Advanced Lane Detection

Rubric

- Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.
- Provide an example of a distortion-corrected image.
- 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.
- Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.
- Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?
- 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.
- Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.
- 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!)
- 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?

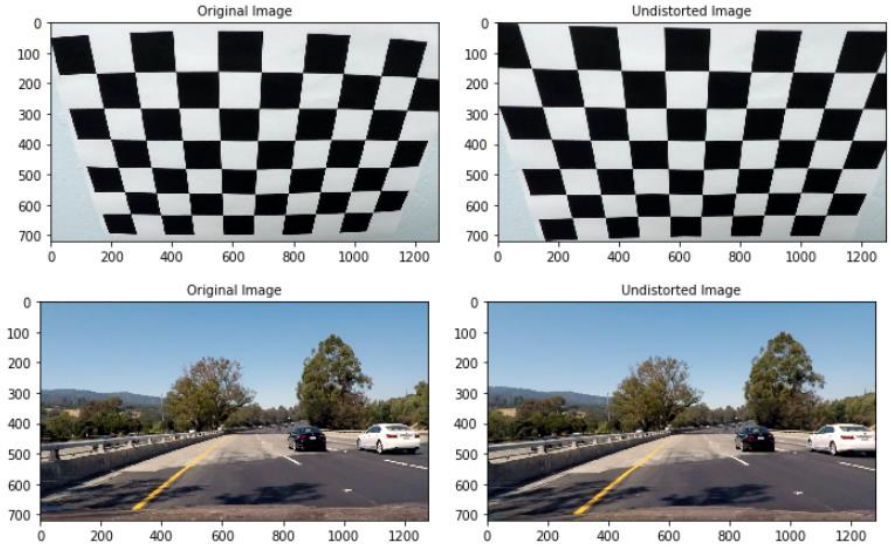
Pipeline

The pipeline was implemented within a class named LaneDetection_From_Video. Below are the methods of the class.

- calibrate_camera
- cal_distort
- color_thresh
- abs_sobel_thresh
- mag_thresh
- dir_thresh
- get_top_down_view
- sliding_window_lane_search
- informed_lane_search
- get_road_params
- transform_top_down_to_world_view
- draw_avg_lanes_top_down
- annotate_img
- process image

When the class is initialized, camera calibration is executed. VideoFileClip takes LaneDetection_From_Video.process_image() as an input and processes each image sequentially. The processing steps of the pipeline are described below in order of application. The methods used within each step are written in *italics*.

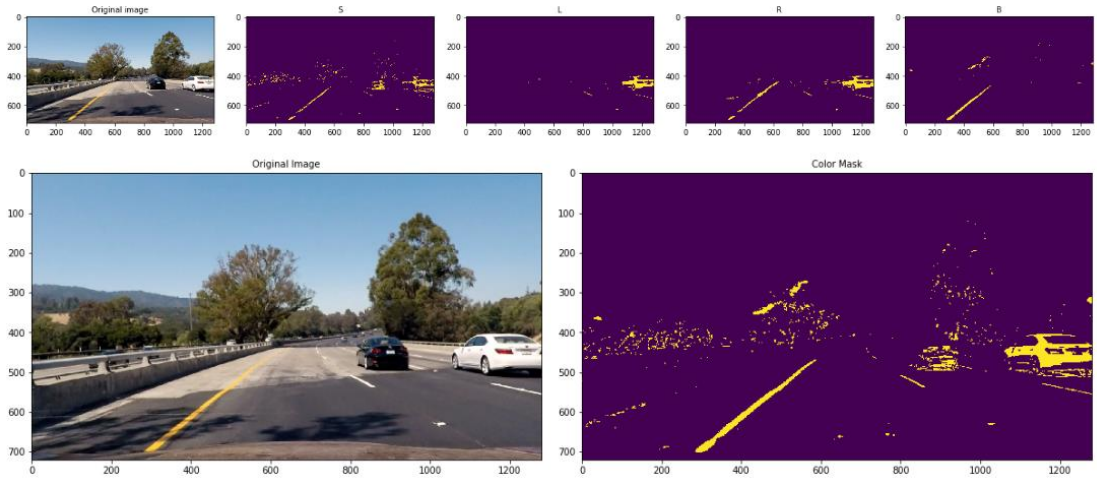
1. **Camera Calibration:** The *calibrate_camera* method takes in a list of filenames of chessboard images taken at different poses. The OpenCV function *cv2.findChessboardCorners* is used to identify the corners of the checkboard in each image. Given the 2D pixel locations of the corners and the 3D position of the corners in the world (assumed to be planar and zero depth), cv2.calibrateCamera function calculates the camera parameters: distortion matrix and coefficients. The *cal_undistort* method uses the camera calibration parameters to undistort a given image.



2. **Color Thresholding:** To extract lane line pixels, I applied a combination of color thresholding and sobel operators. The *color_thresh* method accepts an RGB image, the color space and color channel, and the desired threshold range. To threshold L channel in HSL space, the color_thresh method can be called as follows:

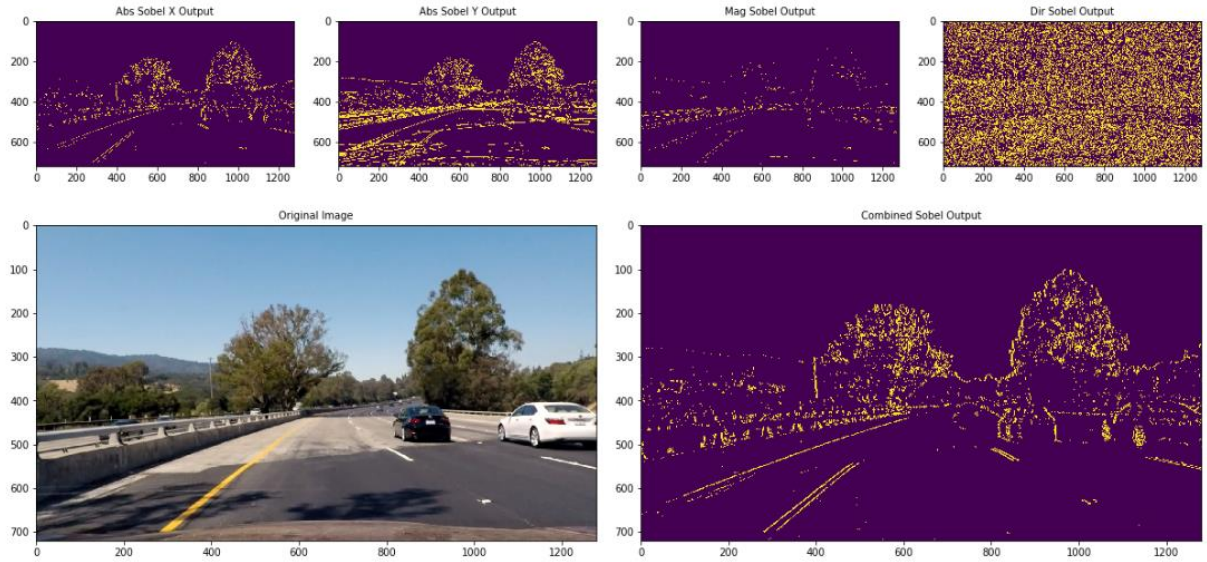
   color_thresh(rgb_img, color_space= 'HSL',channel_num =2 , low_threshold=100, high_threshold=255)

   I applied thresholding to S and L channels in HSL color space, R channel in RGB color space, and L channel in LAB color space. The binary outputs obtained by thresholding each channel is shown below.



   I originally used S, R, and L channels. In the pipeline, S channel does a good job of highlighting yellow lines, L channel adds some robustness to light variations, and R channel does a reasonable job of highlighting all lane lines. B channel was also recommended in one of the posts I came across to have robustness to shadows [#https://medium.com/@vamsiramakrishnan/robust-lane-finding-using-python-open-cv-63eb66fa2616]. I ended up using B channel instead of R. The improvement over using R, however, was negligible.

3. **Edge detection:** I played around with 4 methods. *abs_thresh* generates a binary image by calculating the intensity gradient along the width (x) or height of the image (y), and normalizes the output to [0,1] range. *mag_thresh* calculates the magnitude of the gradients, and dir_thresh calculates the direction of the gradient. After trial and error, I used all the methods except the gradient along the height as it creates shadow artifacts (see abs sobel y output below). The final image is shown below and is titled as Combined Sobel Output.

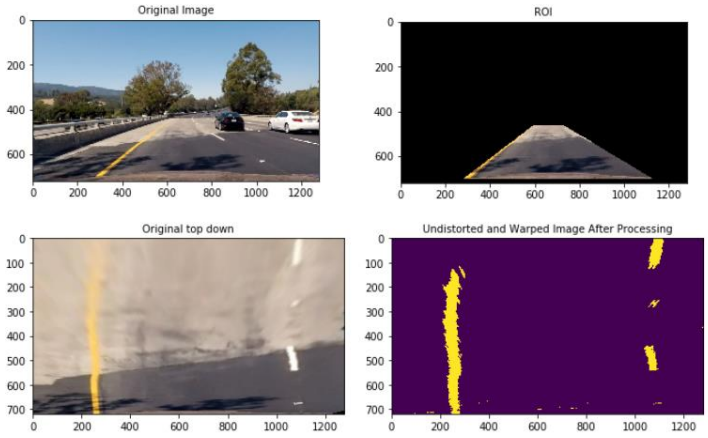


Finally, I did a binary addition of color thresholding and edge detection outputs.

4. **Perspective Transform:** Assuming the road is a flat plane, I pick four points in a trapezoidal shape (like region masking) that would represent a rectangle when looking down on the road from above. I picked an image where the lane lines are more or less straight and picked 4 points (*src*) that are roughly on the lane lines. The method *get_top_down_view* calculates a perspective transform between 4 source points and 4 destination points. cv2.getPerspectiveTransform is used to compute the perspective transformation from *src* to *dst*. This transformation is then applied to the image using *cv2.warpPerspective.*

	src	**dst**
Bottom left	280,700	250,700
Top left	595,460	250,0
Top right	725,460	1065,0
Bottom right	1125,700	1065,720

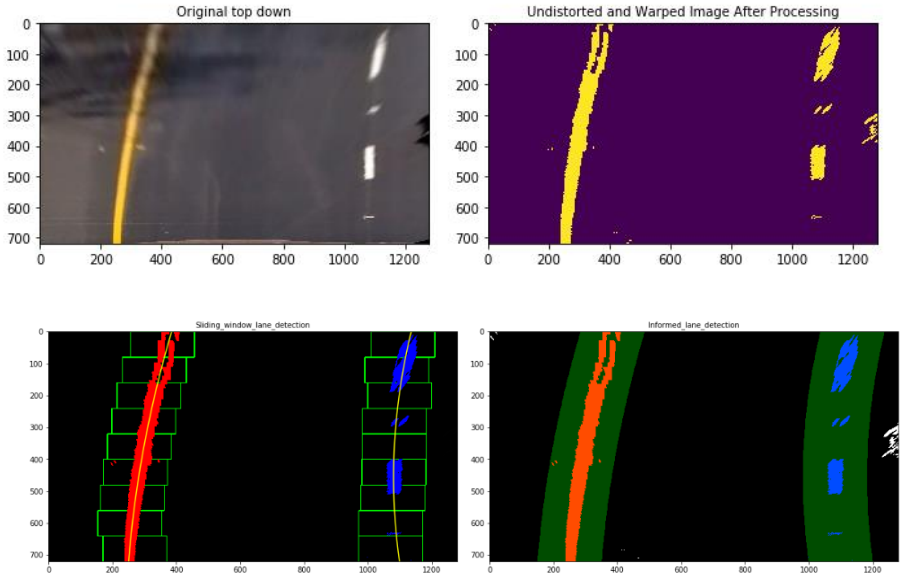
After applying perspective transform from src to dst on an image where the road seems to be straight, the lines should look straight and vertical from a bird's eye view perspective.



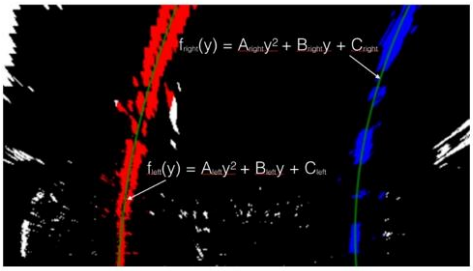
5. **Detect Lane Lines**
   - Sliding Window Lane Search: This algorithm is based on the observation that If we take the histogram along all columns in the lower half of the image, two most prominent peaks in the histogram correspond to the base of the lane lines. If we trace the histogram peaks moving upward in the image, we can determine where the lane lines go. In this pipeline, the image is divided into a number of predefined rows, and a fixed window margin is used. A polynomial is fit to the peaks found in each window to obtain the lane line. The lane line on the left half of the image corresponds to the left lane, where as the right half of the image corresponds to the right lane.

- Informed Lane Search: Informed lane search algorithm uses a prior on the expected lane position and performs a search in a margin around it. The expected lane position is calculated using a linear weighted average of n recent frames as suggested in [http://davidaventimiglia.com/advanced_lane_lines.html].

  Sliding window lane search is inefficient. The lanes do not change drastically from one frame to the next. Therefore, it is used for the first frame, and only when informed lane search fails to find the lanes.

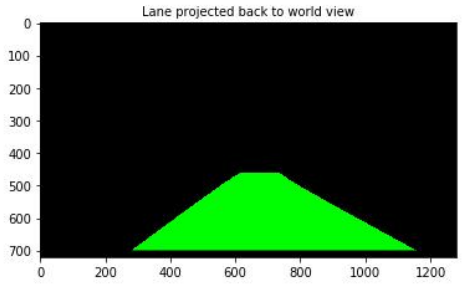


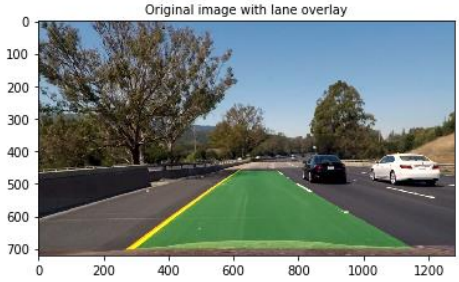
6. **Road curvature and car position calculations:** *get_road_params* method computes the left and right lane curvatures, and the location of the car with respect to the center of the lane. The curvature is calculated using the line fit parameters - output of the polyfit function. Note that y values of the image increase from top to bottom. To measure the radius of curvature closest to the vehicle, we evaluate the formula below at the y value corresponding to the bottom of the image. The car position is calculated by measuring the right polyline intersection and left polyline intersection with respect to the center of the image at the bottom. The last step is to convert the pixel values to real world space. The lane is about 30 meters long and 3.7 meters wide. In the perspective view, the length is 720 pixels, the width is around 800 pixels. These numbers are used to scale the pixel measurements to world measurements.



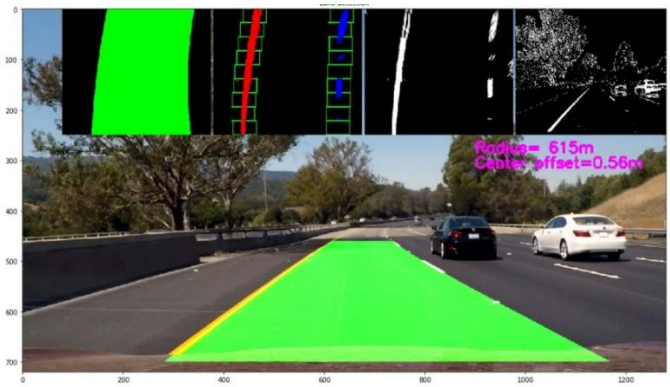
$$R_{curve} = \frac{(1+(2Ay+B)^2)^{3/2}}{|2A|}$$

7. **Inverse perspective transform:** *draw_lanes_top_down* method defines a polygon whose vertices are defined by the polyline fit. Finally, *transform_top_down_to_world_view* method transforms the image back to world view by applying the inverse of the perspective transformation using *cv2.warpPerspective*.





8. **Image annotation:** To understand failure modes during debugging; the output of preprocessing, top down view, sliding window lane search and polygon fit to the lane lines were overlaid on the video. The *annotate_img* method was used to achieve an annotated image.



Discussions:

- This was a very time-consuming project. Most of the time was spent on finding the right color space and threshold parameters, and tuning edge detection parameters. Overlaying the output of each processing layer on the video helped identifying frames that fail.
- The Lane_Detection_From_Video class was not optimized for speed. *abs_thresh,mag_thresh, and dir_thresh* methods can be combined in one method to prevent applying sobel operator each time one of them is called.
- I tried rejecting lane lines if the curvature change within two consecutive frames exceeds a threshold. It was difficult to tune. The change in calculated curvature can be very large when the lane is somewhat straight. This method was commented out and not used in the final pipeline.
- The pipeline did a poor job of removing tree shadow in one of the frames. The image below shows the output of the pipeline when only 12 most recent lanes were used to compute the prior for the lane parameters in the next frame. The pipeline output was constrained to a subregion within the lane. I increased the lane buffer to 25 most recent lanes. Increasing further results in a lag in detecting the lane lines. Changing the order of pipeline steps and applying color thresholding and edge detection after perspective transformation may help removing the shadow better. However, adaptive histogram equalization seems to be necessary. The video output is in test_videos_output/project_solution.mp4

- The pipeline doesn't perform well in challenge videos.