Camera Calibration

The code for this step is contained in the first function, camera_calibration(), where I iterate through the images in the camera_cal/ directory using the glob API. When I have finished going through all the images, I save the camera matrix and distortion coefficients in pickle file called dist_pickle.

I start by preparing the object points, 'objp', which will be the (x, y, z) coordinates of the chessboard corners in the world for each image. A larger list, 'objpoints' is created to hold each set of objp as I iterate through the image and successfully find the corners. 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. The image points list, 'imgpoints', is also created and will be appended with the (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection. While iterating through the images, the corners are found with the cv2 function, 'findChessboardCorners()' which returns a boolean determining successful completion. Here is a photo showing the detected corners drawn on a distorted chessboard image.

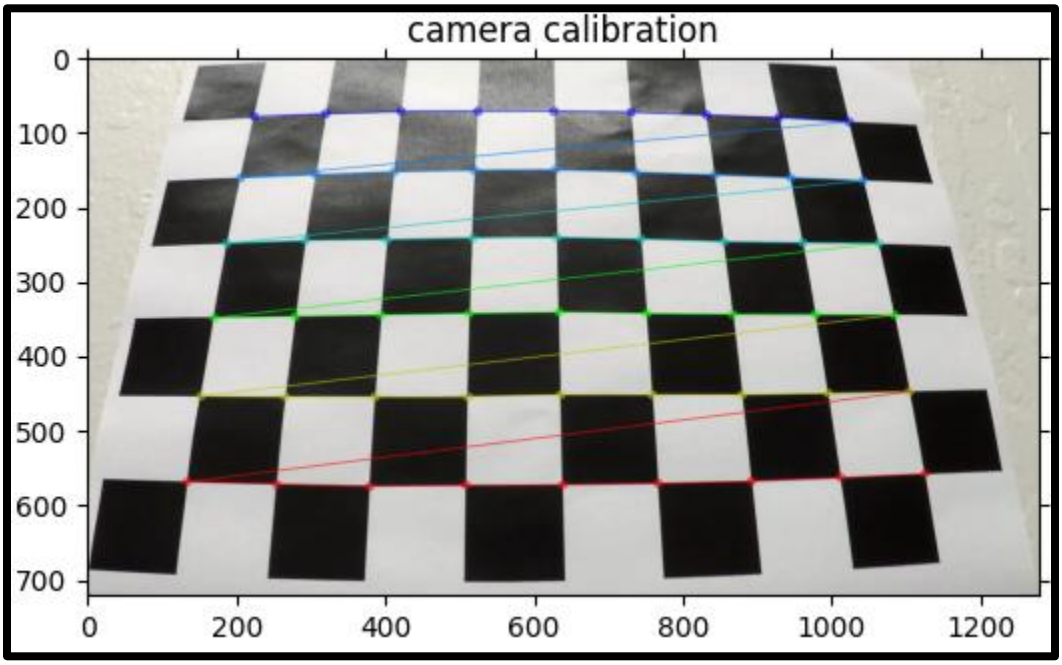

Figure 1: Detected Corners on camera_cal/calibration1.jpg file (subplot 9)

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 pipelined images using the `cv2.undistort()` function. Here is a calibrated example of the camera_cal/calibration1.jpg file processed:

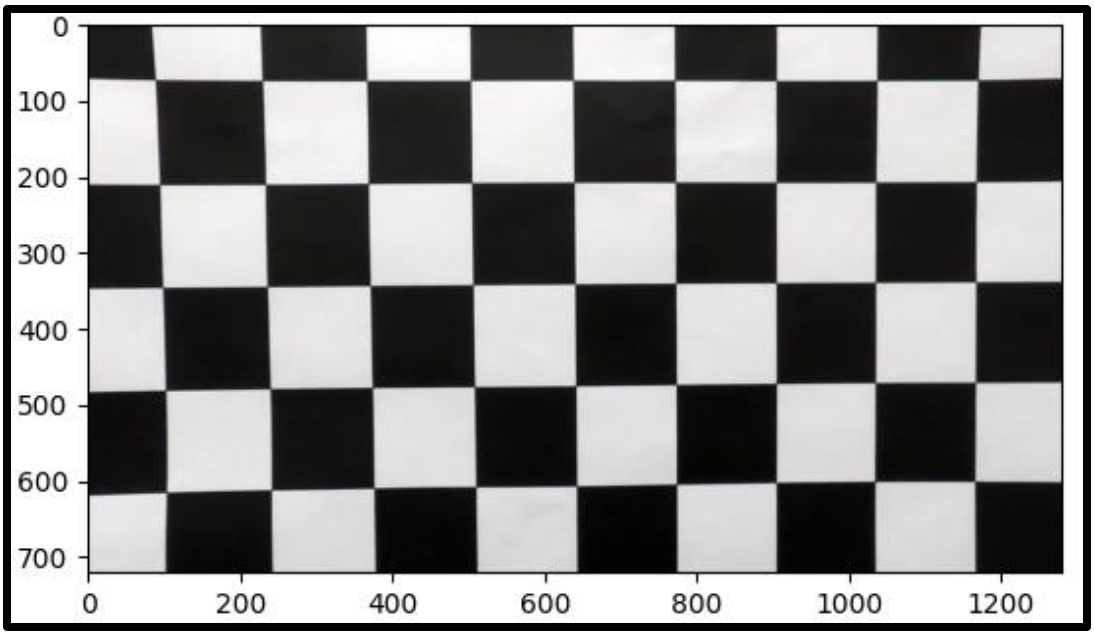

Figure 2: Undistorded camera_cal/calibration1.jpg file

Pipeline

1. **Distortion-corrected image**

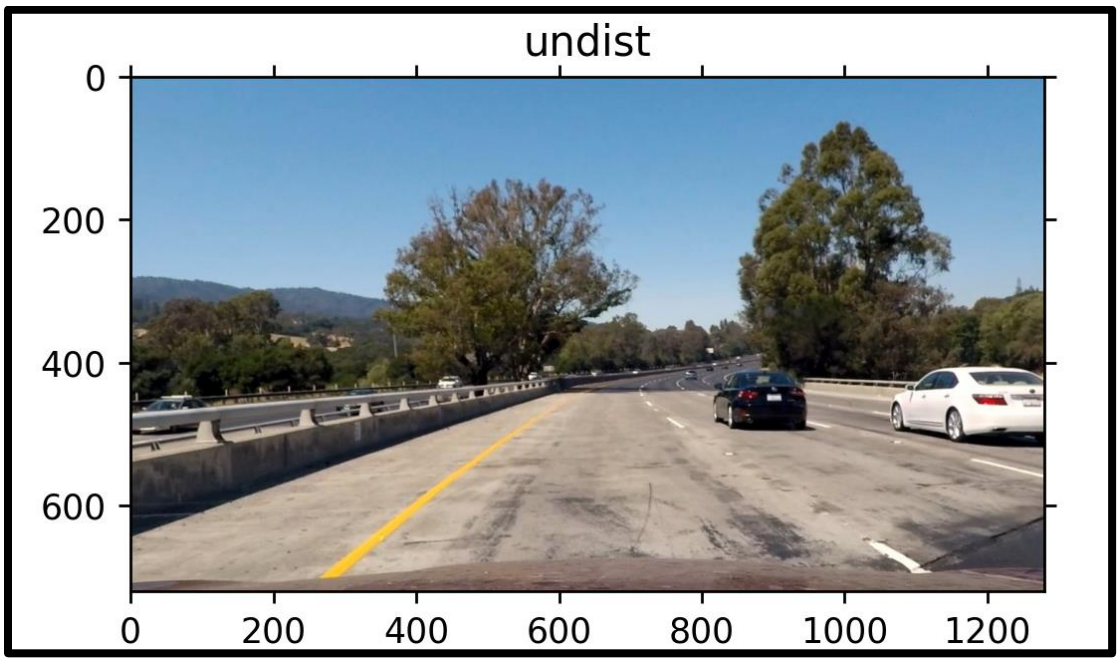

Figure 3: Distortion-Corrected test_images/test1.jpg file (subplot 1)

The figure above is an example of the result of applying a distortion correction to one of the test images. In line 48 of the code in the function distortion_correction(), the pickle file dist_pickle is immediately opened and loaded. This file contains the camera matrix and distortion coefficients from the camera calibration routine performed earlier. These values are loaded in the cv2.undistort() function along with the test image from the same camera as the calibration photos. The result is an the undistorted image.

2. Color Transforms, Gradients, etc. for Thresholded Binary Images

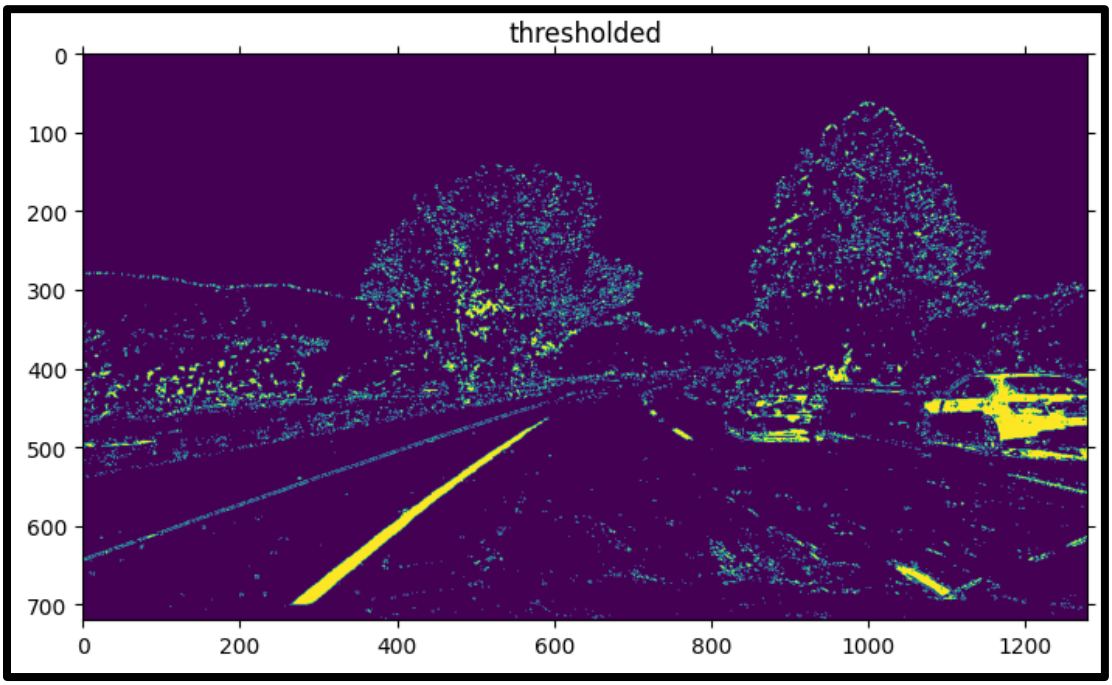

Figure 4: Thresholded test_images/test1.jpg file (subplot 2)

The figure above is an example result from applying the thresholds() function from line 58 to an undistorted image. Three thresholds are performed in the image. First is on the saturation channel of the HSV transformed image. This channel is useful because it holds its range well up to varying lighting conditions. Secondly, a magnitude threshold is performed and is bitwise ANDed with thirdly a direction threshold. This pair give us strong linear tendencies in the image. Bitwise Oring the magnitude-direction threshold with the saturation threshold brings out strong enough road lines to perform lane detection in varying lighting conditions. This image is then masked to expose just the area around the front of the vehicle where the lines should be. The figure below shows the result of the mask performed on Figure 4.

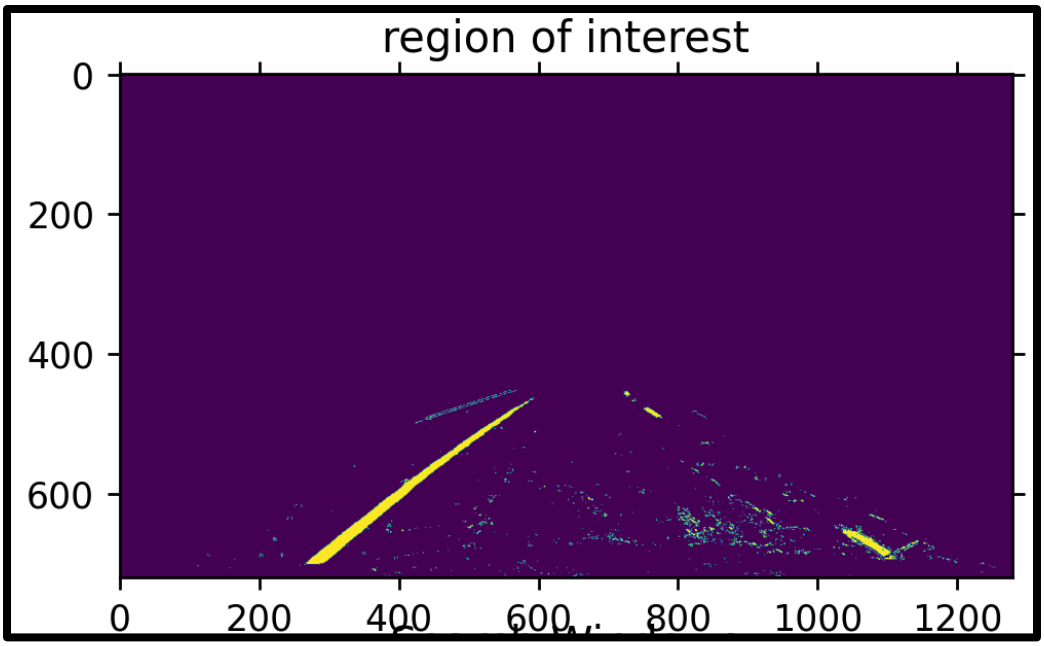

Figure 5: Masked Region of thresholded test_images/test1.jpg file (subplot 3)

3. Perspective Transform

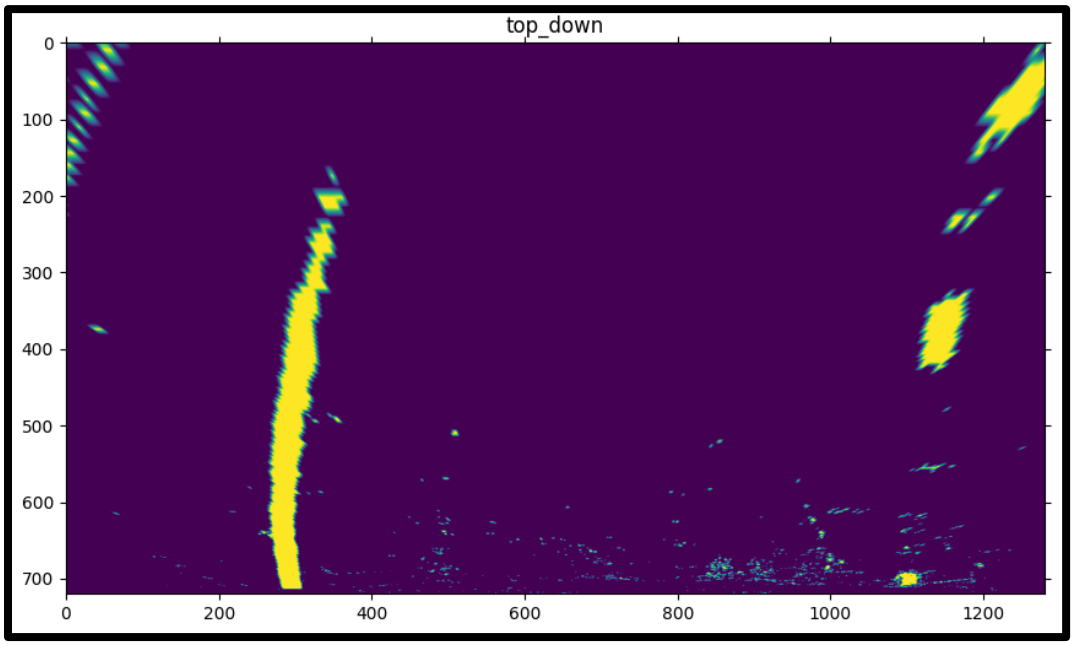

Figure 6: Perspective Transform of test_images/test1.jpg file (subplot 4)

The figure above is a resulting image from applying the warp() function on line 122. The vertices arrays src and dst had to be tuned to matched to what felt comfortable. It became clear very quickly that attempting to process the entire lane is challenging but not grabbing enough of it leaves too few lane pixes for line detection. A compromise was made and I chose to scale the image dimensions for the points in the following manner:

```
# define 4 source points
    src = np.float32(
        [[.44*shape_x,.63*shape_y],
        [.56*shape_x,.63*shape_y],
        [.95*shape_x, shape_y],
        [.05*shape_x, shape_y]])
    # define 4 destination points
    dst = np.float32(
        [[.035*shape_x,0],
        [.965*shape_x,0],
        [.90*shape_x,shape_y],
        [.10*shape_x,shape_y]])
```

It also came to light that the perspective transform does much of the masking for you if the lanes are fitted wide enough to the new perspective. The image below shows a parallel lines resulting from warp() being performed on the test_images/straight_lines1.jpg file. A similar view can be seen on the output_images/plots_from_pipeline.jpg project output file.

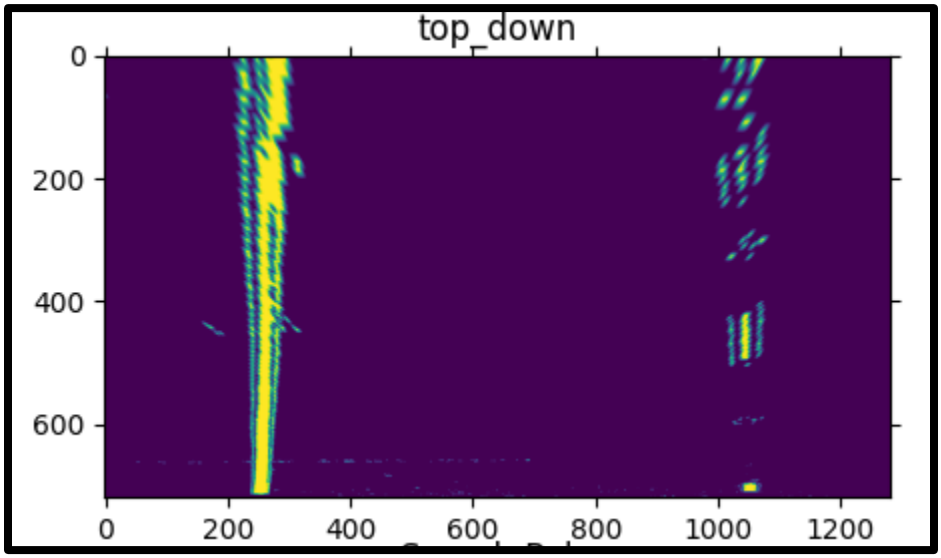

Figure 7: Perspective Transform on thresholded test_images/straight_lines1.jpg file

4. Lane-Line Pixel Identification and Polynomial Fitting

I use two routines to detect lanes in this project, search_windows(), line 201 and search_poly(), line 147. Search_windows is performed initially and upon every failure to detect a minimum of 5,000 lane pixels. It uses auto-centering rectangles cascading from the bottom of the image too the top to gather lane-line pixels. Search_poly uses the previously calculated polynomial and searches within a pre-determined margin around it to gather lane-line pixels. The outputs of both these functions are a collection of left and right lane pixels that are then passed to the line objects, rline and lline, for polynomial fitting. In the class method Line.set_current_fit(), line 392, the collection of pixels are

inputs to the numpy.polyfit() function to yield second order polynomial coefficients. The image below shows a result of these functions at work. The red lines are the plotted result.

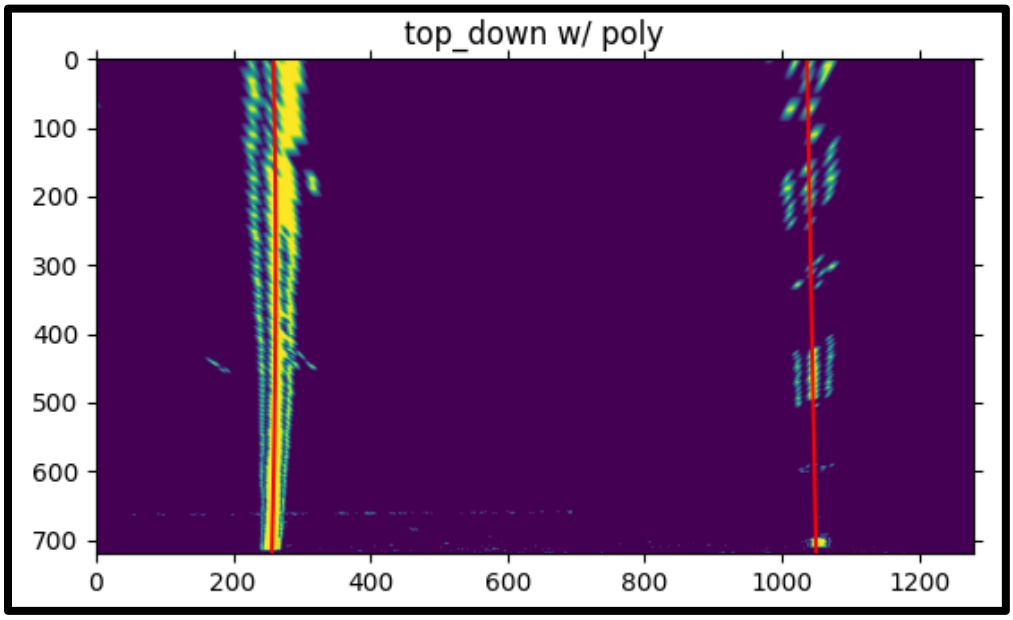

Figure 8: Pixel Detection and Line Fitting (subplot 5)

5. Lane Curvature

Lines 279 to 299 is where my curvature function is defined. I had to adjust the scaling factor (pixels to meters) according to my unique perspective transformed. The center of the vehicle is calculated in lines 291 and 292 by taking the midpoint of the bottom of the two lane polynomials (road center), the midpoint of the image (vehicle center), and differencing them. Then the difference is scaled to give us our real-world measurement of alignment to the center of the road.

```
# subtract image midpoint from polynomials' midpoint for distance from center
   delta_center = ((rightx[-1]+leftx[-1])/2) - (1280/2)
   delta_center = delta_center*xm_per_pix
```

6. Result

Figure 9 is a final result of the pipeline, where the polynomials calculated from the lane-line pixels are used to bound a green highlight on the road. This undoing of the prior perspective transform is performed in the function unwarp() located on line 302. The inverse transform matrix was returned from the prior warp() function and is passed this time to the cv2.warpPerspective() function. Our 'normal' perspective is now our destination and our birds-eye view is our source. The transform is made with a blank image having only the polynomials and the polygon shading between them. Then it is lightly laid over a bare 'undistored' image using the cv2.addWeighted() function.

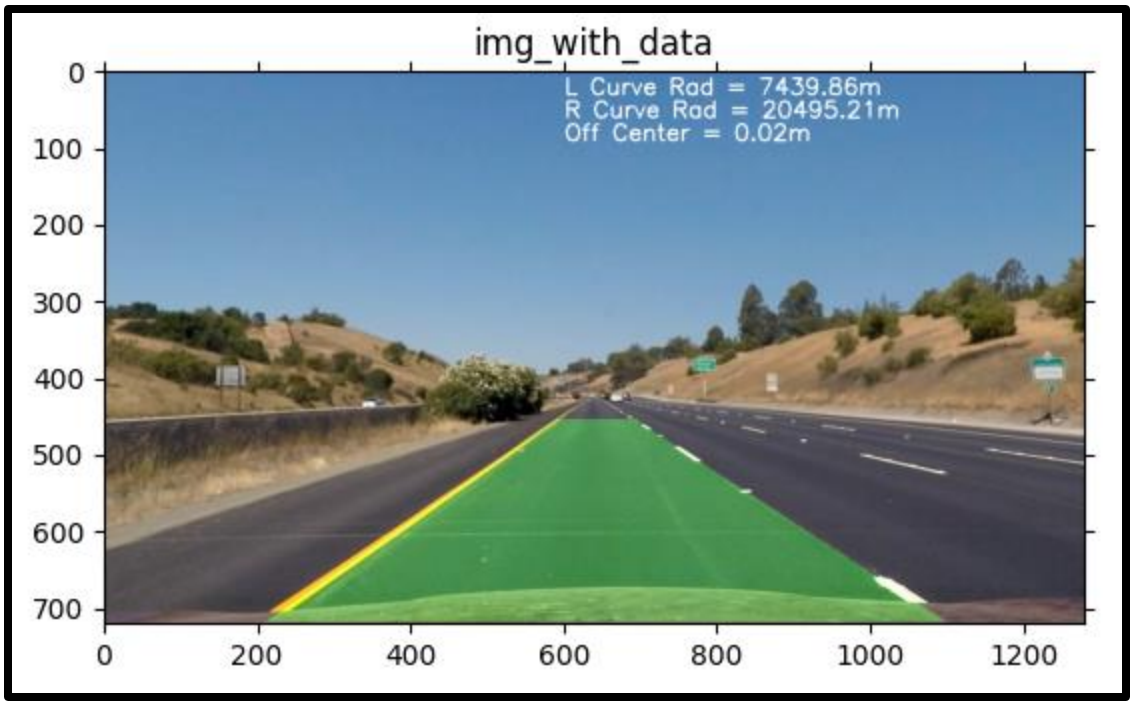

Figure 9: Pipeline Result

Pipeline (video)

The pipeline video is located in output_images/project_video_Output.mp4. There is also in the same directory project_video_outputSubplots.mp4 outlining the pipeline's progression. The same subplotting is used on test_images/test1.jpg and has had its plots referenced several times in this report.

Discussion

I found it most difficult to tune the threshold margins to create the binary images. I even went as far as to try some adaptive thresholding techniques throughout the completion of this project. I found an referenced article in the Udacity help forums but was not able to get the technique to work for my purpose. This is the link: https://scikit-image.org/docs/dev/auto_examples/segmentation/plot_niblack_sauvola.html . It is a great concept of adjust the threshold values to the immediate area around each pixel and not using a constant one throughout the entire image. However, it looks like the Sauvola thresholding is more commonly used on text and I was able to just tune my thresholds a bit better. I suspect my pipeline would break is drastically dark or bright conditions, where the saturation channel could not longer provide the data needed to detect lane pixels. It would also fail in very curvy streets as second order polynomials will not curve as must as we would need them to. One way to improve this project would be to break up the segments of detection. One segment for the area closer to the vehicle and another segment (or segments) for the area(s) further away. I also considered changing the shape of the mask I applied to the image. It could be very useful to have an image mask that adapts in a similar way that the search_poly() function adapts its search area. Thank you for the involved project.