Writeup Template

You can use this file as a template for your writeup if you want to submit it as a markdown file, but feel free to use some other method and submit a pdf if you prefer.

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.

Writeup / README

1. Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf. Here is a template writeup for this project you can use as a guide and a starting point.

You're reading it!

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 and second code cells of the IPython notebook located in "./Advanced-lane-lines.ipynb"

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 (I used one of the calibration images as test images, as there was not a explicit test image in camera_cal dir) using the `cv2.undistort()` function and obtained this result:

output_images/undistorted_calibration.jpg

Image in: output_images/undistorted_calibration.jpg

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: output_images/undistorted_test3.jpg

Image in: output_images/undistorted_test3.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.

The code for my perspective transform includes a function called `pipeline()`, which appears in the 3rd code cell of the IPython notebook. I have used a combination of color and gradient thresholds to generate a binary image. I have thresholded on RGB, HLS, HSV, LUV and LAB channels, and selected the channels that contribute better to yellow and white.

output_images/test3-gradient-threshold.jpg

Image in: output_images/test3-gradient-threshold.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 the 4th 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 the hardcode the source and destination points in the following manner:

```
src = np.float32(
 [[(img_size[0] / 2) - 55, img_size[1] / 2 + 100],
 [((img_size[0] / 6) - 10), img_size[1]],
 [(img_size[0] * 5 / 6) + 60, img_size[1]],
 [(img_size[0] / 2 + 55), img_size[1] / 2 + 100]])
```

```
    dst = np.float32(
     [[(img_size[0] / 4), 0],
     [(img_size[0] / 4), img_size[1]],
     [(img_size[0] * 3 / 4), img_size[1]],
     [(img_size[0] * 3 / 4), 0]])
```

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.

output_images/test3-perspective-transform.jpg

Image in: output_images/test3-perspective-transform.jpg

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

The code for fitting polynomials includes a function called `fit_polynomials()`, which appears in the 5th code cell of the IPython notebook. The function takes a binary warped image as input, and uses the sliding window search approach in order to fit the polynomials to the left and right lines. Image below shows how polynomials fit reasonably well to one of the test images

output_images/test3-polynomial-fit.jpg

Image in: output_images/test3-polynomial-fit.jpg

I also created a `fit_polynomials_lite()` function for the case when we already have a good fit and can be reused, and thus there is no need to run the sliding window search. The function is in the iPython notebook, but I did not end up using it because always using sliding window search seemed fast enough.

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 the 8th code cell of the iPython notebook, along with an offset function in the 9th code cell.

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

The code for fitting polynomials includes a function called `display_lane()`, which appears in the 11th code cell of the IPython notebook. Here is an example of my result on a test image:

output_images/test1-lane-overlap.jpg

Image in: output_images/test1-lane-overlap.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!).

I created a `process_frame` function defined in the 11th code cell of the iPython notebook to compute all the processing done every frame in the video. I tested the function with the test images (12th code cell), and I then applied it to the project video (13th cell). Video result is displayed in code cell 14th.

Here's a link to my 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?

- The images needed to be in binary format for the pipeline to work. It took me a while to figure this out, given that the example code in the course was using 3D format for color thresholds. When I used 1D format for color thresholds, then I was able to make it work.
- I have thresholded on RGB, HLS, HSV, LUV and LAB channels, and selected the channels that contribute better to yellow and white. I played a bit with different thresholds parameters, until I had a combination of params that generated good results in several test images, specially in the ones where the road had a lot of sunlight and shadows for that sunlight(test5). This is the combination that did the work for me:

```
    color_binary[#(s_binary == 1) # best
                 ((r_binary == 1) & (h_binary == 1)) # lines
                 | (l_binary == 1) # white
                 | ((v_binary == 1) & (b_binary == 1)) # yellow
                ] = 1
```

The key was to ignore the S channel in HLS space, even though the course material mentioned it was the best option, and instead focus on channels that contributed to lines, white and yellow explicitly.

- I have created a Line class, to keep track of polynomial coefficients, curvature and offset, from detected-correctly lines, to be used with line detection failed. I have also created a `lane_detected` function to take into account distance between lines and similar curvatures to infer if lines are well detected or not. I have also modified the `process_frame` function to incorporate this functionality. I have followed a hybrid approach, where if there is a single frame missed, I reuse the lines from the previous frame, whereas if there are more than one consecutive frames missed, I reuse the average of the lines across past good frames.

The combination of this hybrid approach and the changes in the thresholding is what has allowed me to detect the line in shadowy situations while maintaining the detection in both sunny and regular situations.