Advanced Lane Finding Project

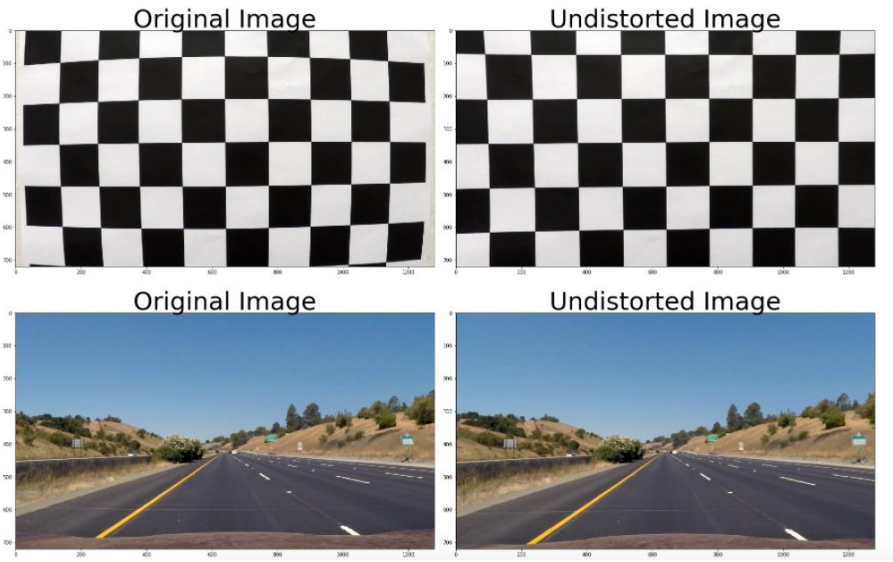
The goals / steps of this project are the following:

- Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
- Apply a distortion correction to raw images.
- Use color transforms, gradients, etc., to create a thresholded binary image.
- Apply a perspective transform to rectify binary image ("birds-eye view").
- Detect lane pixels and fit to find the lane boundary.
- Determine the curvature of the lane and vehicle position with respect to center.
- Warp the detected lane boundaries back onto the original image.
- Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

Camera Calibration

OpenCV library has two helper methods: findChessboardCorners and calibrateCamera. These methods were used on chessboard images for finding corners and matrix coefficients of image distortion. The functions read_points and cal_undistort used for this.

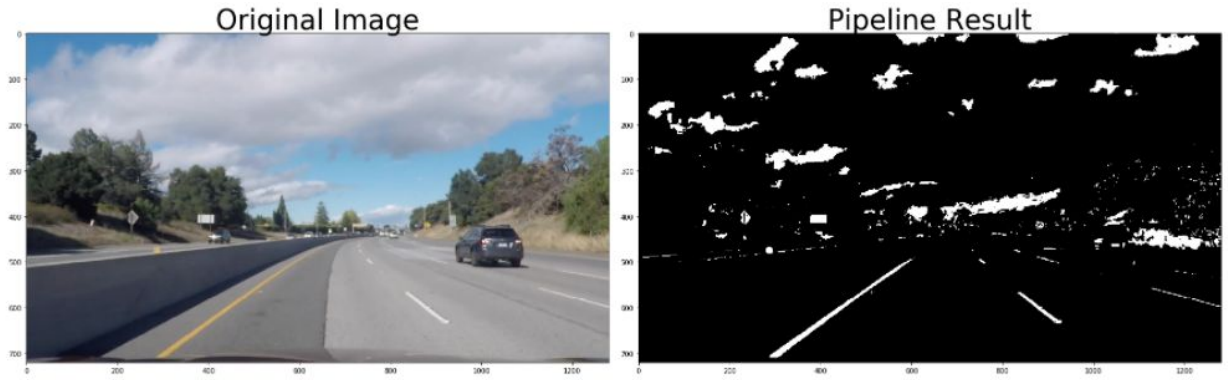
Here are the examples of undistorted images.



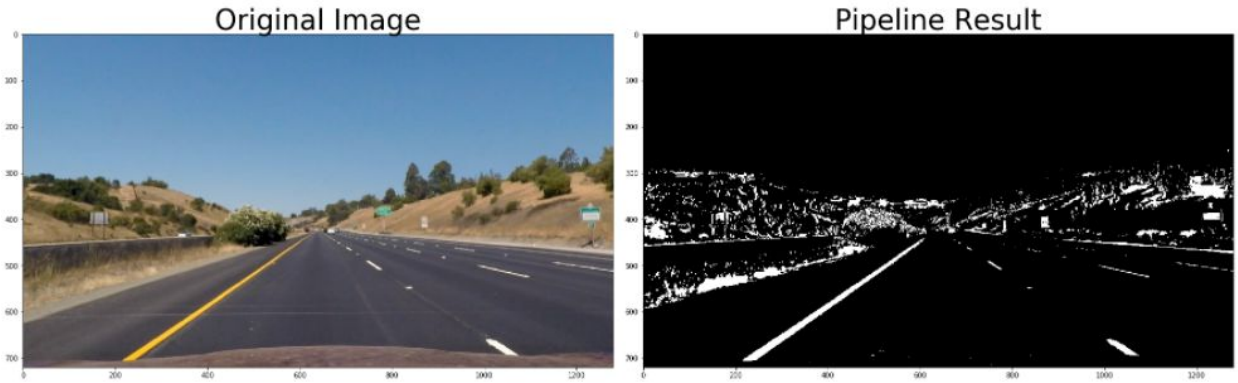
Thresholded binary image

I used several approaches for this. OpenCV can convert images to different color spaces, the most useful are HSV, LAB, and HLS, as described in this article. These spaces can help to find white and yellow pixels. I also added Sobel gradient of gray color space and combined all them in a single score_pixel array. If such array has more than 2 scores I recognize such pixel as a good one. Default Udacity method which combines only S channel of HSV space and gradient gave the bad result on challenge video, but such function gives more robust line. You can find implementation in score_pixels method.

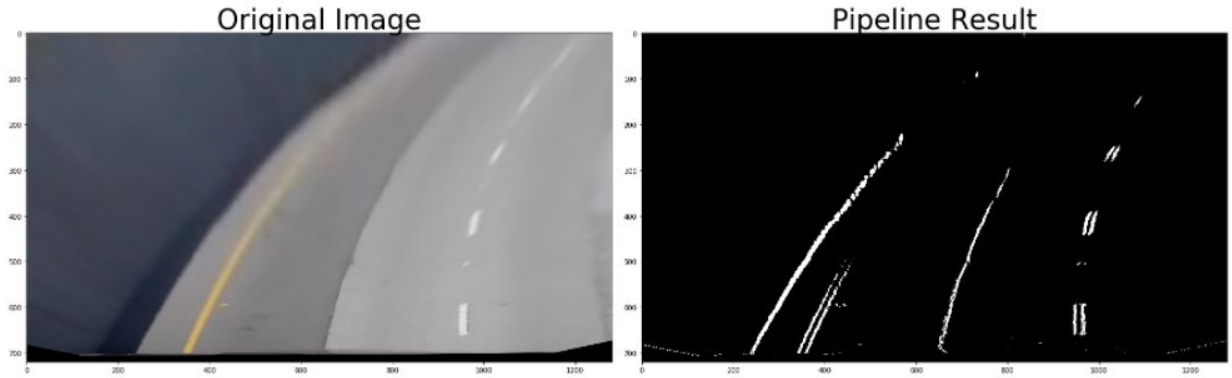
Here are some examples.



Challenge image



Straight line image



Default edge pipeline result

The first image has shadow and two different road covers. Combining several color spaces helped to found more robust lane lines pixels.

Table below contains settings for each color space.

Space	Selected channel	Threshold	Clip Limit for CLAHE method
LAB	B	150	2.0
HSV	V	190	6.0
HLS	L	200	2.0

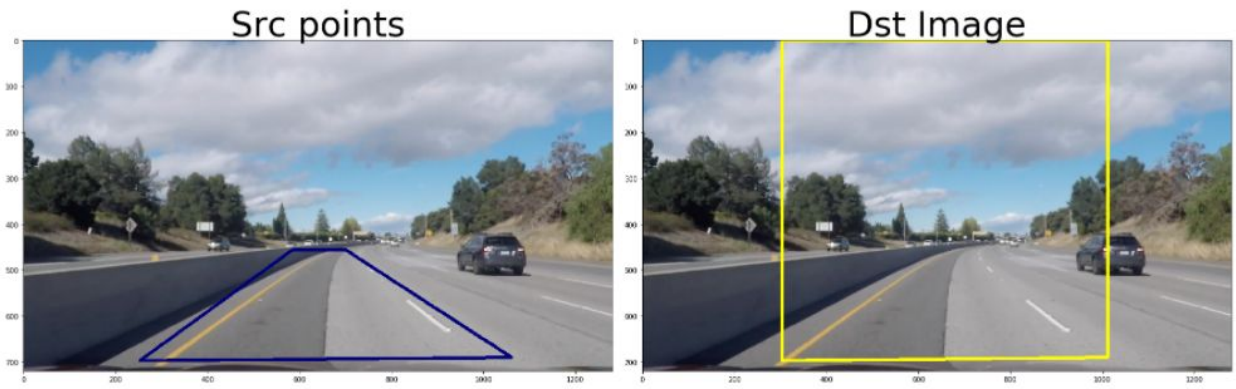
Perspective transform

This transform is really straightforward we need to solve matrix equation with four points, thankfully OpenCV has methods for this purpose.

```
def perspective_tr(img, src, dst, img_size):
        t_m = cv2.getPerspectiveTransform(src, dst)
        tr_img = cv2.warpPerspective(img, t_m, img_size, flags=cv2.INTER_LINEAR)
        inv_m = cv2.getPerspectiveTransform(dst, src)
        return tr_img, t_m, inv_m
```

Where src - is source corners and dst - destination corners, tr_img - result image output, t_m - direct transform matrix, inv_m - reverse one.

Results are below



Detect lane pixels

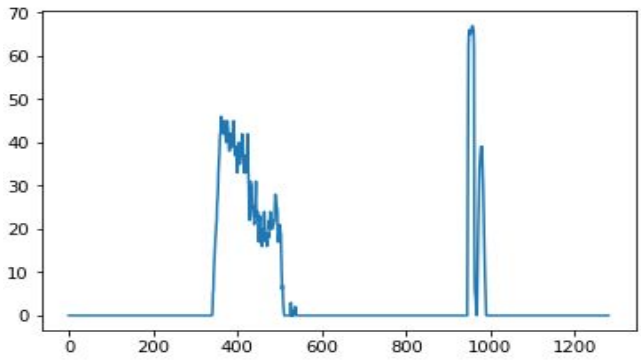
Line Finding Method: Peaks in a Histogram

After applying calibration, thresholding, and a perspective transform to a road image, you should have a binary image where the lane lines stand out clearly. However, you still need to decide explicitly which pixels are part of the lines and which belong to the left line and which belong to the right line.

I first take a histogram along all the columns in the lower half of the image like this:

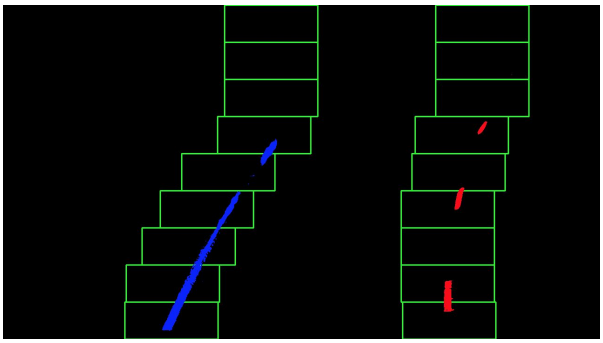
```
import numpy as np
histogram = np.sum(img[img.shape[0]//2:,:], axis=0)
plt.plot(histogram)
```

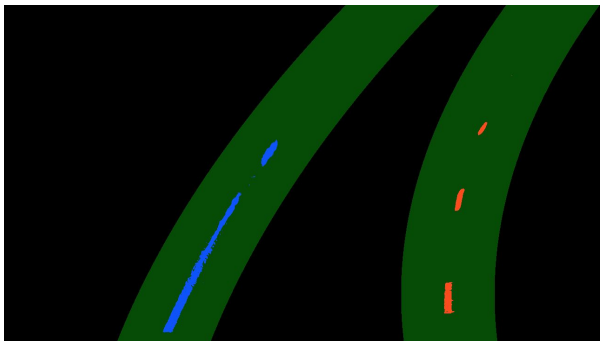
The results look like this:



With this histogram I am adding up the pixel values along each column in the image. In my thresholded binary image, pixels are either 0 or 1, so the two most prominent peaks in this histogram will be good indicators of the x-position of the base of the lane lines. I can use that as a starting point for where to search for the lines. From that point, I can use a sliding window, placed around the line centers, to find and follow the lines up to the top of the frame.

If a polynomial ROI is given, we could search within the region





Finding curvature

I will use second order polynomial to approximate lane line.

$$f(y) = Ay^2 + By + C$$

Geometry

The radius of curvature (awesome tutorial here) at any point x of the function $x = f(y)$ is given as follows:

$$R = \frac{(1 + \frac{dx}{dy}^2)^{3/2}}{\frac{d^2x}{dy^2}}$$

Since we assume that the camera is mounted at the center of the car, the offset can be calculated as follows:

lane_center = (left_fitx[-1] + right_fitx[-1])/2

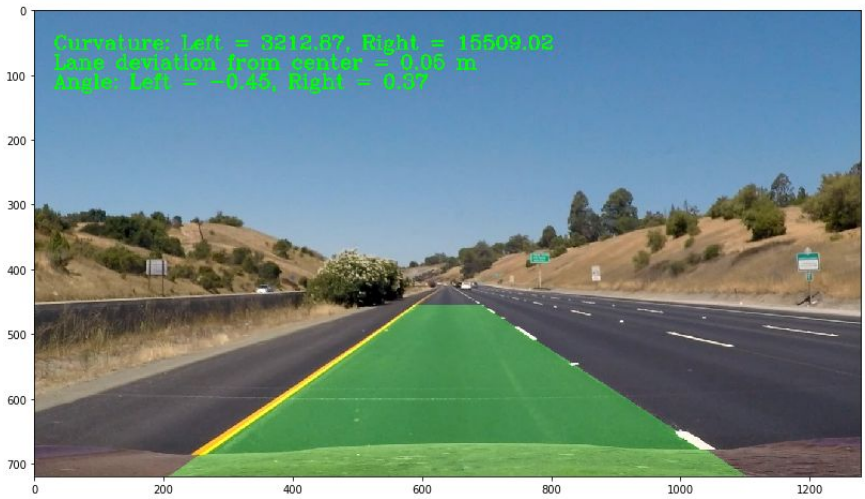
offset = np.abs(lane_center – image.shape[1]/2)

Finally, we need to convert the curvature and offset into the unit of meter. According to Udacity, we can assume that meter per pixel in y direction is 20/720 and in x direction is 3.7/900.

We can also find angles of tangent lines (atan(df(y0) / dy)):

top_angle = m.atan(2 * fit_cr[0] * scene_height + fit_cr[1]) * 180 / m.pi

bottom_angle = m.atan(left_fit_cr[1]) * 180 / m.pi



Line Object

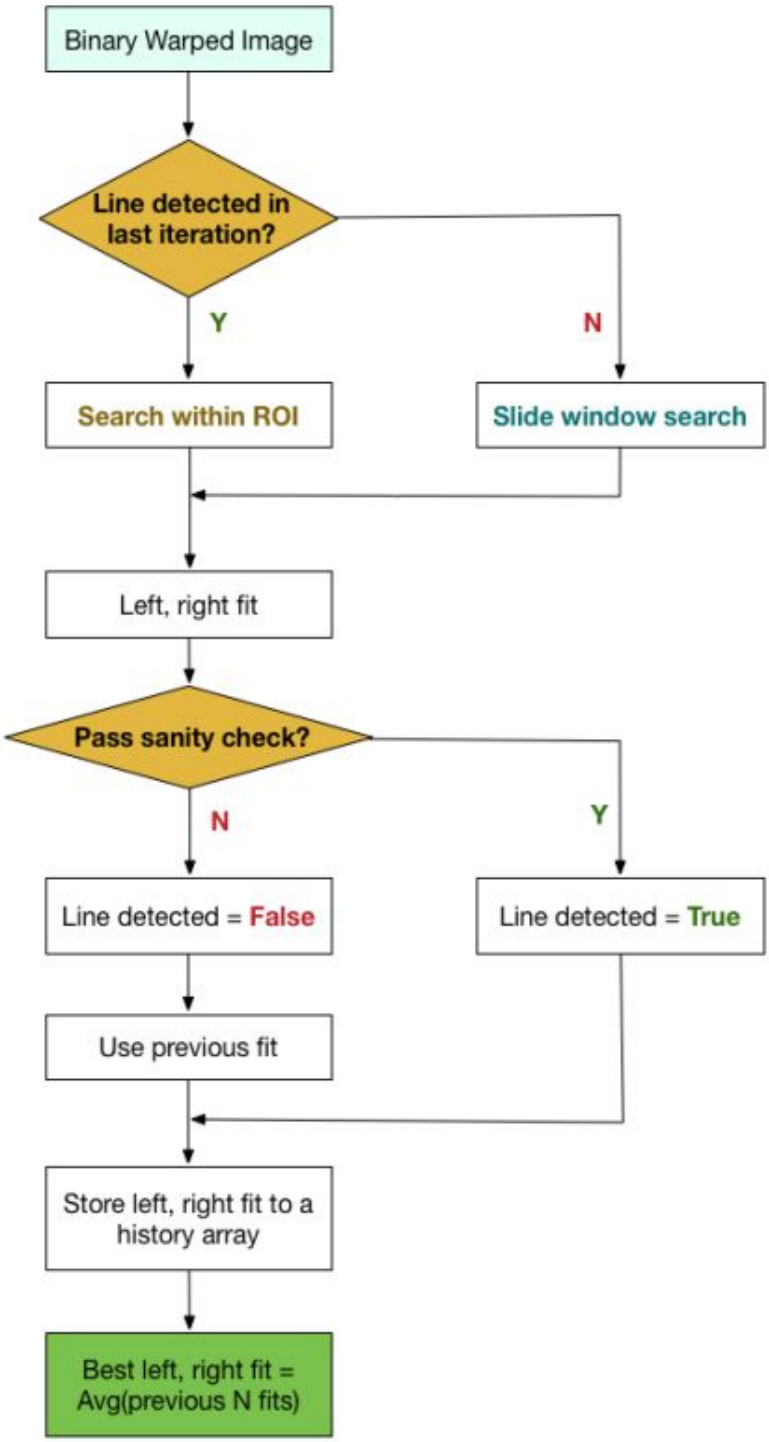
I found this guide very useful, it summarizes all tips from Udacity on a single page.

In order to produce a more stable output, I create an object called Line. Its responsibilities include:

1. Keep track of lane polynomial functions for previous N frame
2. Intellectually choose sliding window algorithm or ROI search algorithm to locate lane lines
3. Perform sanity check on lane lines
4. Take average on current N frame to produce a more stable output
5. Return curvature,offset and angles of the lane

The core algorithm of how a line object detect the polynomial functions for a line is as follows:

My sanity check contains checking that tangent lines of curves have roughly same angle on the top and bottom.


Binary Warped Image
Line detected in last iteration?
Y
N
Search within ROI
Slide window search
Left, right fit
Pass sanity check?
N
Y
Line detected = False
Line detected = True
Use previous fit
Store left, right fit to a history array
Best left, right fit = Avg(previous N fits)

My sanity check contains checking that tangent lines of curves have a roughly same angle on the top and bottom.

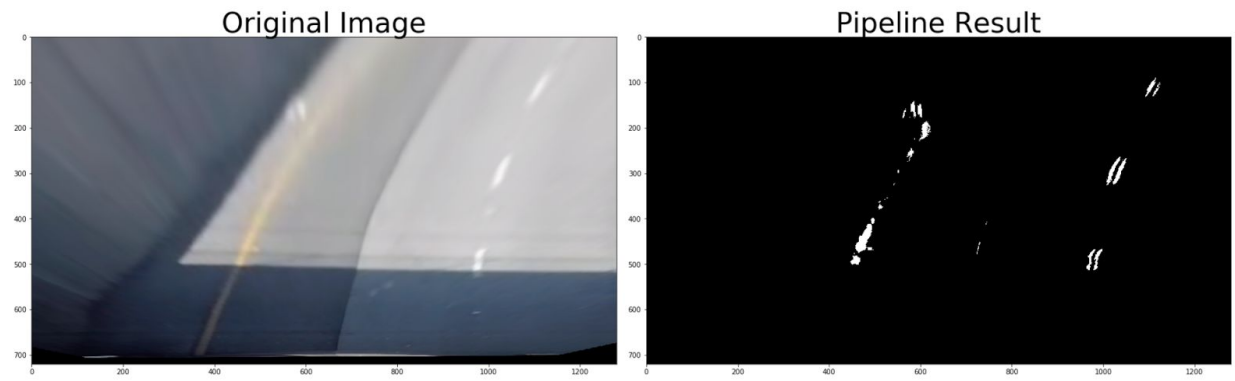
Results

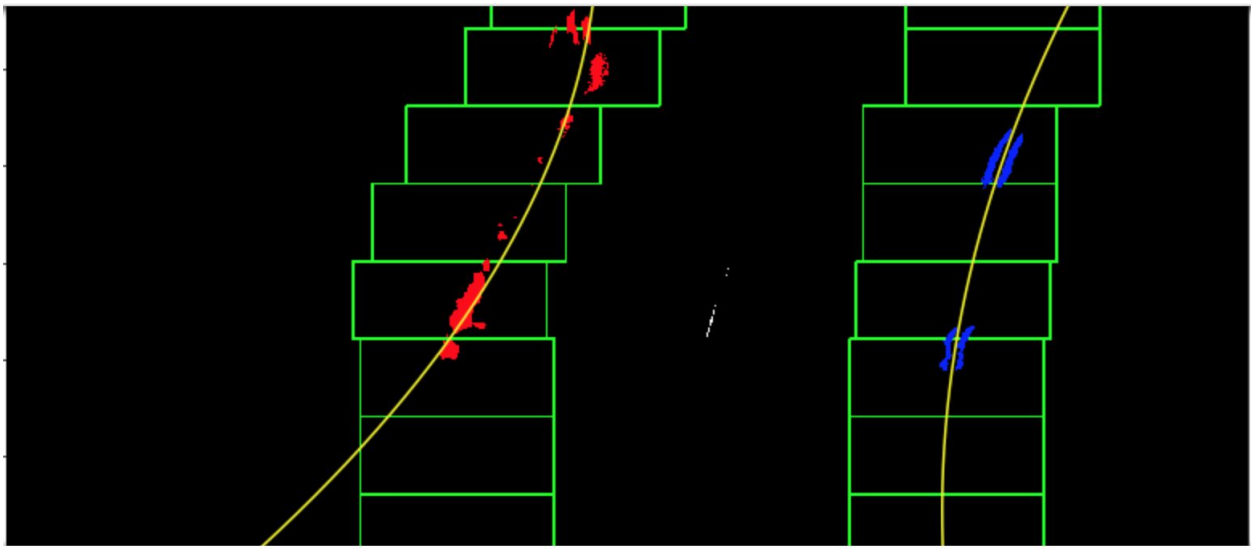
Folder output_images contains results of project_video.mp4 and chalenge_video.mp4. The second video is more harder, lines are shifting under the bridge for some seconds and have more curvature in the top of the image in the end of the video, but project_video.mp4 results are smooth and robust.

But I did not manage to solve the harder challenge video.

Hard coded corners are a problem, I need to change frames for the every different videos, this peace of pipeline should be automated also.

Detecting lane line pixels is a real challenge. I changed some thresholds for channels and got better results for first two videos, but it did not work for harder one. And I am still facing problems with shadows.





If there are small amount of pixels (some could be covered by shadow), the line model will predict sharp turn, and lines will be unstable.

Pipeline performs badly on sharp turns and sloppy terrain and it is very difficult to find threshold parameters for lane line pixels that will work on every video. Light and weather conditions could vary a lot, so I assume that it is impossible to find a single amount of parameters for every situation.

Files with code:

Notebook.py contains visualization.

Methods.py contains methods which were used for single image pipeline, they were copied from Udacity.

Line.py contains Line.class which has methods for video pipeline, I refactored some Udacity implementations and added moving average over last n image with sanity check, block-schema of algorithm is above.