Finding Lane Lines on the Road Project

The goal of my project is to develop a pipeline for finding lane lines on the road.

Reflection

Describe your pipeline. As part of the description, explain how you modified the draw_lines() function.

In the following steps I explains the whole of my solution section by section.

Import Packages

Most of the time I use the first section of the Jupyter notebook for importing the packages, which I will use in my solution later. Of course, it is possible to import the packages in the middle section, but I use the first section to have more legibility.

Read in an Image

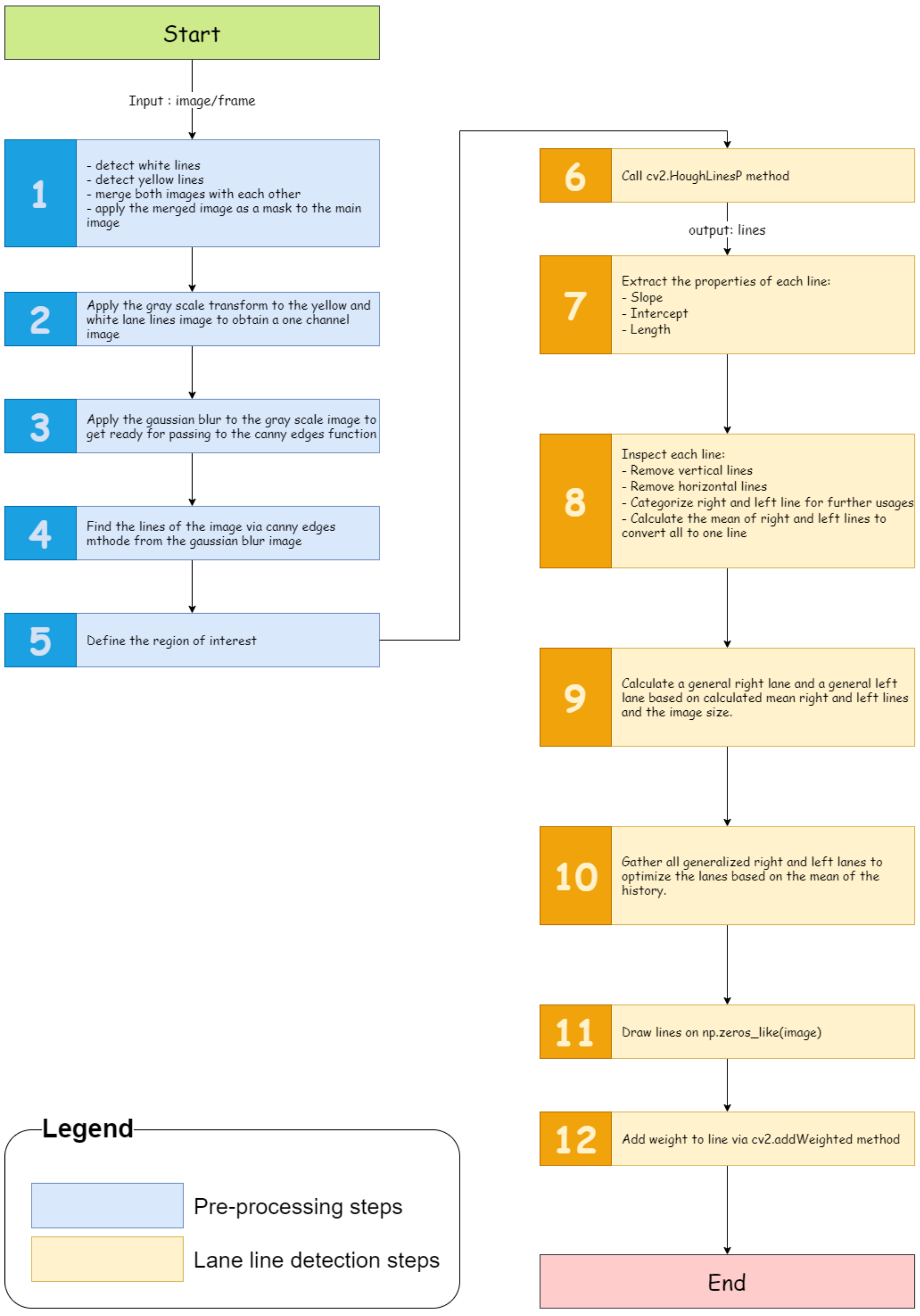
The "read in an image" section is only for reading in an image and displaying it the Jupyter notebook.

Helper Functions

Helper functions section was already in the Notebook, I only developed some new methods to the helper function. The helper functions are somehow remarkably interesting. They are like a step waterfall.

The output of a helper function is the input for the next helper function. The following diagram demonstrates the sequence of calling functions in my pipeline.

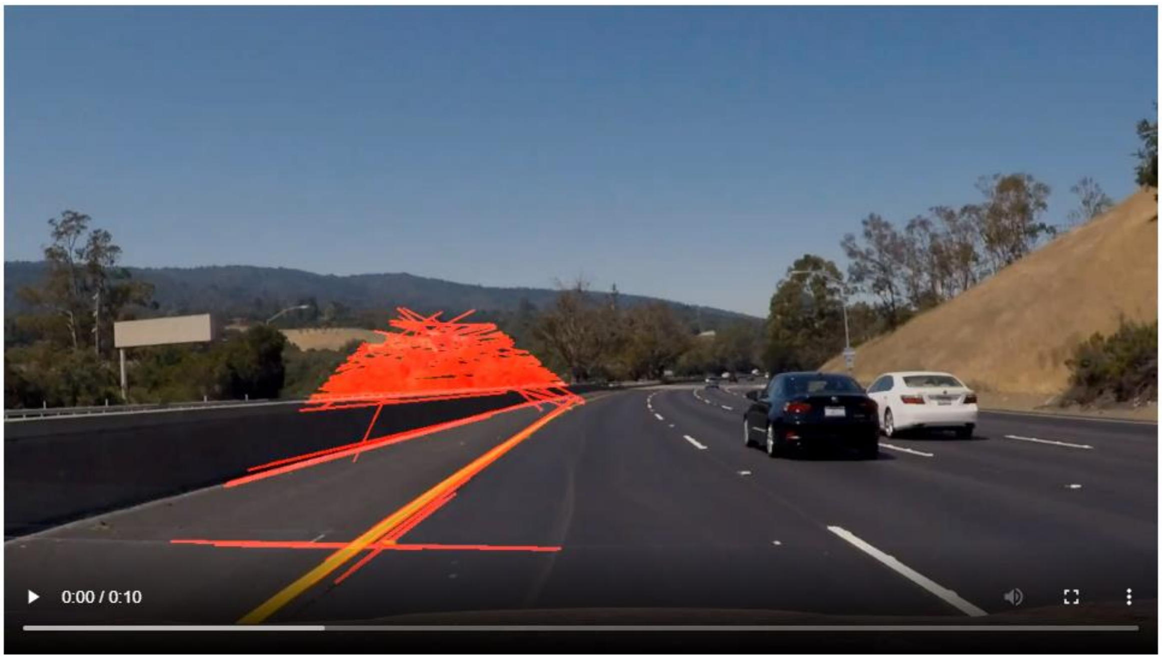
Lane line detection and drawing logic


Start
Input : image/frame
1
- detect white lines
- detect yellow lines
- merge both images with each other
- apply the merged image as a mask to the main image
2
Apply the gray scale transform to the yellow and white lane lines image to obtain a one channel image
3
Apply the gaussian blur to the gray scale image to get ready for passing to the canny edges function
4
Find the lines of the image via canny edges mthode from the gaussian blur image
5
Define the region of interest
6
Call cv2.HoughLinesP method
output: lines
7
Extract the properties of each line:
- Slope
- Intercept
- Length
8
Inspect each line:
- Remove vertical lines
- Remove horizontal lines
- Categorize right and left line for further usages
- Calculate the mean of right and left lines to convert all to one line
9
Calculate a general right lane and a general left lane based on calculated mean right and left lines and the image size.
10
Gather all generalized right and left lanes to optimize the lanes based on the mean of the history.
11
Draw lines on np.zeros_like(image)
12
Add weight to line via cv2.addWeighted method
End
Legend
Pre-processing steps
Lane line detection steps

Detect white and yellow lane lines with RGB

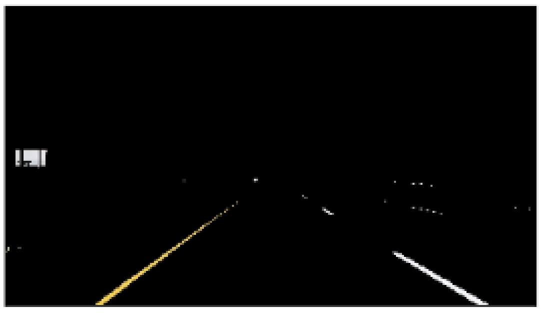
[This function is added to my second pipeline]

The **detect_white_yellow_lanes** function detects the white and yellow part of the input image. My first pipeline did not have this helper function. Next figure was the output of my lane line detection for the challenge video.



Therefore, I decided to remove the unnecessary part of the image and pass an optimized image to detect **canny** edge step. Because a better output from canny function can cause a better output from **hough_line** function.

Another advantage of detecting the white and yellow lane lines at the beginning of the process, that is to have a lightweight process.



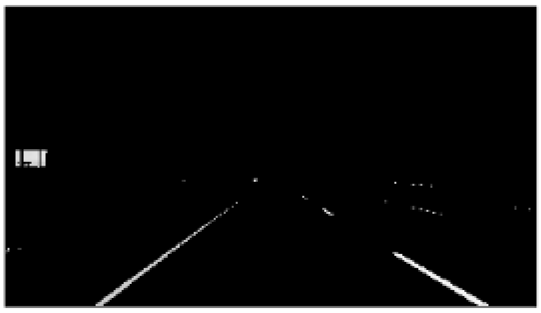
The above figure is the image, which is used in the whole lane line detection process.

Convert to gray scale

The **grayscale** function was already available in Helper function section and it is one of the steps which must be applied to an image that will be passed to the **canny** function later. I consider the **grayscale** function as a preprocessing step. This function applies the gray scale to the input image and the output is a one channel image.

Before using the white-yellow color detection

After using the white-yellow color detection

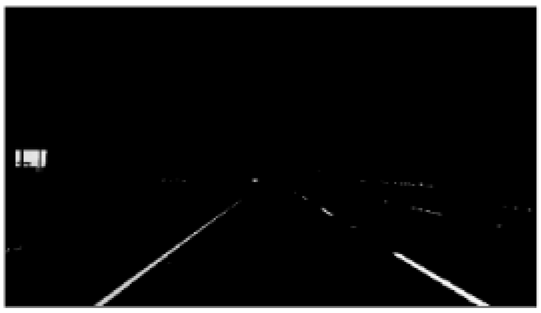


Generate the Gaussian blur of the image

The **gaussian_blur** function applies the gaussian noise to the input image based on the kernel-size. This function is also a preprocessing function because these functions are preparing the image for the main process, which is lane line detection.

Before using the white-yellow color detection

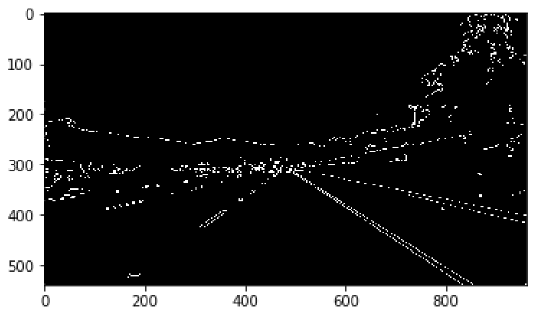
After using the white-yellow color detection



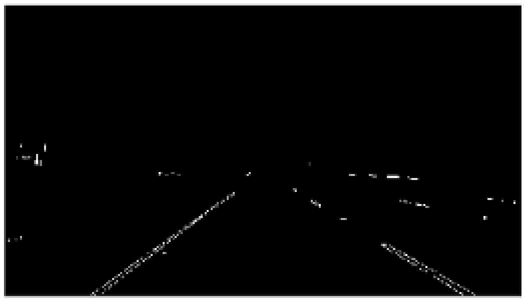
Detect Canny edges

After applying the **gray scale transformation** and **gaussian blur**, the image is ready for detecting the edges. The **canny** function detects the edges of the input image.

Before using the white-yellow color detection



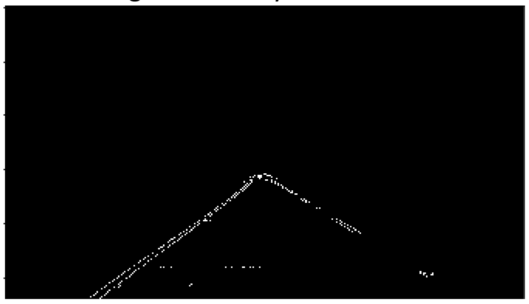
After using the white-yellow color detection



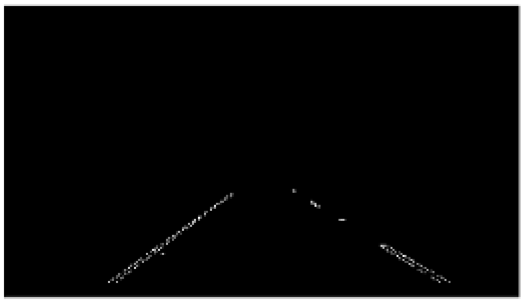
Define the region of the interest

The **region_of_interest** function specifies the area of the image which is more interesting for lane line detection processing. This function applies a mask based on the input vertices to the input image. As the output of this step is the input of the next step, the lane line detection processes only this area. It helps to have a lightweight image processing and a better performance.

Before using the white-yellow color detection



After using the white-yellow color detection



Detect the lines with Hough transform

The **hough_lines** function detects lines in the input image based on the threshold. The output of this function are only lines with the following format.

```
( [[[ 669, 417, 787, 417]],
   [[ 490, 313, 658, 406]]] )
```

Extract line properties

[This function is added to my second pipeline]

The **line_properties_extractor** function receives a line as an input parameter. In this step I decided to do the following clean ups:

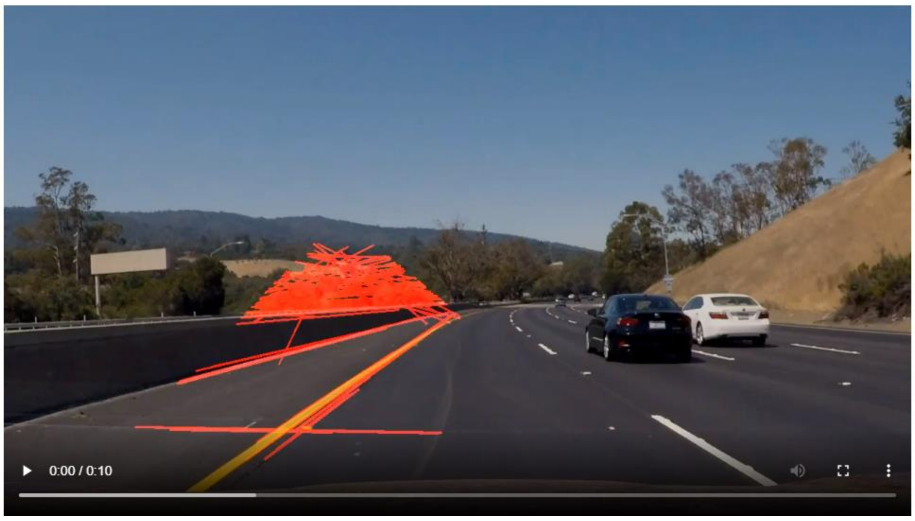
- Remove the vertical lines
- Remove the horizontal lines

Because the vertical and horizontal lines are not the good line candidates for my project. The output of this function is used in the **inspect_lines** function and has been explained in the inspect lines section.

Inspect lines

[This function is added to my second pipeline]

As I explained before the output of my first pipeline was not what I expected, see the figure below. In addition to select the white and yellow part of the image at the beginning of the pipeline to eliminate the unnecessary part of the image, I decided to inspect the lines (hough_line's output) properties and calculate the best line candidates for left and right lanes. The output of this function is only two lines. One for left lane line and the second is the right lane line.



The **inspect_lines** function inspects each output lines from **hough_line** function and only consider the best candidate for the next step. The **inspect_lines** function calls the **line_properties_extractor** function to extract the line properties like slope, intercept, and length and each line is inspected based on these properties.

If the slope or the intercept or the length of a line is None, this line is not a good line candidate. Then based on the value of the slope is distinguished if a line is the left and the right lane line. The left and right lane lines are gathered in two different lists. The (slope, intercept) in a list and the length in another list.

And in the last part of the function the average of slope and intercept is calculated separately for left and right lane lines. The output of this function is the average slope and intercept for the left and right lane lines.

Calculate general lane lines

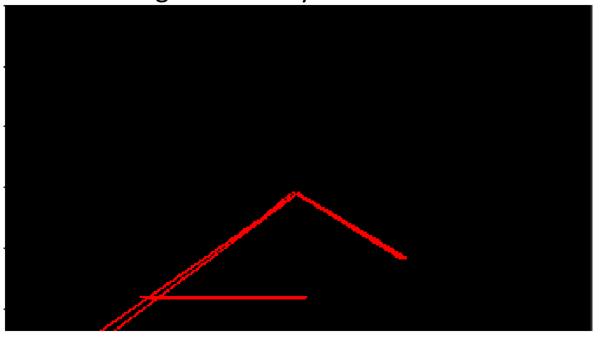
[This function is added to my second pipeline]

The **calculate_general_lane_lines** function is for extending the best candidates lane lines to have a full-length line. Because the previous functions are working with slope and intercept, in this step I can calculate the two endpoints of left and right lane lines easily.

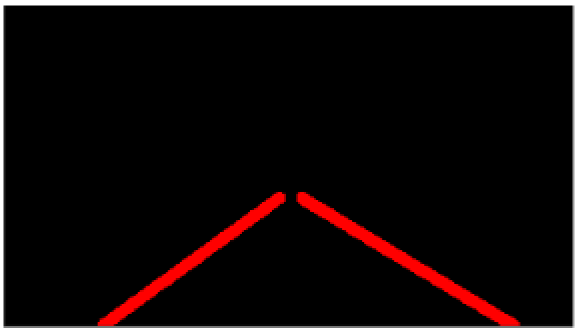
Draw lane lines

The **draw_lane_lines** function draws the full-length lines that calculated in the previous step on a zero array with the same shape of the image, with red color and defined thickness.

Before using the white-yellow color detection

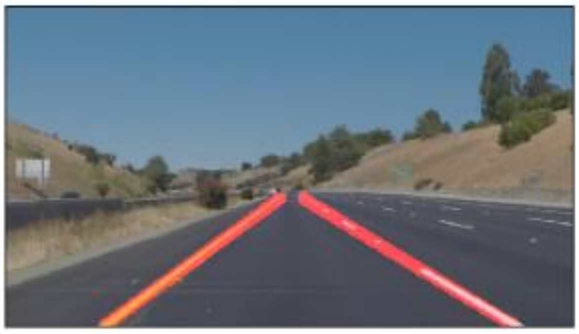


After using the white-yellow color detection



Add weight to the drew lane lines

The **weighted_img** function merges the lane lines image from the previous step with the initial image but with the defined weight that the calculated lane lines from pipeline are a bit transparent.



Test Images

In this section of the Jupyter notebook I have defined a class to manage the methods and properties. All the default values are defined in this class. Right now, most of the properties are private properties.

The main function in this class is the **processor** function. This is responsible for processing each frame of the video and the sequence of the process in this function is demonstrated in the diagram above.

After testing my second pipeline with an optimized and improved logic, I noticed that the lane line is not drawn in some frame. Therefore, I improved the pipeline for the third time.

[This logic added to my third pipeline]

To solve the above problem I defined two lists for the right and left lanes and gathered the all the lane lines which are drawn on the frames to calculate the mean of them and then the newly calculated mean lane line is drawn on the image.

Identify potential shortcomings with your current pipeline

- If I use another video with a different camera view, the region of interest cannot adjust automatically.

Suggest possible improvements to your pipeline

- Calculate the region of the interest based on the camera view.