Write up Template

You can use this file as a template for your write up 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.

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.

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

Camera Calibration

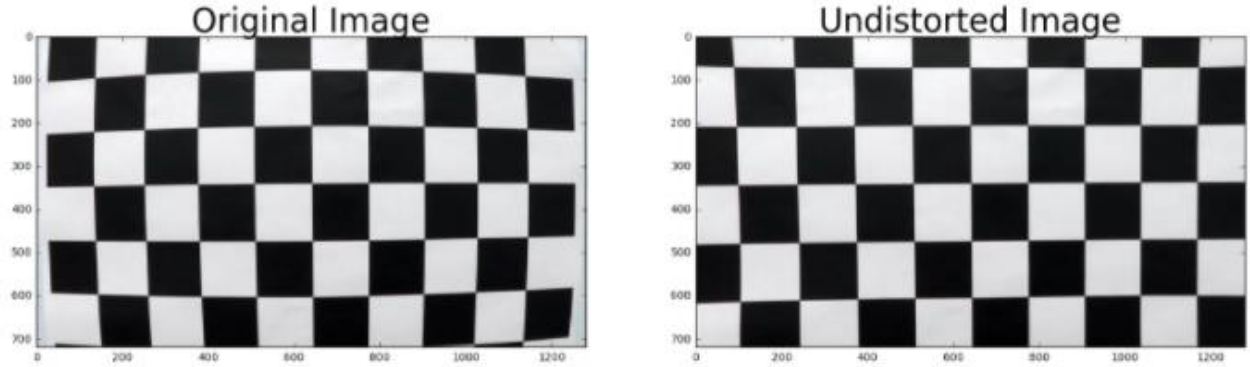
1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

The code for this step is contained in the first code cell of the IPython notebook P2.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 using the `cv2.undistort()` function and obtained this result:



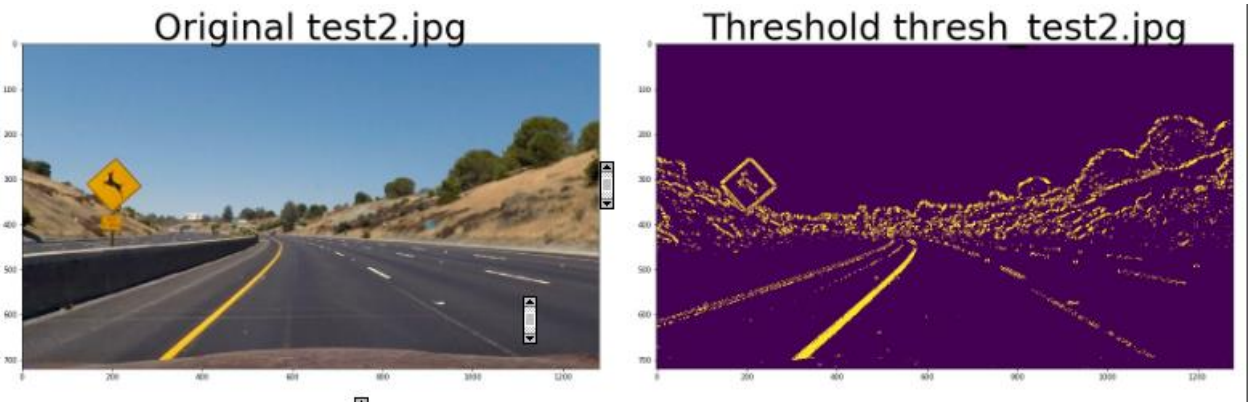
Pipeline (single images)

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.
Once the function to undistort image was tested successfully on the chessboard image, i tested it on one of the sample test images. The results are as shown below. Got a successful distort free image .

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
First I experimented with color thresholds like Red and S and L in HLS. Then with a combination of R, S and L, I found good lane detection with less noise.
Here's an example of my output for this step.
The code can be found in the 'color_binary()' function.



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 file `P2.py`. 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:
The image provided to the 'warper()' function should be undistorted and binary thresholded.
This function transforms the image into a bird's eye view i.e. top view. This enables us to view the lane curves in more detail and then would be useful for further processing of the lanes.

```python
corners = np.float32([[253, 697], [585, 456], [700, 456], [1061, 690]])

new_top_left = np.array([corners[0, 0], 0])
new_top_right = np.array([corners[3, 0], 0])
offset = [50, 0]
```

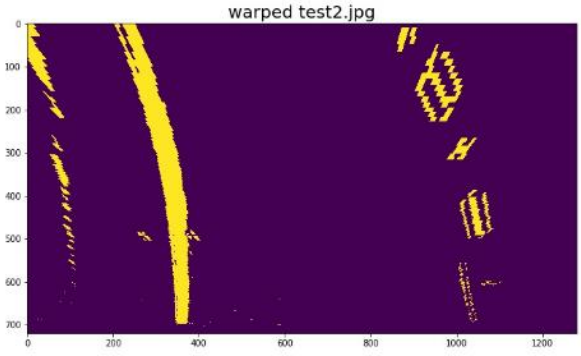
```
    img_size = (image.shape[1], image.shape[0])

    src = np.float32([corners[0], corners[1], corners[2], corners[3]])
    dst = np.float32([corners[0] + offset, new_top_left + offset, new_top_right - offset, corners[3] - offset])
```

This resulted in the following source and destination points:

Source	Destination
263, 697	303,697
585,456	303, 0
700,456	1011, 0
1061, 690	1011,690

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.



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

After applying calibration, thresholding, and a perspective transformation to the road image, I got a binary image where the lane lines stand out clearly. But here we need to still decide which pixels are part of the line and which belongs to the left line and which belongs to the right line.
To get this ,we plot a histogram of where the binary activations occurs across the image.

With this histogram we are adding up the pixel values along each column in the image. In our 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. We can use that as a starting point for where to search for the lines. From that point, we can use a sliding window, placed around the line centers, to find and follow the lines up to the top of the frame.

Then, set a few hyperparameters like, no of windows, margin, etc. related to our sliding windows, and set them up to iterate across the binary activations in the image

Once we found all our pixels belonging to each line through the sliding window method, we fit a polynomial to the line such as below.

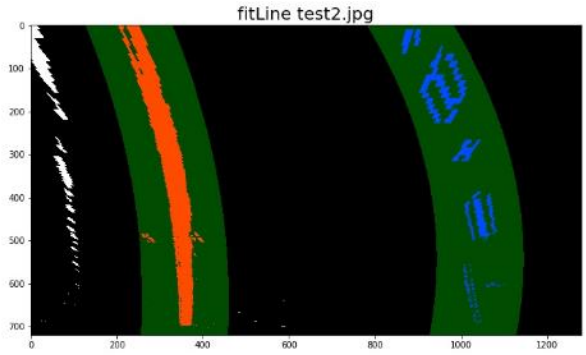
```
left_fit = np.polyfit(lefty, leftx, 2)
right_fit = np.polyfit(righty, rightx, 2)
```

However, using the above algorithm and starting fresh on every frame may seem inefficient, as the lines don't necessarily move a lot from frame to frame.

In the next frame of video we don't need to do a blind search again, but instead we can just search in a margin around the previous line position, like in the below image. The green shaded area shows where we searched for the lines this time. So, once we know where the lines are in one frame of video, we can do a highly targeted search for them in the next frame.

This is equivalent to using a customized region of interest for each frame of video, and should help us track the lanes through sharp curves and tricky conditions.

Thus we have estimated which pixels belong to the left and right lane lines(shown in red and blue respectively) and fitted a polynomial to those pixel positions.



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.

Now we find the radius of curvature and the position of the vehicle with respect to center. This has been coded in the function 'get_curvature()'.

Here we first convert x and y from the pixel space to meters to convert it into real world measurements.

```
# Define conversions in x and y from pixels space to meters
ym_per_pix = 30/720 # meters per pixel in y dimension
xm_per_pix = 3.7/700 # meters per pixel in x dimension
```

Then, we find the radius of curvature of each line using the formula..

$$R_{curve} = \frac{(1+(2Ay+B)^2)^{3/2}}{|2A|}$$

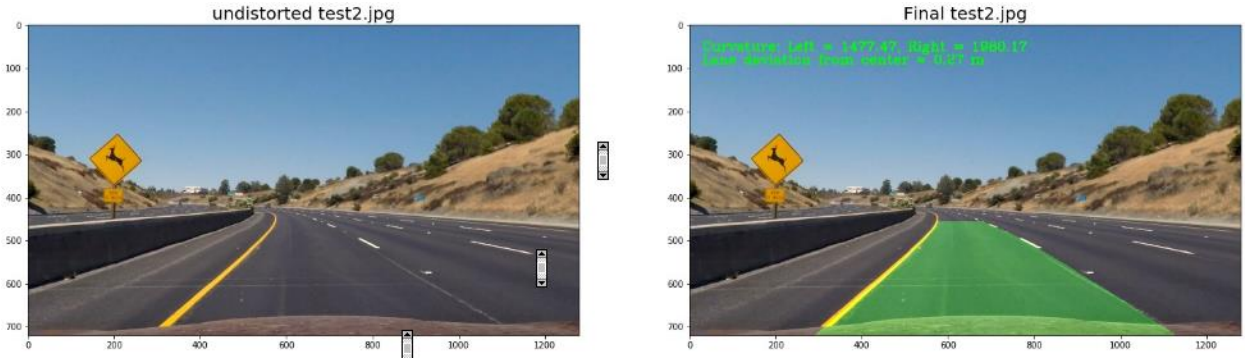
Where A,B and C are the second order polynomial coefficients and y is the ypixles converted to meters.

For position of the vehicle with respect to the center, we first calculate the intercepts points at the bottom of our image using second order polynomial using width and height of our image.
Then calculate the center using these intercepts and then the deviation from the center.

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

The final output image is as shown below. The code is in the function called "draw_lanes_on_image()".

Once the polynomial is fit and we detect the pixels of the image, we do an inverse perspective transform and fit the found pixels on to the original undistorted image



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 code:
https://github.com/bnnair/Advanced_Lane_find_Project2.git

Link to the video:
https://github.com/bnnair/Advanced_Lane_find_Project2/tree/master/videoimages

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 validation used for detecting whether a line exist or not in the image is not as good as it should be.
- Also the smoothness of the line in case of vast deviations need more work.
- The challenge project does not work properly.

- Plotting of lane lines after fitting the polynomial was a challenge. The plot of the yellow line was all over the place and not where it should be, even though the lanes are correctly detected.

Here I'll talk about the approach I took, what techniques I used, what worked and why, where the pipeline might fail and how I might improve it if I were going to pursue this project further.

- I used the second order polynomial to detect the lane lines and draw a margin over the lines. This method has served its purpose however would like to try out the convolution window sliding method as well to see if there is any difference in the accuracy with which it tracks the lanes.

- Line class is used here for tracking history and using the same for validations and smoothening. However, in my case, these do not seem to work, even though I have implemented it. I would be working more on the same.