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.

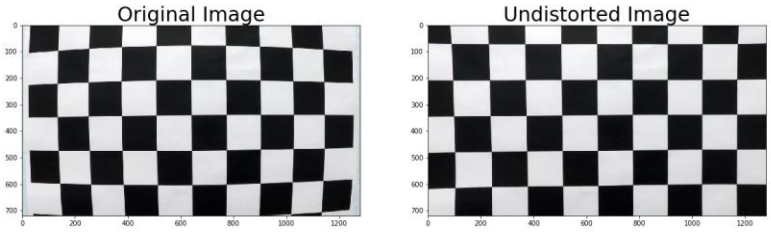
Camera Calibration

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

The code for this step is contained in the first 2 code cells of the IPython notebook located in "./project.ipynb" (or in lines 11 through 69 of the file called `camera_cal.py`).

I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. Here I am assuming the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. Thus, `objp` is just a replicated array of coordinates, and `objpoints` will be appended with a copy of it every time I successfully detect all chessboard corners in a test image. `imgpoints` will be appended with the (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection.

I then used the output `objpoints` and `imgpoints` to compute the camera calibration and distortion coefficients using the `cv2.calibrateCamera()` function. I applied this distortion correction to the test image using the `cv2.undistort()` function and obtained this result:



Pipeline (single images)

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

To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like this one:

Following image shows the distortion-corrected image.

I have saved all distortion-corrected images as `output_images/*_undist.jpg`

2. 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.
To detect white lines, I used a combination of color thresholding from L channel of LAB color space and magnitude and direction gradient thresholds from L channel of LAB color space. For the yellow lines, I used color thresholding from B channel of LAB color space. Then I combined these two binary image results to generate a binary image containing both white and yellow lines. (thresholding steps at lines 49 through 82 in `pipeline.py` or, in the 5th and 6th code cells of the IPython notebook). Here's an example of my output for this step.

I have saved all binary images as `output_images/binary_*.jpg`

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

The code for my perspective transform includes a function called `warper()`, which appears in lines 12 through 18 in the file `image_gen.py` (./image_gen.py) (or, for example, in the 8th code cell of the IPython notebook). The `warper()` function takes as inputs an image (`img`), as well as source (`src`) and destination (`dst`) points. I chose to hardcode the source and destination points in the following manner which I manually picked from a test image with straight road lines:

```
src = np.float32([[252, 698], [573, 465], [712, 465], [1064, 698]])dst =
np.float32([[412, 720], [412, 0], [888, 0], [888, 720]])
```

This resulted in the following source and destination points:

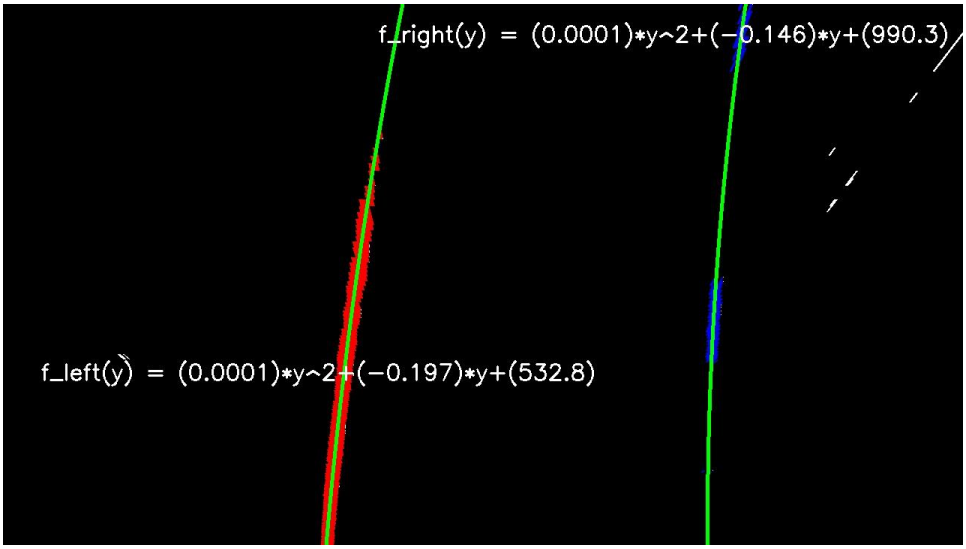
Source	Destination
252, 698	412, 720
573, 465	412, 0
712, 465	888, 0
1064, 698	888, 720

I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.

I have saved all warped binary images as `output_images/warped_*.jpg`

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

Then I used convolutions in conjunction with sliding window search to find the hot pixels belongs to the lane lines and fit my lane lines with a 2nd order polynomial to those identified pixels kind of like this:



Code to identify lane line pixels using convolution in conjunction with sliding window search is contained in `find_lane_pixels(self, warped)` method in my LineTracker class. (10th code cell in my IPython notebook or for example line 116 through 186 in the file `tracker.py`)

I then used `np.polyfit()` function to fit a 2nd order polynomial function to fit their positions. (I did this in `fit_line(self, smooth_factor, margin)` function in Line class contained in the same code cell. or for example line 35 through 44 in the file `tracker.py`)

In those procedures some statistics are calculated which help determine if the found line is good or not. The statistics include:

- Number of points in the fit
- Vertical distance between uppermost and lowermost lane line pixel detected
- Horizontal difference of lowest and uppermost point of polynomial fit from that of previous fit
- Deviation of width of detected lane at nearest, halfway and farthermost positions from standard width of a lane (3.7m)

These checks are included in `filter_fitx(self, fitx, margin)` method in Line class and `lane_sanity_checks(self)` method in LineTracker class contained in 10th code cell of my IPython notebook or line 54 through 83 and 231 through 243 respectively in the file `tracker.py`

Note: When processing video files, I have skipped the sliding window search on subsequent frames after initial frame and just searched in a margin around the averaged polynomial fit of the line so far as indicated in same code cell of IPython notebook or for example line 187 through 209 in the file `tracker.py`

I have saved all images of line fits as `output_images/*_fit_lines.jpg`

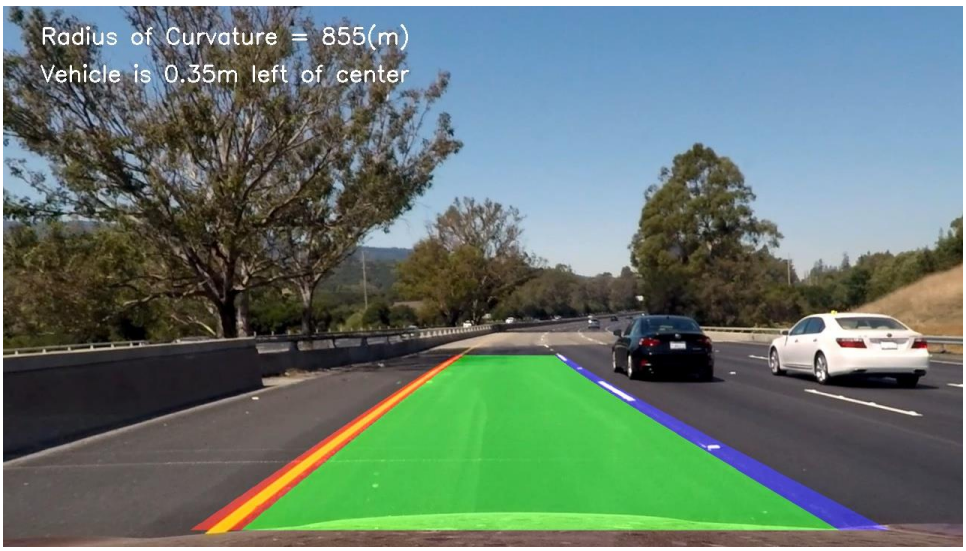
5. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

I did this in lines 50 through 57 in my code in `image_gen.py` (or, in the 15th code cell of the IPython notebook) For this, first I averaged the x values from both polynomial fit lines (thus taking the approximated middle line) and then converted the pixel positions to x, y positions in real world using `xm_per_pix` and `ym_per_pix` values. (I took these values assuming lane width is 3.7m and length of a dashed line and separation is 5m and 10m respectively, which I observed from google maps satellite images of the same road.) Then I again fit a polynomial function in this domain to get the radius of curvature of the road in meters.

Similarly, I calculated the position of the vehicle with respect to center by taking the average of offset of both lines respect to center of the frame.

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

First, I generated top down view of the result plotted on warped road image in lines 246 through 276 in `tracker.py` file (or in the 10th code cell of the IPython notebook). Then I implemented this step in lines 21 through 27 in my code in `image_gen.py` in the function `map_lane()` (or, for example, in the 14th code cell of the IPython notebook) to map the result back to undistorted image of the road. Here is an example of my result on a test image:



I have saved all images of lanes mapped as `output_images/*_output.jpg`

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

Pipeline was tuned and run on challenge video as well. It works great except losing lane momentarily under the bridge due to change in lighting. (Note that tuned parameters for challenge video also works for project video as well though distance of mapped lane segment is reduced to 20m from 25m)

Here's a link to my challenge video result

However harder challenge video required extensive tuning and reducing mapped lane segment length to 10m since it contains sharp turns. It works ok but loses the lane a few times due to very sharp turns and frequent changes in lighting.

Here's a link to my harder challenge video result

Furthermore, I have implemented this on my own video I've recorded myself. For this step, I also captured my own calibration images (These are saved under `./pg_camera_cal` folder) and calculated calibration and distortion coefficients. I also derived a separate set of source and destination points for perspective transformation of my own video.

Here's a link to my own video result

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?
Here I have used a combination of color thresholding from L and B channels and magnitude and direction thresholds from gradient of L channel of LAB color space. I have hand tuned those thresholds along with other parameters like source and destination points of perspective transform as well as smoothing parameters from LaneTracker class.

It is observed that while selected thresholds works fairly robust to detect lane line pixels, sudden changes of lighting make it hard to consistently detect lane line pixels. On the other hand, reducing the length of warped lane by tuning source and destination points helps in situations where sharp turns, but it requires solid lane lines on either side. (So, this won't work for either project video or challenge video where dashed lines have as much as 10m gap between them.)

To overcome above limitations, we could try to dynamically set the thresholds for pipeline based on the lighting condition and curvature of the lane. We could also examine more advanced filtering and brightness equalization techniques to make the pipeline more robust under sudden changes of light conditions.