ADVANCED LANE FINDING

GOAL: Given a sample video showing a vehicle driving on a highway, detect and display the approximate lane position, and include information about the current radius of curvature of the road and the vehicle distance from center of the lane.

STEPS: In order to find the lane, here are the steps in the processing pipeline:

1) Calibrate the camera to find the calibration matrix and distortion coefficients.
2) Using the camera calibration, undistort the video frame.
3) Detect edges in each video frame using a combination of color and gradient thresholds to generate a binary image.
4) Create a "birds eye view" from the binary image of each video frame by transforming the image perspective.
5) Use a sliding window (or prior measurement) to find the right and left lane lines.
6) Fit each lane line to a polynomial to determine the road radius of curvature.
7) Determine the vehicle offset from center using the left and right lane lines averaged compared to the center of the image frame.
8) Merge the found lane lines back onto the original image frame and include the curvature and centering data.
9)

1. CAMERA CALIBRATION

In order to use each video frame to accurately calculate road curvature in the real world, we need to make sure the camera is calibrated. Every camera requires some form of calibration, to offset any image distortion caused by the lens of the camera when taking video/photos. For this project, we use a test sample of chessboard images and the Python CV2 `findChessboardCorners` and `calibrateCamera` functions library to determine the camera transformation matrix and the distortion coefficient. Because the calibration step is only required ONCE (every frame will use the same matrix and coefficient), I created a separate `CalibrateCamera` class to do the calibrations one time at the start of the pipeline, and then used the results for every video frame.

2. UNDISTORTING IMAGES

This is where our image pipeline really starts: undistorting the video frame. Using both the transformation matrix and distortion coefficient found in step 1, we run each image through the CV2 `undistort` function to generate an undistorted version:

```
undistorted_image = cv2.undistort(self.image, mtx, dist, None, mtx)
```

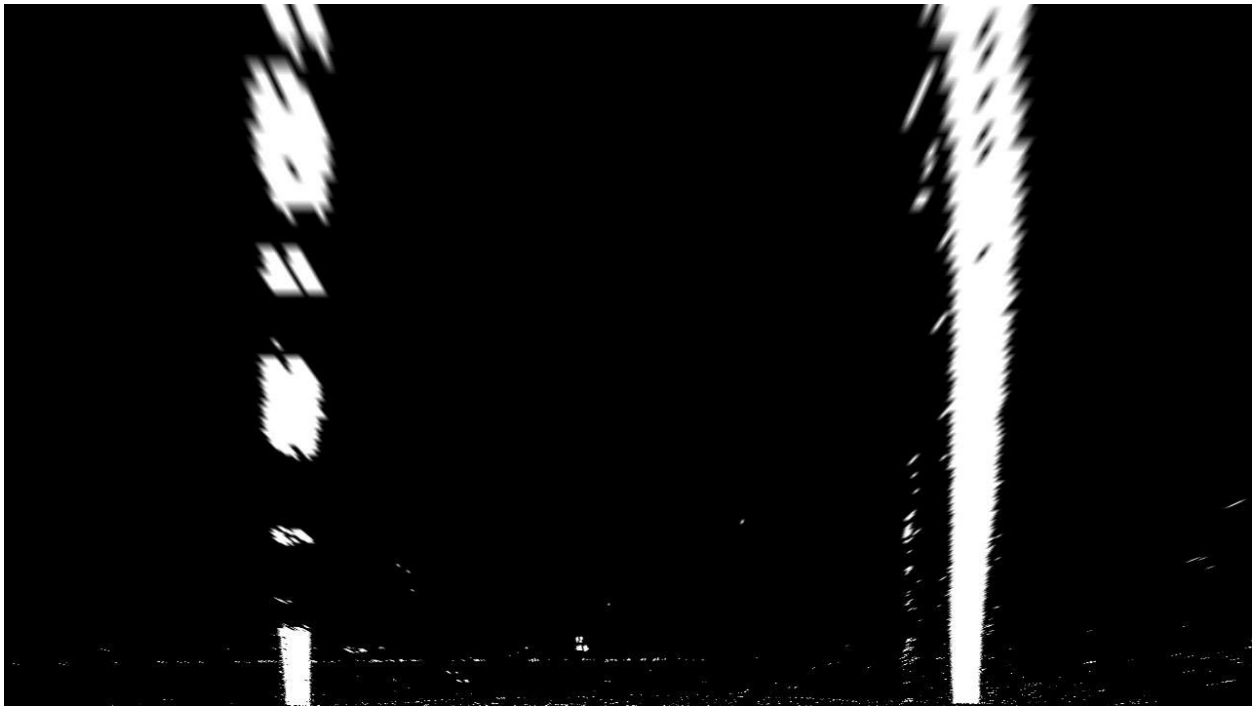
Where `mtx` is the calibration matrix and `dist` is the distortion coefficient determined in step 1 when we calibrated the camera.

3. DETECT EDGES

In order to find the lane lines, we convert the image to HLS colorspace and then apply both a color threshold and a Sobel X transformation threshold to the undistorted image created in step 2. Because the lane lines tend to be vertical, we use the Sobel X to find vertical lines from the image, using the "L" channel where the lanes are brightest. We combine this with a simple threshold on the "S" channel to create a combined image that gives us "the best of both worlds" helping to find images in shadows, etc. An example binary image looks like:

4. BIRDS EYE VIEW

In order to determine the road radius of curvature and to better track the lane lines for pixel detection, the binary image created in step 3 is perspective transformed into a "birds eye view" representation of the road. By choosing 4 source points from the original image and 4 destination points, the image is transformed to provide a truer sense of the road, as if looking directly from above (as opposed to the oblique angle seen in the camera). A sample birds-eye-image from the pipeline (from a sample *straight* portion of the road):



5. DETECT LANE LINES

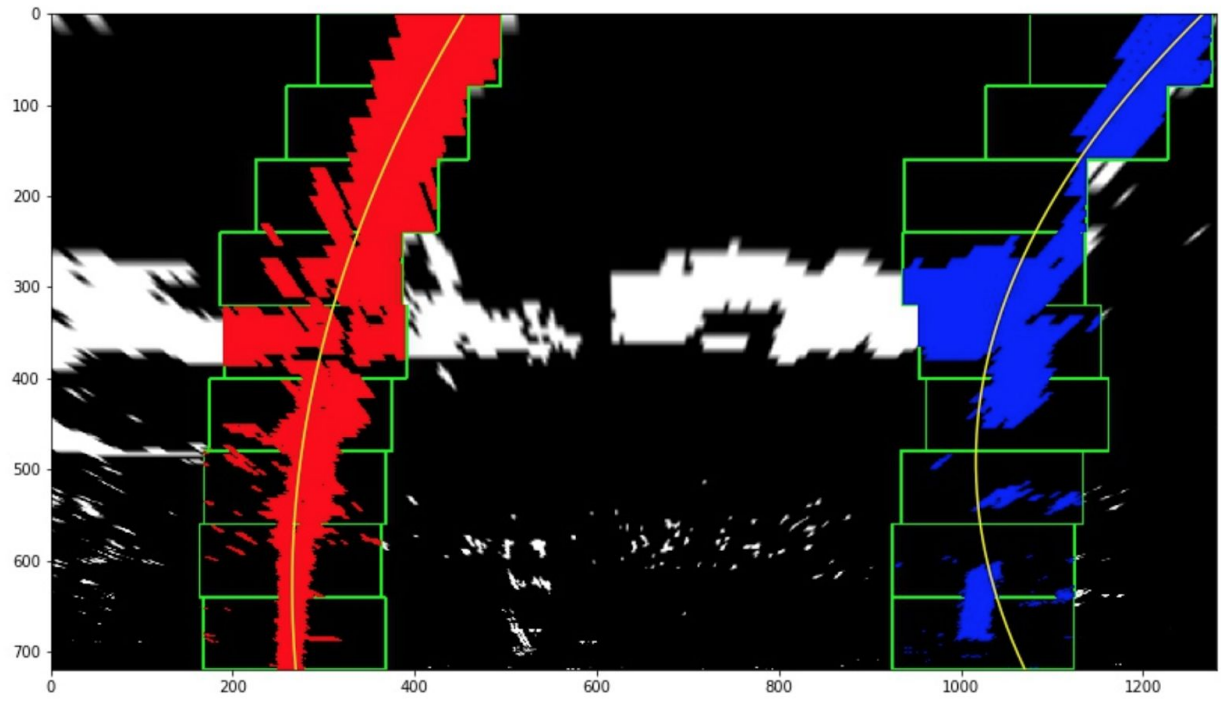
Using the bird's-eye perspective from step 4, we use a sliding window technique to trace the lane line pixels. We first create a histogram of the image to determine the most likely starting point at the bottom of the image for each left and right lane line. Using these starting points, we move upwards in steps looking at a small window each time and recalculating the lane line center at each point. We track the lane line to the top of the image and save the locations of each lane pixel that was found inside each window.

6. FIT POLYNOMIAL

Once we've found the lane line pixels, we can determine an approximate second order polynomial fit using the numpy `fitpoly` function and a generic second order equation of the form:

$$x = Ay^2 + By + C$$

The result is a "pretty good" approximation of the lane curvature. The resulting image shows the sliding window searches (in green), the left and right lanes (in red and blue) and the final polynomial fit for each lane line (in yellow):



7. CURVATURE AND CENTER

Using the polynomial fit from step 6, we can calculate the real-world curvature of the road in meters. By converting from "pixel space" and using the formula

$$curvature = mx/(my^2)*Ay^2 + (mx/my)*By + C$$

where `mx` and `my` are the pixel-to-meter conversions, we can calculate the curvature of the road in meters. And by using the polynomial fit from step 6, we can also determine the vehicle offset from the center of the lane by comparing the location of the center of the image to the approximate center of the lane line, using our same second order formula from step 6.

8. GENERATE FINAL OUTPUT IMAGE

Now that we have found the lane lines, we can project the lane back onto the original video frame using the inverse of the transform we used initially in step 2. We also take the location and curvature stats calculated in step 7 and draw them directly on each image frame. This final output image shows the detected lane in green (even through the shadows of the trees!):

PIPELINE

The final output video can be found in the "output_videos" folder for the project. The "output_images" folder also contains images from each step of the pipeline for each of the test images given.

DISCUSSION

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?

Although I'm super happy with the results for the "project_video", the results for both of the challenge videos leaves a bit to be desired. With more parameter tuning I could likely find better results. And although this project was relatively straight forward, it did take me *quite a bit longer* to complete than I thought it might initially. Each step of the pipeline required tuning and tweaking, and creating a clean, easy to understand and follow Python class to hold all of the functions took some doing. The main areas of improvement would be to create a new `Line` class to store information about each line for use in each subsequent video frame. My class includes the code to search for lane lines based on a prior result, but the actual "pipeline" doesn't implement this optimization. Other techniques could likely be integrated into this pipeline to improve the quality of the lane detection, like using other color channels to better search for lane lines. Overall I'm super impressed with this pipeline and had a blast working through it. Thank you, Udacity!