Advanced Lane Detection

Files Submitted & Code Quality

1. My project includes the following files:

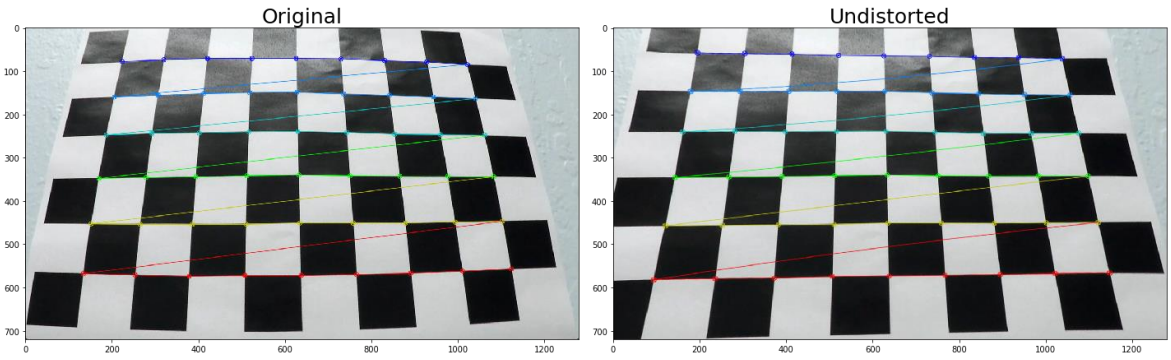
- **advanced_lane_finding.ipynb** iPython notebook containing the scripts for various tasks of this projects
- **MyUtilities.py** containing scripts for some utilities used in the projects
- **output_images** folder containing images obtained at various stages in an algorithm
- **writeup_report.pdf** summarizing the results
- **project_video_output.mp4** video showing my final result

Camera Calibration

2. 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 **second cell** of the IPython notebook – the '**get_camera_calibration**' method which is similar to the one in class lessons. It processes all images in 'camera_cal\' path, converts them to grayscale, gets their object points and image points (corners), and finally computes the camera matrix and camera distortion coefficients using the cv2.calibrateCamera function. The object points are constructed with an assumption that the chessboard is two-dimensional (i.e. z = 0).
- The **second cell** of the IPython notebook – the '**undistort_image**' method which is the same as the one in class lessons. It takes the camera matrix and the distortion coefficients and undistorts the input image using the cv2.undistort function.
  An example of undistorted chessboard image is shown below.



Pipeline (single images)

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

Using the camera matrix and the distortion coefficients, every image in the pipeline will be corrected using the '**undistort_image**' method. Undistorted version of an example road image is shown below:

2. 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 includes two functions called **perspective_transform**() and **warp_image**(), which appear in **cell five** of the Ipython notebook. The **perspective_transform()** function takes as inputs source (src) and destination (dst) points and computes transformation matrix and inverse transformation matrix using the cv2.getPerspectiveTransform function. The **warp_image**() function takes as inputs – an image and the perspective transformation matrix and gives a different view point of the image using the cv2.warpPerspective function.

I chose hardcoded source and destination points as below:

src_points = np.float32([[240., 720.], [575., 460.], [715., 460.], [1150., 720.]])

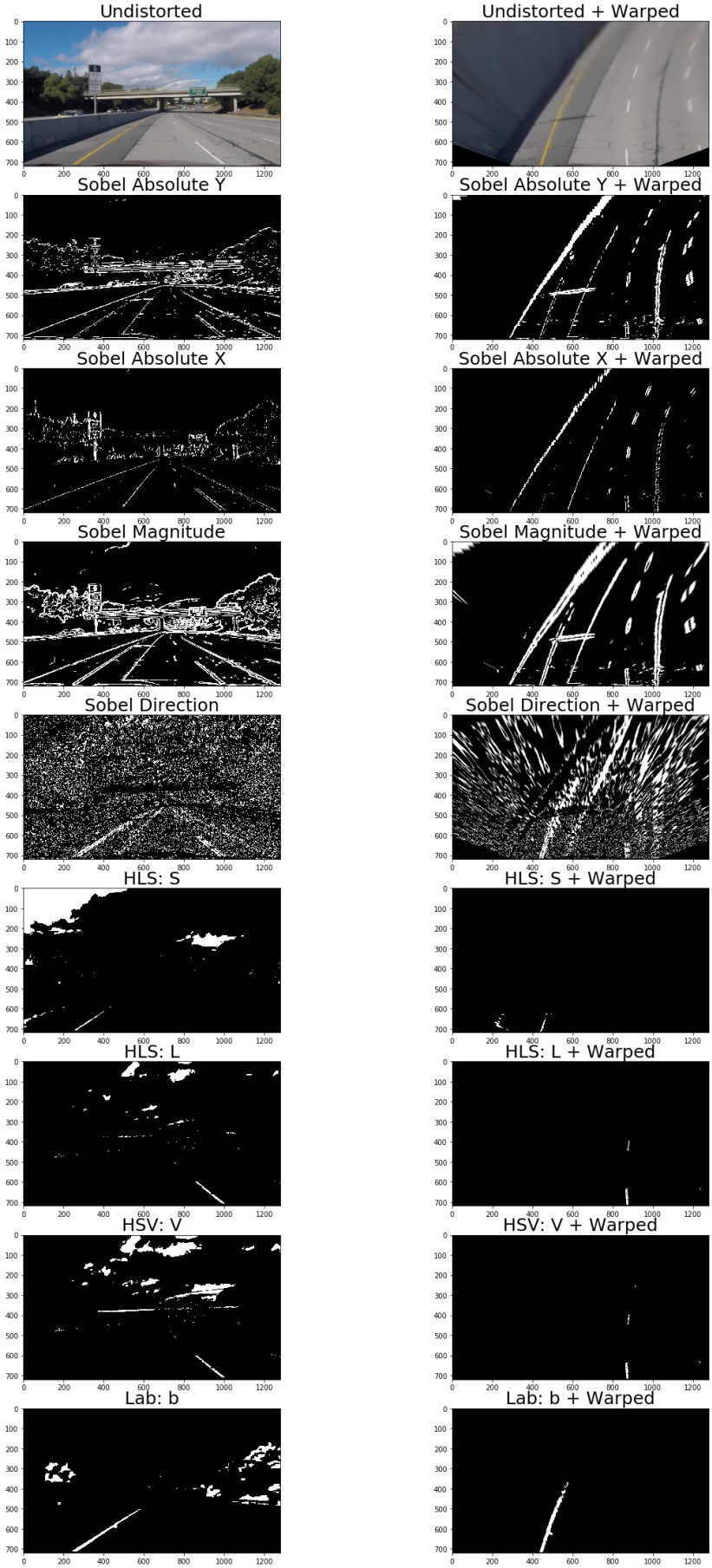
dst_points = np.float32([[440., 720.], [440., 0.], [950., 0.], [950., 720.]])

An example of undistorted and perspective transformed image is shown above. The red lines on the above two images (Undistorted and Warped) give an intuition as to how the image is transformed.

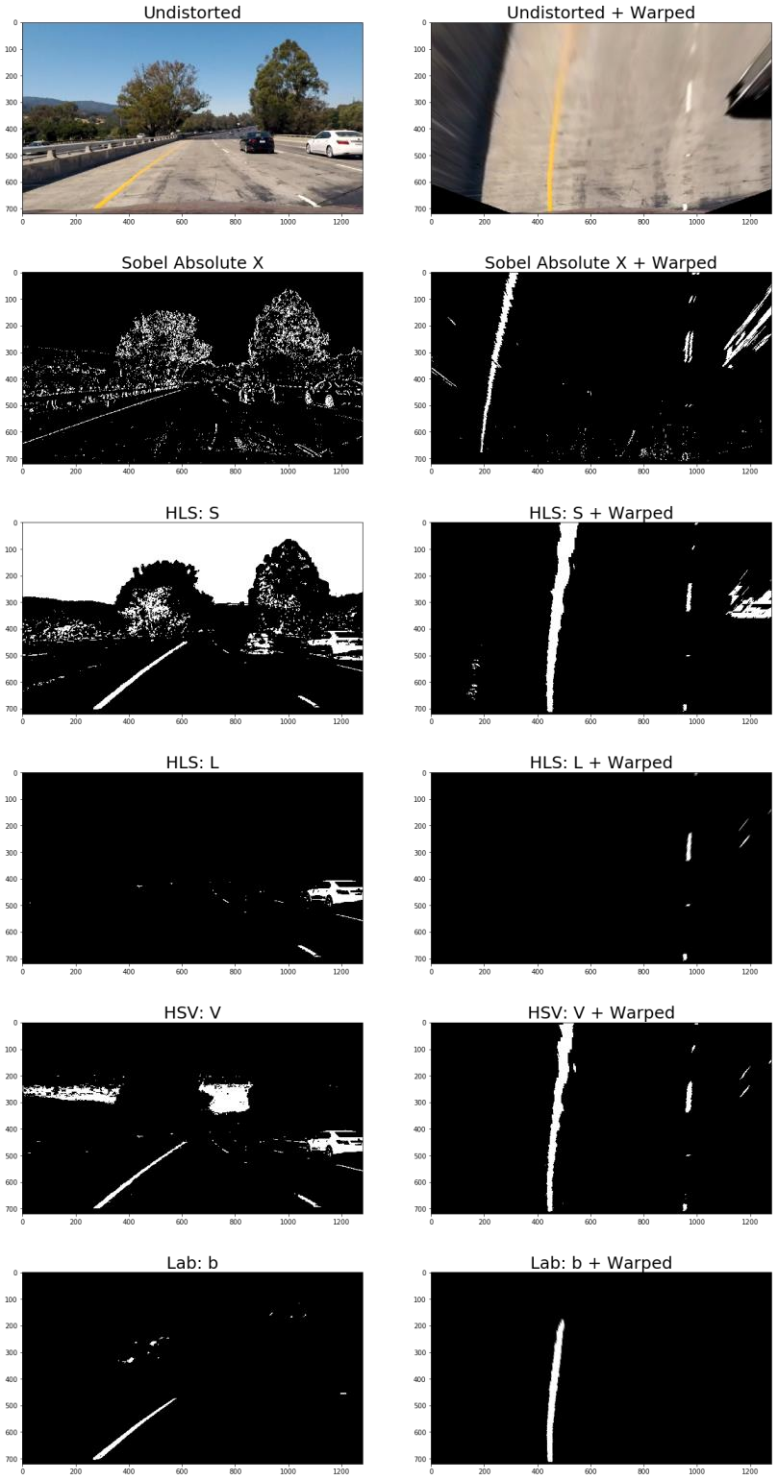
3. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

The chosen color transforms, gradients or other methods must suitably work on diverse types of test images and video streams. Initialy I did not know what color transforms, gradients or other method would be best suitable, and therefore first I tested all types of binary thresholding techniques using a test image. In **cell 7 and 8** of the Ipython notebook, the following thresholding techniques are tested:

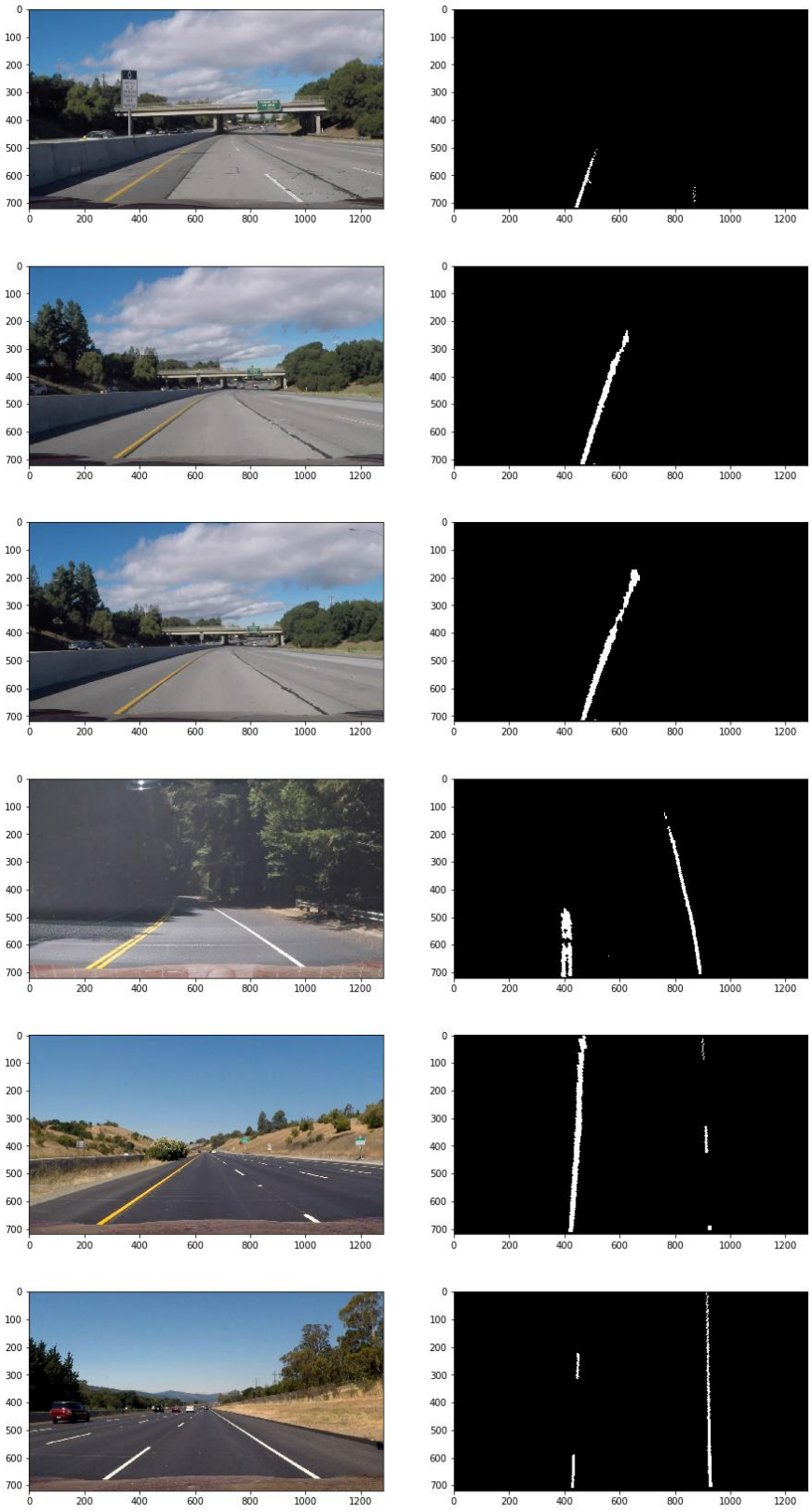
Absolute sobel gradient in Y, Absolute sobel gradient in X, Magnitude sobel gradient, Directional sobel gradient, S channel in HLS color space, L channel HLS color space, V channel in HSV color space and the 'b' channel in the Lab color space. The positive side of 'b' channel in 'Lab' color space indicates the presence of 'Yellow' color in an image. I chose this colorspace specifically because of the presence of 'Yellow' tracks in images. Lab can be identify the 'yellow' color under diverse lighting conditions with the help of its 'b' channel. Below image show different binary transformations for an example image.

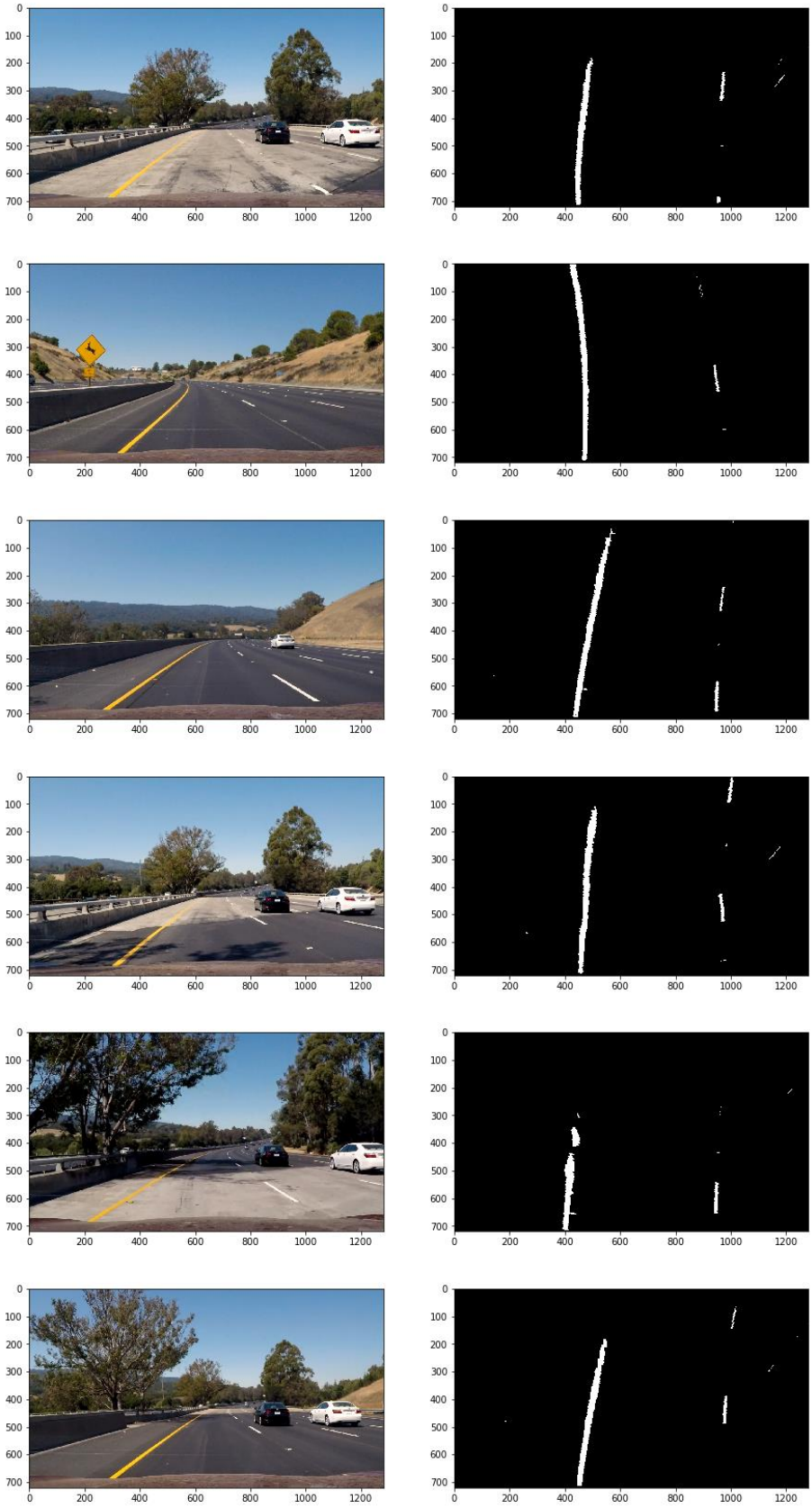

Undistorted
Undistorted + Warped
Sobel Absolute Y
Sobel Absolute Y + Warped
Sobel Absolute X
Sobel Absolute X + Warped
Sobel Magnitude
Sobel Magnitude + Warped
Sobel Direction
Sobel Direction + Warped
HLS: S
HLS: S + Warped
HLS: L
HLS: L + Warped
HSV: V
HSV: V + Warped
Lab: b
Lab: b + Warped

As can be seen above, the first four transformations do not work satisfactorily. Therefore I decided to test extensively - the HLS, HSV and Lab color spaces and choose appropriate threshold values for those. I chose a different test image this time. Its binary thresholded version is shown below (**cell 9** of the notebook). As can be seen below, the S channel of HLS color space can be noisy sometimes. However the combination of L channel of HLS and b channel of Lab color space seem to show nice lane pixel identifications. Therefore I decided to combine those two color spaces (**cell 10** of the notebook)



Further, all test images are tested using the pipeline described in **cell 10** of the notebook (images below). Lane pixel identifiaction seems to be satisfactory. Next step is to construct the lines using pixel search and polyfit operation.

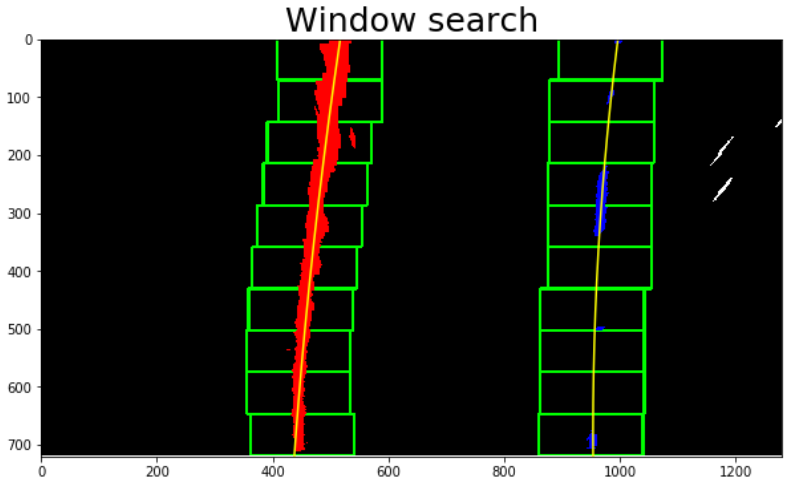


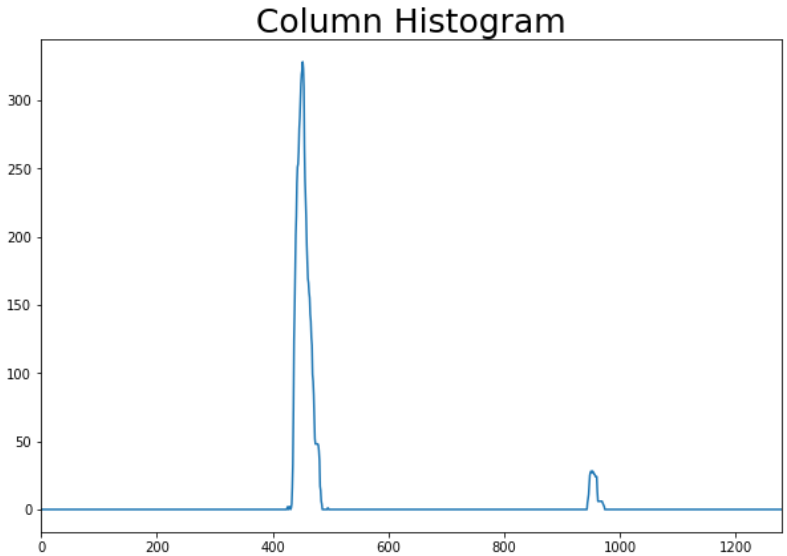


4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

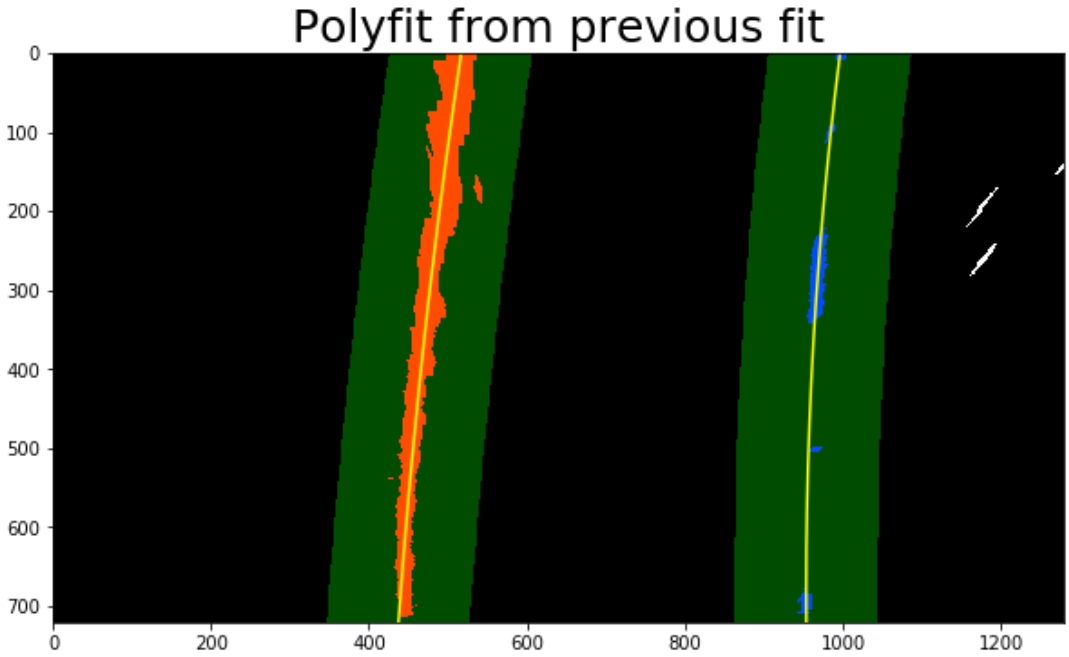
I used two methods to identify the lane line images as suggested in the lecture notes, depending on whether a best estimate of the lane line polynomials is available or not.

1. If the best estimate of polynomials (for both left and right lanes) are not available (generally happens at the beginning of a video frame or when there is a long gap in lane identification), then the sliding window search method is used. It is written in **cell 12** of the notebook. Here the wrapped image is searched for a predefined number of pixels within a moving box that hovers across the image. Then the pixels within these boxes are classified into left and right lane pixels. Then the centroid pixel in each detected box is identified and the polynomials are fitted for the detected pixels in left and right lanes. The start of the search begins at the bottom of the image (actually top of the image in inverted axis), where the highest concentration of pixels (or lanes) is identified with the help of a histogram taken over the lower half of the image.





2. Once the polynomials are identified, the window search is not really necessary for the subsequent images. Since the location of lanes does not jump much from frame to frame, a small search area around the previously identified area can be defined and searched for potential lane pixels. Eventually the polyfit of the pixels detected this way can be constructed. This is called the polyfit with the help of previously fitted polynomial (**cell 14** of the notebook). An example image using this method is shown below:



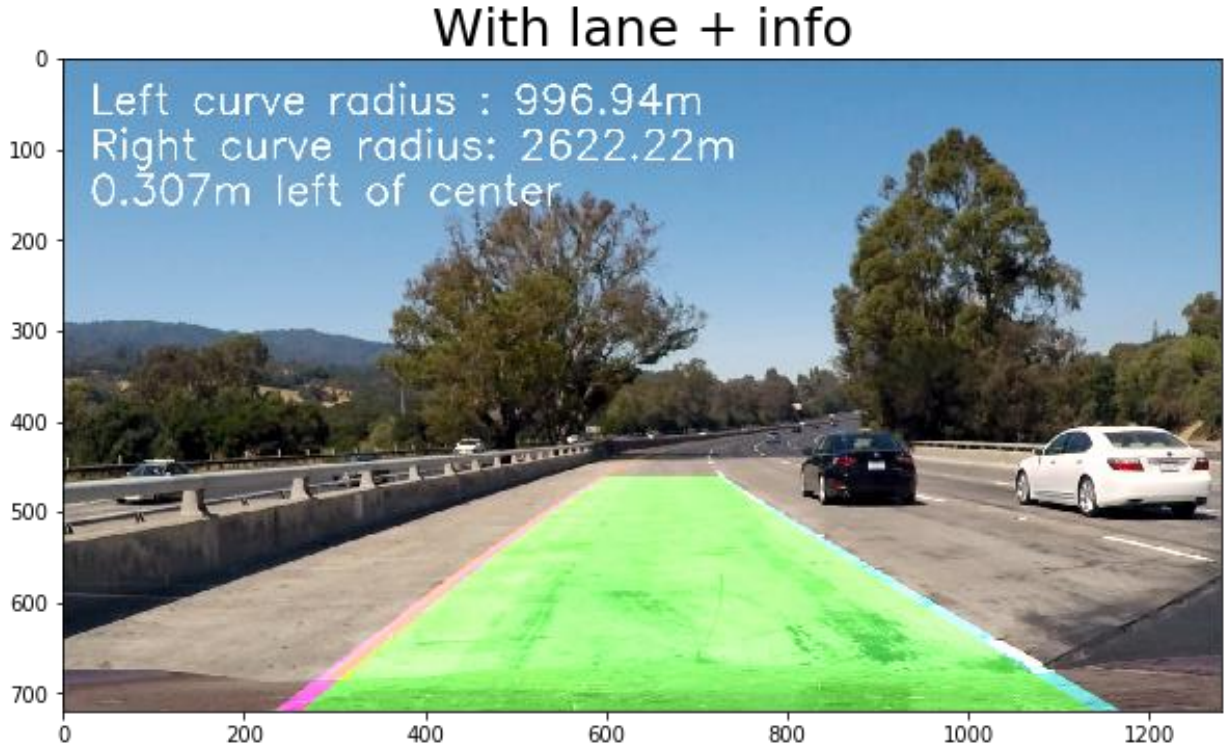
5. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

   This code is written in **cell 16.**
   First I defined conversion ratios to transform pixel distance from image space to physical space. Then using the radius of curvature formula and the polynomial coefficients (this formula is given in lecture notes), the left and right radius of curvatures are computed.
   The offset of vehicle from the center is simply the difference between the image midpoint and the lane midpoint. The lane midpoint is computed as follows:

```
l_fit_x_int = l_fit[0]*img_height**2 + l_fit[1]*img_height + l_fit[2]
r_fit_x_int = r_fit[0]*img_height**2 + r_fit[1]*img_height + r_fit[2]
lane_midpoint = (r_fit_x_int + l_fit_x_int) //2

offset_from_center_dist = (image_midpoint - lane_midpoint) * xm_per_pix #in meters
```

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



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!).

The pipeline to process images from videoframe is given in method "**process_image**". It is self-explanatory.

Video is provided

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?

I faced following major issues during my implementation and solved those issues with some modifications/enhancements in the software.

Issue1: Initially I was working with the combination of S-channel of HLS color space and sobel X gradient transformations as suggested in the lecture notes. However, despite tuning corresponding thresholds rigorously, I did not achieve good results.

Solution: I tried different color spaces and their binary thresholded images simultaneously, as discussed earlier. Finally I decided to go with L channel of HLS colorspace and the 'b' channel of Lab color space. The 'b' channel independenly identifies yellow lines under various lighting conditions. Combination of L and b worked the best.

Issue2: Detected lanes were jumping from frame to frame.
Solution: I performed sanity checks such as - checking the the right and left lanes must be almost parallel, and the distance between right and left lanes must be reasonable. If the lanes did not satisfy these checks, they are rejected. These checks are performed in method '**process_image'**.
I also used 'smoothing' of detected polynomials. I saved detected polynomials in a moving array 'current_fit' declated in a class **Line()** [**cell 18** of the notebook]. The function **get_best_fit** would perform some housekeeping checks to decide whether to perform averaging or not depending on how reliable the latest polynomial is. These checks are performed inside the function 'get_best_fit' in the class 'line'. If everything is good, an average of past 5 polynomial coefficients is performed so that the constructed lane lines will move smoothly.

Other problems: Although my code works best for project_video.mp4, it works kind of 'okay' for 'challenge_video.mp4' and fails miserably for 'harder_challenge_video.mp4'. It fails at those palces when the light is too bright or too dark. I think I should explore other color spaces for challenging videos (currently I use lightness channel - L).
Nonetheless it looks like there is more to do in pixel search and in getting the best_fit, rather than the color space explaoration.