Image Processing
Assignment 2 – Report

Tang Wing Ho 鄧永豪 (310551004)

Introduction

This assignment is to experiment with one of several image segmentation techniques mentioned in the class. The one we pick is Canny Edge Detection. The techniques used in this assignment are implemented in Python 3.8 without using advanced image processing toolbox functions like image resizing, intensity transformation, histogram computation and spatial filtering. We would do the experiment on several images in order to try different combinations of the image enhancement techniques.

Methodology

1. Noise Reduction

The first step of Canny Edge Detection is to reduce the noise in image.
In this experiment, we would apply Gaussian filter to denoise our image. For each element's Gaussian filter coefficient, I calculate it through the following formula:

$$G_i = \alpha * e^{-\left(i-\frac{kernal\_size-1}{2}\right)^2 /(2*sigma^2)}$$

2. Sobel Filter

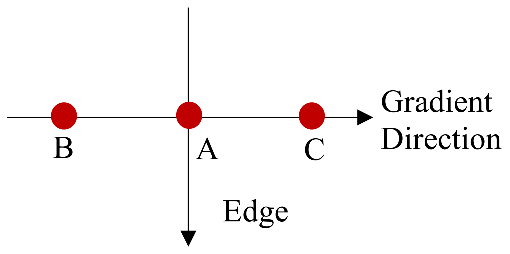
After noise reduction, the second step is to filter the smoothed image with Sobel kernel in both horizontal and vertical direction to get first derivative in horizontal direction (G_x) and vertical direction (G_y). From these two images, we can find edge gradient (G) and direction (θ) for each pixel as follows:

$$G = \sqrt{G_x^2 + G_y^2}$$

$$\theta = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

3. Non-maxima Suppression

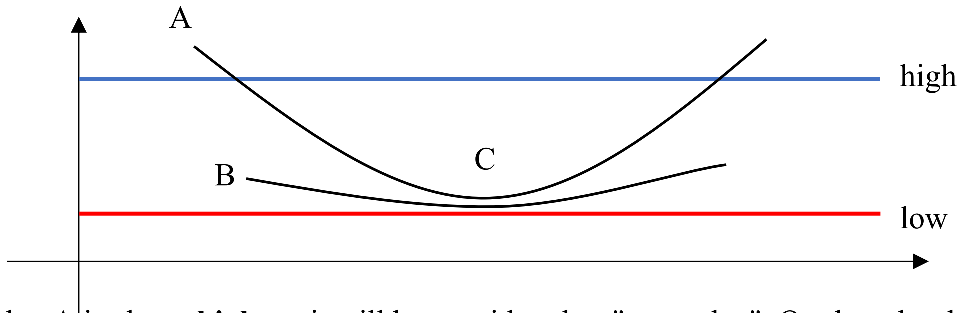
With gradient magnitude and direction, we need to scan the whole image to remove any unwanted pixels which may not constitute the edge. Every pixel is checked if it is a local maximum in its neighborhood in the direction of gradient.



In the image above, **A** is on the edge which is in vertical direction. Gradient direction is normal to the edge. Since **B** and **C** are in gradient directions, so **A** is checked with **B** and **C** to see if it forms a local maximum. If so, it is considered for next stage, otherwise, it is suppressed which become zero. After repeating these steps, we would get the thin edges as a result.

4. Thresholding

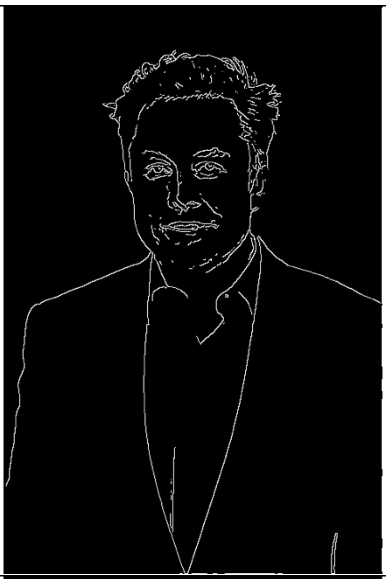
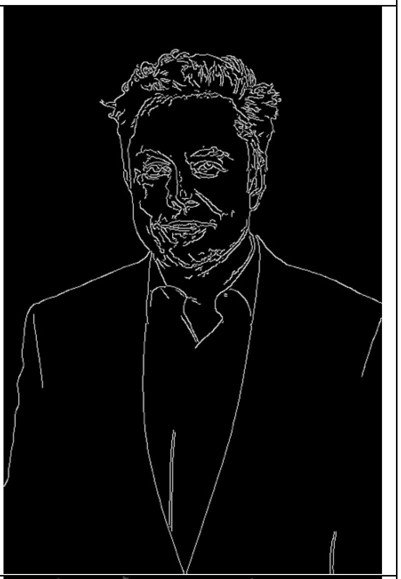
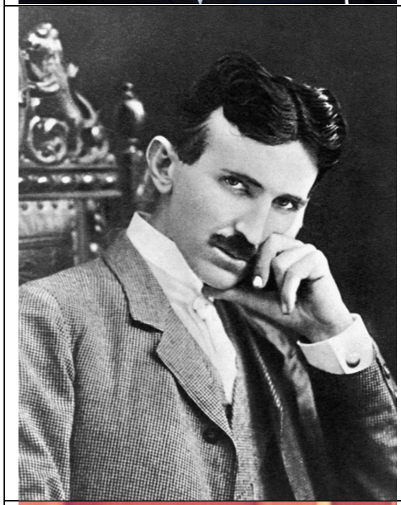
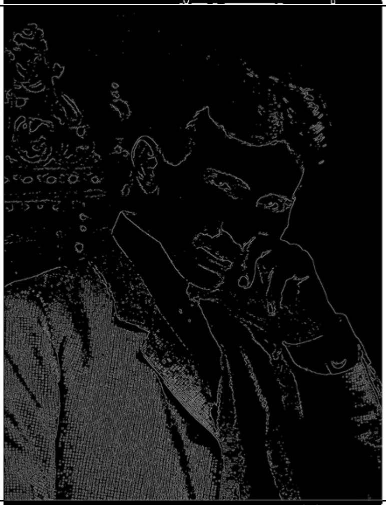
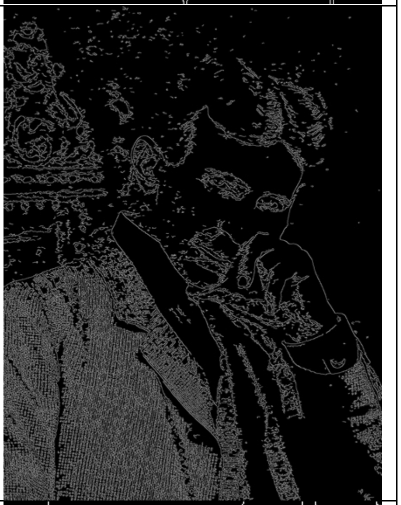
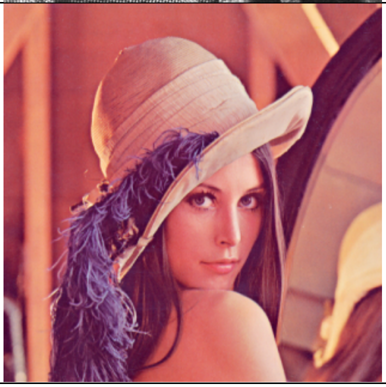
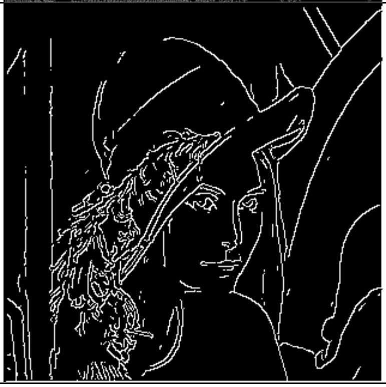
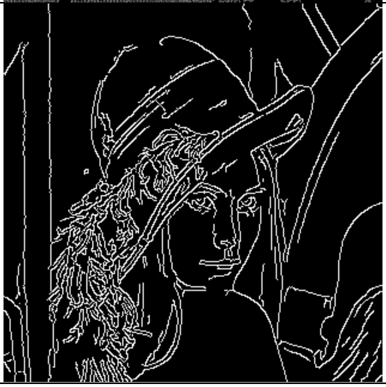
In this stage, we will decide if the edges that we have got in the last step are “sure-edge”. For this, we will define the threshold values, **low** and **high**. Any edges with intensity gradient more than **high** are sure to be edges and those below **low** are sure to be non-edges which are discarded. Those who lie between these two thresholds are classified edges or non-edges based on their connectivity. If they are connected to "sure-edge" pixels, they are considered to be part of edges. Otherwise, they are also discarded. Here is a concrete example:

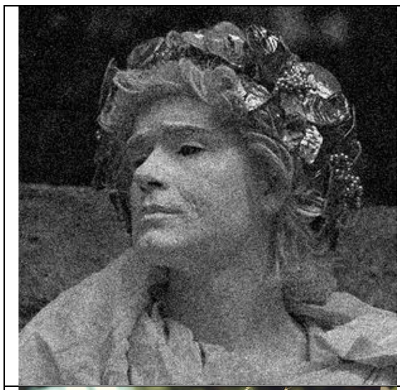
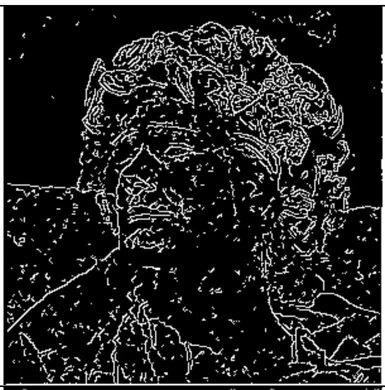
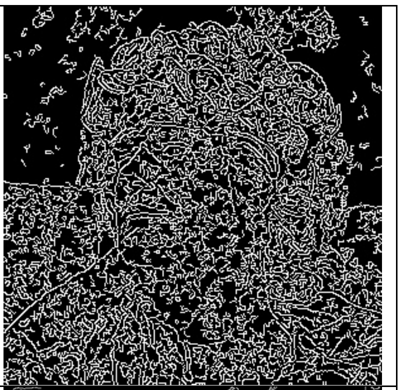
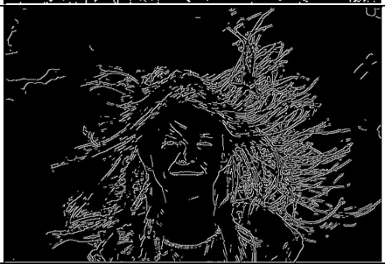
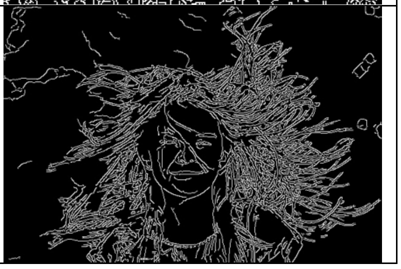


The edge A is above **high**, so it will be considered as "sure-edge". On the other hand, although edge C is below **high**, it is connected to edge A, so that also considered as valid edge and we get that full curve. But edge B, although it is above **low** and is in same region as that of edge C, it is not connected to any "sure-edge", so that is discarded. So, it is very important that we have to select **low** and **high** accordingly to get the correct result.

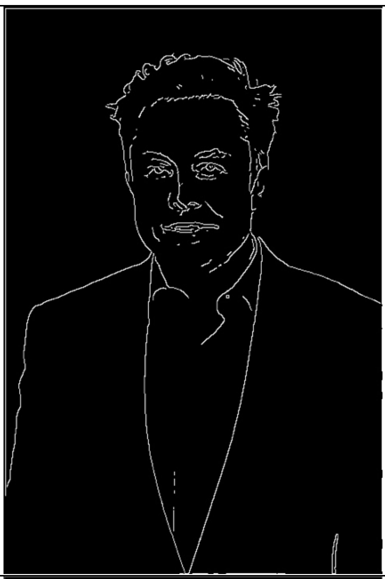
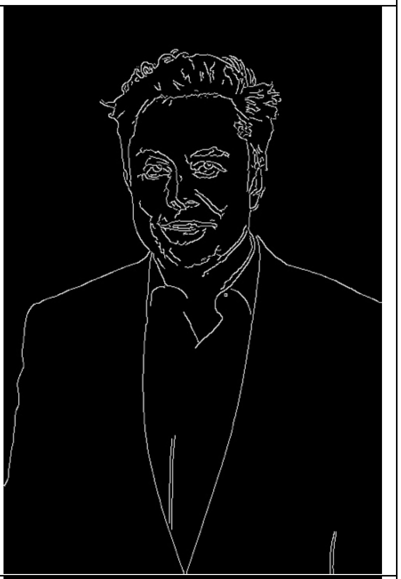
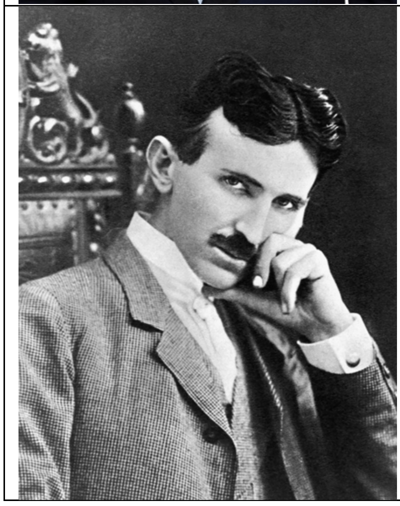
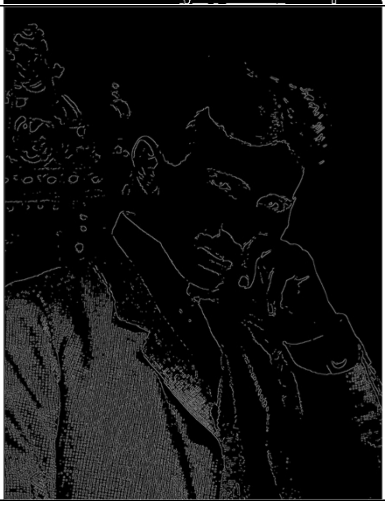
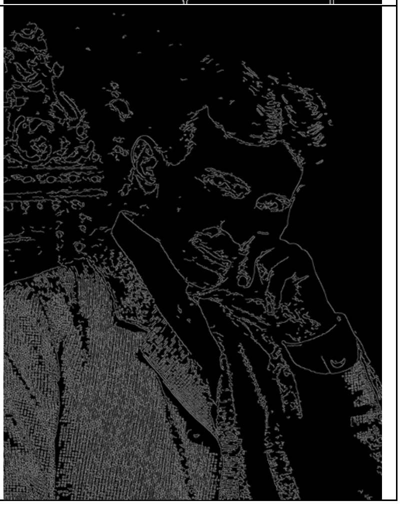
Experiments

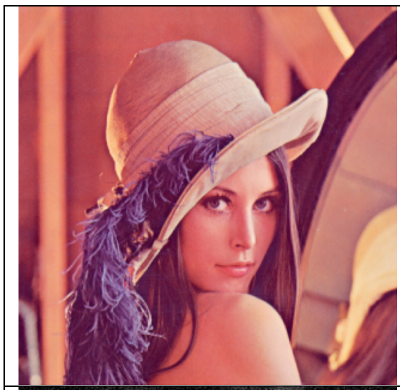
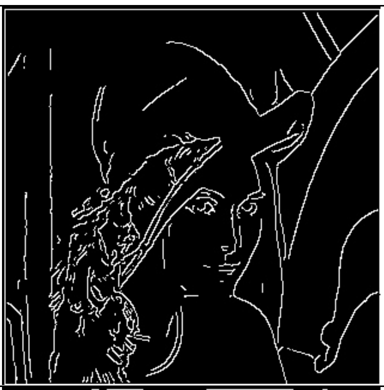
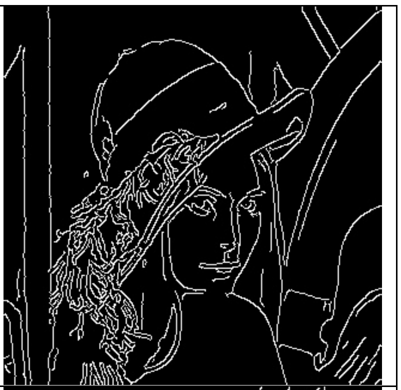
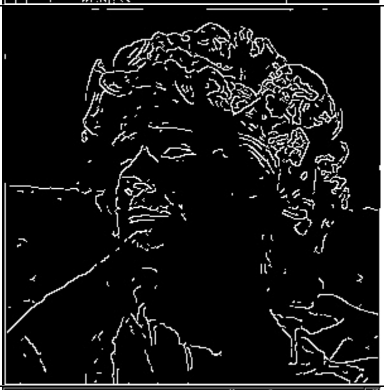
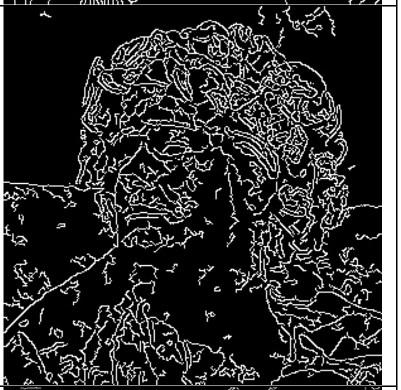
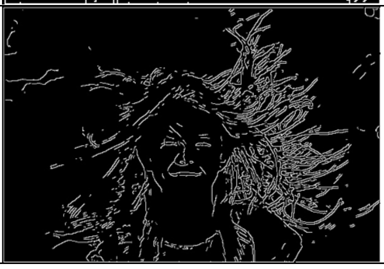
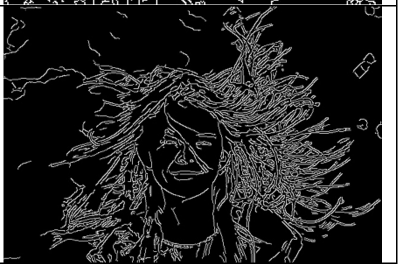
1. **Gaussian Kernel Size = 3, Low = 50, High = 100**

Input	Our Detector	CV Dector
		
		
		

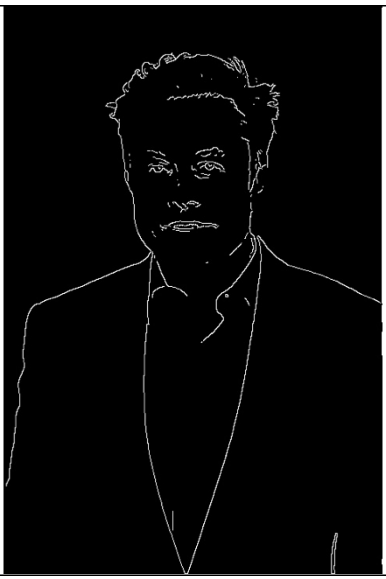
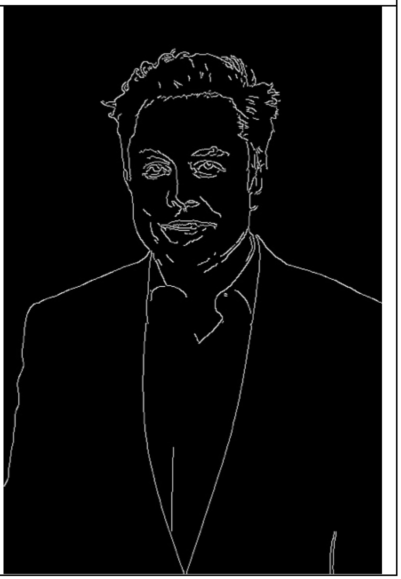
		
		

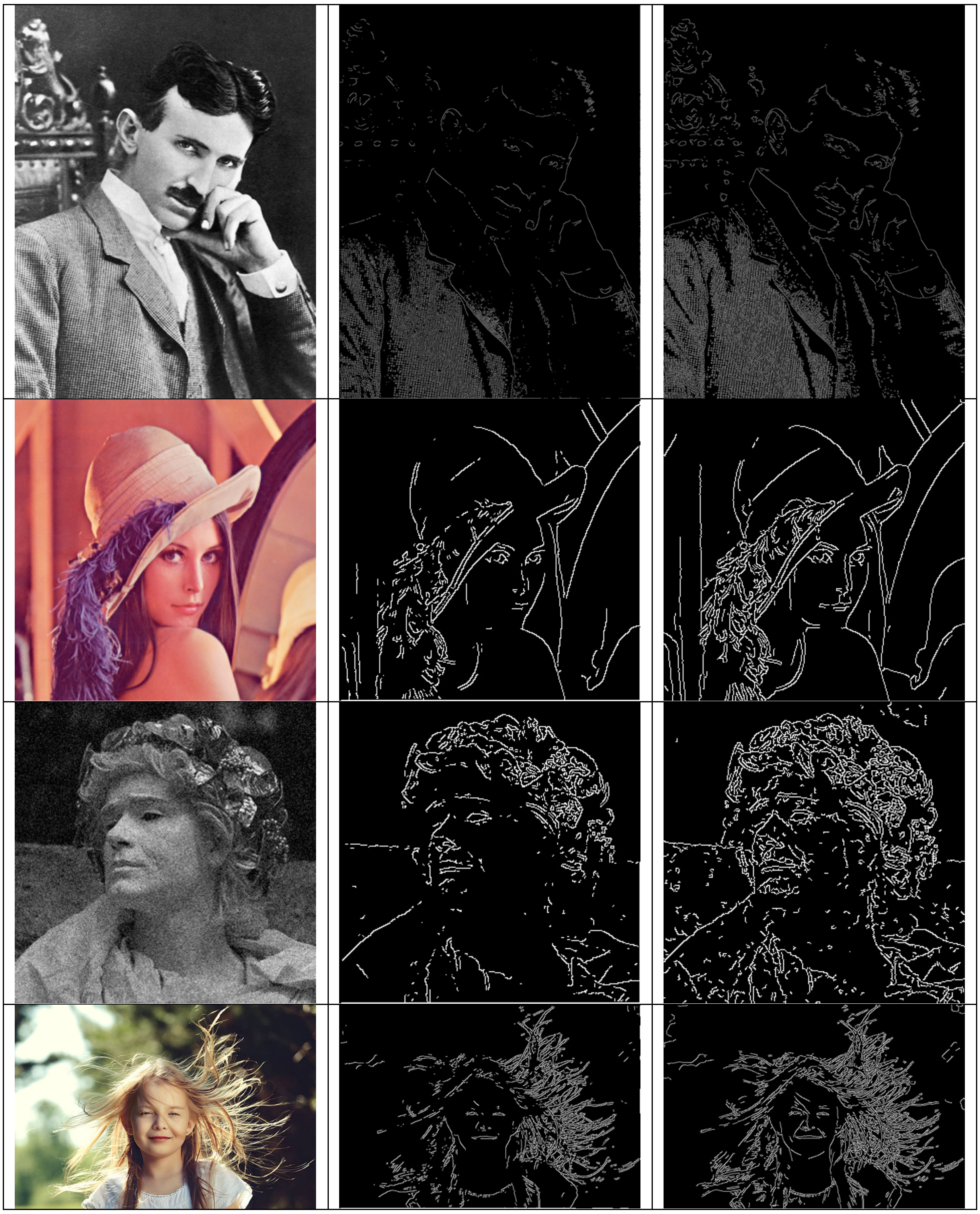
2. **Gaussian Kernel Size = 5, Low = 50, High = 100**

Input	**Our Detector**	**CV Dector**
		
		

3. Gaussian Kernel Size = 3, Low = 100, High = 150

Input	Our Detector	CV Dector
		



Discussion

1. **Pros and Cons of Using Sobel Filter**
   Sobel filter is an efficient way to detect the edges but it seems to fail to get the weak edges. A larger kernel size used in Sobel Kernel seems to be a way to improve its performance.
   Also, its localization is poor, which means you will see many edges where we actually should have only one edge.
   On the other hand, we may miss edges which are neither vertical or horizontal.

2. **Pros and Cons of Canny Edge Detection**
   With the help of gaussian filter and non- maxima suppression, it overcomes some of the issues when we use Sobel Filter alone. The signal can be enhanced with respect to the noise ratio by non-maxima suppression which results in one-pixel wide ridges as the output. Also, it gives a good localization, response and is immune to a noisy environment due to noise reduction.
   The result is quite decent if you pick the threshold right due to non-maxima suppression. However, it is hard to decide the threshold value from scratch.
   Also, the computing resources required is quite large due to its complexity, which is impossible to do it in real time.

3. **Pros and Cons of Gaussian Filter in Noise Reduction**
   Gaussian Filter is a quick solution of noise reduction but it may not perform well and may blur the edges sometimes.

Appendix

```
import numpy as np
import cv2
import math

def image_padding(image, padded_size):
    padded_image = np.zeros((image.shape[0] + padded_size,
image.shape[1] + padded_size))
    padded_image[padded_size : image.shape[0] + padded_size,
padded_size : image.shape[1] + padded_size] = image
    return padded_image

def gaussian_filter(image, kernel_size=3):
    sigma = 0.3*((kernel_size-1)*0.5 - 1) + 0.8
    padded_size = kernel_size // 2

    # initialize kernel
    ax = np.linspace(-(kernel_size - 1) / 2.0, (kernel_size - 1) / 2.0,
kernel_size)
    x, y = np.meshgrid(ax, ax)
    gaussian_kernel = np.exp(-((np.sqrt(x ** 2 + y ** 2)) ** 2 / (2.0 *
sigma ** 2)))
    gaussian_kernel = gaussian_kernel / np.sum(gaussian_kernel)

    # image padding
    padded_image = image_padding(image, padded_size)

    output = np.zeros((image.shape[0], image.shape[1]))
    for row in range(padded_size, padded_image.shape[0] - padded_size):
        for col in range(padded_size, padded_image.shape[1] -
padded_size):
            output[row - 1, col - 1] = np.sum(
                padded_image[
                    row - padded_size : row + padded_size + 1,
                    col - padded_size : col + padded_size + 1,
                ]
                * gaussian_kernel
            )
    return output

def sobel_filter(image):
    kernel_size = 3
    padded_size = kernel_size // 2
    sobel_dx = np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]])
    sobel_dy = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])

    gradient_magnitude = np.zeros((image.shape[0], image.shape[1]))
    gradient_angle = np.zeros((image.shape[0], image.shape[1]))

    # image padding
    padded_image = image_padding(image, padded_size)

    for row in range(padded_size, padded_image.shape[0] - padded_size):
        for col in range(padded_size, padded_image.shape[1] -
padded_size):
            gx = np.sum(
                padded_image[
                    row - padded_size : row + padded_size + 1,
                    col - padded_size : col + padded_size + 1,
                ]
                * sobel_dx
            )
            gy = np.sum(
                padded_image[
                    row - padded_size : row + padded_size + 1,
                    col - padded_size : col + padded_size + 1,
                ]
                * sobel_dy
            )
            theta = np.arctan(gy / gx)
            if theta is float("nan"):
                theta == 0
            gradient_angle[row - 1, col - 1] = theta
            magnitude = np.sqrt(gx ** 2 + gy ** 2)
            gradient_magnitude[row - 1, col - 1] = round(magnitude)
    return gradient_magnitude, gradient_angle

def non_maximum_suppression(gradient_magnitude, gradient_angle):
    output = np.zeros((gradient_magnitude.shape[0],
gradient_magnitude.shape[1]))
    for i in range(1, gradient_magnitude.shape[0] - 1):
        for j in range(1, gradient_magnitude.shape[1] - 1):
            before_pixel = 255
            after_pixel = 255
            if (0 <= gradient_angle[i, j] < PI / 8) or (15 * PI / 8 <=
gradient_angle[i, j] <= 2 * PI):
                before_pixel = gradient_magnitude[i, j + 1]
                after_pixel = gradient_magnitude[i, j - 1]
            elif (PI / 8 <= gradient_angle[i, j] < 3 * PI / 8) or (9 *
PI / 8 <= gradient_angle[i, j] < 11 * PI / 8):
                before_pixel = gradient_magnitude[i + 1, j - 1]
                after_pixel = gradient_magnitude[i - 1, j + 1]
            elif (3 * PI / 8 <= gradient_angle[i, j] < 5 * PI / 8) or
(11 * PI / 8 <= gradient_angle[i, j] < 13 * PI / 8):
                before_pixel = gradient_magnitude[i - 1, j]
                after_pixel = gradient_magnitude[i + 1, j]

            else:
                before_pixel = gradient_magnitude[i - 1, j - 1]
                after_pixel = gradient_magnitude[i + 1, j + 1]

            if (gradient_magnitude[i, j] >= before_pixel) and
(gradient_magnitude[i, j] >= after_pixel):
                output[i, j] = gradient_magnitude[i, j]
            else:
                output[i,j] = 0
    return output
def thresholding(suppression, low, high):
    output = np.zeros((suppression.shape[0], suppression.shape[1]),
dtype=np.uint8)
    padded_suppresion = image_padding(suppression, 2)

    for i in range(1, suppression.shape[0] + 1):
        for j in range(1, suppression.shape[1] + 1):
            if padded_suppresion[i, j] > high:
                output[i - 1, j - 1] = 255
            elif padded_suppresion[i, j] < low:
                continue
            else:
                for dx in range(i - 1, i + 2):
                    for dy in range(j - 1, j + 2):
                        if padded_suppresion[dx, dy] > high:
                            output[i - 1, j - 1] = 255
                            break
                        break

    return output

def canny_edge_detection(image, low, high, kernel_size=3):
    blurred_image = gaussian_filter(image, kernel_size)
    gradient_magnitude, gradient_angle = sobel_filter(blurred_image)
```

```python
    suppression = non_maximum_suppression(gradient_magnitude,
gradient_angle)
    output = thresholding(suppression, low, high)

    return output

LOW = 50
HIGH = 100

for i in range(1,6):
  img_bgr = cv2.imread(f"{i}.jpeg", cv2.IMREAD_COLOR)
  img_gray = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2GRAY)
  output = canny_edge_detection(img_gray, LOW, HIGH, kernel_size=5)
  cv2.imwrite(f"{i}-my.jpeg", output)
  # cv2.imshow("output", output)

  cv_blurred = cv2.GaussianBlur(img_gray, (5, 5), 0)
  cv_canny = cv2.Canny(cv_blurred, LOW, HIGH)
  cv2.imwrite(f"{i}-cv2.jpeg", cv_canny)
  # cv2.imshow("CV output", cv_canny)
  # cv2.waitKey(0)
```