Advanced Lane Finding Project: Krishna

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

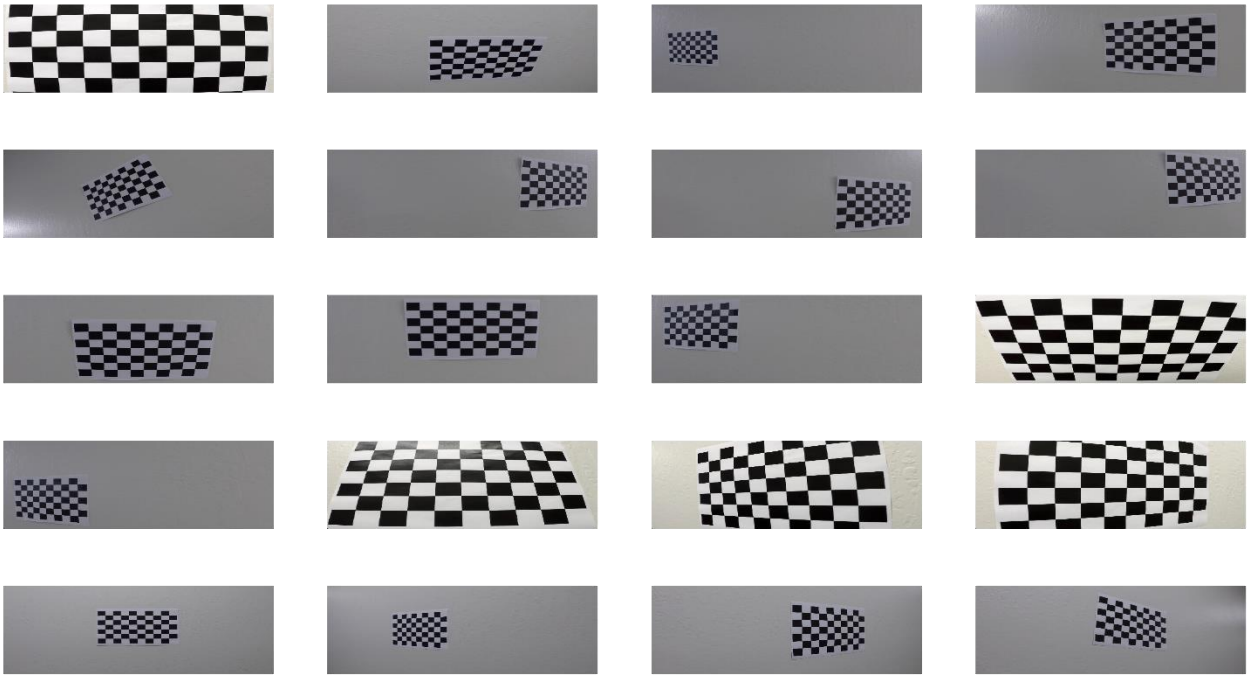
1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

The code for this step is contained in the first code cell of the IPython notebook located in the 1st cell block.

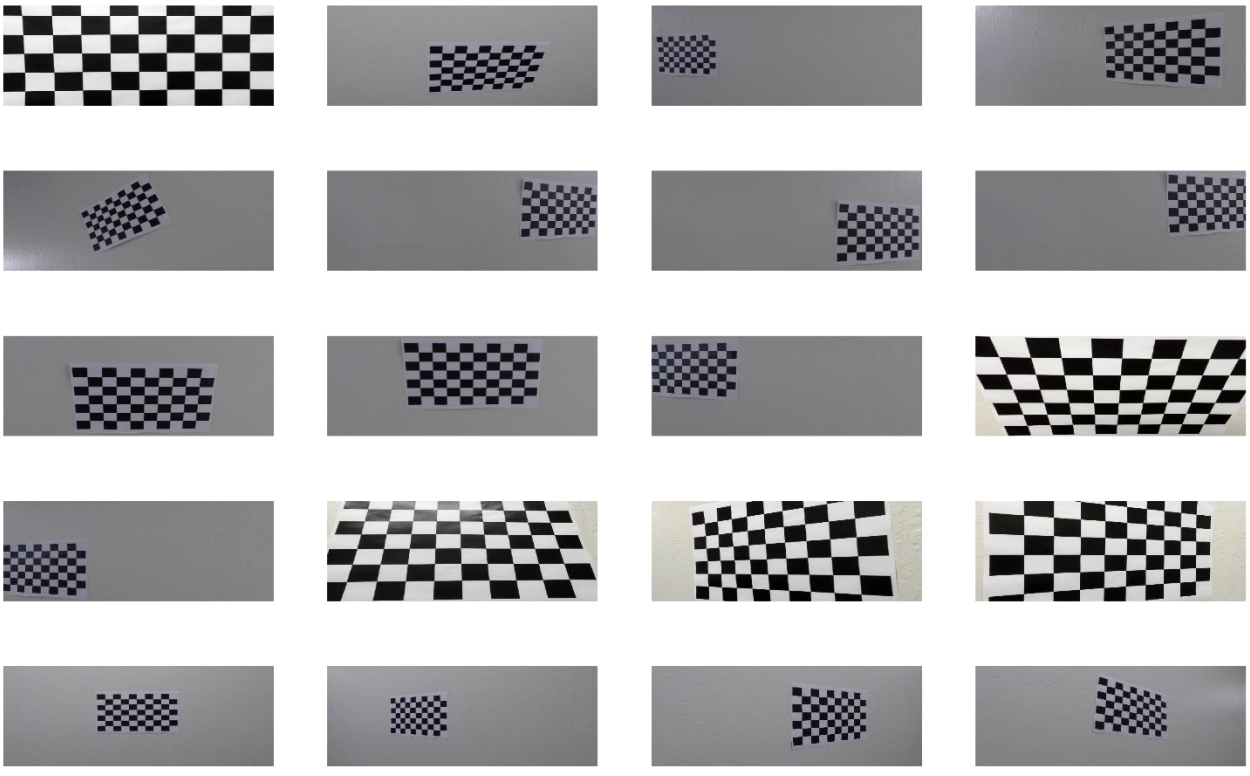
For doing camera calibration, I have used the chessboard images provided by Udacity. I start by preparing "object points", which will be the (x, y, z) coordinates of the chessboard corners in the world. Here I am assuming the chessboard is fixed on the (x, y) plane at z=0, such that the object points 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. In this way, get the distortion coefficients `dist` and camera calibration matrix `mtx`.

I created a separate function named undistort which takes input as images. Using the distortion coefficients and the camera calibration matrix, the function returns an undistorted image using the `cv2.undistort()` function of opencv. Here is the result of undistorted image obtained after camera calibration.

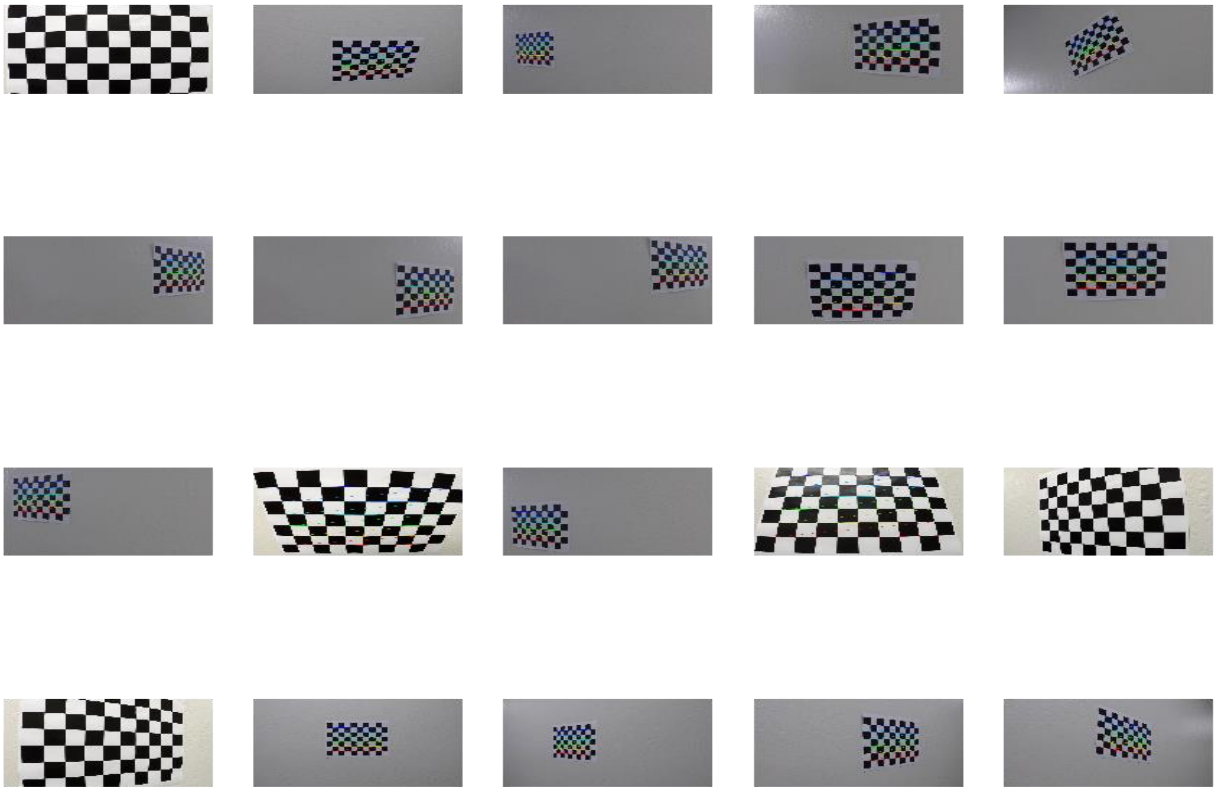
Distorted Images



Undistorted Images



Images with Chessboard Corners:



Pipeline (single images):

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

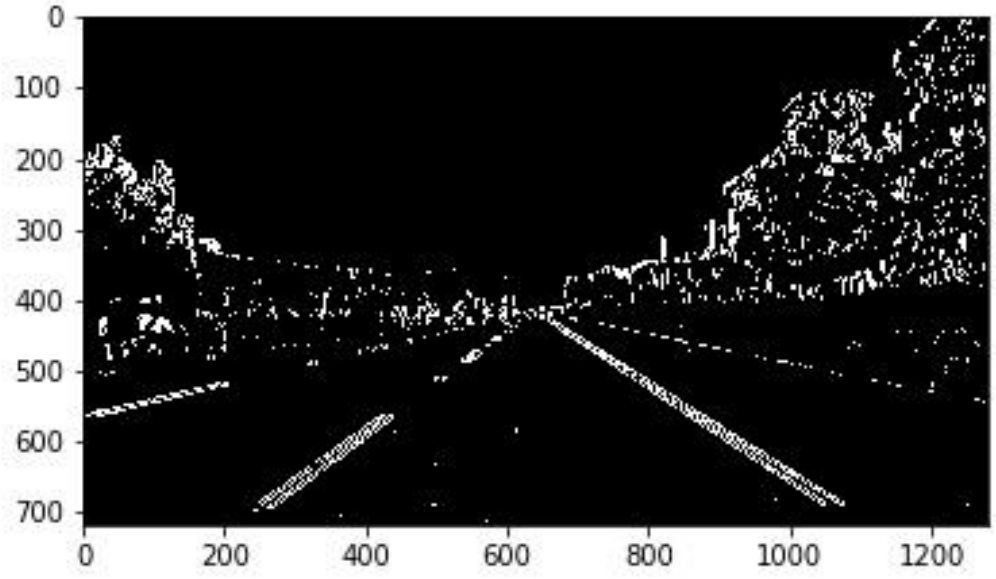
Distorted Test Image

Undistorted Test Image

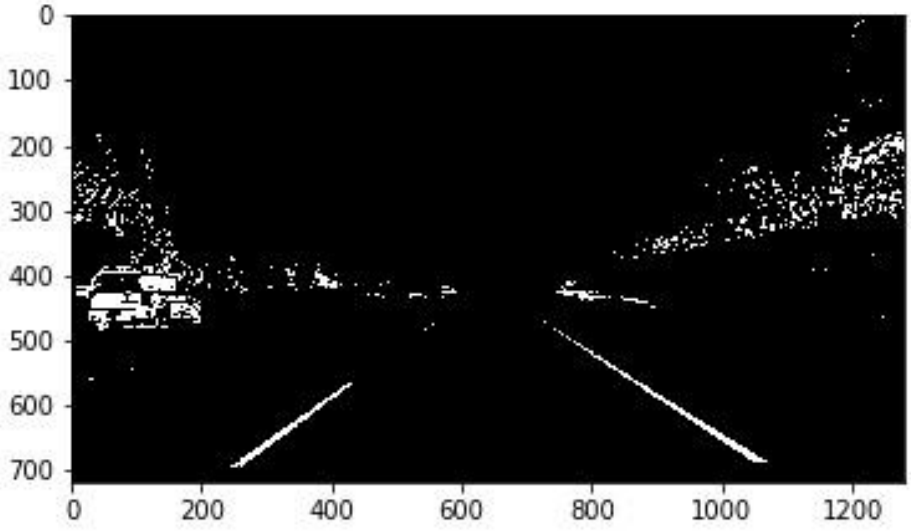
2. *Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.*

Initially I started with combination of HLS color format and gradient thresholds to generate a binary image. I also tried combination of RGB format and their individual channels. I also tried on with taking the gradient along y-axis, then the magnitude and also direction gradient. But, the combination of the S channel of HLS color format and gradient along x-direction performed well but still had some errors.

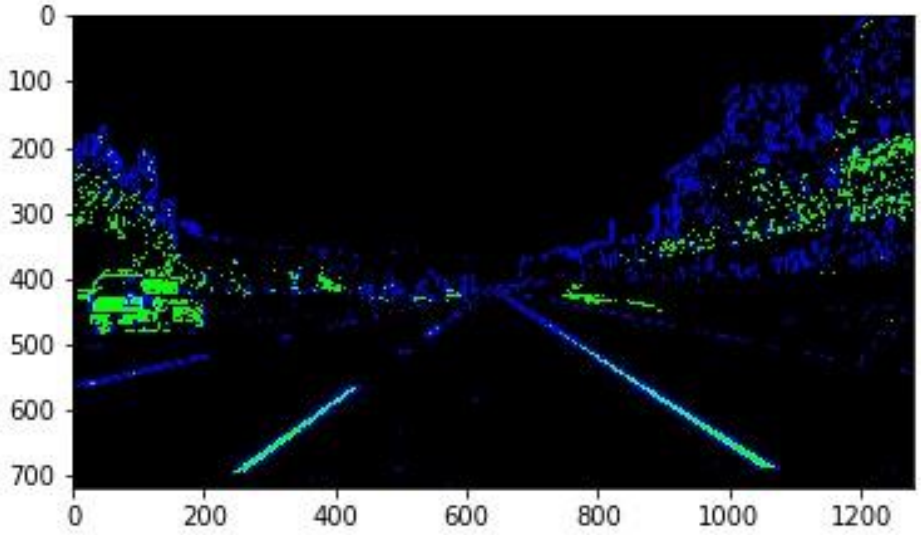
Then I moved on to LAB color format and used the B channel of this. Combining this with the gradient-x worked very well for me. For, B channel, I used a thresholding between (150,255). While for the gradient along X direction using **sobel**, I used the threshold between (30,130). Below are the binary images using the above method.



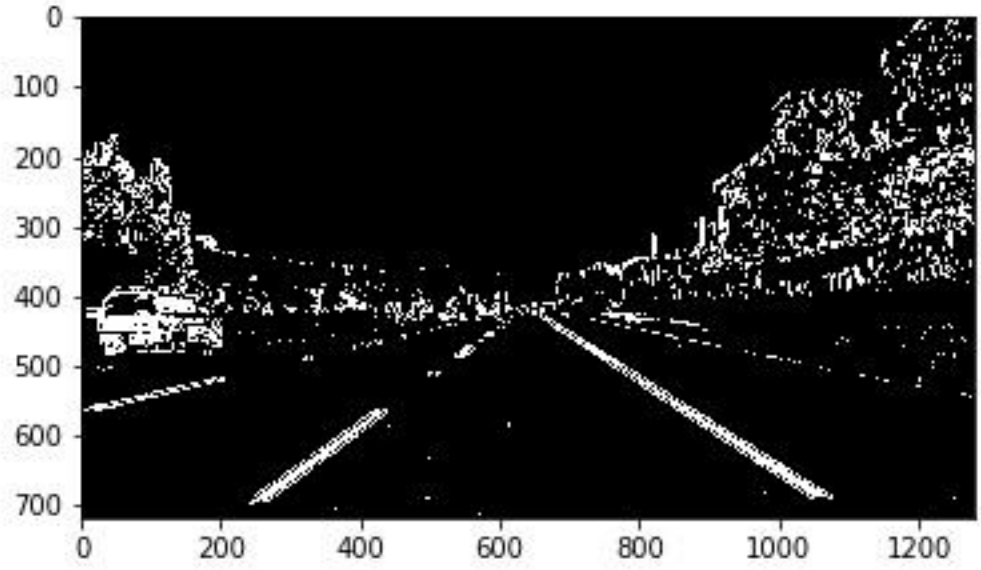
Sobel-X



LAB using only B channel



Colored Combined image for both S and gradient. Green represents the B channel and Blue represents the gradient x.



Combined Binary image for the previous image

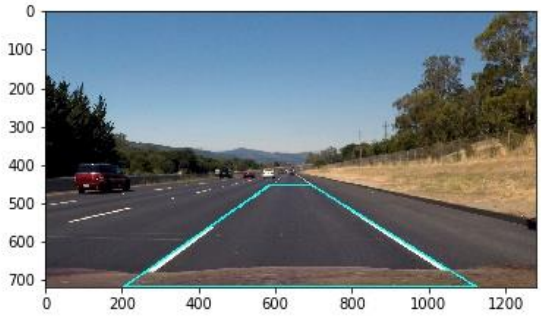
3. *Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.*

I have used the same values for the src and dst as given by Udacity, however I have tweaked the values of the source a little bit. Also, I have a defined a separate function for "warp" which does the perspective transform. This function takes an image and img_size as its input and returns warped image as output. This function is defined in the 2nd block of the ipython notebook file.

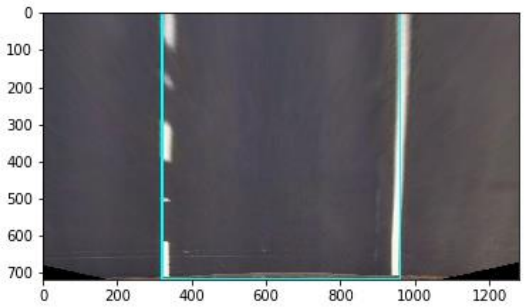
```
src = np.float32(
        [[(img_size[0] / 2) - 55, img_size[1] / 2 + 95],
        [((img_size[0] / 6) - 10), img_size[1]],
         [(img_size[0] * 5 / 6) + 60, img_size[1]],
         [(img_size[0] / 2 + 55), img_size[1] / 2 + 95]])

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]])
```

This resulted in the following source and destination points:

Source	Destination
585, 455	320, 0
203, 720	320, 720
1127, 720	960, 720
695, 455	960, 0



Undistorted Image with source points

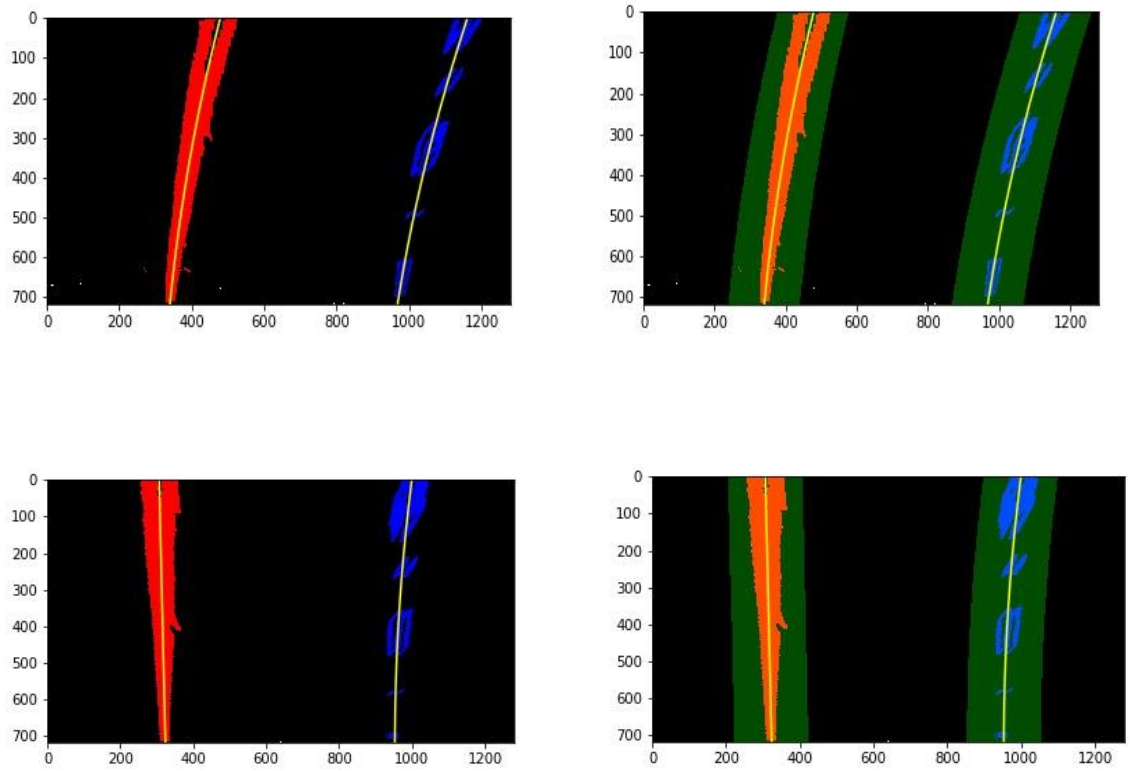


Warped Image with destination points

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

For identifying the lane-line pixels and fit them with a polynomial, I used the same technique as given the Udacity classroom. I fitted the 1st frame with a histogram and then calculated the maximum density of points to find the left and right bottom starting points of my images. Using these points and sliding window technique, I fitted rectangular windows on the warped image. Using the mean of the points lying in the windows we got the points required for fitting the polynomial. Then, using the "polyfit" function of numpy, I fitted a 2nd order polynomial and in this we get the coefficients of the polynomial. This part of the code is applied in a function named "Hist" in the 2nd block of the .ipynb file. It takes a binary warped image as an input and returns array of coefficients for 2nd order polynomial for both left and right lanes.

For next upcoming frames, I created another function “next_frames” which takes input as binary warped image, left and right polynomial coefficients. This part of code is in the 2nd block of the ipynb file. I used the same udacity technique, that is, to search in a specific region near the polynomial found in the previous frame.



Then, I defined a new function “draw” which draws in the region between the two detected left and right lanes.

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 have done the calculation of the radius of curvature in the “draw” function I created in the 2nd code cell of my notebook file. The “draw” function also takes the left lane & right lane polynomial coefficeints as their inputs. Once I had these coefficients, I converted them to real world coefficients by taking the conversion ratio between pixel and real world distance. I used the ratio provided in the classroom which are

ym_per_pix = 30/720 # meters per pixel in y dimension

xm_per_pix = 3.7/700 # meters per pixel in x dimension

Using these, I got coefficients in real world. Now, using these new coefficients, I got left and right curvatures and by taking their average, I got the curvature of the road. In a similar fashion, I found the difference between the center of car and center of lane. All of these are done in the last few blocks of code in the “draw” function.

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

Pipeline (video)

I have attached the video along with in the project folder.

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?

While doing the project, I felt that whenever there is a sudden change in the lightness or darkness in the video, then the lanes are detected very faintly. As a result, it might give errors in the detection. One solution to this is to consider more number of thresholding patterns. But however, this won’t be a good solution. I think a more robust solution to this problem would be to apply deep learning for the lane detection. By giving it training data and working on a good architecture to find lane lines would be better solution. Yes, of course we would need a lot of data for that, and also error might persist in that case as well, but it will be very less as compared to the present technique.