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.

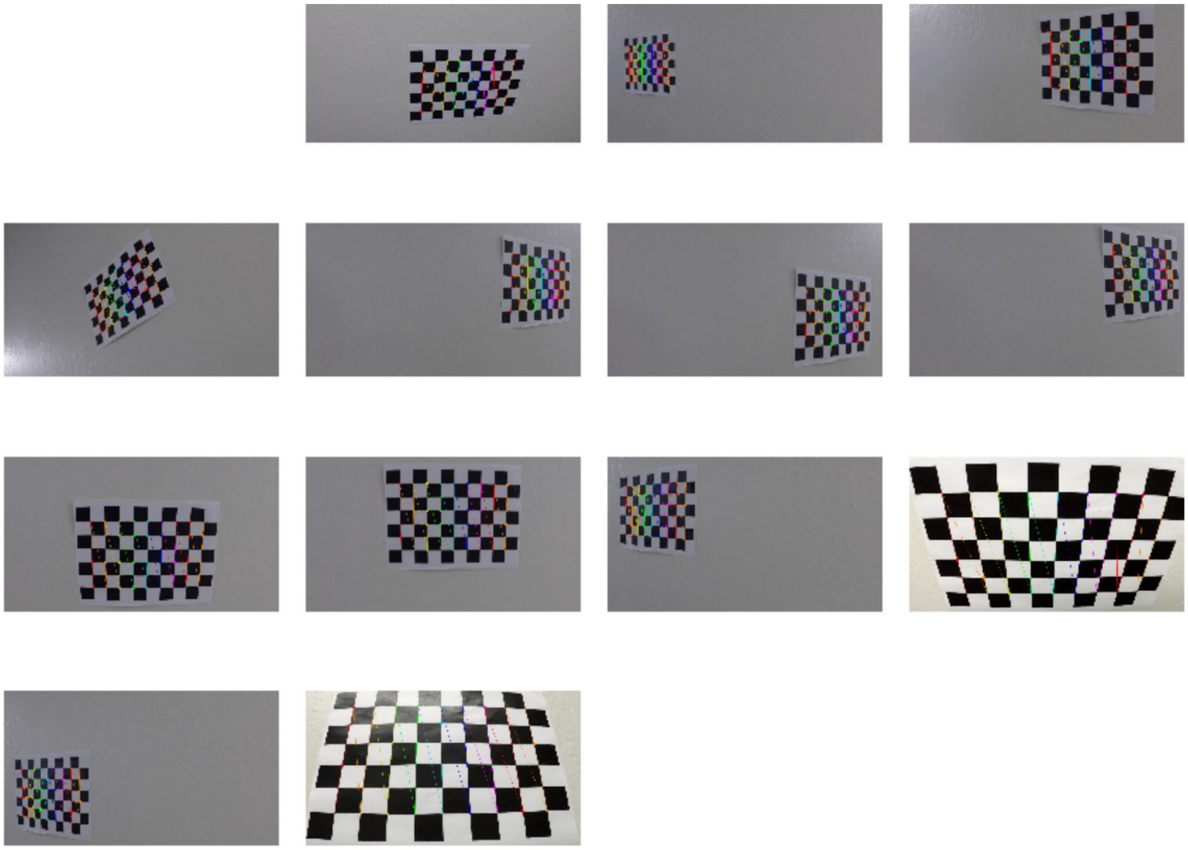
Rubric Points

Camera Calibration

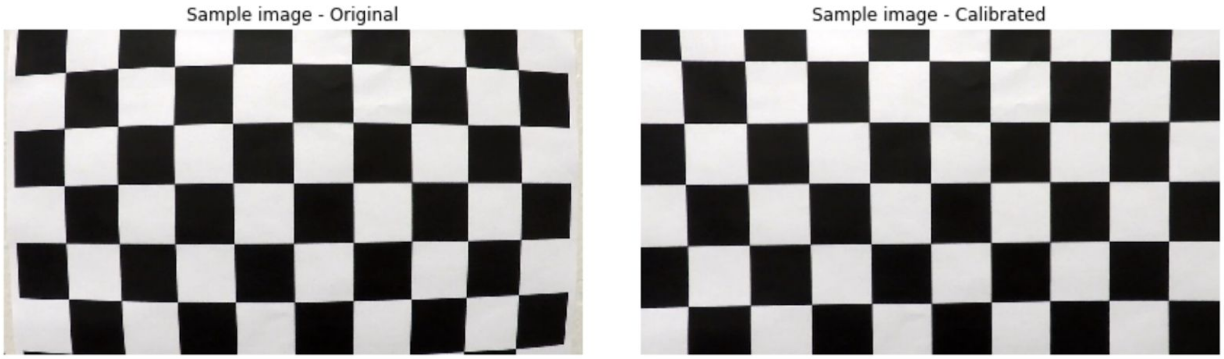
1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

The code for this step is contained in the second code cell of the Jupyter notebook Lane_find-combined.ipynb in this repository https://github.com/JesseR2/CarND-Advanced-Lane-Lines-JR.

Sample 9 x 6 checkerboard images from the camera were provided and used to calibrate video frames later in the project. OpenCV's findChessboardCorners function was used to detect corners and drawChessboardCorners was used to overlay lines showing them. They are shown here with the detected corners.



Once the objpoints and imgpoints were identified, calibrateCamera from cv2 was used to determine the calibration matrix. Here is a sample image along with a corrected version of it using this matrix with cv2.undistort.



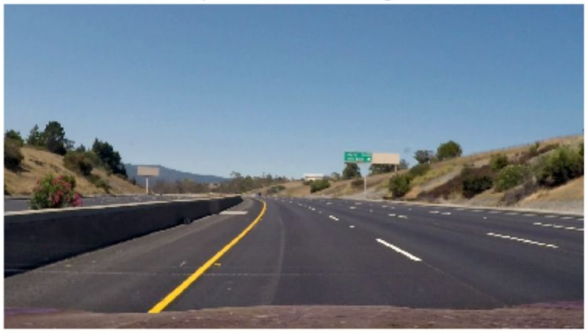
Sample image - Original

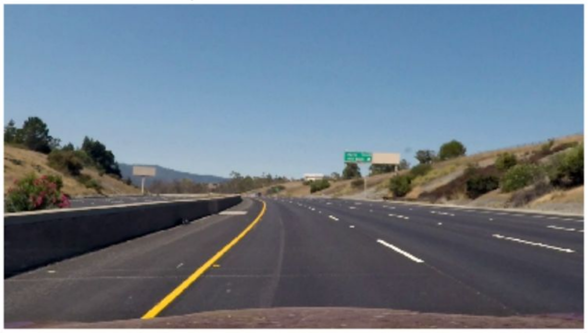
Sample image - Calibrated

Pipeline (single images)

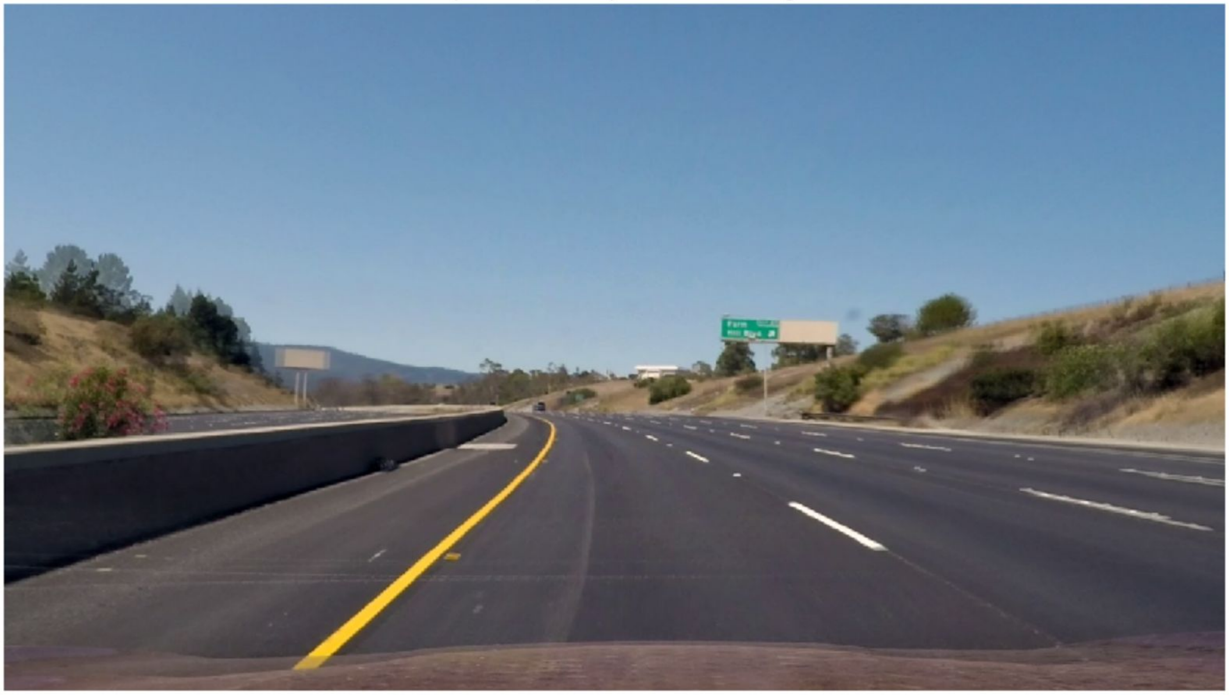
1. Provide an example of a distortion-corrected image.

Using the calibration matrix described above, I again used cv2.undistort too correct a sample frame from the test video. Before and after images are show here.


Sample video frame - Original


Sample video frame - Calibrated

The correction is subtle and hard to see by just looking at the two images so i also included a composite image using cv2.addWeighted using a 50%/50% alpha blend. The differences in the images, especially around the perimeter are much easier to see in the overlay.

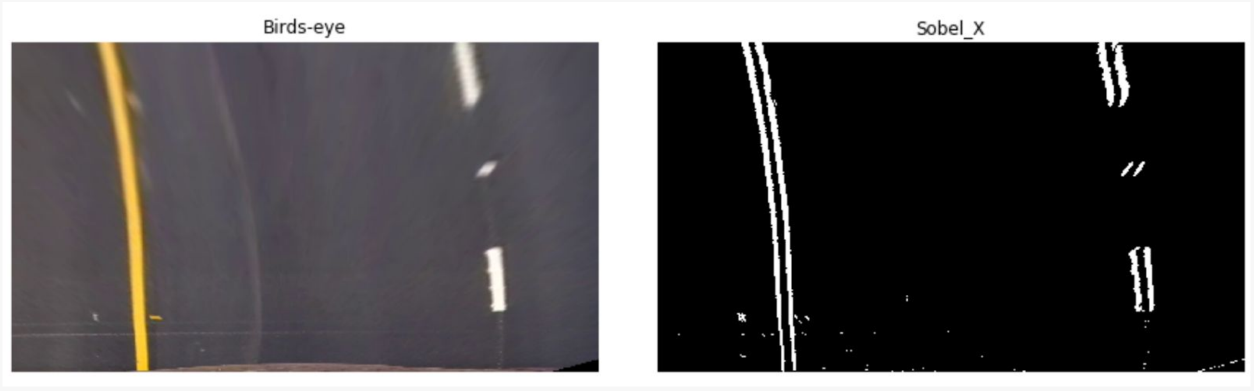

Composite of pre and post calibration images

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.

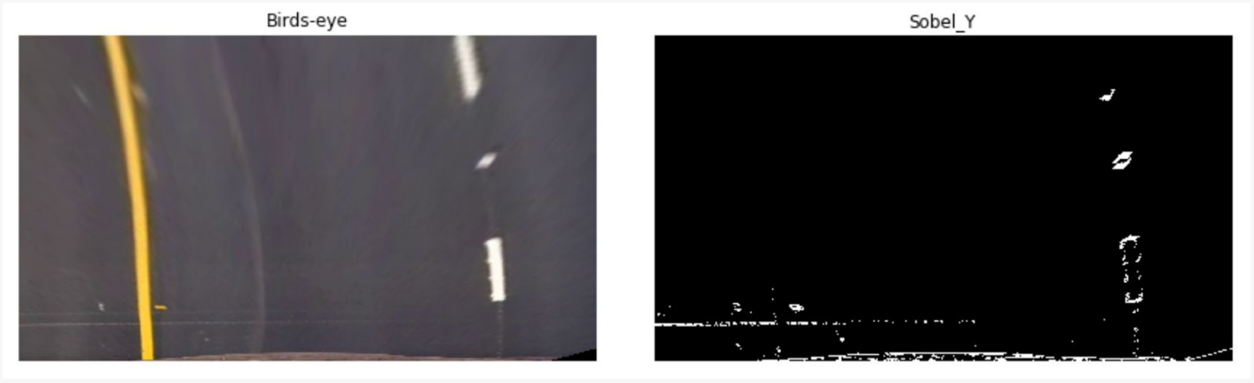
I chose to warp the image to birdseye view prior to doing any binary thresholding. The warping process is described in section 3 below. The birdseye images were processed using several algorithms to identify the lane lines including Sobel X, Sobel Y, Sobel Magnitude, Sobel Direction, as well at two color channel thresholds (Hue and Saturation) after conversion to HSV colorspace. This code is in the same Jupyter notebook from In [96]: to In [107]:.

The Soble masks were generated using cv2.Sobel. Several parameters were tested for each one and the following seemed to provide the best results. Here are the final parameters and sample images for each.

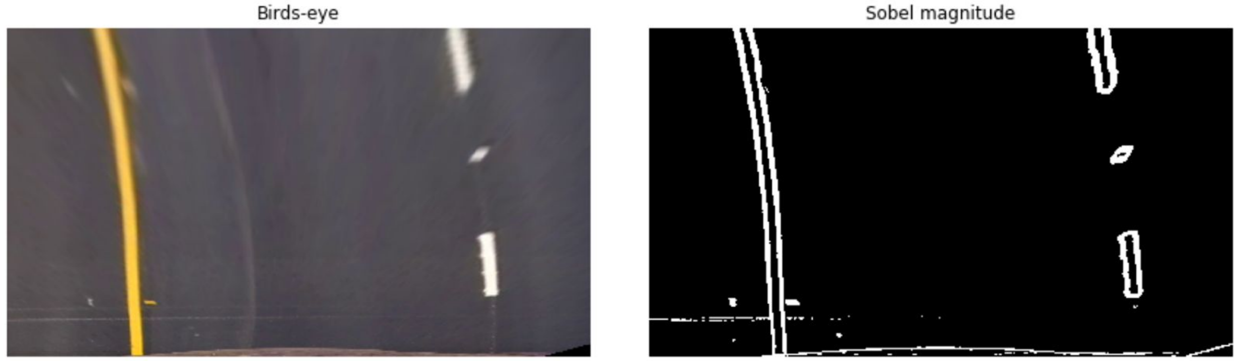
Sobel X - orient='x', thresh_min=25, thresh_max=255



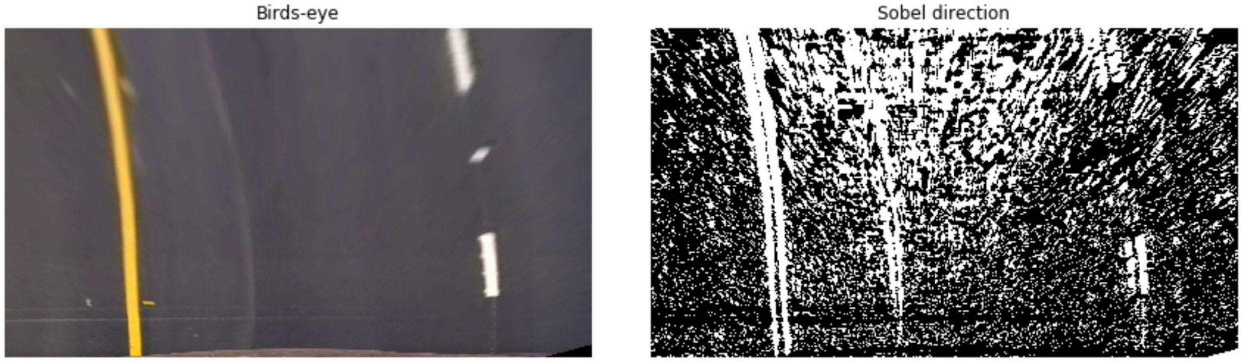
Sobel Y - orient='y', thresh_min=25, thresh_max=255



Sobel Magnitude - sobel_kernel=5, mag_thresh=(30,255)

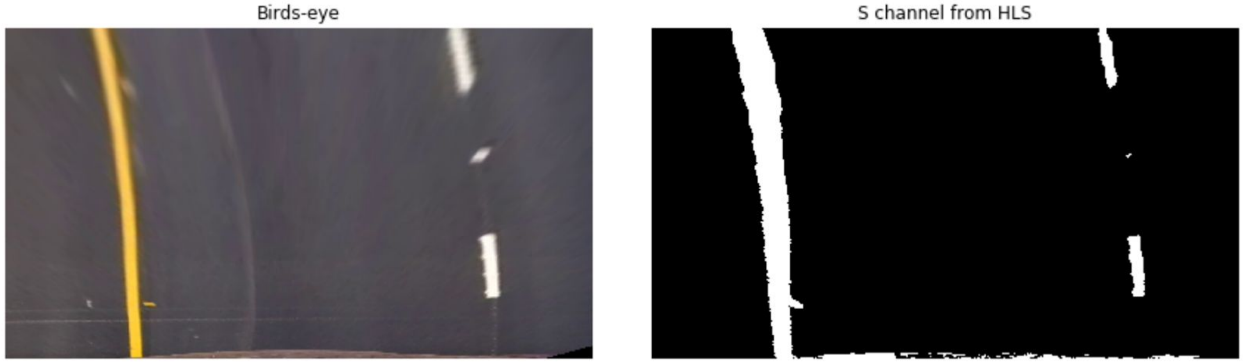


Sobel Direction - sobel_kernel=9, thresh=(0, 0.4)

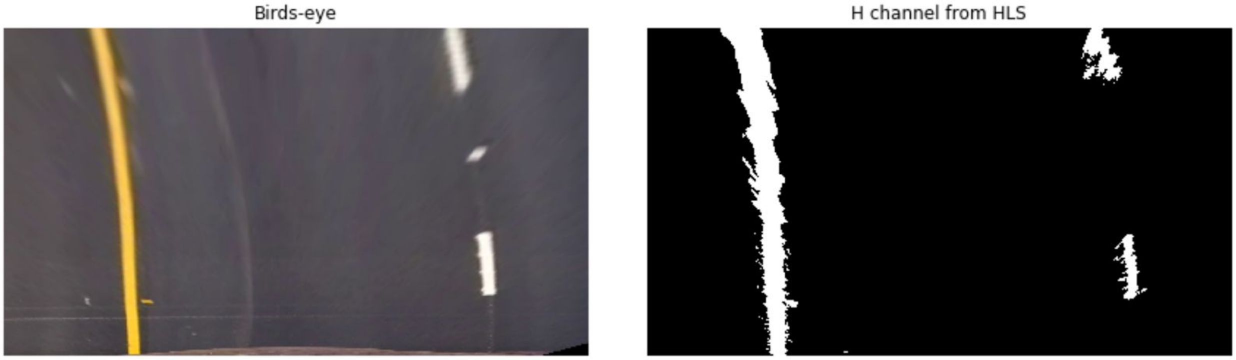


Several color thresholding techniques were tested in various color spaces. Ultimately Hue and Saturation channels of the HSV color space provided good results. OpenCV was used for color space conversions and the thresholds were set as follows.

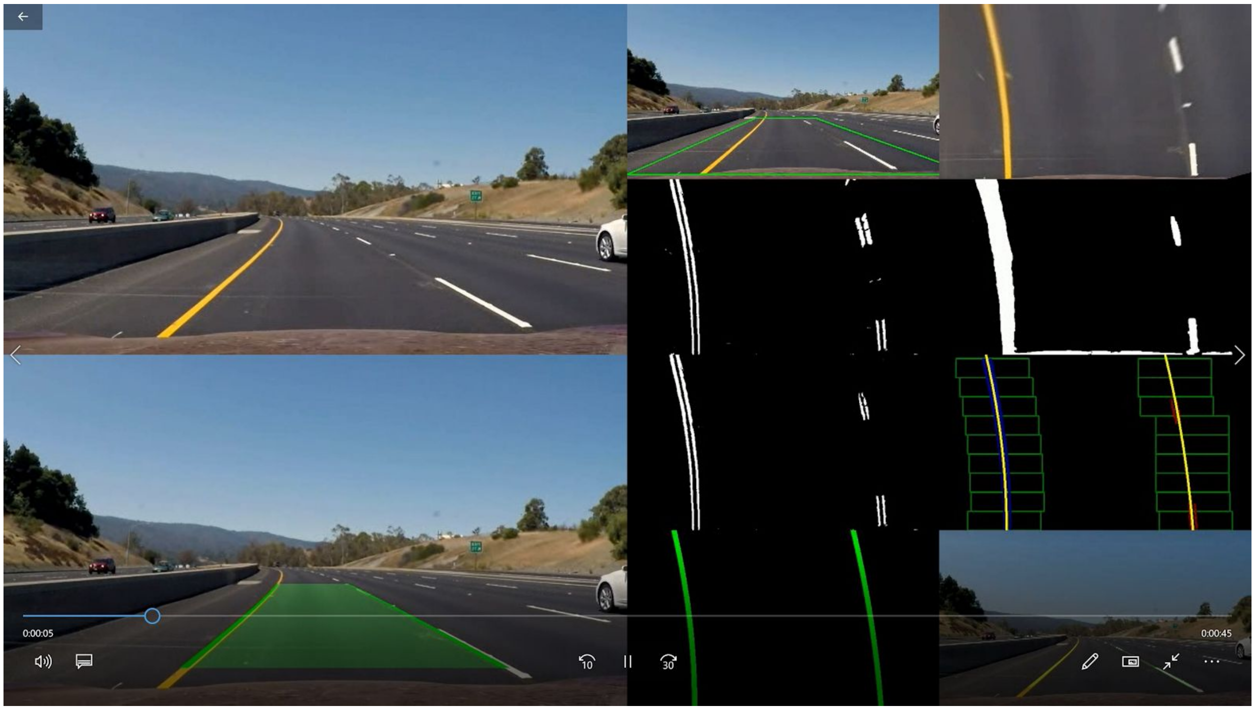
HSV - S channel - 30 to 255



HSV - H channel - 15 to 100



These Sobel binary filters were combined by (sobelx_bin | sobely_bin | sobelm_bin) & sobeld_bin. The color filters were ORed (s_bin | h_bin) and then dilated using a 5x5 kernel and cv2.dilate before being ANDed with the composite Sobel mask. This is done in In [125]: to achieve the final binary lane output. Since there were several steps in the pipeline I found it helpful to generate a diagnostic video showing various steps in the process. This is a sample frame.



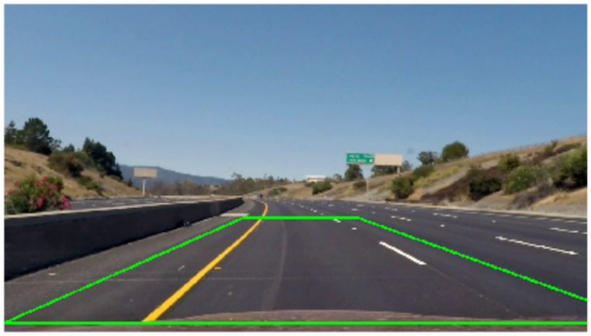
The video diagnostic full is at https://www.youtube.com/watch?v=f6-I5ZnKfEw

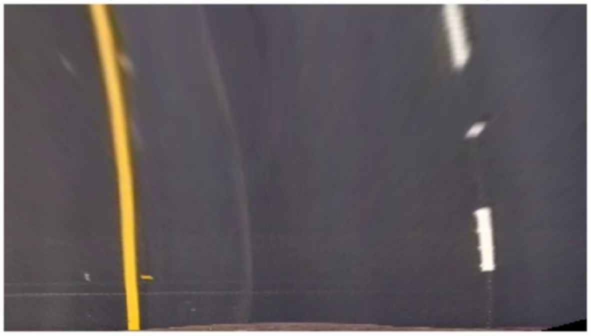
3. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

As seen in In [93]: through In [95]:, I warped the images to birdseye prior to the binary thresholding using the following source and destination points.

```
src = np.float32([[525,470], [775,470], [1390,700], [0,700]])
dst = np.float32([[0,0], [1279,0], [1279,719], [0,719]])
```

Using getPerspectiveTransform and warpPerspective from the cv2 library I warped these sample images (and all video frames) to a top down view for processing. I also drew the a green trapezoid connecting the source points to show the being area warped.

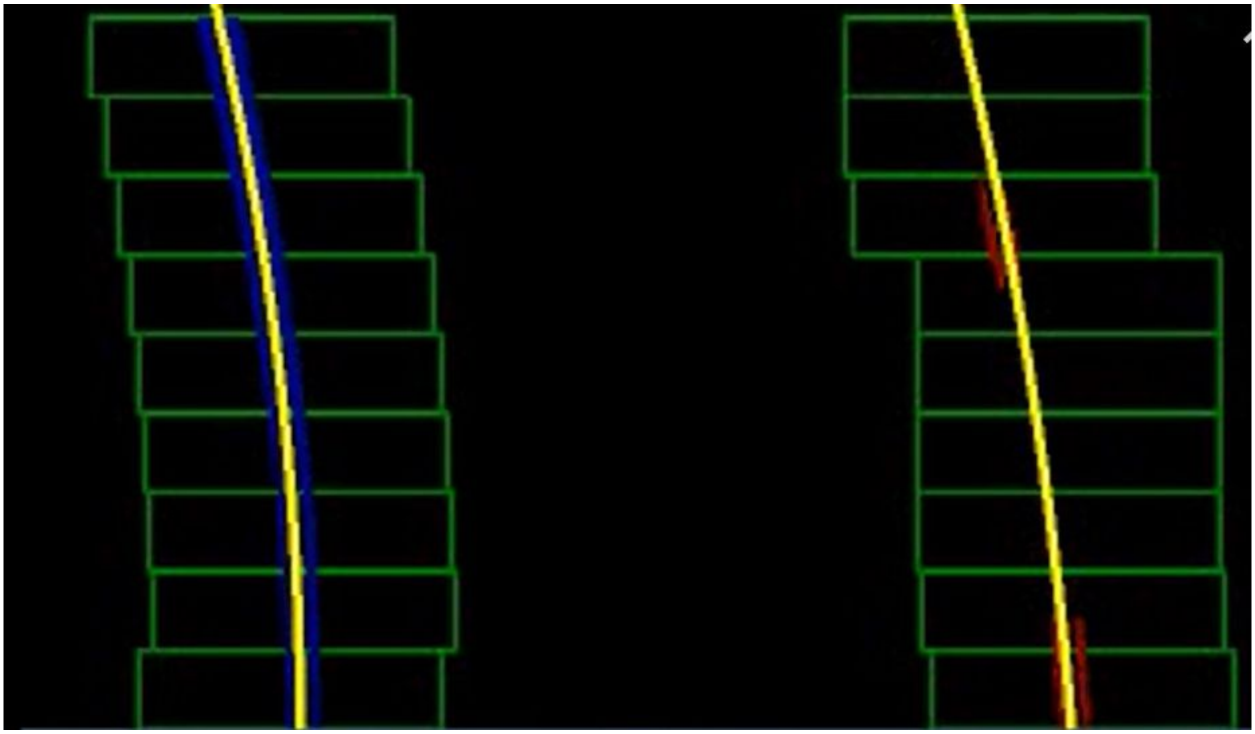

Sample video frame - Calibrated with ROI


Sample video frame - ROI - Calibrated and Warped

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

On In [122]:, run all the lind finding logic. The function brings in the current video frame and previous lien fit data to allow frame to frame smoothing. The code breaks each lane line up into nine vertical sections and finds the most likely lane location starting at the bottom and working its way up building an array of points that is passed into Numpy's polyfit which determines a best fit second order curve. Here is a sample image showing the regions that were evaluated (green), the binary mask points found (blue and red), and the best fit curves (yellow).

To reduce the noise caused by poorly detected lines, I ignored lines that were outside of a reasonable range and did some averaging with previous found lines.

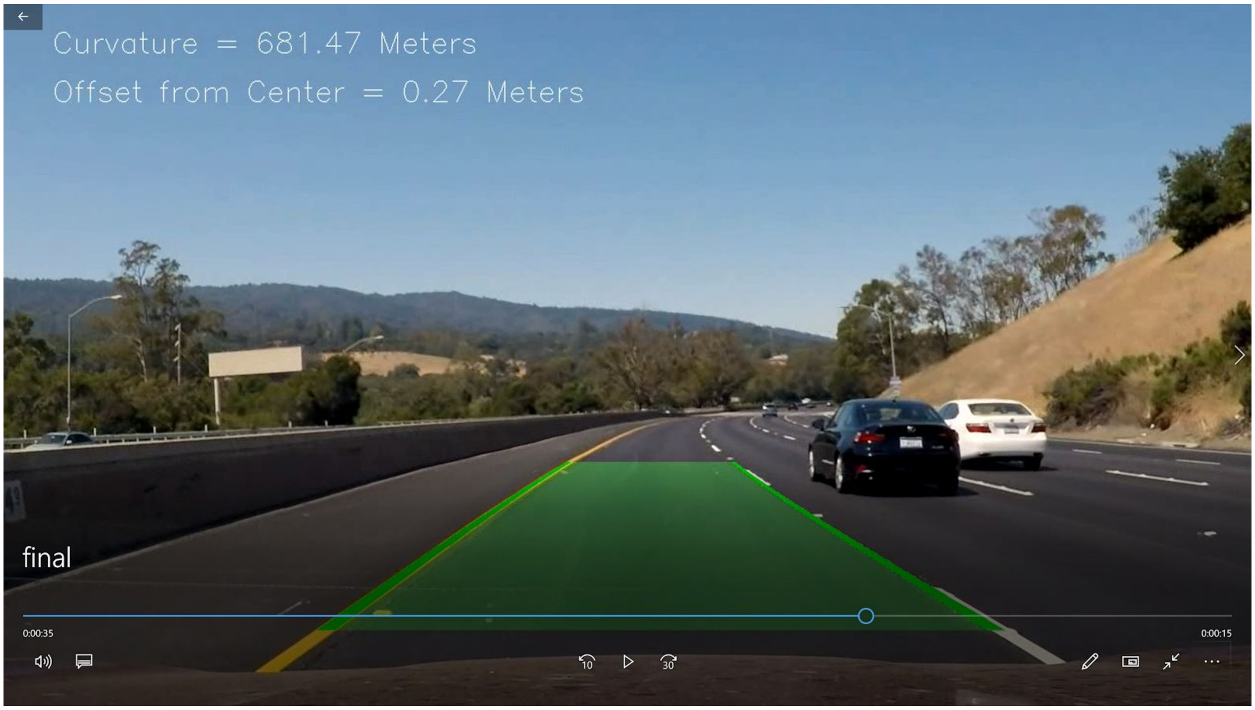


5. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

I did this in In [122]: in the last few lines of code commented with # Calculate curvature. The offset from center is also calculated there, all units are in Meters. Data can be seen in the output of cell In [125]:.

There was quite a bit of frame to frame noise so I summed 10% of the current value and 90% of the previous running average to get a more stable number. This was rounded to to two decimal places and displayed as a text overlay using cv2.putText.

Here is a sample image with curvature and offset information.



Curvature = 681.47 Meters

Offset from Center = 0.27 Meters

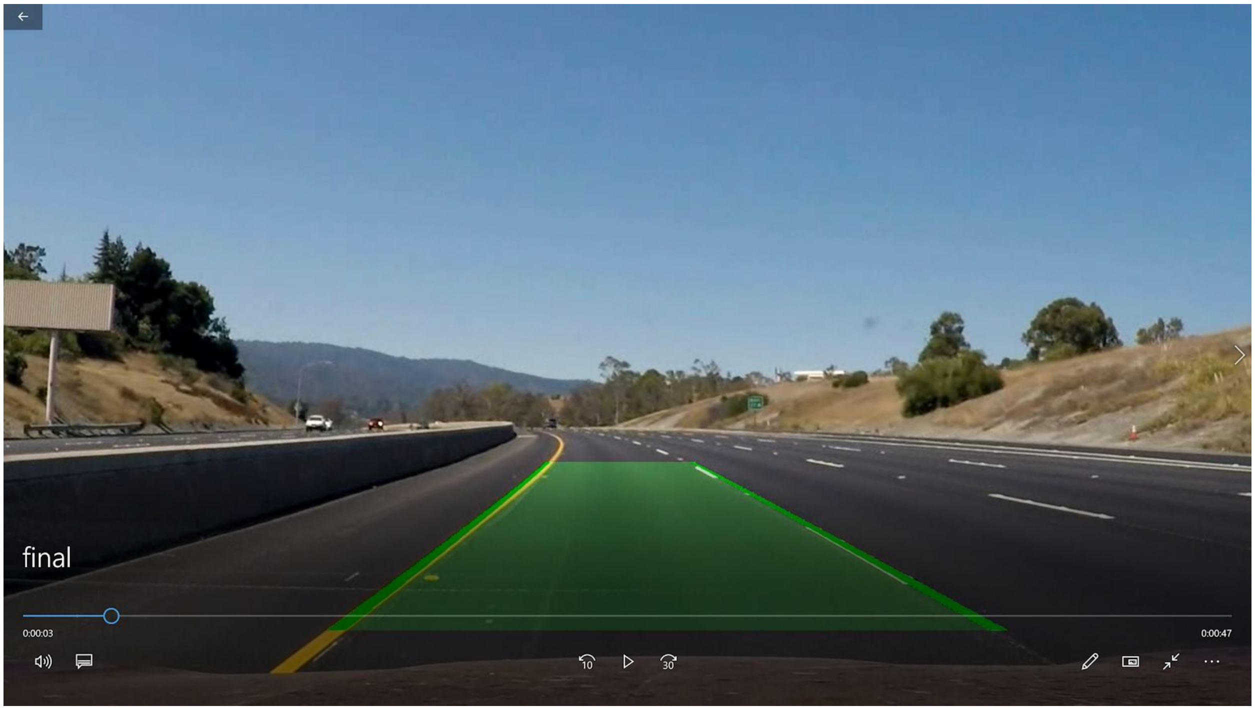
6. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.

Warping the images of the lines back was done using the same method described above but using the following source and destination points. Much of the processing was done on images that were reduced 50% to increase speed. The lines were then scaled back up to 1280x720 to overlay.

src = np.float32([[0,0], [639,0], [639,360], [0,360]])

dst = np.float32([[525,470], [775,470], [1390,700], [0,700]])

Here is a sample of the image with lane lines overlaid on top.



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 are github and youtube links to the video…

https://github.com/JesseR2/CarND-Advanced-Lane-Lines-JR/blob/master/final.mp4

https://www.youtube.com/watch?v=e3tGcYgd97M

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 results on the test video were relatively stable but did have some small issues when the road conditions change, particularly around bridges and in area that had shadows. These issues were masked by the frame to frame smoothing but can be seen in the pipeline diagnost video in the mask stage. I suspect that the parameters that were found to work well on this video would fail if the lighting or weather conditions changed significantly or the lane lines were not as well defined or had faded.

I suspect that using manual rules and hand tuned parameters like this project uses are far outclassed today by well trained neural networks that can be built on millions of miles of road data in all weather and lighting conditions and in various geographic regions and road types.

A CNN would be the approach I would use to improve on this project. The manual process this project covered may be useful in the early stages of data labeling to avoid having to annotate all lane lines by hand.