# Writeup Template

## 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 code in lines 167 of the file called `pipeline.py`, and that Image Processing Utils are from lines 24 through 29 fo the file called `ImageProcessUtils.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:
alt text

## 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: alt text

### 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.

I used a combination of color and gradient thresholds to generate a binary image (thresholding steps at lines 170 in `pipeline.py`, and that is defined in lines 44 to 50). Here's an example of my output for this step. (note: this is not actually from one of the test images)
alt text

### 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 `warp()`, which appears in lines 144 through 147 in the file `pipeline.py`. The `warp()` 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([[770, 480], [510, 480], [0, 720], [1280, 720]])
dst = np.float32([[self.img_size[0], 0], [0, 0], [0, self.img_size[1]], [self.img_size
```

This resulted in the following source and destination points:

| Source | Destination |
|---|---|
| 770, 480 | 1280, 0 |
| 510, 480 | 0, 0 |

| | |
|---|---|
| 0, 720 | 0, 720 |
| 1280, 720 | 1280, 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.
alt text

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

Then I did some other stuff and fit my lane lines with a 2nd order polynomial kinda like this:
alt text
And in this picture the coefficients are below:

| **Coefficients** | **Left** | **Right** |
|---|---:|---:|
| Second order | 8.21 * 10^-5 | 2.05 * 10^-4 |
| First order | -8.94 * 10^-2 | -2.19 * 10^-1 |
| Zero order | 2.72 * 10^2 | 1.22 * 10^3 |

### 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 213 through 219 in my code in `pipeline.py`, and that is defined in lines 96 through 111 in my code in `Line.py`

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

I implemented this step in lines 195 through 210 in my code in `pipeline.py` in the function `pipeline()`. Here is an example of my result on a test image:
alt text

---

## 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

## 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?

I had a hard time especially to take out the signal of Lane beautifully. For example, simply using gradient with various types of sobel filters did not work well, so we can detect it relatively coarse at that point and obtain desired data well at the time of subsequent superposition Parameter tuning was done with such a way of thinking. Also, I had difficulty with dotted lines. I tried to avoid this by superimposing the input image data of each frame in a specific section. Because I thought that dots are lines by temporal overlay. Although this may be effective for detection of lanes with relatively little change such as expressway, it may be useful to detect lanes such as urban areas where lane changes are relatively frequent and there are large changes, and other I think that it is an unsuitable technique for detecting objects.