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.

Writeup / README

1. Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf. Here is a template writeup for this project you can use as a guide and a starting point.

You're reading it!

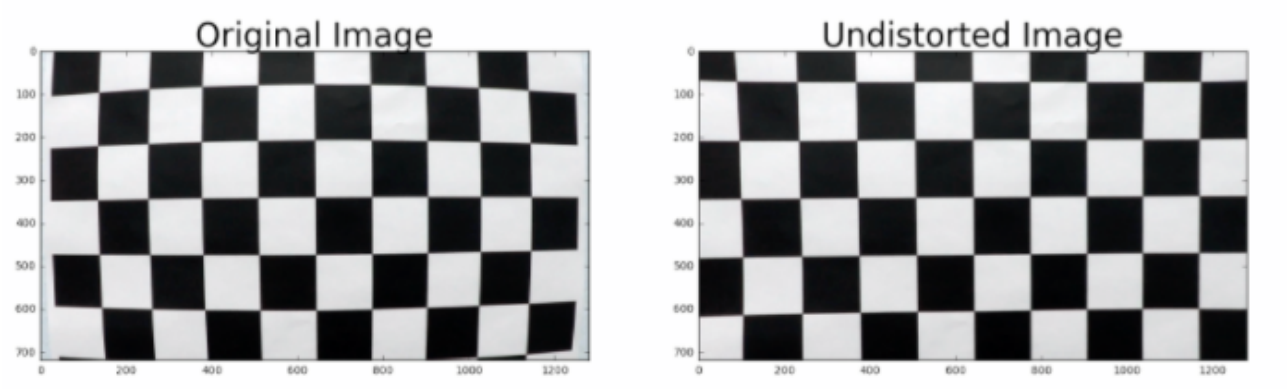
Camera Calibration

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 AdvanceLaneLines.ipynb notebook.

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

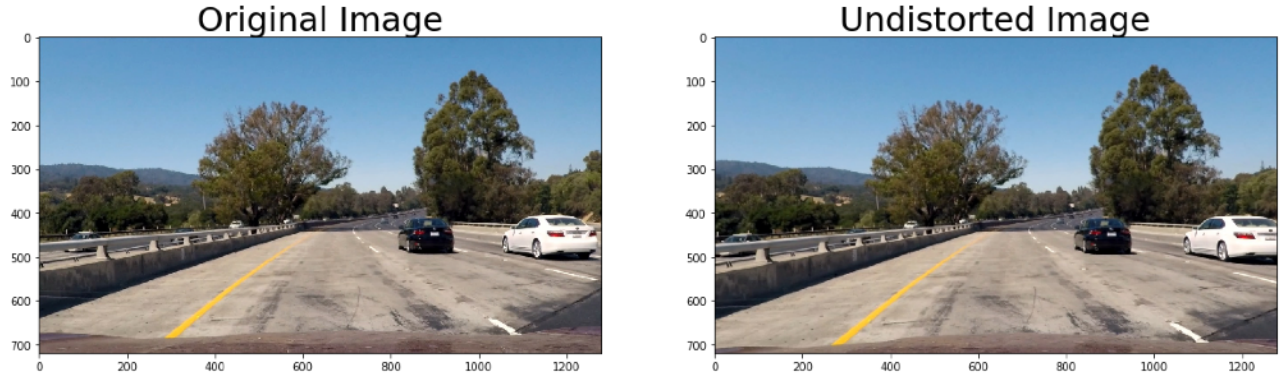
is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. Thus, `objp` is just a replicated array of coordinates, and `objpoints` will be appended with a copy of it every time I successfully detect all chessboard corners in a test image. `imgpoints` will 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. I applied this distortion correction to the test image using the `cv2.undistort()` function and obtained this:



Pipeline (single images)

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

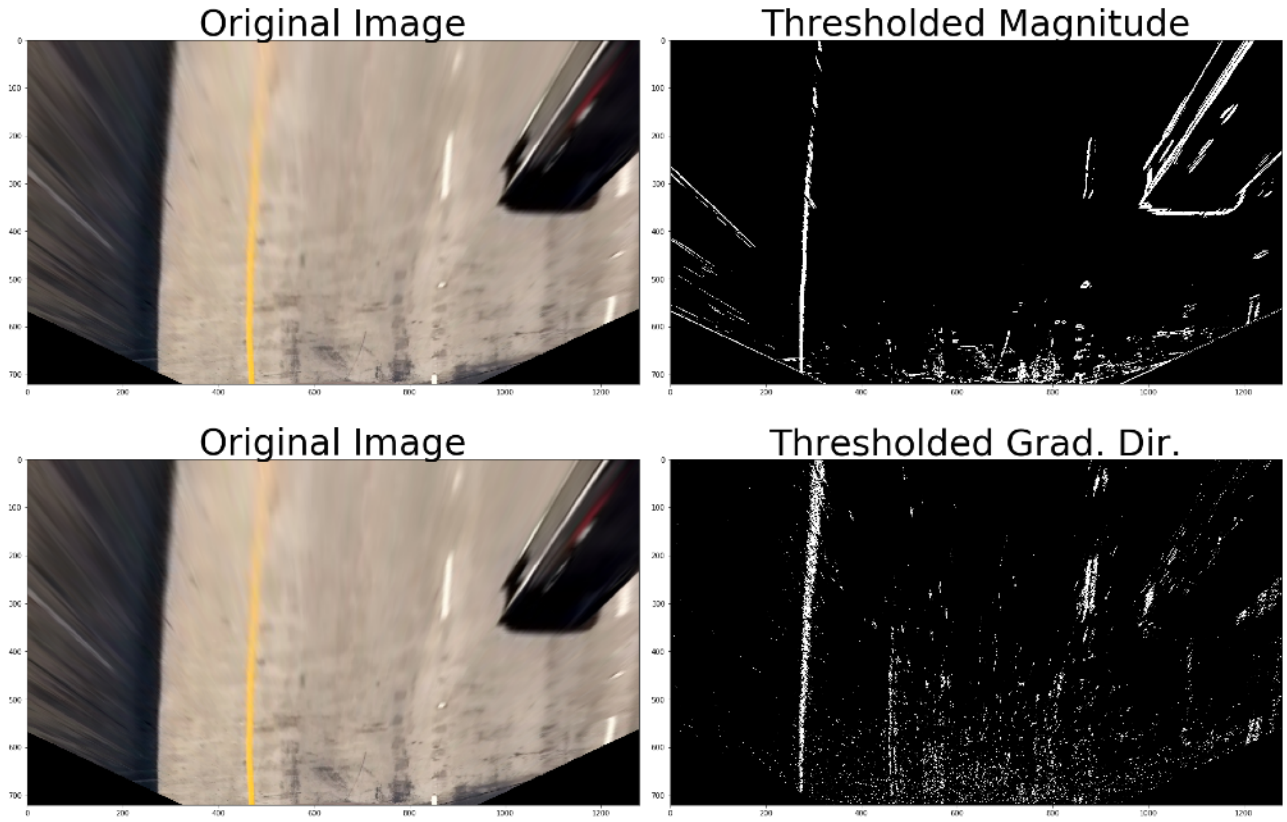
The image below depicts the results of applying `undistort` to one of the project dashcam images:



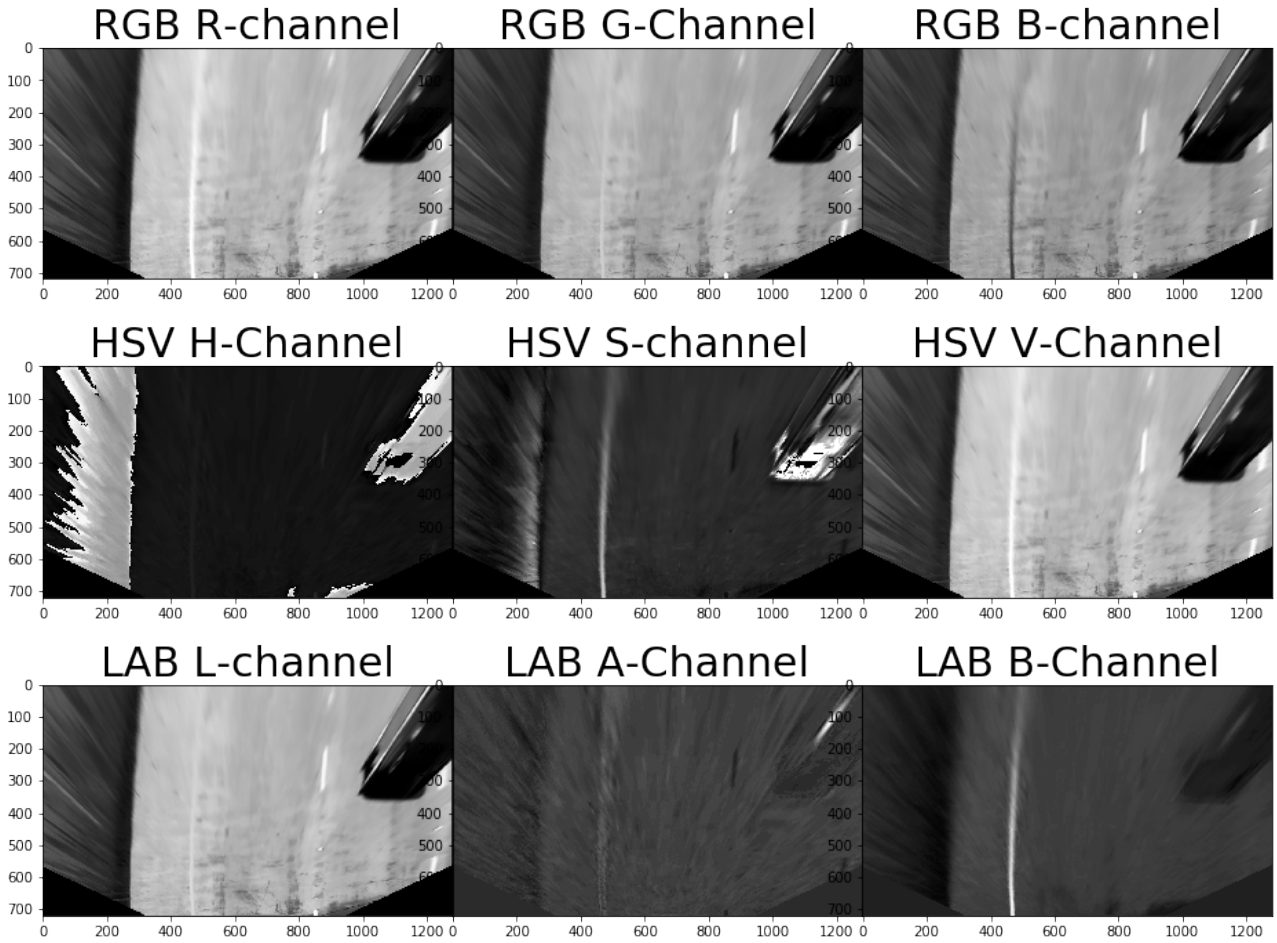
The effect of `undistort` is subtle, but can be perceived from the difference in shape of the hood of the car at the bottom corners of the image.

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.

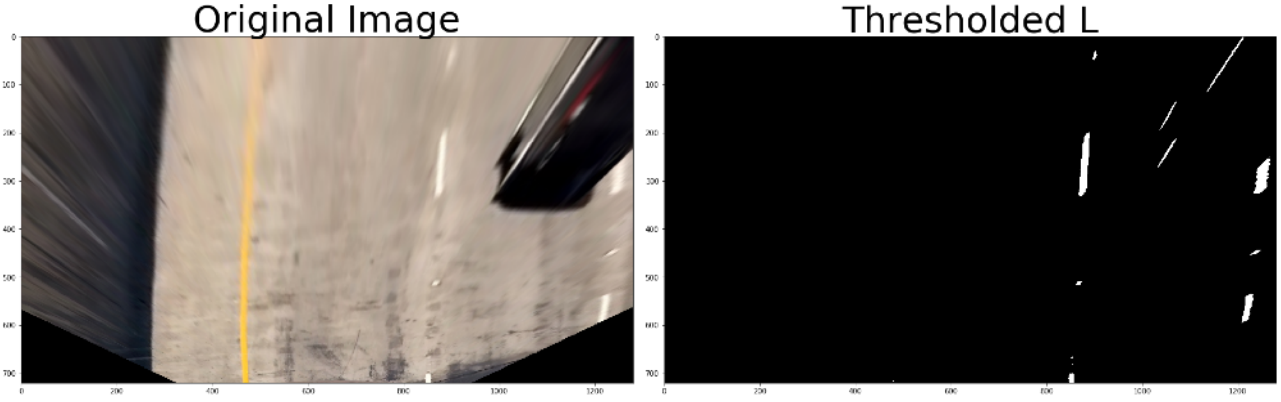
I explored several sobel gradient thresholds and color channel thresholds in multiple color spaces. These are labeled clearly in the Jupyter notebook. Below is an example of the sobel magnitude and direction thresholds:

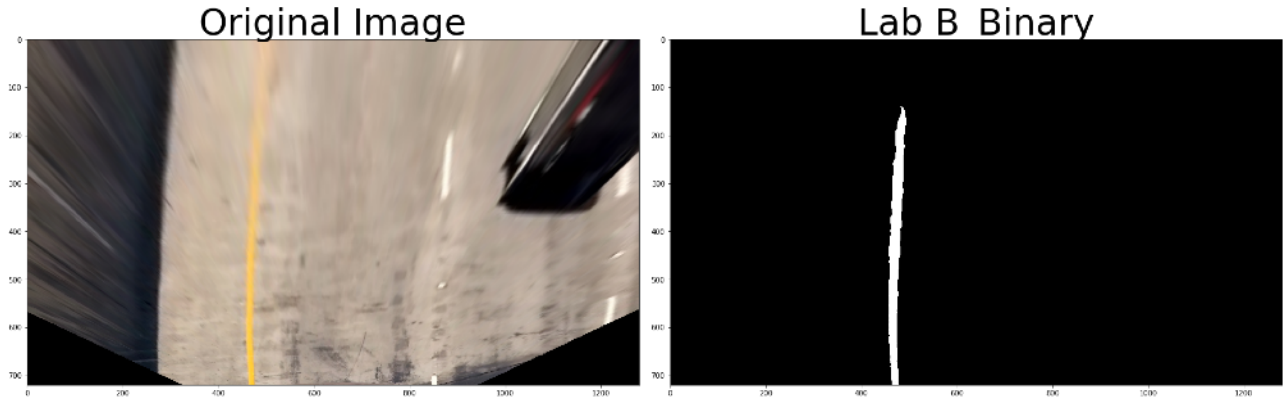


The below image shows the various channels of three different color spaces for the same image:



Ultimately, I chose to use just the L channel of the HLS color space to isolate white lines and the B channel of the LAB colorspace to isolate yellow lines. I did not use any gradient thresholds in my pipeline. I did, however finely tune the threshold for each channel to be minimally tolerant to changes in lighting. As part of this, I chose to normalize the maximum values of the HLS L channel and the LAB B channel to 255, since the values the lane lines span in these channels can vary depending on lighting conditions. Below are examples of thresholds in the HLS L channel and the LAB B channel:





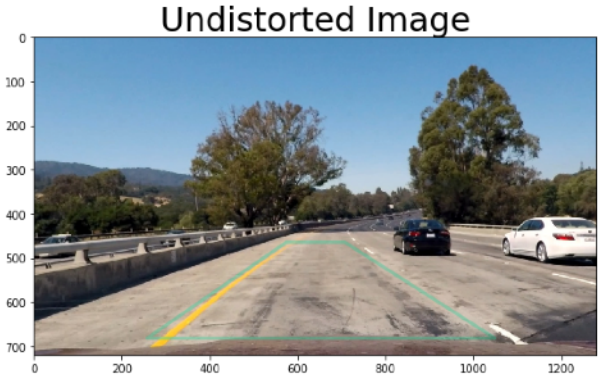
Below are the results of applying the binary thresholding pipeline to various sample images:

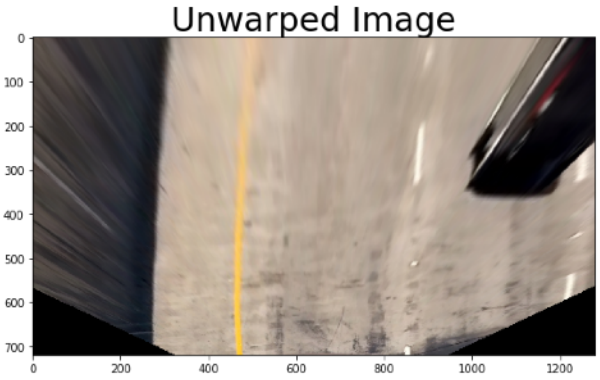
3. Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

The code for my perspective transform is titled "Perspective Transform" in the Jupyter notebook, in the seventh and eighth code cells from the top. The `unwarp()` function takes as inputs an image (`img`), as well as source (`src`) and destination (`dst`) points. I chose to hardcode the source and destination points in the following manner:

```
src = np.float32([(575,464),
                  (707,464),
                  (258,682),
                  (1049,682)])
dst = np.float32([(450,0),
                  (w-450,0),
                  (450,h),
                  (w-450,h)])
```

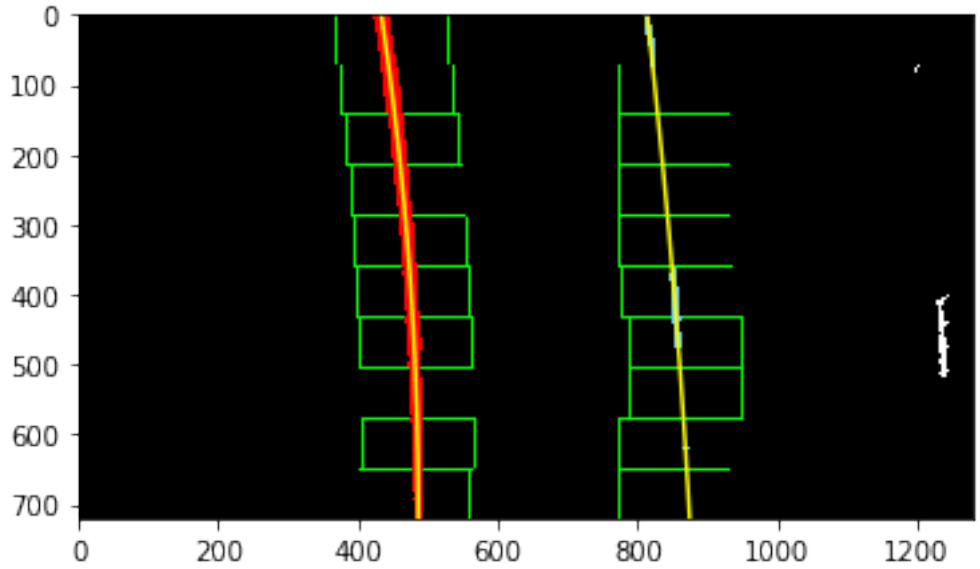
I had considered programmatically determining source and destination points, but I felt that I would get better results carefully selecting points using one of the `straight_lines` test images for reference and assuming that the camera position will remain constant and that the road in the videos will remain relatively flat. The image below demonstrates the results of the perspective transform:



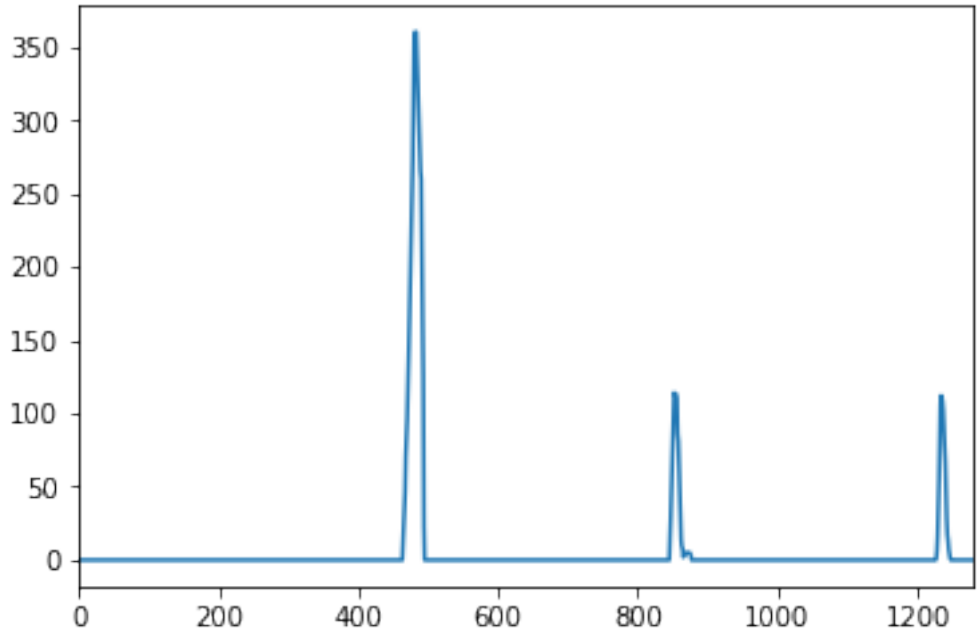


4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

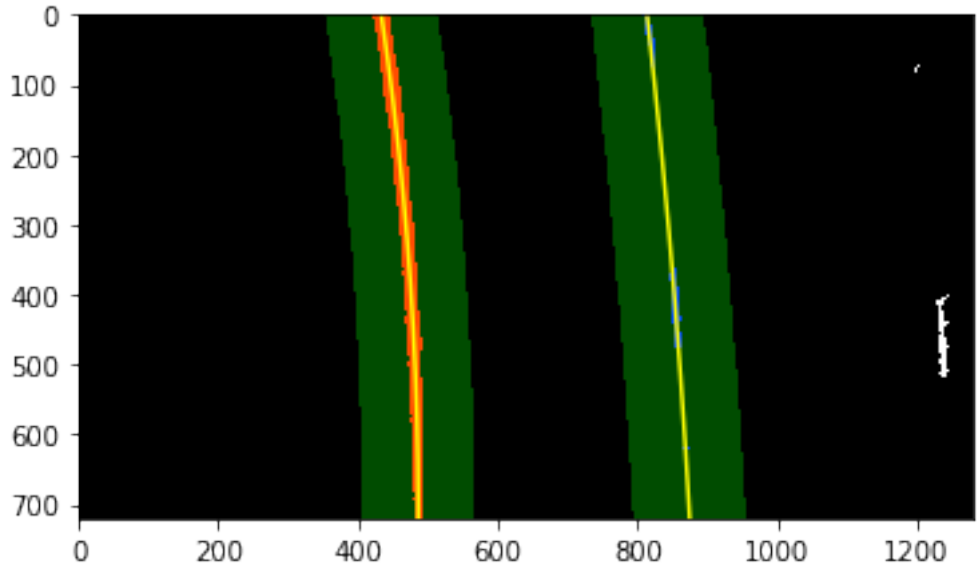
The functions `sliding_window_polyfit` and `polyfit_using_prev_fit`, which identify lane lines and fit a second order polynomial to both right and left lane lines, are clearly labeled in the Jupyter notebook as "Sliding Window Polyfit" and "Polyfit Using Fit from Previous Frame". The first of these computes a histogram of the bottom half of the image and finds the bottom-most x position (or "base") of the left and right lane lines. Originally these locations were identified from the local maxima of the left and right halves of the histogram, but in my final implementation I changed these to quarters of the histogram just left and right of the midpoint. This helped to reject lines from adjacent lanes. The function then identifies ten windows from which to identify lane pixels, each one centered on the midpoint of the pixels from the window below. This effectively "follows" the lane lines up to the top of the binary image, and speeds processing by only searching for activated pixels over a small portion of the image. Pixels belonging to each lane line are identified and the Numpy `polyfit()` method fits a second order polynomial to each set of pixels. The image below demonstrates how this process works:



The image below depicts the histogram generated by `sliding_window_polyfit`; the resulting base points for the left and right lanes - the two peaks nearest the center - are clearly visible:



The `polyfit_using_prev_fit` function performs basically the same task, but alleviates much difficulty of the search process by leveraging a previous fit (from a previous video frame, for example) and only searching for lane pixels within a certain range of that fit. The image below demonstrates this - the green shaded area is the range from the previous fit, and the yellow lines and red and blue pixels are from the current image:



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.

The radius of curvature and calculated in the code cell titled "Radius of Curvature and Distance from Lane Center Calculation" using this line of code (altered for clarity):

```
curve_radius = ((1 + (2*fit[0]*y_0*y_meters_per_pixel + fit[1])**2)**1.5) / np.absolute(2*fit[0])
```

In this example, `fit[0]` is the first coefficient (the y-squared coefficient) of the second order polynomial fit, and `fit[1]` is the second (y) coefficient. `y_0` is the y position within the image upon which the curvature calculation is based (the bottom-most y - the position of the car in the image - was chosen). `y_meters_per_pixel` is the factor used for converting from pixels to meters. This conversion was also used to generate a new fit with coefficients in terms of meters.
The position of the vehicle with respect to the center of the lane is calculated with the following lines of code:

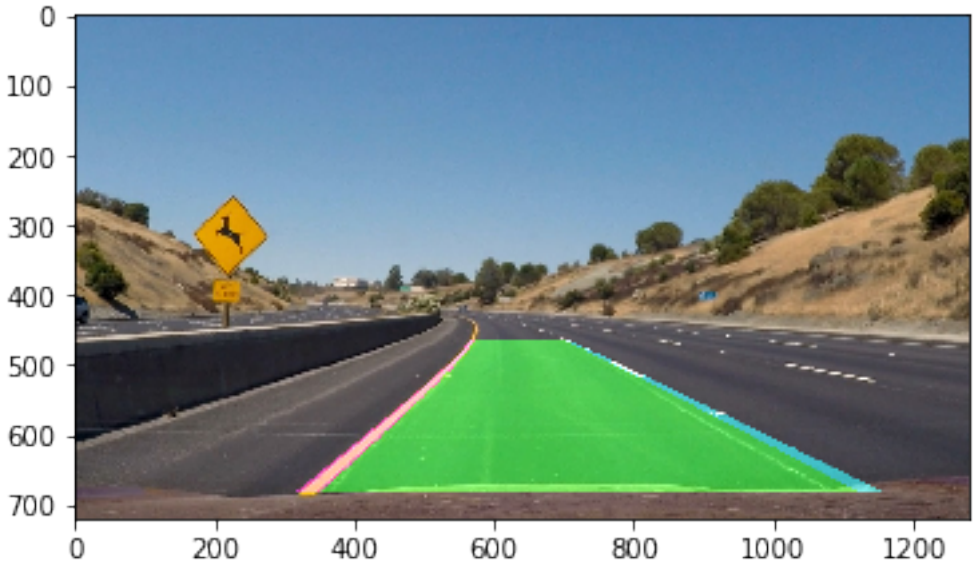
```
lane_center_position = (r_fit_x_int + l_fit_x_int) /2
center_dist = (car_position - lane_center_position) * x_meters_per_pix
```

`r_fit_x_int` and `l_fit_x_int` are the x-intercepts of the right and left fits, respectively. This requires evaluating the fit at the maximum y value (719, in this case - the bottom of the image) because the minimum y value is actually at the top (otherwise, the constant coefficient of each fit would have sufficed). The car position is the difference between these intercept points and the image midpoint (assuming that the camera is mounted at the center of the vehicle).

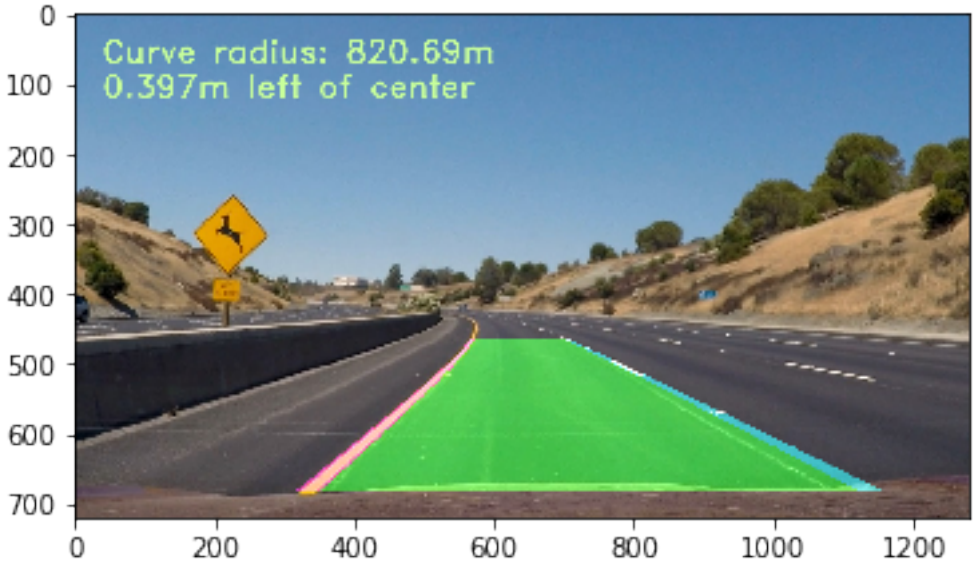
6. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.

I implemented this step in the code cells titled "Draw the Detected Lane Back onto the Original Image" and "Draw Curvature Radius

and Distance from Center Data onto the Original Image" in the Jupyter notebook. A polygon is generated based on plots of the left and right fits, warped back to the perspective of the original image using the inverse perspective matrix `Minv` and overlaid onto the original image. The image below is an example of the results of the `draw_lane` function:



Below is an example of the results of the `draw_data` function, which writes text identifying the curvature radius and vehicle position data onto the original image:



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!).

Video can be found in the folder AdvanceLaneLines

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 problems I encountered were almost exclusively due to lighting conditions, shadows, discoloration, etc. It wasn't difficult to dial in threshold parameters to get the pipeline to perform well on the original project video (particularly after discovering the B channel of the LAB colorspace, which isolates the yellow lines very well), even on the lighter-gray bridge sections that comprised the most difficult sections of the video. It was trying to extend the same pipeline to the challenge video that presented the greatest challenge. The lane lines don't necessarily occupy the same pixel value range on this video that they occupy on the first video, so the normalization/ scaling technique helped here quite a bit, although it also tended to create problems when the white lines didn't contrast with the rest of the image enough. This would definitely be an issue in snow or in a situation where, for example, a bright white car were driving among dull white lane lines. Producing a pipeline from which lane lines can reliably be identified was of utmost importance (garbage in, garbage out - as they say), but smoothing the video output by averaging the last `n` found good fits also helped. My approach also invalidates fits if the left and right base points aren't a certain distance apart under the assumption that the lane width will remain relatively constant.

I've considered a few possible approaches for making my algorithm more robust. These include more dynamic thresholding, designating a confidence level for fits and rejecting new fits that deviate beyond a certain amount or rejecting the right fit if the confidence in the left fit is high and right fit deviates too much.