Project1 Report

Wenbo Du

Readme

All input images are in the folder named ‘lane’.
The generated images will be saved in the same folder as the code.
The image named ‘output image.jpeg’ is the final result.
After the calculation, press keyboard 0 to process the next image.

Method

The algorithm we proposed contains three parts: pre-processing module, processing module and post-processing module.

a. Pre-processing module

First, we calculate the size of the image, and if it is too large (height > 1000 or width > 1000), then we need to reduce its size:

```
frame = cv2.imread(path)
height, width, channels = frame.shape
print(frame.shape)
if height > 1000 or width > 1000:
    factor = max(height, width)
    frame = cv2.resize(frame, None, fx=1000/factor, fy=1000/factor, interpolation=cv2.INTER_LINEAR)
    height, width, channels = frame.shape
cv2.imshow("original image", frame)
cv2.waitKey(0)
```

The, we convert the original image into RGB format:

```
frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
print(frame.shape)
cv2.imshow("RGB image", frame)
cv2.waitKey(0)
```

The third step is to call function named detect_lane to process the image:

```
def detect_lane(frame, height, width):
```

b. Processing module

First, we convert RGB image into HLS format and separate s channel:

```
frame_0 = cv2.cvtColor(frame, cv2.COLOR_RGB2HLS)
h_channel = frame_0[:, :, 0]
cv2.imshow("h_channel image", h_channel)
cv2.waitKey(0)
l_channel = frame_0[:, :, 1]
cv2.imshow("l_channel image", l_channel)
cv2.waitKey(0)
s_channel = frame_0[:, :, 2]
cv2.imshow("s_channel image", s_channel)
cv2.waitKey(0)
```

Next, we use functions named cv2.kmeans and cv2.threshold to do the Optimal Thresholding to the s channel:

```
Z = s_channel.reshape((-1, 1))
# convert to np.float32
Z = np.float32(Z)
# define criteria, number of clusters(K) and apply kmeans()
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
K = 3
ret, label, center = cv2.kmeans(Z, K, None, criteria, 10, cv2.KMEANS_RANDOM_CENTERS)
# Now convert back into uint8, and make original image
center = np.uint8(center)
res = center[label.flatten()]
res2 = res.reshape((s_channel.shape))
cv2.imshow("threshold after s_channel image", res2)
cv2.waitKey(0)
print(center)
max_num = max(center[0][0], max(center[1][0], center[2][0]))
min_num = min(center[0][0], min(center[1][0], center[2][0]))
print(max_num)
print(min_num)
for i in range(K):
    if center[i][0] != max_num and center[i][0] != min_num:
        mid_num = center[i][0]
ret, binary_0 = cv2.threshold(res2, min_num, 255, cv2.THRESH_BINARY)
print(binary_0)
cv2.imshow("threshold after s_channel image", binary_0)
cv2.waitKey(0)
```

After that, we convert RGB image into gray level image:

```
frame_0 = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
```

Now, we introduce Sobel detection by using three different types of cv2.Sobel functions:

```
edges = cv2.Sobel(frame_0, cv2.CV_16S, 1, 1)
edges = cv2.convertScaleAbs(edges)
cv2.imshow("sobel image", edges)
cv2.waitKey(0)

edgesh = cv2.Sobel(frame_0, cv2.CV_16S, 1, 0)
edgesh = cv2.convertScaleAbs(edgesh)
cv2.imshow("sobel horizontal image", edgesh)
cv2.waitKey(0)

edgesv = cv2.Sobel(frame_0, cv2.CV_16S, 0, 1)
edgesv = cv2.convertScaleAbs(edgesv)
cv2.imshow("sobel vertical image", edgesv)
cv2.waitKey(0)
```

The we first combine last two images:

```
grad = cv2.addWeighted(edgesh, 0.5, edgesv, 0.5, 0)
cv2.imshow("vertical + horizontal image", grad)
cv2.waitKey(0)
```

Then we combine the first image and the new image:

```
gradd = cv2.addWeighted(grad, 0.7, edges, 0.3, 0)
cv2.imshow("sobel + vertical + horizontal image", gradd)
cv2.waitKey(0)
```

Thirdly, we combine the above image and the processed s channel image:

```
graddd = cv2.addWeighted(gradd, 0.75, binary_0, 0.25, 0)
cv2.imshow("sobel + vertical + horizontal + s_channel image", graddd)
cv2.waitKey(0)
```

In the next step, we also do the Optimal Thresholding to the combined image:

```
Z = graddd.reshape((-1, 1))
# convert to np.float32
Z = np.float32(Z)
# define criteria, number of clusters(K) and apply kmeans()
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
K = 2
ret, label, center = cv2.kmeans(Z, K, None, criteria, 10, cv2.KMEANS_RANDOM_CENTERS)
# Now convert back into uint8, and make original image
center = np.uint8(center)
print(center)
res = center[label.flatten()]
res2 = res.reshape((graddd.shape))

ret, binary = cv2.threshold(res2, int((center[0] + center[1])/2), 255, cv2.THRESH_BINARY_INV)
cv2.imshow("threshold after sobel + vertical + horizontal + s_channel image", binary)
cv2.waitKey(0)
print(binary)
```

Then, we need to create ROI. We know most lane are white or yellow, so we need to

select yellow and white parts from the original image:

```
frame_hsv_0 = cv2.cvtColor(frame, cv2.COLOR_RGB2BGR)
frame_hsv = cv2.cvtColor(frame_hsv_0, cv2.COLOR_BGR2HSV)
lower_yw = np.array([15, 85, 85], dtype="uint8")
upper_yw = np.array([30, 255, 255], dtype="uint8")
mask_yw = cv2.inRange(frame_hsv, lower_yw, upper_yw)
mask_wt = cv2.inRange(frame_0, 200, 255)
mask = cv2.bitwise_or(mask_wt, mask_yw)
frame_mask = cv2.bitwise_and(frame_0, mask)
cv2.imshow("mask image", frame_mask)
cv2.waitKey(0)
```

And we also need to combine the ROI and the previous image:

```
graddd = cv2.addWeighted(graddd, 0.8, frame_mask, 0.2, 0)
cv2.imshow("sobel + vertical + horizontal + s_channel + ROI image", graddd)
cv2.imwrite('sobel + vertical + horizontal + s_channel + ROI image.jpeg', graddd)
cv2.waitKey(0)
```

Then, we do the Gaussian blur by using cv2.GaussianBlur function:

```
kernel_size_temp = round(((height * width) / (80 * 80)) ** 0.5)
if kernel_size_temp % 2 == 0:
    kernel_size_temp = kernel_size_temp + 1
kernel_size = max(9, kernel_size_temp)
print(kernel_size)
frame_1 = cv2.GaussianBlur(binary, (kernel_size, kernel_size), 1)
cv2.imshow("GaussianBlur image", frame_1)
cv2.waitKey(0)
```

The kernel size is self-adaptive to the image size. If the image size is large, then the kernel size will also be relatively large.

After doing the Gaussian blur, we continue do the median blur to the image by using cv2. medianBlur function:

```
kernel_size_temp = round(((height * width) / (180 * 180)) ** 0.5)
if kernel_size_temp % 2 == 0:
    kernel_size_temp = kernel_size_temp + 1
kernel_size = max(5, kernel_size_temp)
print(kernel_size)
frame_2 = cv2.medianBlur(frame_1, kernel_size)
cv2.imshow("medianBlur image", frame_2)
cv2.waitKey(0)
print('frame_2', frame_2)
```

The kernel size is also self-adaptive.

Then, we use cv2.Canny to do the Canny detection:

```
low_bound = 600
upper_bound = 800
frame_3 = cv2.Canny(frame_2, low_bound, upper_bound)# + cv2.Canny(s_channel, low_bound, upper_bound)
cv2.imshow("Canny image", frame_3)
cv2.waitKey(0)
```

And we use cv2. HoughLinesP to detect lines:

```
lines_0 = cv2.HoughLinesP(frame_3, rho=1, theta=np.pi/180, threshold=round((height + width) / 30),
                          minLineLength=round((height + width) / 30), maxLineGap=round((height + width) / 30))
```

Finally, we add detected lines to the original image:

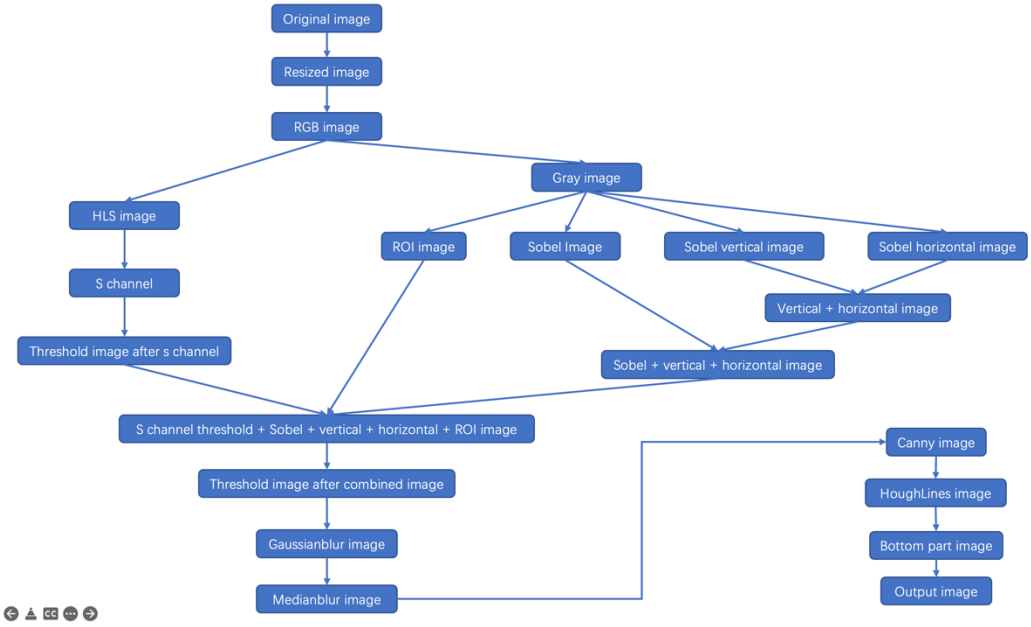
```
lines = []
if lines_0 is not None:
    lines.extend(lines_0)
#if lines_1 is not None:
#    lines.extend(lines_1)
print(type(lines_0))
if lines is not []:
    for line in lines:
        for x1, y1, x2, y2 in line:
            if (y1 >= height / 3) and (y2 >= height / 3):
                cv2.line(frame, (x1, y1), (x2, y2), color=(255, 0, 0), thickness=3)
            elif (y1 < height / 3) and (y2 >= height / 3):
                y3 = height / 3
                x3 = ((y3 - y1) / (y2 - y1)) * (x2 - x1) + x1
                print(x3, y3)
                cv2.line(frame, (round(x3), round(y3)), (x2, y2), color=(255, 0, 0), thickness=round((height + width) / 500))
            elif (y1 >= height / 3) and (y2 < height / 3):
                y3 = height / 3
                x3 = ((y3 - y1) / (y2 - y1)) * (x2 - x1) + x1
                print(x3, y3)
                cv2.line(frame, (x1, y1), (round(x3), round(y3)), color=(255, 0, 0), thickness=round((height + width) / 500))
```

c. Post-processing module

From the empirical perspective, the upper part of the picture is generally independent of the lane, so we create the ROI which only cover the bottom half of the picture:

```
elif (y1 < height / 3) and (y2 >= height / 3):
    y3 = height / 3
    x3 = ((y3 - y1) / (y2 - y1)) * (x2 - x1) + x1
    print(x3, y3)
    cv2.line(frame, (round(x3), round(y3)), (x2, y2), color=(255, 0, 0), thickness=round((height + width) / 500))
elif (y1 >= height / 3) and (y2 < height / 3):
    y3 = height / 3
    x3 = ((y3 - y1) / (y2 - y1)) * (x2 - x1) + x1
    print(x3, y3)
    cv2.line(frame, (x1, y1), (round(x3), round(y3)), color=(255, 0, 0), thickness=round((height + width) / 500))
```

After the calculation, press keyboard 0 to process next image.
The algorithm flowchart is shown below:



Experiment Results

The original image is shown below:

The s channel of HLS image is shown below:

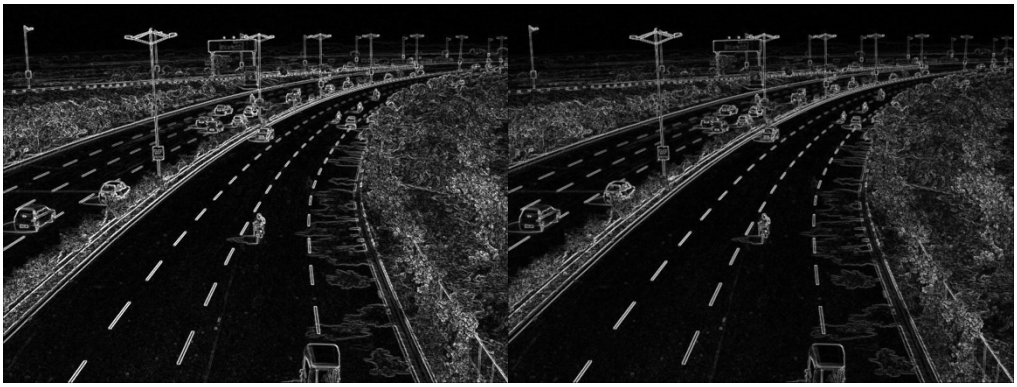
The image after Optimal Thresholding is shown below:

The gray image is shown below:

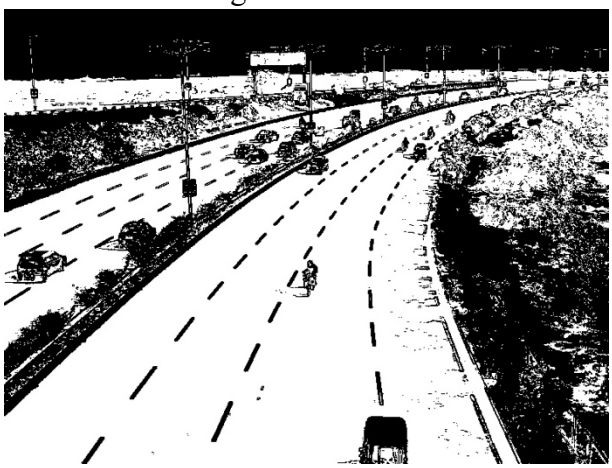
Images after three different type of Sobel detection are shown below:

The ROI image is shown below:

The combined images are shown below:



The image after Optimal Thresholding is shown below:

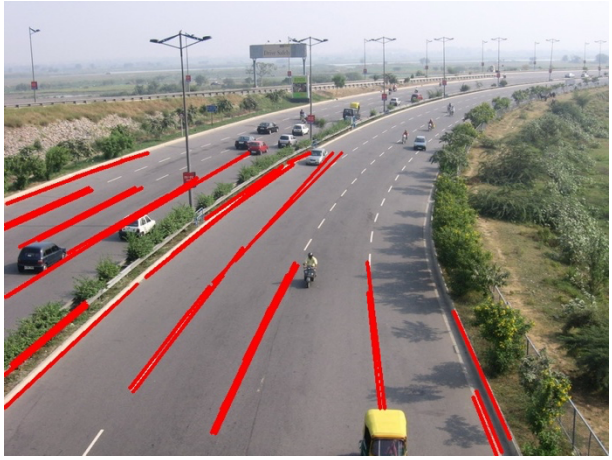


The image after Gaussian blur is shown below:

The image after median blur is shown below:

The image after Canny detection is shown below:

The output image after HoughLines detection is shown below:



Future Work

We also attempted to introduce the outline of object as a part of features by using function called cv2.findContours, however, the results are not idea. So, we abandoned it finally. In the future, we plan to find another method to capture the outline of object, which we believe can improve the performance of the algorithm and robustness.

And we also want to improve the function of the algorithm which can merge adjacent segments. However, due to limited time, we didn’t finish that part.
Besides, we are also working on self-adaptive ROI.