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.

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Camera Calibration

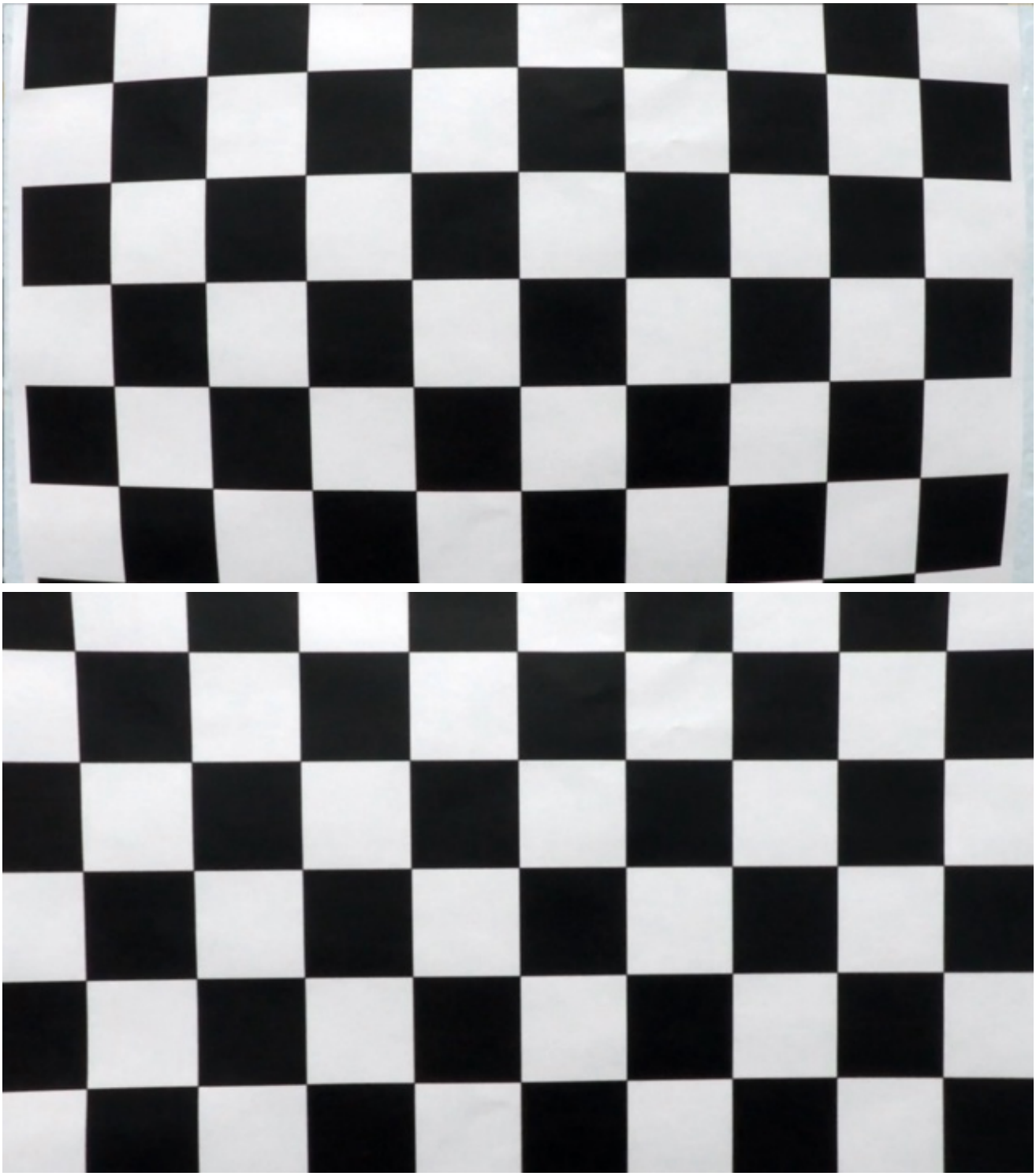
1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

Code for camera calibration is contained in `script/calibrate.camera.py` in function `compute_camera_calibration()` starting on line `19`. First a 3D grid of (x, y, z) size of (9, 6, 1) is prepared to represent true coordinates of chessboard square corners, called object points below. Z-value is set to 0 for all entries, since all points lie on the same z-plane.

Then `cv2.findChessboardCorners()` is used to find chessboard corners in calibration images.

Calibration is performed with `cv2.calibrateCamera()` based on object space and image space points pairs and resulting camera matrix and distortion coefficients are stored for later use.

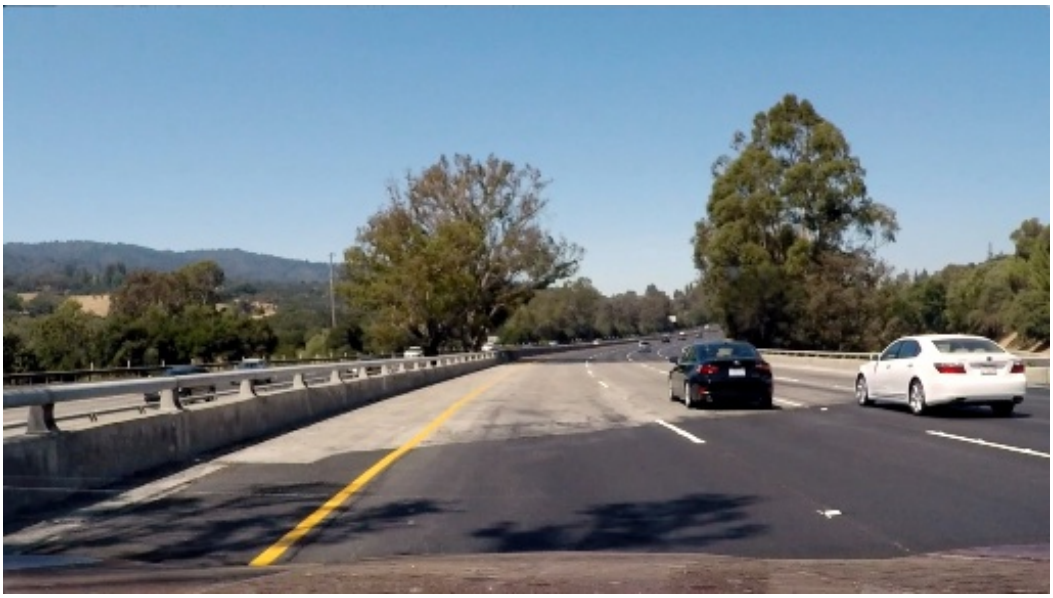
`script/calibrate.camera.py` also contains `undistort_sample_image()` function on line 71 that undistorts an input image based on previously computed camera parameters. A sample input and undistorted output are presented below:



Pipeline (single images)

1. Provide an example of a distortion-corrected image.

As a first step of the pipeline I compute undistorted image based on previously established camera calibration. Below is an example of original image and its undistorted counterpart.



As can be seen undistortion doesn't affect image too much. This is as expected, since distortion effects are only significant around image edges (more specifically, lense edges), and pronounced only when object is close to the lens, but our area of interest is right in front of the camera and spans a significant distance. Pay attention to car hood though, you can see subtle changes to its shape between original and undistorted image.

Undistorted image was computed with `car.processing.ImageProcessor::get_undistorted_image()`, which makes a simple call to `cv2.undistort()`.

2. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

As a second step of my preprocessing pipeline I warped image so as to give a bird-eye view of area likely occupied by the lane. This has several advantages:
- making lane lines appear more straight even around turns
- making lane lines width more uniform across whole image
- making it easier to mask out areas outside of the lane

Below is a sample image with source points used for warpind marked, as well as the same image after warping.

Source and destination points for warping are specified in `car.processing.get_perspective_transformation_source_coordinates()` and `car.processing.get_perspective_transformation_destination_coordinates()` on lines 267 and 279, respecively. Source and destination points are chosen so as to map likely lane area to a rectangle spanning middle band of destination image.

Warped image is computed with `car.processing.ImageProcessor::get_warped_image()` on line 111, which makes a simple call to `cv2.warpPerspective()`.

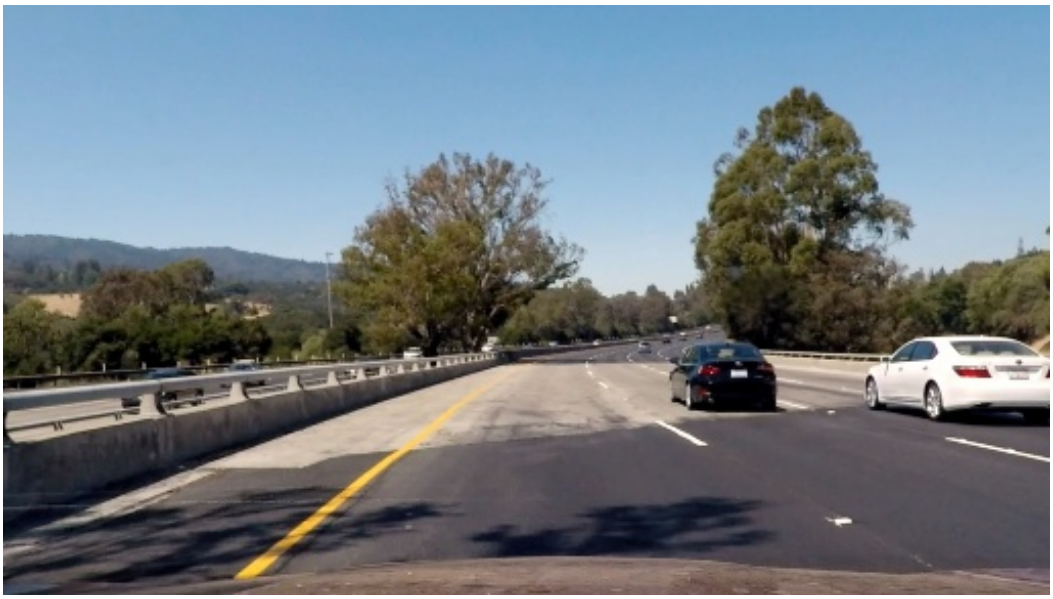
3. Shadow removal

Recongnizing that shadow areas affected my binary mask images (shown later), I added a simple shadow removal method based on a paper *"A robust approach for road detection with shadow removal technique"* by Salim, Cheng and Degui. For implementation details please refer to the paper, but a quick intuition is that shadows are areas that have lively colors (thus high saturation in HSV space), but appear dark (thus have low

value in HSV space). Once shadow areas are identified, shadows can be removed (or at least their impact attenuated) by making their image statistics closer to these of surrounding non-shadow areas.

In practice above method works well on small to medium size shadow patches and proves useful in removing some of shadows that happen to fall within car lanes.

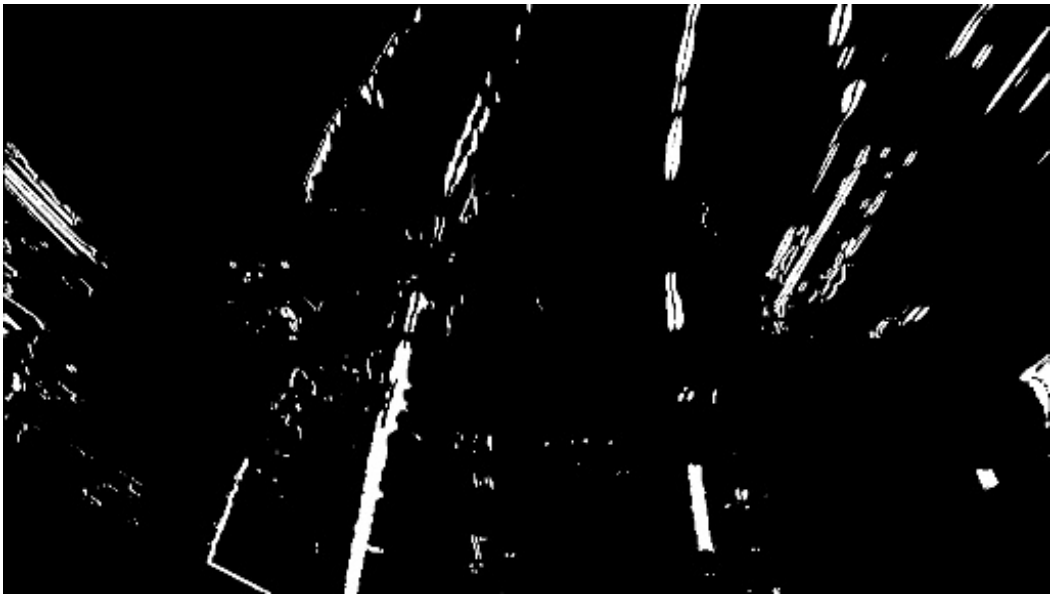
While I perform shadow removal on warped images (`car.processing.ImageProcessor.get_preprocessed_image()`, line 125, below I present a sample unwarped image, first in original form then with shadow removed, followed by a saturation-based mask for both cases.



As can be seen most of shadow right in front of car hood was removed, which simplifies further detection stages. A careful reader would note that black car in right lane also got removed from saturation mask - this isn't a problem in our scenario, but it highlights that above shadow removal method shoud be used with care.

4. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

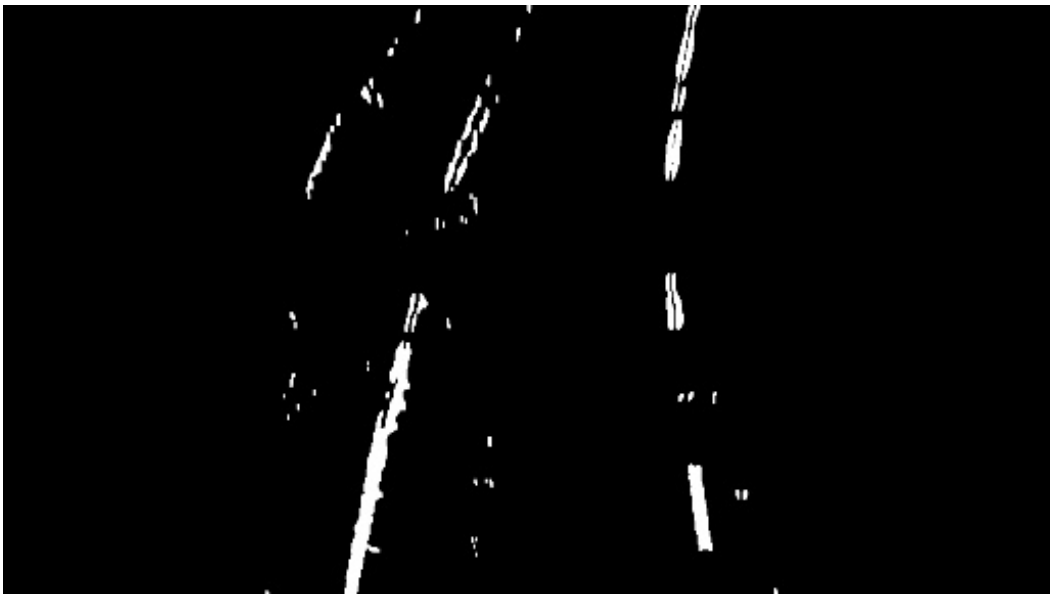
Once shadows are removed from my warped image, I compute a simple binary mask as an OR operation of saturation and x_gradient based mask. This computation is done in `car.processing.ImageProcessor.get_preprocessed_image()` on lines 127~130. Below is a sample input and output (also please note positive effect of shadow removal):



5. Cropping area of interest, erosion and dilation

As a final stage of my preprocessing pipeline, I crop out areas unlikely to be inside the lane, as well as perform an erosion-dilation operation with a vertical kernel, so that small horizontal specs unlikely to represent lane lines get removed. This computation is done in `car.processing.ImageProcessor.get_preprocessed_image()` on lines 135~137.

Image below is the mask presented in previous section after cropping and erosion-dilation process applied.



It should be noted that the cropping stage is the most restrictive assumption in the whole processing pipeline. Applied cropping mask works very well at removing unwanted pixels in `project_video.mp4`, but has a potential to mask out lane lines on roadswith sharper turns than in above footage, thus degrading overall performance.

6. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

Class `car.processing.LaneLineFinder`, found on line 291, is responsible for finding lane lines. It exposes two methods, `get_lane_equation_without_prior_knowledge()` and `get_lane_equation_using_prior_knowledge(recent_lane_equations)`.

When searching for lane line from a scratch, `LaneFinder` first computes a histogram of number of non-zero pixels in each column and identifies column with largest response as likely to contain lane line. A similar search over a small band of x-values within that column reveals a starting point for extensive lane line search. This computation is performed in `car.processing.LaneLineFinder.get_lane_starting_coordinates()` on line 304.

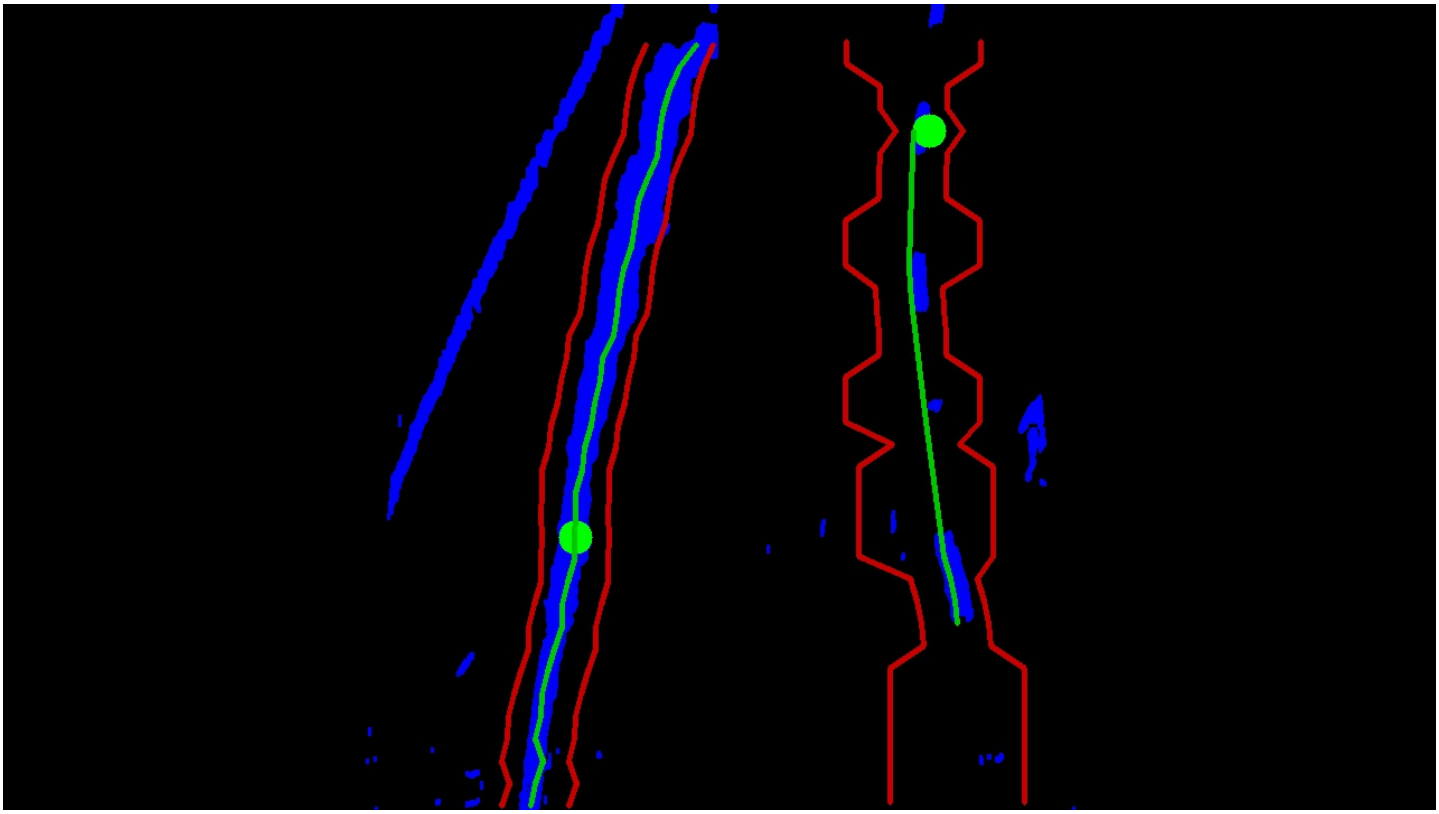
Once starting point is identified, search is done in two directions - up and down. In each case a correlation with a small kernel made of 1s is made to identify coordinate with largerst response - deemed to be the most likely lane center candidate. At each step once candidate coordinate is identified, algorithm moves (up or down depending on direction) and performs another search within a band centered on y-coordinate of last identified lane candidate, since lines shoud be continuous.

Line center candidate is identified only if kernel response was high enough - this prevents adding candidate points in "blank" areas between dashed line segments.

While search area is kept relatively narrow (about double to triple of expected lane line width), if a very low response was obtained at a given search step, search area width is doubled until a high response is obtained again. This helps to tackle lines with large curvature.

Above algorithm is implemented in `car.processing.LaneLineFinder.scan_image_for_line_candidates_without_prior_knowledge()` on line 325. `scan_image_for_line_candidates_using_prior_knowledge()` is very similar, except it uses last known lane equation to constrain search band.

Image below presents search result for two lanes lines. Green circles represent identified starting points. With right line it can be seen that search area is widened where no good response to lane kernel was found and that found line spans only areas with high response. Please note that while in my code I perform search for left and right lane separately, image below shows both search results concatenated in a single image.



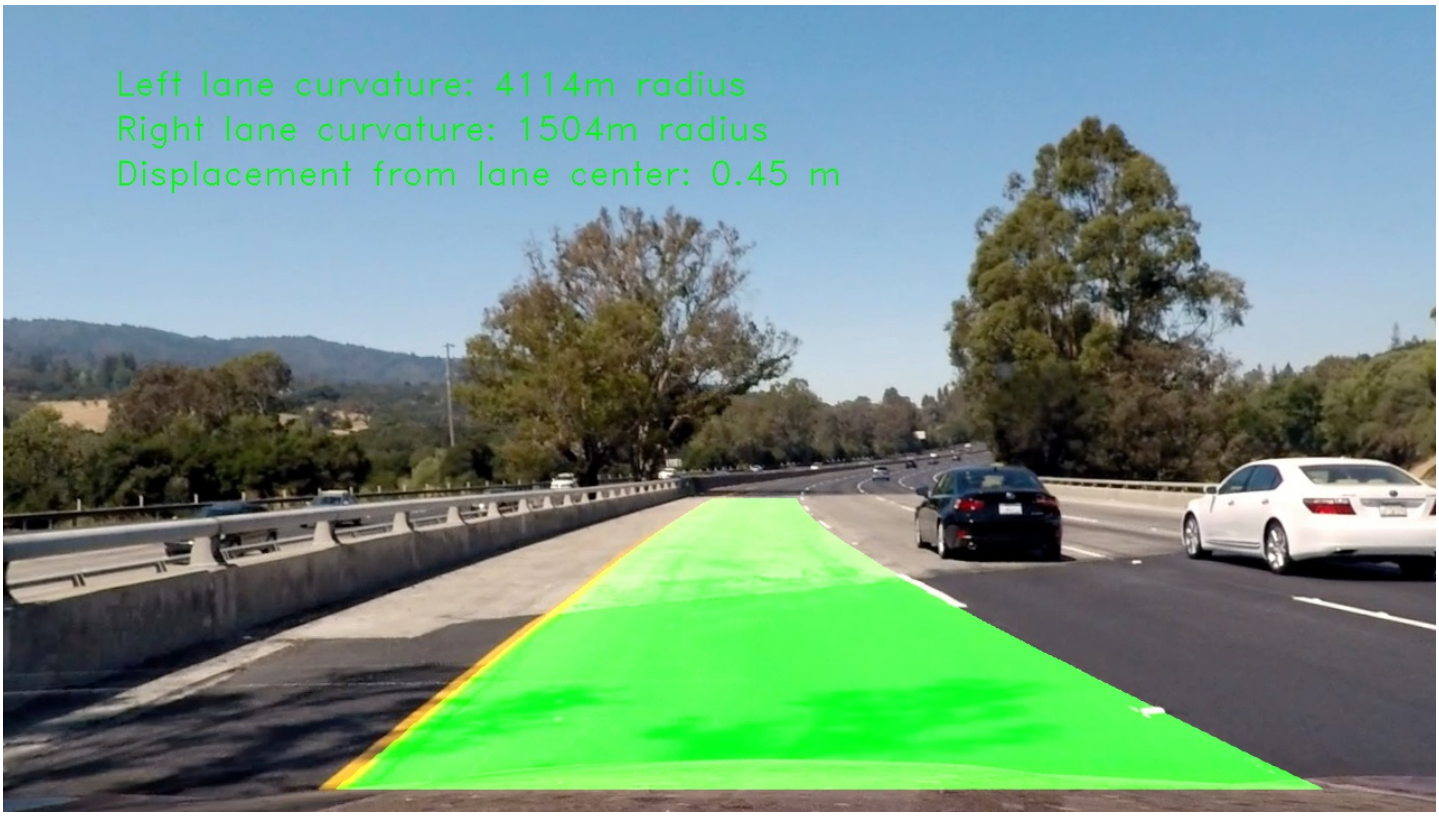
Once line center points are identified, a simple call to `np.polyfit()` identifies lane lines equations. `get_lane_equation_using_prior_knowledge(recent_lane_equations)` returns an average of a line equations found for a few last frames, which yields more stable results than estimates based on a single frame only.

7. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

Radius of curvature and position of vehicle with respect to center are calculated by class `car.processing.LaneStatisticsComputer` in methods `get_line_curvature()` and `get_lane_displacement()` respectively. `get_line_curvature()` uses equation for lane curvature provided in Udacity's notes, `get_lane_displacement()` simple computes x positions of both lane lines at lower image border (thus where the car is), then takes average to compute x-coordinate of lane center and compares it to half the image width. The difference is the converted to meters.

8. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.

Finding lanes in images is implemented in `script/find_lanes_lines.py` in function `find_lane_lines_in_test_images()` on line 19, which also shows some of the processing steps. Below is a single example of identified lane:



Pipeline (video)

1. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (wobbly lines are ok but no catastrophic failures that would cause the car to drive off the road!).

Here's a link to my video result. It was created with `find_lane_lines_in_videos_smooth()` function of `script/find_lanes_lines.py`, and most of the heavy lifting is performed by `car.processing.SmoothVideoProcessor` class.

Discussion

1. Briefly discuss any problems / issues you faced in your implementation of this project. Where will your pipeline likely fail? What could you do to make it more robust?

By far my largest problem was forgetting that while OpenCV reads images in BGR order, moviepy does so in

RGB order. This led to some head scratching why my detection works well in images, but hardly so in video. I'm probably not the only one who got bitten by this difference between OpenCV and moviepy.

The following assumptions in my pipeline are likely to break in more generic settings:
- aggressive cropping mask - it helps to remove a lot of pixels that can't represent lanes in project video, but would also remove road segments on sharp turns
- using a single kernel for lines detection - my kernel only looks for straing lines. A more flexible approach would use a family of kernels for differt line angles. This would help pick up lines better, as well as steer search region for next step in correct direction
- not examining nature of identified lines - in challenge video a vertical line formed by a connection between two different road surfaces gets picked up by x-gradient mask. While it's difficult to make robust assumptions in colorspaces, examining colors of positive pixels returned by x-gradient mask might help to remove some wrong candidates, since lane lines should be white or yellow, not e.g. different shades of gray
- searching for each lane line in only one half of the image - for simplicity my pipeline only searches for left lane line in left half of image and right lane line in right half of image. This could break on sharp turns and a more robust approach would allow lane lines to cross over to other half ot the image