresis(image, low, high):
    print('Double Thresholding...')
    weak = 0
    strong = 255
    size = image.shape
    result = np.zeros(size)
    weak_x, weak_y = np.where((image > low) & (image <= high))
    strong_x, strong_y = np.where(image >= high)
    result[strong_x, strong_y] = strong
    result[weak_x, weak_y] = weak
    dx = np.array((-1, -1, 0, 1, 1, 1, 0, -1))
    dy = np.array((0, 1, 1, 1, 0, -1, -1, -1))
    size = image.shape

    while len(strong_x):
        x = strong_x[0]
        y = strong_y[0]
        strong_x = np.delete(strong_x, 0)
        strong_y = np.delete(strong_y, 0)
        for direction in range(len(dx)):
            new_x = x + dx[direction]
```

```
                new_y = y + dy[direction]
                if((new_x >= 0 & new_x < size[0] & new_y >= 0 & new_y < size[1])
and (result[new_x, new_y]    == weak)):
                    result[new_x, new_y] = strong
                    np.append(strong_x, new_x)
                    np.append(strong_y, new_y)
    result[result != strong] = 0
    return result

def Filtering_Generalized_Chains(image, low, high):
    print('Filtering Generalized Chains...')
    weak = 0
    strong = 255
    size = image.shape
    result = np.zeros(size)
    weak_x, weak_y = np.where((image > low) & (image <= high))
    strong_x, strong_y = np.where(image >= high)
    grad_avg = np.average(image)
    result[strong_x, strong_y] = strong
    result[weak_x, weak_y] = weak

    '''
    Select strong edge points as the edge of the starting point and link them with weak
points to form edge chains,
    and calculate the average of edge chains by (8),
    then remove the generalized edge chains, which are smaller than the average of
the gradient maximum of the image.
    '''
    strong_points_list = []
    for i in range(len(strong_x)):
        point = (strong_x[i], strong_y[i])
        strong_points_list.append(point)

    weak_points_list = []
    for i in range(len(weak_x)):
        point = (weak_x[i], weak_y[i])
        weak_points_list.append(point)
```

```
# arr = np.array(distance_list)
# dmax = np.max(arr)
# dmin = np.min(arr)
# davg = (dmax + dmin)/2

size = image.shape
distance_list = []
progress = tqdm(total=len(strong_x))
while len(strong_x):
    progress.update(1)
    local_distance_list = []
    x = strong_x[0]
    y = strong_y[0]
    strong_x = np.delete(strong_x, 0)
    strong_y = np.delete(strong_y, 0)
    for i in range(len(weak_x)):
        d = abs(image[x,y] - image[weak_x[i], weak_y[i]])
        local_distance_list.append(d)
        avg_local_dis = np.average(np.array(local_distance_list))
        # distance_list.append(avg_local_dis)

    local_dmax = np.max(np.array(local_distance_list))
    local_dmin = np.min(np.array(local_distance_list))
    if avg_local_dis > (local_dmax + local_dmin)/2:
        if image[x,y] < avg_local_dis:
            result[x,y] = 0

    # for p in weak_points_list:
    #     d = distance((x,y), p)
    #     local_distance_list.append(d)
    # avg_local_dis = np.average(np.array(local_distance_list))
    # distance_list.append(avg_local_dis)

# d_avg = np.average(np.array(distance_list))
# p_strong_x, p_strong_y = np.where(image < d_avg)
# result[p_strong_x, p_strong_y] = 0

# result[result != strong] = 0
```

```
    return result, avg_local_dis

def Canny(image, low, high):
    image, angles = SobelFilter(image, method='gaussian')
    image = non_maximum_suppression(image, angles)
    gradient = np.copy(image)
    image = double_threshold_hysteresis(image, low, high)
    return image, gradient

def Imp_Canny(image):
    image, angles = SobelFilter(image, method='median')
    image = non_maximum_suppression(image, angles)
    gradient = np.copy(image)
    high, low = adaptive_threshold(image)
    image, average = Filtering_Generalized_Chains(image, low, high)
    return image, gradient, average

if __name__ == "__main__":

    input_path = ('jet.png')
    #output_path = ('jet_2.png')
    image = cv2.imread(input_path)
    image1, gradient1 = Canny(image, low = 0, high = 25)
    image2, gradient2, average = Imp_Canny(image)
    print('average: ', average)

    result = {}
    result['Original'] = image
    result['Original_Canny'] = image1
    result['Improved_Canny'] = image2

    show_all_img(result)
```

Paper2:

```
import cv2
import os
import numpy as np
import matplotlib.pyplot as plt
import math

#Show Image
def show_all_img(result, columns = None, rows = 1):

    if columns is None:
        columns = len(result)
    fig = plt.figure(figsize=(20, 20))

    for i, img_n in enumerate(result):
        ax = fig.add_subplot(rows, columns, i+1)
        ax.title.set_text(img_n)
        # ax.set_title(i)
        img = result[img_n]
        if len(img.shape) == 2:
            plt.imshow(img, cmap='gray')
        else:
            img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
            plt.imshow(img)
    #plt.tight_layout()
    plt.show()

def Grayscale(image):
    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    return image

def GaussianBlur(image):
    image = cv2.GaussianBlur(image, (3, 3), 0)
    return image

def MedianBlur(image):
    image = cv2.medianBlur(image, 3, 0)
    return image
```

```
def SobelFilter(image, method = 'median'):
    if method == 'median':
        image = Grayscale(MedianBlur(image))
    else:
        image = Grayscale(GaussianBlur(image))

    convolved = np.zeros(image.shape)
    G_x = np.zeros(image.shape)
    G_y = np.zeros(image.shape)
    size = image.shape
    kernel_x = np.array(([-1, 0, 1], [-2, 0, 2], [-1, 0, 1]))
    kernel_y = np.array(([-1, -2, -1], [0, 0, 0], [1, 2, 1]))

    '''
    The gradient magnitude and gradient direction of the central pixel are calculated as follows:
    Gxy = [[a0, a3, a6],
           [a1, a4, a7],
           [a2, a5, a8]]
    px=(a6+2a7+a8)−(a0+2a3+a6)
    py=(a6+2a7+a8)−(a0+a1+a2)
    '''

    for i in range(1, size[0] - 1):
        for j in range(1, size[1] - 1):
            target_image = image[i - 1 : i + 2, j - 1 : j + 2]
            G_x[i, j] = np.sum(np.multiply(target_image, kernel_x))
            G_y[i, j] = np.sum(np.multiply(target_image, kernel_y))

    convolved = np.sqrt(np.square(G_x) + np.square(G_y))
    convolved = np.multiply(convolved, 255.0 / convolved.max())

    '''
    The gradient direction is calculated as follows:
    G1(i,j)=root(p2x+p2y) 2:square
```

```
        theta(i,j) = arctan(px/py)
        '''
        angles = np.rad2deg(np.arctan2(G_y, G_x))
        angles[angles < 0] += 180
        convolved = convolved.astype('uint8')
        return convolved, angles

def Imp_SobelFilter(image, method = 'gaussian'):

        if method == 'median':
                image = Grayscale(MedianBlur(image))
        else:
                image = Grayscale(GaussianBlur(image))

        convolved = np.zeros(image.shape)
        G_x = np.zeros(image.shape)
        G_y = np.zeros(image.shape)
        G_45 = np.zeros(image.shape)
        G_135 = np.zeros(image.shape)

        size = image.shape
        kernel_x = np.array(([-1, 0, 1],
                             [-2, 0, 2],
                             [-1, 0, 1]))

        kernel_y = np.array(([-1, -2, -1],
                             [0, 0, 0],
                             [1, 2, 1]))

        kernel_45 = np.array(([-2, -1, 0],
                              [-1, 0, 1],
                              [0, 1, 2]))

        kernel_135 = np.array(([0, 1, 2],
                               [-1, 0, 1],
                               [-2, -1, 0]))

        for i in range(1, size[0] - 1):
```

```
            for j in range(1, size[1] - 1):
                target_image = image[i - 1 : i + 2, j - 1 : j + 2]
                G_x[i, j] = np.sum(np.multiply(target_image, kernel_x))
                G_y[i, j] = np.sum(np.multiply(target_image, kernel_y))
                G_45[i, j] = np.sum(np.multiply(target_image, kernel_45))
                G_135[i, j] = np.sum(np.multiply(target_image, kernel_135))

        Gx1 = G_x + (math.sqrt(2)/2)*G_45 + (math.sqrt(2)/2)*G_135
        Gy1 = G_y + (math.sqrt(2)/2)*G_45 + (math.sqrt(2)/2)*G_135
        convolved = np.sqrt(np.square(Gx1) + np.square(Gy1))
        convolved = np.multiply(convolved, 255.0 / convolved.max())

        angles = np.rad2deg(np.arctan2(Gy1, Gx1))
        angles[angles < 0] += 180
        convolved = convolved.astype('uint8')
        return convolved, angles


def non_maximum_suppression(image, angles):
        size = image.shape
        suppressed = np.zeros(size)
        for i in range(1, size[0] - 1):
            for j in range(1, size[1] - 1):
                if (0 <= angles[i, j] < 22.5) or (157.5 <= angles[i, j] <= 180):
                    value_to_compare = max(image[i, j - 1], image[i, j + 1])
                elif (22.5 <= angles[i, j] < 67.5):
                    value_to_compare = max(image[i - 1, j - 1], image[i + 1, j + 1])
                elif (67.5 <= angles[i, j] < 112.5):
                    value_to_compare = max(image[i - 1, j], image[i + 1, j])
                else:
                    value_to_compare = max(image[i + 1, j - 1], image[i - 1, j + 1])

                if image[i, j] >= value_to_compare:
                    suppressed[i, j] = image[i, j]
        suppressed = np.multiply(suppressed, 255.0 / suppressed.max())
        return suppressed
```

```
def adaptive_threshold(image):

    dif = math.inf
    times = 1
    threshold = 10
    T =    (np.max(image) + np.min(image))/2
    T1 = T
    while True:

        '''
        Where, T is the initial threshold of the image, K is the number of iterations of the algorithm,
        Zmax is the maximum gray value, and Zmin is the minimum gray value.
        T{Tk|K=0}
        T=(Zmax+Zmin)/2
        '''
        print('Adaptive Choosing Thresholding...')
        print('No. {} iteration'.format(times))

        '''
        Ho and Hr according to the initial threshold, H0 is above the initial threshold value portion,
        Hr is below the initial threshold value portion.
        '''
        Ho_x, Ho_y = np.where(image>T1)
        Hr_x, Hr_y = np.where(image<=T1)
        Ho = image[Ho_x, Ho_y]
        Hr = image[Hr_x, Hr_y]

        '''
        To calculate the gray mean values TH and TL of the two parts H0 and Hr respectively:
        '''
        Th = np.average(Ho)
        Tl =    np.average(Hr)

        '''
        Calculate the new threshold TT
```

```
            '''
            TT = (Th + Tl)/2

            '''
            The iteration stops when the final iteration threshold is equal to the initial threshold
            or satisfies the set reasonable error range, otherwise the iteration is always running.
            '''
            dif = abs(TT - T)
            if dif == 0 or dif<threshold:
                h_threshold = Th
                l_threshold = Tl
                break

            times += 1
            T1 = TT

            if times % 15 == 0:
                threshold += 3

        print('Final Difference Threshold: ', threshold)
        print('Final Threshold: ', h_threshold, l_threshold)
        return h_threshold, l_threshold


def double_threshold_hysteresis(image, low, high):
        print('Double Thresholding...')
        weak = 0
        strong = 255
        size = image.shape
        result = np.zeros(size)
        weak_x, weak_y = np.where((image > low) & (image <= high))
        strong_x, strong_y = np.where(image >= high)
        result[strong_x, strong_y] = strong
        result[weak_x, weak_y] = weak
        dx = np.array((-1, -1, 0, 1, 1, 1, 0, -1))
        dy = np.array((0, 1, 1, 1, 0, -1, -1, -1))
```

```
    size = image.shape

    while len(strong_x):
        x = strong_x[0]
        y = strong_y[0]
        strong_x = np.delete(strong_x, 0)
        strong_y = np.delete(strong_y, 0)
        for direction in range(len(dx)):
            new_x = x + dx[direction]
            new_y = y + dy[direction]
            if((new_x >= 0 & new_x < size[0] & new_y >= 0 & new_y < size[1])
and (result[new_x, new_y]    == weak)):
                result[new_x, new_y] = strong
                np.append(strong_x, new_x)
                np.append(strong_y, new_y)
    result[result != strong] = 0
    return result


def Canny(image, low = 0, high = 25):
    image, angles = SobelFilter(image, method='gaussian')
    image = non_maximum_suppression(image, angles)
    gradient = np.copy(image)
    image = double_threshold_hysteresis(image, low, high)
    return image, gradient

def Imp_Canny(image):
    image, angles = Imp_SobelFilter(image)
    image = non_maximum_suppression(image, angles)
    gradient = np.copy(image)
    high, low = adaptive_threshold(image)
    image = double_threshold_hysteresis(image, low, high)
    return image, gradient

if __name__ == "__main__":

    input_path = ('jet.png')
    #output_path = ('jet_2.png')
```

```
image = cv2.imread(input_path)
image2, gradient2 = Imp_Canny(image)
image1, gradient1 = Canny(image)

result = {}
result['Original'] = image
result['Original_Canny'] = image1
result['Improved_Canny.py '] = image2

show_all_img(result)
```

Paper3:

```
import cv2
import os
import numpy as np
import matplotlib.pyplot as plt


#Show Image
def show_all_img(result, columns = None, rows = 1):

    if columns is None:
        columns = len(result)
    fig = plt.figure(figsize=(20, 20))

    for i, img_n in enumerate(result):
        ax = fig.add_subplot(rows, columns, i+1)
        ax.title.set_text(img_n)
        # ax.set_title(i)
        img = result[img_n]
        if len(img.shape) == 2:
            plt.imshow(img, cmap='gray')
        else:
            img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
            plt.imshow(img)
    #plt.tight_layout()
    plt.show()

def Grayscale(image):
    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    return image

def GaussianBlur(image):
    image = cv2.GaussianBlur(image, (3, 3), 0)
    return image

def MedianBlur(image):
    image = cv2.medianBlur(image, 3, 0)
    return image
```

```
def Morphology_filter(image):
    print('Morphology Filtering...')
    '''
    A3 = 0.1*( (F open B1)close B2) + 0.9*( (F close B2) open B1)
    '''
    B1 = np.array([[0, 1, 0],
                   [1, 1, 1],
                   [0, 1, 0]], np.uint8)

    B2 = np.array([[0, 0, 1, 0, 0],
                   [0, 1, 1, 1, 0],
                   [1, 1, 1, 1, 1],
                   [0, 1, 1, 1, 0],
                   [0, 0, 1, 0, 0]], np.uint8)

    # opening
    opening = cv2.morphologyEx(image, cv2.MORPH_OPEN, B1, iterations=1)
    # closing
    A1 = cv2.morphologyEx(opening, cv2.MORPH_CLOSE, B2, iterations=1)
    # opening
    A2 = cv2.morphologyEx(image, cv2.MORPH_CLOSE, B2, iterations=1)
    # closing
    A2 = cv2.morphologyEx(A2, cv2.MORPH_OPEN, B1, iterations=1)

    A3 = 0.1*A1 + 0.9*A2
    return A3


def SobelFilter(image, method = 'median'):
    if method == 'median':
        image = Grayscale(MedianBlur(image))
    else:
        image = Grayscale(GaussianBlur(image))

    convolved = np.zeros(image.shape)
    G_x = np.zeros(image.shape)
    G_y = np.zeros(image.shape)
```

```
    size = image.shape
    kernel_x = np.array(([-1, 0, 1], [-2, 0, 2], [-1, 0, 1]))
    kernel_y = np.array(([-1, -2, -1], [0, 0, 0], [1, 2, 1]))

    '''
    The gradient magnitude and gradient direction of the central pixel are calculated as follows:
    Gxy = [[a0, a3, a6],
           [a1, a4, a7],
           [a2, a5, a8]]
    px=(a6+2a7+a8)−(a0+2a3+a6)
    py=(a6+2a7+a8)−(a0+a1+a2)
    '''

    for i in range(1, size[0] - 1):
        for j in range(1, size[1] - 1):
            target_image = image[i - 1 : i + 2, j - 1 : j + 2]
            G_x[i, j] = np.sum(np.multiply(target_image, kernel_x))
            G_y[i, j] = np.sum(np.multiply(target_image, kernel_y))

    convolved = np.sqrt(np.square(G_x) + np.square(G_y))
    convolved = np.multiply(convolved, 255.0 / convolved.max())

    '''
    The gradient direction is calculated as follows:
    G1(i,j)=root(p2x+p2y) 2:square
    theta(i,j) = arctan(px/py)
    '''
    angles = np.rad2deg(np.arctan2(G_y, G_x))
    angles[angles < 0] += 180
    convolved = convolved.astype('uint8')
    return convolved, angles

def Imp_SobelFilter(image):

    image = Morphology_filter(Grayscale(image))
    convolved = np.zeros(image.shape)
    G_x = np.zeros(image.shape)
```

```
    G_y = np.zeros(image.shape)
    G_45 = np.zeros(image.shape)
    G_135 = np.zeros(image.shape)

    size = image.shape
    kernel_x = np.array(([-1, 0, 1],
                         [-2, 0, 2],
                         [-1, 0, 1]))

    kernel_y = np.array(([-1, -2, -1],
                         [0, 0, 0],
                         [1, 2, 1]))

    kernel_45 = np.array(([-2, -1, 0],
                          [-1, 0, 1],
                          [0, 1, 2]))

    kernel_135 = np.array(([0, 1, 2],
                           [-1, 0, 1],
                           [-2, -1, 0]))

    for i in range(1, size[0] - 1):
        for j in range(1, size[1] - 1):
            target_image = image[i - 1 : i + 2, j - 1 : j + 2]
            G_x[i, j] = np.sum(np.multiply(target_image, kernel_x))
            G_y[i, j] = np.sum(np.multiply(target_image, kernel_y))
            G_45[i, j] = np.sum(np.multiply(target_image, kernel_45))
            G_135[i, j] = np.sum(np.multiply(target_image, kernel_135))

    convolved = np.sqrt(np.square(G_x) + np.square(G_y) + np.square(G_45) +
np.square(G_135))
    convolved = np.multiply(convolved, 255.0 / convolved.max())

    angles = np.rad2deg(np.arctan2(np.square(G_y), np.square(G_x)))
    angles[angles < 0] += 180
    convolved = convolved.astype('uint8')
    return convolved, angles
```

```
def non_maximum_suppression(image, angles):
    size = image.shape
    suppressed = np.zeros(size)
    for i in range(1, size[0] - 1):
        for j in range(1, size[1] - 1):
            if (0 <= angles[i, j] < 22.5) or (157.5 <= angles[i, j] <= 180):
                value_to_compare = max(image[i, j - 1], image[i, j + 1])
            elif (22.5 <= angles[i, j] < 67.5):
                value_to_compare = max(image[i - 1, j - 1], image[i + 1, j + 1])
            elif (67.5 <= angles[i, j] < 112.5):
                value_to_compare = max(image[i - 1, j], image[i + 1, j])
            else:
                value_to_compare = max(image[i + 1, j - 1], image[i - 1, j + 1])

            if image[i, j] >= value_to_compare:
                suppressed[i, j] = image[i, j]
    suppressed = np.multiply(suppressed, 255.0 / suppressed.max())
    return suppressed

def Otsu_threshold(image, lowrate = 0.1):
    print('Otsu thresholding...')
    # Otsu's thresholding
    image = image.astype("uint8")
    ret, th = cv2.threshold(image, 0, 255, type=(cv2.THRESH_BINARY +
cv2.THRESH_OTSU))
    l_threshold = ret * lowrate
    h_threshold = ret

    return h_threshold, l_threshold

def adaptive_threshold(image):
    '''
    generate the gradient amplitude histogram as G1
    '''
    print('Adaptive Choosing Thresholds...')
    # create the histogram
    histogram, bin_edges = np.histogram(image, bins=256, range=(0, 255))
```

```
    G1 = histogram

    '''
    Select the first zero of the amplitude as a high threshold, and the low value take 0.4 times of this high threshold
    '''
    i = 0
    while i <= 255 :
        #print(i)
        i +=1
        if G1[i+1] - G1[i] == 0:
            break

    h_threshold = G1[i]
    l_threshold = h_threshold*0.4

    return h_threshold, l_threshold



def double_threshold_hysteresis(image, low, high):
    print('Double Thresholding...')
    weak = 0
    strong = 255
    size = image.shape
    result = np.zeros(size)
    weak_x, weak_y = np.where((image > low) & (image <= high))
    strong_x, strong_y = np.where(image >= high)
    result[strong_x, strong_y] = strong
    result[weak_x, weak_y] = weak
    dx = np.array((-1, -1, 0, 1, 1, 1, 0, -1))
    dy = np.array((0, 1, 1, 1, 0, -1, -1, -1))
    size = image.shape

    while len(strong_x):
        x = strong_x[0]
        y = strong_y[0]
        strong_x = np.delete(strong_x, 0)
        strong_y = np.delete(strong_y, 0)
```

```
            for direction in range(len(dx)):
                new_x = x + dx[direction]
                new_y = y + dy[direction]
                if((new_x >= 0 & new_x < size[0] & new_y >= 0 & new_y < size[1])
and (result[new_x, new_y]    == weak)):
                    result[new_x, new_y] = strong
                    np.append(strong_x, new_x)
                    np.append(strong_y, new_y)
        result[result != strong] = 0
        return result

def Canny(image, low = 0, high = 30):
        image, angles = SobelFilter(image, method='gaussian')
        image = non_maximum_suppression(image, angles)
        gradient = np.copy(image)
        image = double_threshold_hysteresis(image, low, high)
        return image, gradient

def Imp_Canny(image):
        image, angles = Imp_SobelFilter(image)
        image = non_maximum_suppression(image, angles)
        gradient = np.copy(image)
        high, low = Otsu_threshold(image)
        image = double_threshold_hysteresis(image, low, high)
        return image, gradient

if __name__ == "__main__":

        input_path = ('jet.png')
        #output_path = ('jet_2.png')
        image = cv2.imread(input_path)
        image2, gradient2 = Imp_Canny(image)
        image1, gradient1 = Canny(image)

        result = {}
        result['Original'] = image
        result['Original_Canny'] = image1
        result['Improved_Canny'] = image2
```

```
show_all_img(result)
```