Udacity SDC – P4: Advanced Lane Finding Write-up Report

By: James Beasley

In this report, I will consider the rubric points individually and describe how I addressed each point in my implementation.

Submission Contents & Execution

My project includes the following directories/files:

- **camera_cal** directory containing Udacity-provided camera calibration images
- **output_images** directory containing example images of each stage of the pipeline
- **output_video** directory containing final output video: *processed_project_video.mp4*
- **test_images** directory containing Udacity-provided pipeline test images (and a few I exported from the project video)
- **test_video** directory containing Udacity-provided project video: *project_video.mp4*
- **calibration_processor.py** containing calibration and distortion correction code
- **lane_processor.py** containing lane detection, mapping, and fitting code
- **main.py** containing initialization, test (export example stage images), and production (export processed video) pipeline execution code
- **perspective_processor.py** containing lane warping/unwarping code
- **production_pipeline.py** containing code that processes each frame of the project video through my computer vision pipeline and produces a video saved to the output_video directory
- **threshold_processor.py** containing color and gradient thresholding code
- **test_pipeline.py** containing code that saves example images of each stage of the pipeline to the output_images directory
- **writeup_report.pdf** summarizing the results (this document)

To execute: first, clone the project from: https://github.com/embeddedcognition/sdcp4.git

Once cloned, examples images for each stage of my pipeline are auto-generated and saved in the **output_images** directory (using a straight-line test image) and a processed version of the provided project video is also auto-generated and saved in the **output_video** directory by executing:

```
python main.py
```

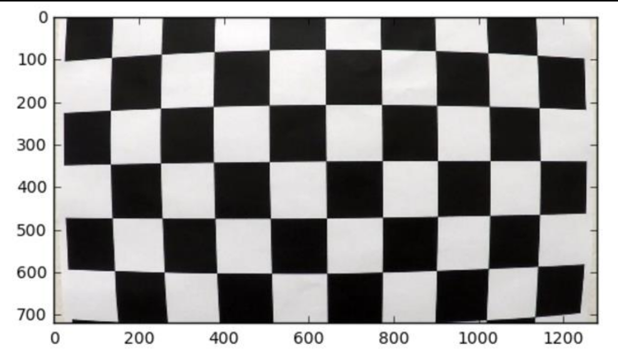
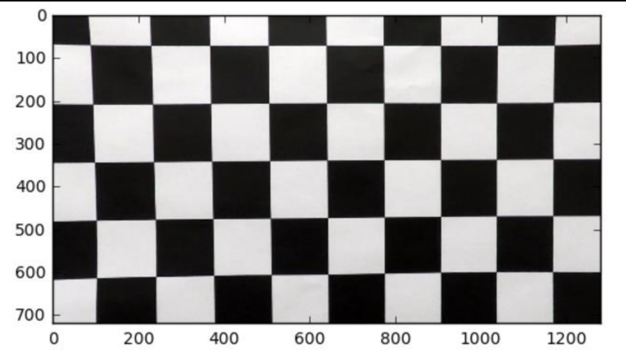
Camera Calibration (Stage 0 – Part 1)

Stage 0 of my pipeline tackles camera calibration and distortion correction. In this section, I'll be describing part 1 of that stage (camera calibration). All code related to this stage is in the **main.py** and **calibration_processor.py** files. You can also find example images for stage 0 (parts 1 & 2) in the **output_images** directory. Part 1 of this stage (camera calibration) is executed only once; all other stages (including stage 0 part 2) are executed per-frame.

In **main.py** (lines 20–34), I set the chessboard dimensions (9x6) and the path to the calibration images (i.e., all files within the **camera_cal** directory). I then make a call to the *generate_calibration_components* function located in the **calibration_processor.py** file (lines 15–44), supplying the chessboard dimensions, the wildcard path to the calibration images, and the camera image size as parameters. This function first loads the chessboard calibration images from the **camera_cal** directory (one at a time), converts each to grayscale, then feeds each to the *findChessboardCorners* function. If the appropriate number of corners are found for a calibration image (17 of the 20 images fit this criteria), they're added to the image points list (list of 2d points on the image plane). To map these image points to the real world (3d space), a template of the object points (lines 21–23) is also added to the object points list for that calibration image (these points are the same for every calibration image). After all image/object points are gathered, the final step is to generate the camera matrix and the distortion coefficients necessary to undistort images taken with the camera that produced the project video. This is accomplished with a call to the *calibrateCamera* function.

Once the calibration components are returned (i.e., camera matrix and distortion coefficients), they are then supplied to the *execute_test_pipeline* (auto-generating example images for each pipeline stage using a straight-line test image) and *execute_production_pipeline* (process frames from the project video through my computer vision pipeline and output a processed video that identifies the lane and prints curvature and vehicle offset values) functions to use in undistorting test images and video frames respectively (lines 77 & 84 of the **main.py** file).

An example undistorted chessboard ("camera_cal/calibration1.jpg") is depicted below:

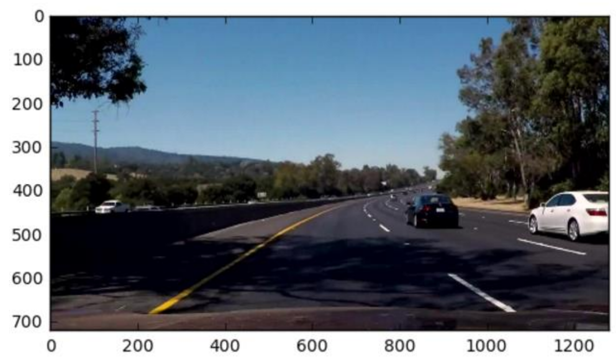
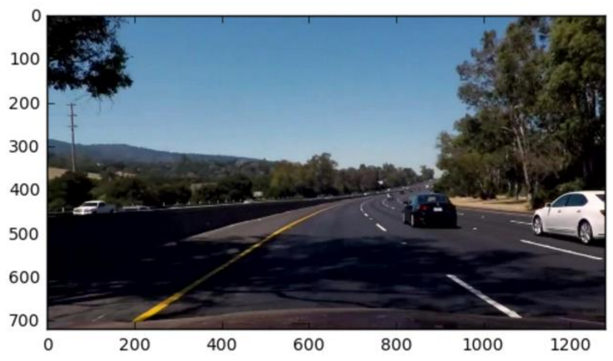
Original	Undistorted
	

Pipeline (description of the stages)

My submission includes two pipelines: The first is the test pipeline (**test_pipeline.py**) which saves example images of each stage (0-part-2 through 4) to the **output_images** directory. This gives you an inside view into how the images are transformed at each stage. The second is the production pipeline (**production_pipeline.py**) which processes each frame of the Udacity-provided project video (*project_video.mp4*) through each stage (0-part-2 through 4) and saves a processed video (processed*_project_video.mp4*) to the **output_video** directory. The following sub-sections will describe the pipeline stages and how each works.

1. Distortion Correction (Stage 0 – Part 2)

Stage 0 part 2 of my pipeline tackles distortion correction of images. For the test and production pipelines to undistort images, they call the *perform_undistort* function in the **calibration_processor.py** file (lines 46–50). This function takes the distorted image and the calibration components (i.e., camera matrix and distortion coefficients) as parameters and returns an undistorted image. An example test image ("test_images/test7.jpg") depicts the comparison between distorted and undistorted below:

Original	Undistorted
	

2. Perspective Transform (Stage 1)

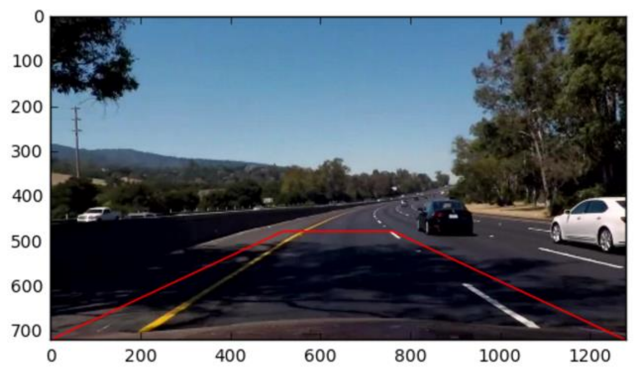
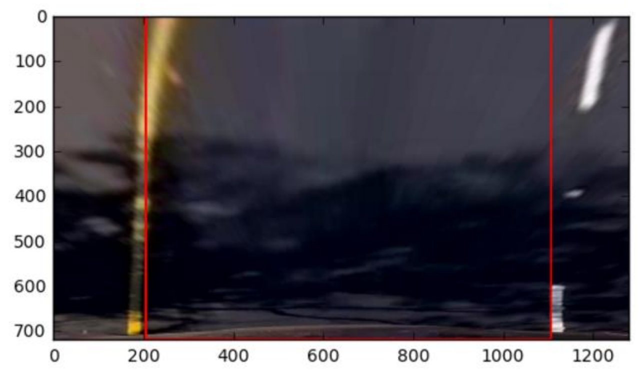
Stage 1 of my pipeline tackles perspective transform of an undistorted image. All code related to this stage is in the **main.py** and **perspective_processor.py** files. You can also find example images for this stage in the **output_images** directory. Lines 40–70 of the **main.py** file handle the one-time initialization of the warp and unwarp perspective matrices used by the pipeline to transform from depth-of-field to bird's-eye view and vice-versa (via a call to the *generate_perspective_transform_components* function, lines 12–19 of the **perspective_processor.py** file).

For the test and production pipelines to transform perspective, they call the *perform_prespective_transform* function (lines 21–24 of the **perspective_processor.py** file), supplying the undistorted image and the appropriate perspective matrix (for warp or unwarp) created during initialization.

Regarding point selection, I experimented with the method outlined in the lesson (tightknit masking around the lane), but found when I got to the thresholding step, I had lots of artifacts that increased the complexity of the lane detection task. After further experimentation, I landed on setting up destination points that let me zoom in on the road.

My **final** points are listed below as well as an example of the transformation:

Source Points	Destination Points
(517, 478)	**(0, 0)**
(762, 478)	**(1280, 0)**
(0, 720)	**(0, 720)**
(1280, 720)	**(1280, 720)**

Undistorted w/ source points (in red)	Warped w/ alignment lines (in red)
	

In the above warped image, the red lines don't represent my destination points; they are just straight lines I drew on the image to understand whether the perspective transform was correct. The destination points I used were the 4 corners of the image frame (allowing me to zoom in).

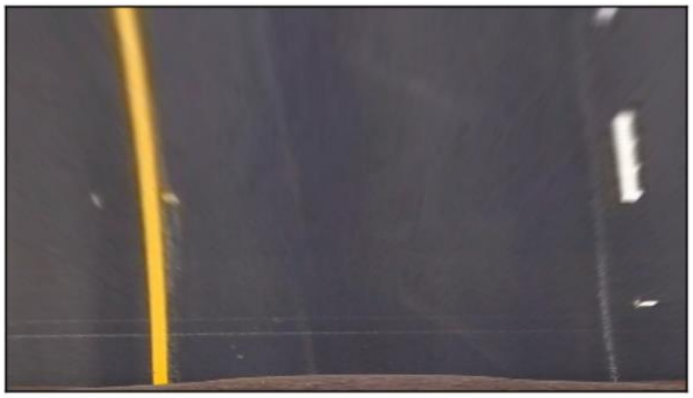
The above depicts a curved road segment (to show you what it looks like), but the **stage 1** images in the **output_images** folder depict a straight road and verify the correctness of my perspective transform.

3. Thresholding (Stage 2)

Stage 2 of my pipeline tackles thresholding an undistorted, perspective transformed image. All code related to this stage is in the **threshold_processor.py** file. You can also find example images for this stage in the **output_images** directory. For the test and production pipelines to threshold and produce binary images, they call the *perform_thresholding* function (lines 89–119 of the **threshold_processor.py** file), supplying the undistorted and perspective transformed image.

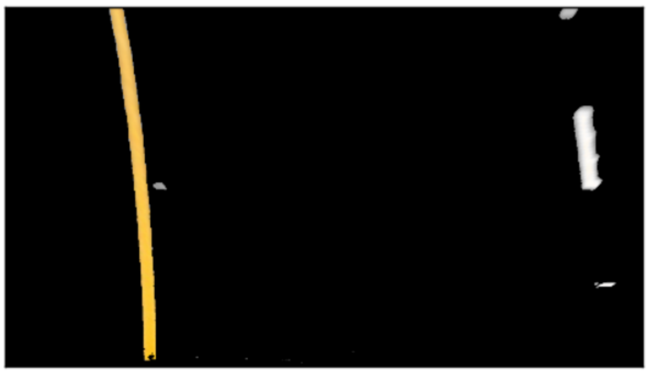
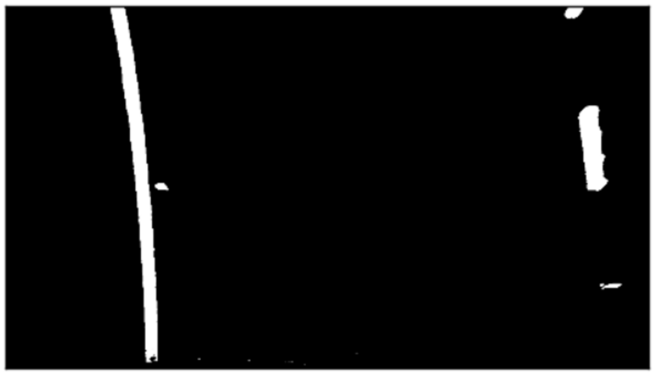
To demonstrate my thresholding process, I'll be using an example test image ("test_images/test2.jpg") from my submission. It was a conscious choice to threshold after perspective transform vs. before, as I found it to be more effective for the thresholding techniques I chose to implement (more on this later).

The warped image is as follows:



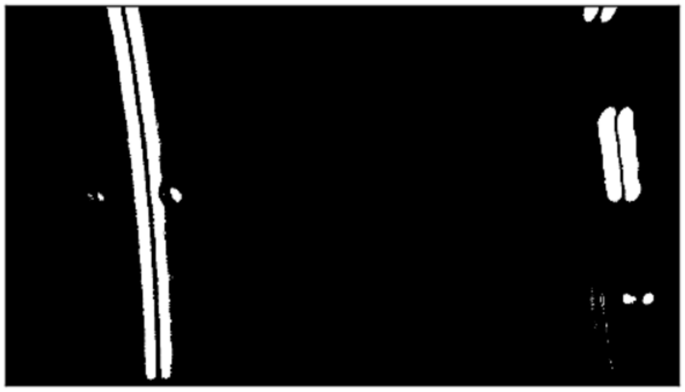
My thresholding process is comprised of 3 different sub-processes (all taking place after a conversion of the image to the HLS color space).

The <u>first sub-process</u> attempts to identify yellow and white lines via value thresholding of the H, L, and S channels through a call to the *apply_hls_channel_color_thresholding* function (lines 41–70 of **threshold_processor.py**).

HLS color thresholding	Converted to binary
	

The <u>second sub-process</u> leverages Gaussian blurring and x-axis gradient thresholding of the L-channel via a call to the *apply_l_channel_gradient_thresholding* function (lines 72–77 of **threshold_processor.py**).

Example thresholded L-channel:

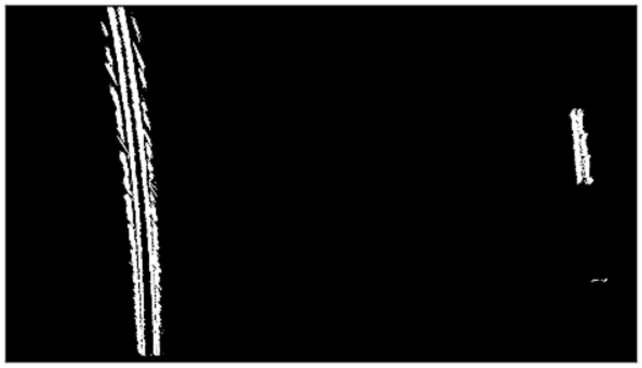
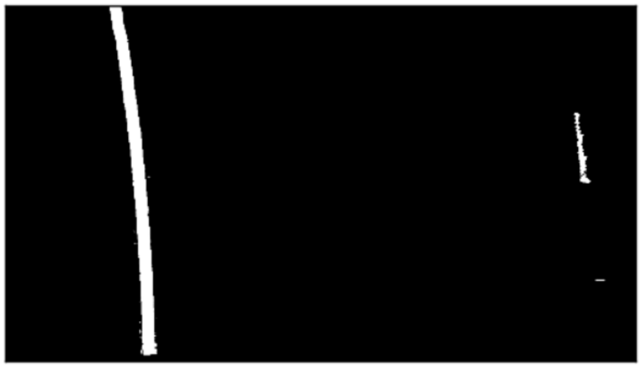


The reason I decided to threshold after warping was the following:

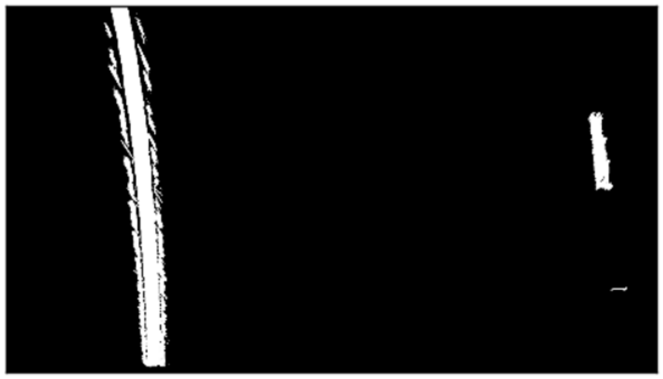
Through experimentation, I found that blurring had a large influence on the effectiveness of difference filtering of the L-channel (or similarly the R-channel in RGB color space). In fact, the larger I made the blurring kernel, the better I could identify the line using the difference filter, and, the better I could reduce noise by setting the low threshold of the difference filter to a higher value.

I also found that this technique was most effective if the blurring kernel size and low threshold of the difference filter where close in value. However, this tactic only worked if thresholding was applied after perspective transform, not before (thresholding was way too aggressive for an unwarped image). I found this tactic made the L channel much more resilient vs. normal value or small kernel gradient thresholding (i.e., kernel values of 3 or 5).

The third sub-process entails separate x-axis gradient and value thresholding of the S-channel via a call to the *apply_s_channel_gradient_and_value_thresholding* function (lines 79–87 of **threshold_processor.py**). This was a tactic I took from one of the lesson quizzes and I think it worked well.

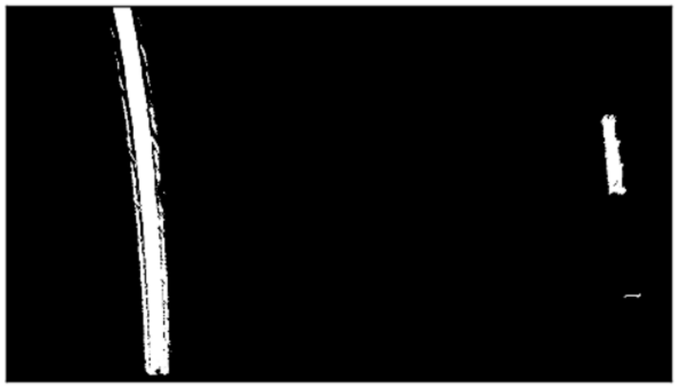
HLS – S-channel x-axis gradient threshold	HLS – S-channel value threshold
	

Final combined S-channel (via bitwise OR):



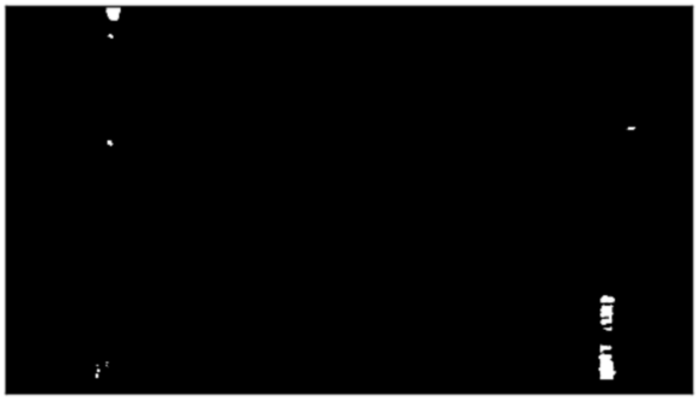
These three sub-processes are then combined to produce a final binary image for use in lane detection.

For the final thresholded image, I found the most resilient combination to be when I performed a bitwise OR of the HLS binary and L-channel binary channels, then followed that with a bitwise AND of the combined S-Channel, producing the following:

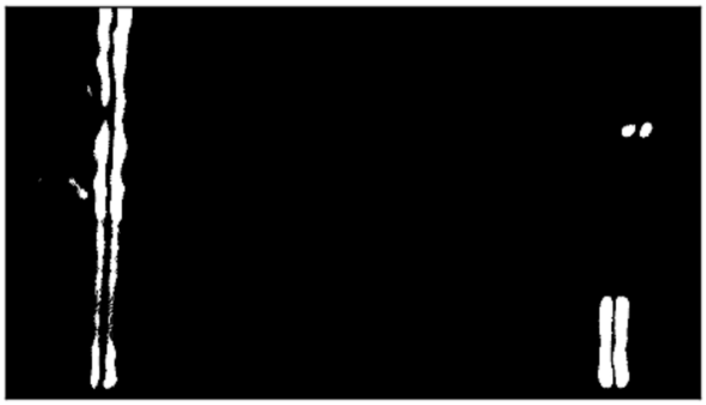


To get better intuition on how my threshold process performed on difficult images, I exported a shadow-rich frame (*test_images/test7.jpg*) from the supplied project video, ran it through my pipeline, and found that it exposed a big flaw:

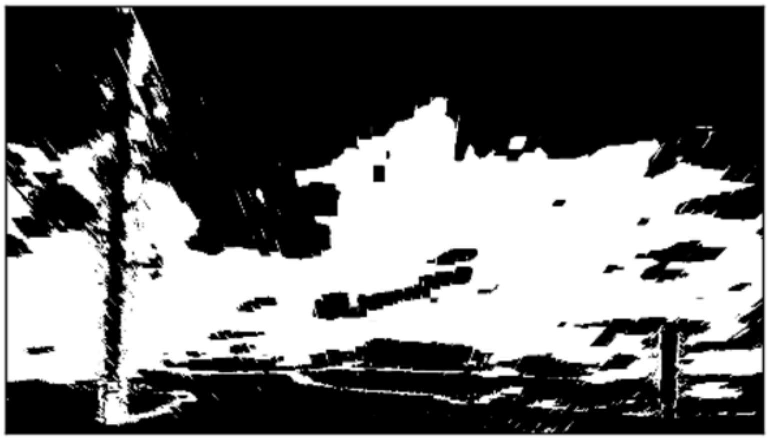
You see how the first sub-process (on the H, L, and S channels) does a poor job of identifying the left line especially:



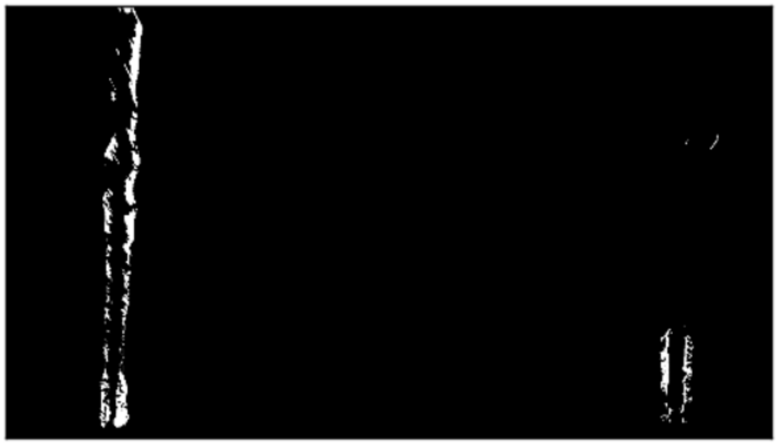
The second sub-process (on the L-channel) does a much better job of identifying both lines:



The third sub-process (on the S-channel) does a poor job of identifying both lines and is full of noise/clouding:

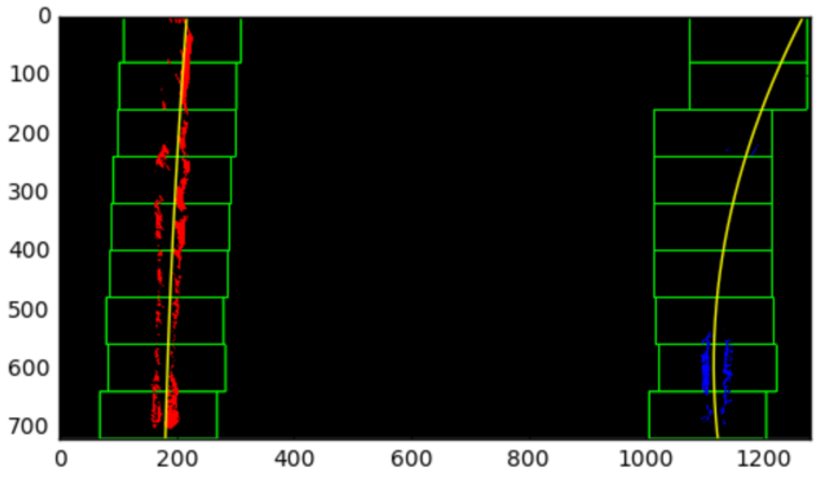


And then the final binary result leaves much to be desired:

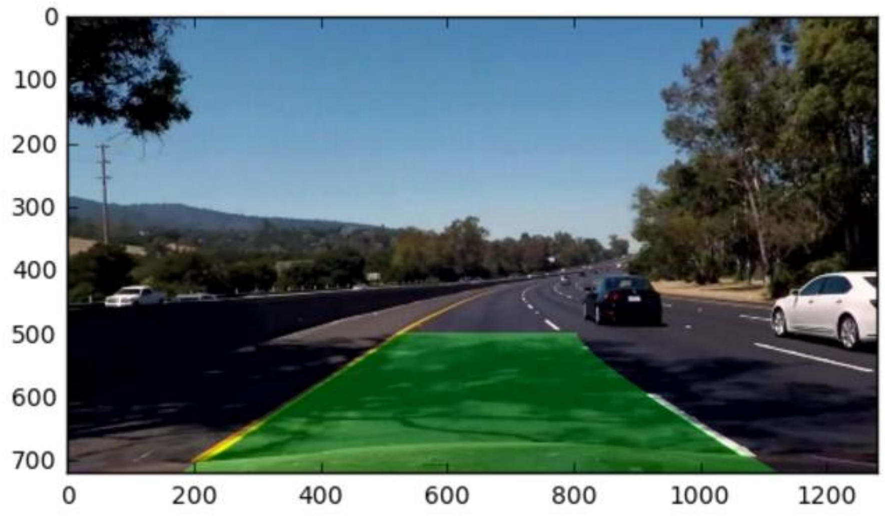


Though the left lane is good enough, the right doesn't give much direction on the top half of the lane.

Fitting polynomials confirms the problem:



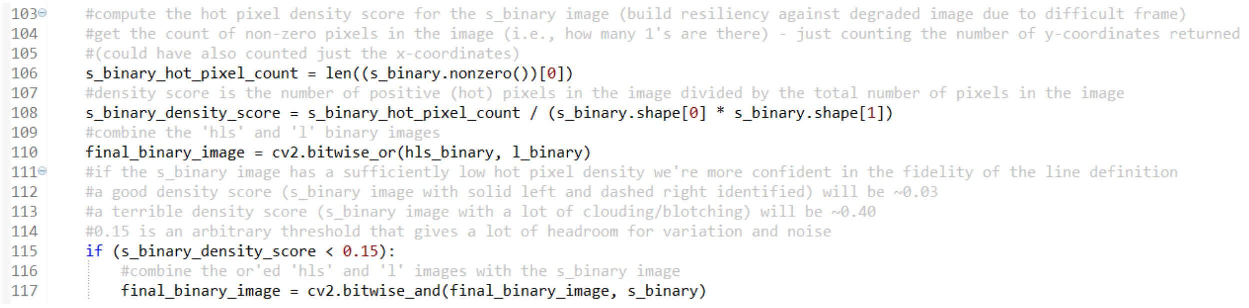
The computed right line is definitely incorrect after seeing the projected lane on the undistorted image:



To combat this issue, I decided to build additional intelligence into my thresholding process to ensure the final image wasn't degraded by bad S-channel information (based on hard-to-read conditions).

The algorithm I decided to employ was to compute the % of hot (value of 1) pixels in the resulting binary thresholded S-channel image, and if it exceeded a threshold, throw the S-channel out completely. Thereby removing result degradation in the final binary image.

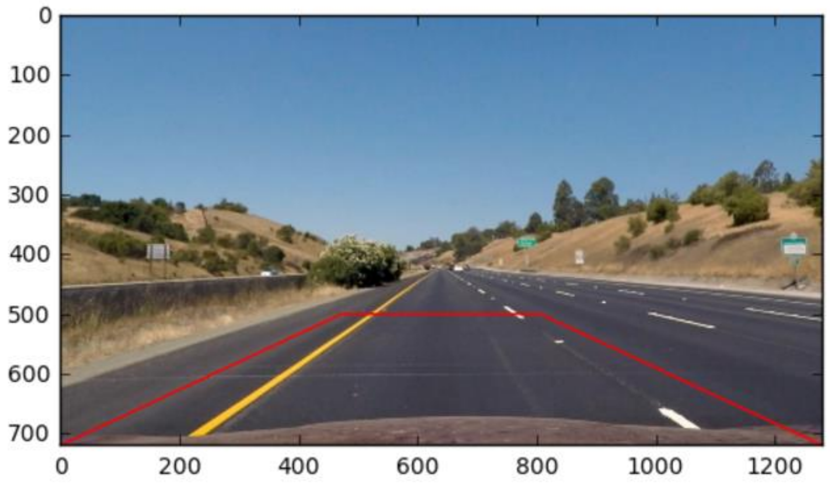
I called this the binary S-channel's *density score* (lines 103–117 in **threshold_processor.py**):

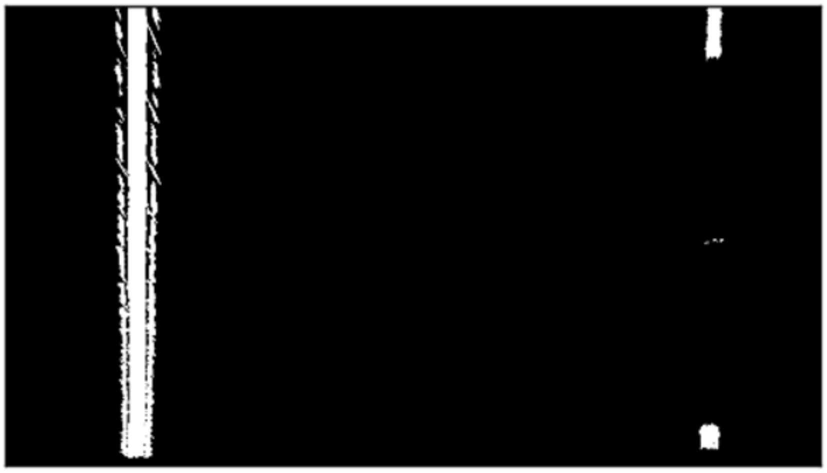


```
103    #compute the hot pixel density score for the s_binary image (build resiliency against degraded image due to difficult frame)
104    #get the count of non-zero pixels in the image (i.e., how many 1's are there) - just counting the number of y-coordinates returned
105    #(could have also counted just the x-coordinates)
106    s_binary_hot_pixel_count = len((s_binary.nonzero())[0])
107    #density score is the number of positive (hot) pixels in the image divided by the total number of pixels in the image
108    s_binary_density_score = s_binary_hot_pixel_count / (s_binary.shape[0] * s_binary.shape[1])
109    #combine the 'hls' and 'l' binary images
110    final_binary_image = cv2.bitwise_or(hls_binary, l_binary)
111    #if the s_binary image has a sufficiently low hot pixel density we're more confident in the fidelity of the line definition
112    #a good density score (s_binary image with solid left and dashed right identified) will be ~0.03
113    #a terrible density score (s_binary image with a lot of clouding/blotching) will be ~0.40
114    #0.15 is an arbitrary threshold that gives a lot of headroom for variation and noise
115    if (s_binary_density_score < 0.15):
116        #combine the or'ed 'hls' and 'l' images with the s_binary image
117        final_binary_image = cv2.bitwise_and(final_binary_image, s_binary)
```

The above S-channel image has a density score of: **0.4**

When I compared this to an input image where I'd expect to have a well-defined S-channel (such as a bright straight-lined road), I saw a large difference:

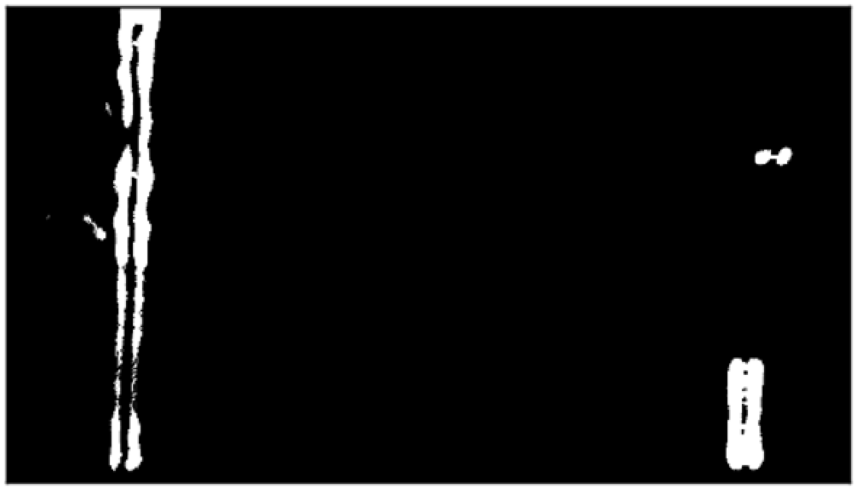




The straight-line “easy” road had a density score of: **0.03**

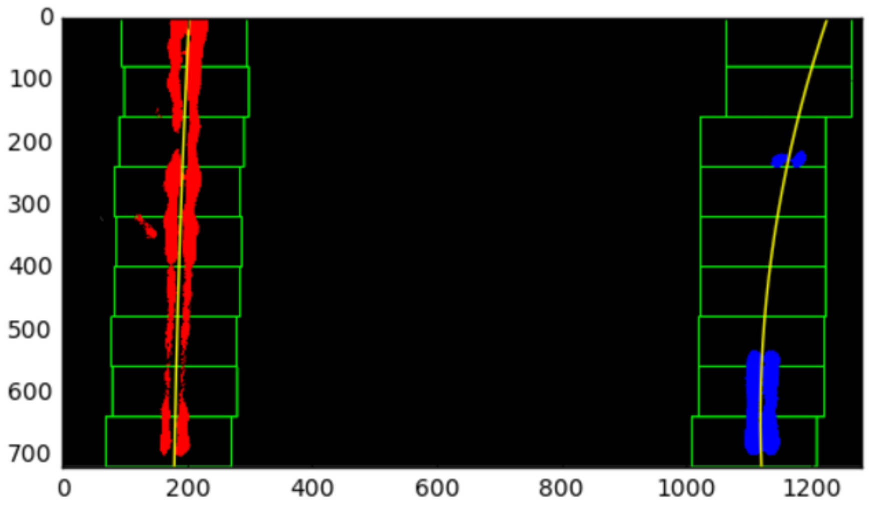
With that in mind, I decided to set my threshold at **0.15** (adding some head-room for variation and noise), but this is a tunable parameter for sure.

When this logic is applied back to the frame in question, the final result is:



As it didn’t include the degraded s-channel…

You also see better performance from the polynomial fit:



4. Lane Detection & Polynomial Fitting (Stage 3 – Part 1)

Stage 3 part 1 of my pipeline tackles lane detection and polynomial fitting using the supplied thresholded binary image. All code related to this stage is in the **production_pipeline.py** (lines 75–115) and **lane_processor.py** files. You can also find example images for this stage in the **output_images** directory.

For the test and production pipelines to detect lane line pixels and fit their positions with polynomials there are a few steps to go through:

First, I decided to keep two queues with the last 10 sets of polynomial coefficients (this also aids in smoothing video playback later) for the left and right lane lines (lines 36–38 of the **production_pipeline.py** file). Now, starting at line 75 of that same file, we next need to determine if we have previous coefficients stored:

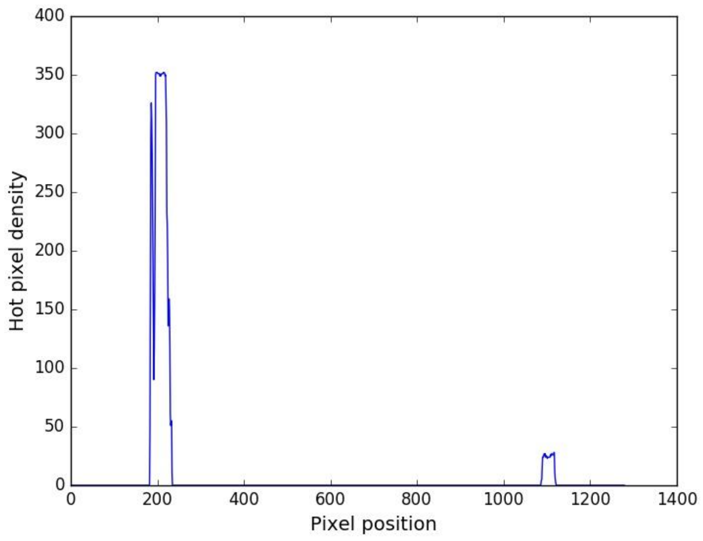
> If not, we'll go about executing a blind search for the lane lines by calling the *perform_blind_lane_line_pixel_search* function (lines 137–223 in the **lane_processor.py** file). I based this approach on the code provided in the lesson and it worked well, but had a high compute cost, so it's only executed on the first frame.
>
> If we do, we'll go about executing an educated search for the lane lines by calling the *perform_educated_lane_line_pixel_search* function (lines 84–135 in the **lane_processor.py** file). I also based this approach on the code provided in the lesson and it worked well. This approach accelerates the search by using the previously fitted polynomial as a starting place to mount the search for the lane line pixels in the next frame. This approach is used on every frame after the first.

After the set of coordinates for the left and right lane line pixels are returned by either method above, I go about computing the left and right lane polynomial coefficients (i.e., fit the polynomials) via a call to the *compute_lane_line_coefficients* function (lines 43–51 in the **lane_processor.py** file).

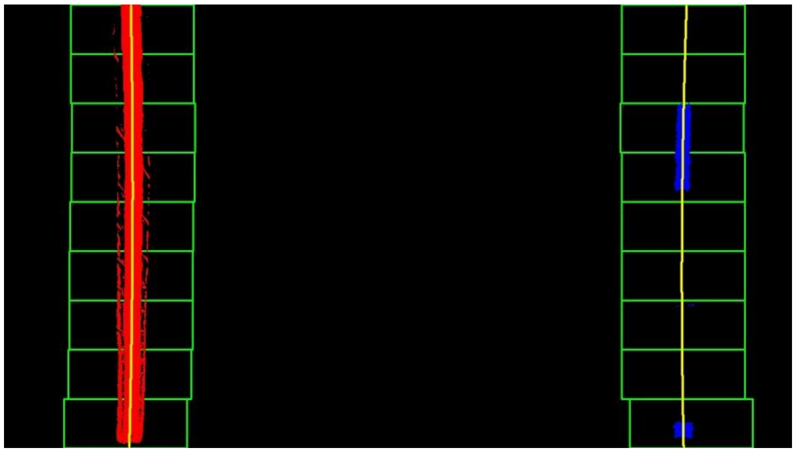
If on a current frame, I come up with left or right lane line coefficients that are different (by greater than 3%) from the coefficients in the previous frame, I drop the coefficients for the current frame and use the ones from the last frame (lines 89–103 in the **production_pipeline.py** file). After adding the chosen coefficients to the queues, I take the mean of all the sets of coefficients in each queue and use the result as my final coefficient values for the second order polynomial equation for each line (lines 105–115 in the **production_pipeline.py** file).

Below are some example images from the sub-stages in this stage:

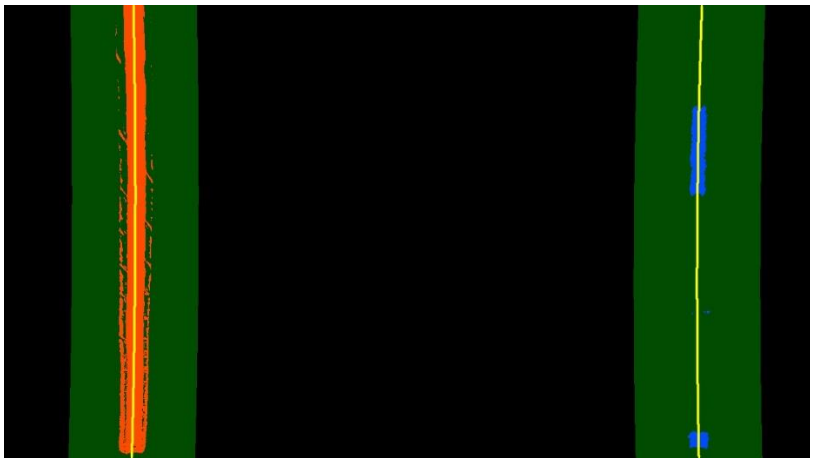
Determining the likely location of the base of the left and right lane lines (lines 15–41 of the **lane_processor.py** file):



Example output of the blind search + polynomial fitting (first time only):



Example output of the educated search + polynomial fitting (for all frames after the first):



5. Lane Curvature & Vehicle Offset (Stage 3 – Part 2)

Stage 3 part 2 of my pipeline tackles lane curvature and vehicle offset/position. All code related to this stage is in the **production_pipeline.py** (lines 117–121) and **lane_processor.py** files. You can also find example images for this stage in the **output_images** directory.

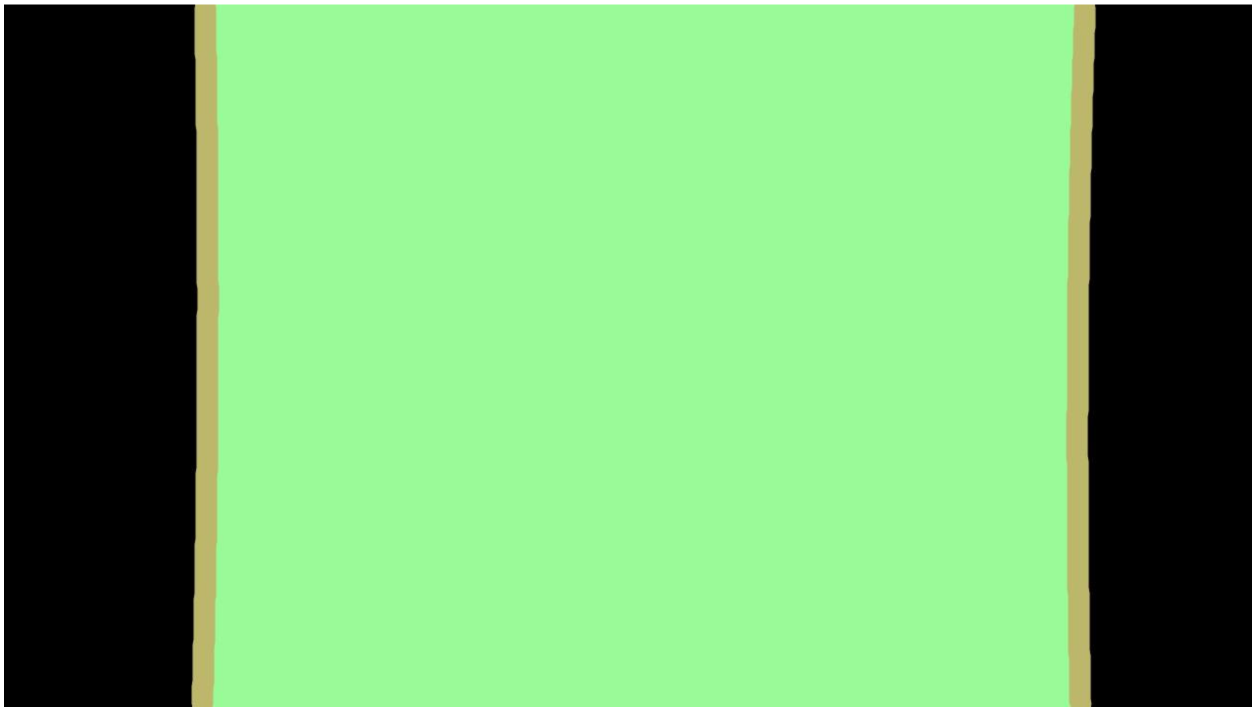
To compute curvature (lines 68–82 of the **lane_processor.py** file), I followed the method detailed in the lesson (with a few tweaks); essentially rescaling the polynomial from pixel space to real world space (using predefined meters-per-pixel scaling factors). I also checked my values on the first left turn of the project video to ensure they stayed in the ~1km range. I computed the curvature for the left and the right lines and then took the mean of both to come up with the curvature value displayed in my final video.

For vehicle offset (lines 53–66 of the **lane_processor.py** file), I also followed the thought process outlined in the lessons by finding the base of each calculated polynomial, the center of the lane based on those values, then comparing that against the center of the image (with a scaling factor as discussed before).

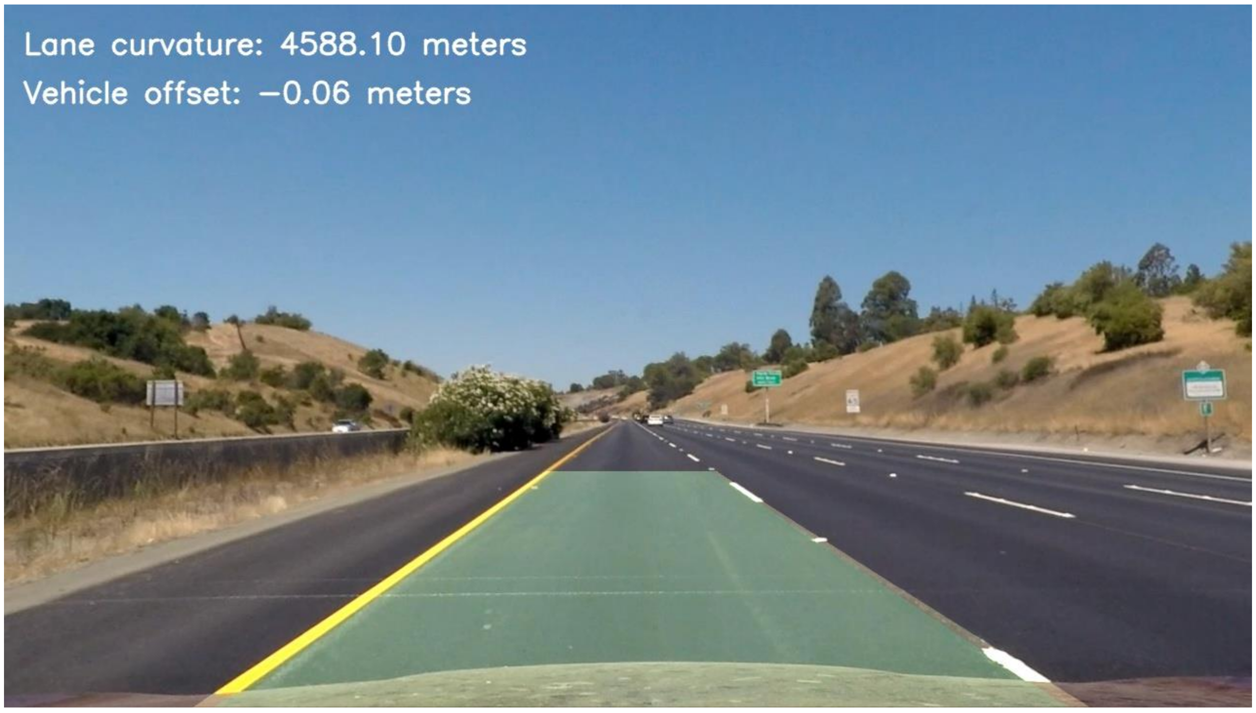
5. Resulting Projected Lane (Stage 4)

Stage 4 of my pipeline tackles projecting the computed lane back on an undistorted/unwarped image. All code related to this stage is in the **production_pipeline.py** (lines 127–151). You can also find example images for this stage in the **output_images** directory. In this stage we're drawing the lane (via a call to the *fillPoly* function) based on the computed polynomials and then projecting it back on the undistorted image using the unwarp perspective matrix.

Example output of the final lane (warped):



Example output of the final lane (unwarped) + lane curvature and vehicle offset stats:



Pipeline (resulting video)

My completed video can be found at:
https://www.youtube.com/watch?v=MuNVkMFeJMEvq=hd720

Alternatively, it's located in the **output_video** folder of my submission, the file is named: **processed_project_video.mp4**

Discussion

I really learned a lot during this project and feel like I was able to implement some interesting things (such as: implementing a density score for the S-channel, rejecting coefficients that had greater than 3% difference from the previous frame's set, smoothing video playback by calculating the mean of the coefficient sets in the queues, building a fully automated test and production pipeline, and building an effective thresholding process).

The biggest hurdle I faced on this project was deciding on the tapestry of thresholding techniques to employ, as there are lots of combinations possible. I evaluated so many variations that I would sometimes find myself "overfitting" a specific image with a particular set of techniques that would then break down on another image. Once I built a good intuition on the strengths and weaknesses of each color space and their specific channels, I was able to keep it simple, but also compensate for edge cases (like when the S-channel breaks down).

I think there is definitely room for improvement in my pipeline, especially in the areas of confidence level in the lanes found and thresholding. It would have been nice to explore additional techniques to build in better-intelligence/resilience into those processes, especially in the situations posed in the challenge videos. I haven't run my pipeline on anything but the project video, but I would expect there to be lane identification issues given that the second challenge video (HOV lanes) has lots of vertical variation in the pavement that could be misconstrued as lane lines by my current implementation. As stated before, additional time spent on thresholding techniques to reject vertical features that don't meet the criteria of a lane line (or dashed line) would be appropriate as well as better intuition on how to isolate lane lines based on visual characteristics vs. just density of pixels.

I will definitely be revisiting this codebase in the future!