Finding Lines on the Road

The goals of this project are the following:

- Make a pipeline that detects lane lines on the road
- Reflect on your work in a written report

Reflection

Pipeline

The pipeline consists of 6 steps:

- Convert the image to grayscale
- Apply gaussian blur filter
- Use Canny edge detector to detect the edges on the image
- Form the region of interest (RoI)
- Apply Hough transform to the masked part of the image
- Draw lines on the original image

To draw a single line on the left and right lanes the function `draw_lines()` was modified in the following way:

1. Check the slope of the line, and separate lines into two categories: right and left. I also kept the slope bounded in order to exclude anomaly cases, i.e. when the detected line is horizontal, vertical or has a very steep angle, etc.
2. Collect coordinates (x, y) for both line types;
3. Update the least squares polynomial fit of the slope and bias terms (a and b in $y = ax + b$ equation for a line). I updated the estimates using the moving average formula for the vector $\boldsymbol{q} = [a, b]$:

$$\boldsymbol{q}_{current} = \frac{\boldsymbol{q}_{previous}(w-1) + \boldsymbol{q}_{new}}{w}$$

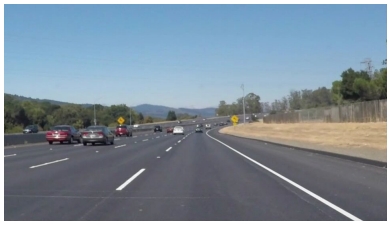
   where w is the sliding window length. [1]

4. Calculate line coordinates based on (a, b) estimates and RoI size;

[1] This required me to use global variables in Python which might not be effective way of doing it...

5. Draw two lines.

Here are the examples of pipeline work for different images:



(a) Original image

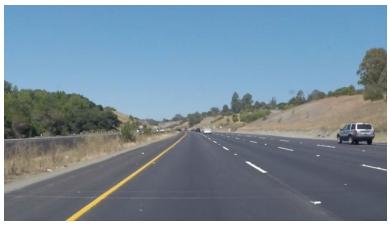
(b) Image with lines detected

Figure 1: solidWhiteCurve.jpg

(a) Original image

(b) Image with lines detected

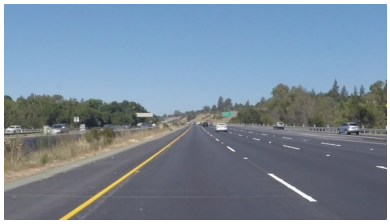
Figure 2: solidWhiteRight.jpg



(a) Original image

(b) Image with lines detected

Figure 3: solidYellowCurve.jpg



(a) Original image

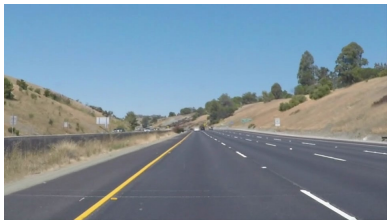
(b) Image with lines detected

Figure 4: solidYellowLeft.jpg

(a) Original image

(b) Image with lines detected

Figure 5: solidYellowCurve2.jpg



(a) Original image

(b) Image with lines detected

Figure 6: whiteCarLaneSwitch.jpg

Potential shortcomings

There are different potential shortcomings:

- Hand-picked parameters (for instance values for the slope bound and RoI vertices, thresholds, etc.).
  - It is easy to come up with a case when chosen values for slope bounds or RoI might not be optimal or could even be wrong. For example if a car goes uphill or downhill the optimal mask required to detect lines successfully will be different from the one used for the flat road. [2]

[2] The problem with nonadaptive RoI choice is very noticeable in the example where I apply

  - Non-adaptive edge filtering will stop working once a car goes under bright sunlight or on a dark road, i.e. when analysed image suddenly changes its average brightness. Moving averaging of the line slopes implemented in the `draw_lines()` function partly solves this problem, however it will not work for the described case.

- Computation time. Proposed algorithm can be demanding for computation resources, especially in the case of increased video resolution or any additional processing.

- Difficulties. Many cars on the road (i.e. traffic jams) - a car in front of you decreases line visibility and has details which the algorithm might consider to be features. Another difficulty - dirty, poor quality roads with many features that need to be carefully analysed.

Possible improvements

Here is a list of possible improvements:

- Robust regression methods for line fitting. They will help to avoid a very tight bounding for the line slopes. A good example could be the RANSAC algorithm from the Scikit Learn package: link. This algorithm would effectively penalise outliers that least squares are sensitive to. The drawback is computation time.

- Another potential improvement could be to choose canny edge detection threshold parameters based on the average brightness of the region of interest. This would improve robustness for bright areas of the road.

- Two separate masks for each line will improve the detectability as well as robustness. However it will lead to the increased computation demands.

- Computations

  - The more effective Hough transform algorithm or a variation. For instance, in the paper "A simple and robust line detection algorithm based on small eigenvalue analysis" (link) by D. S. Guru and others, the authors suggest an alternative approach that provides a similar (or even better) performance as Hough transform, and outperforms the standard realisation by a factor of 5-10 in terms of computation time.
  - Recursive least squares (RLS, link) fitting algorithm which would improve computation time required for estimation, i.e. make the algorithm more efficient for real-time implementation.

---

the line detection algorithm for my own video (see P1 notebook). In this case I need to use completely different mask in order to get acceptable performance