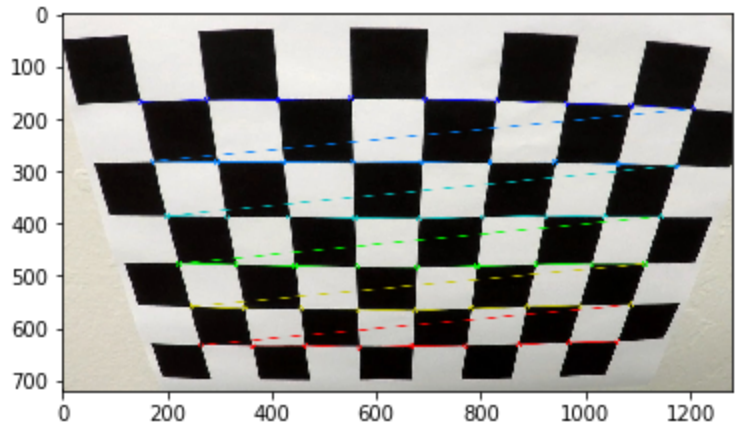
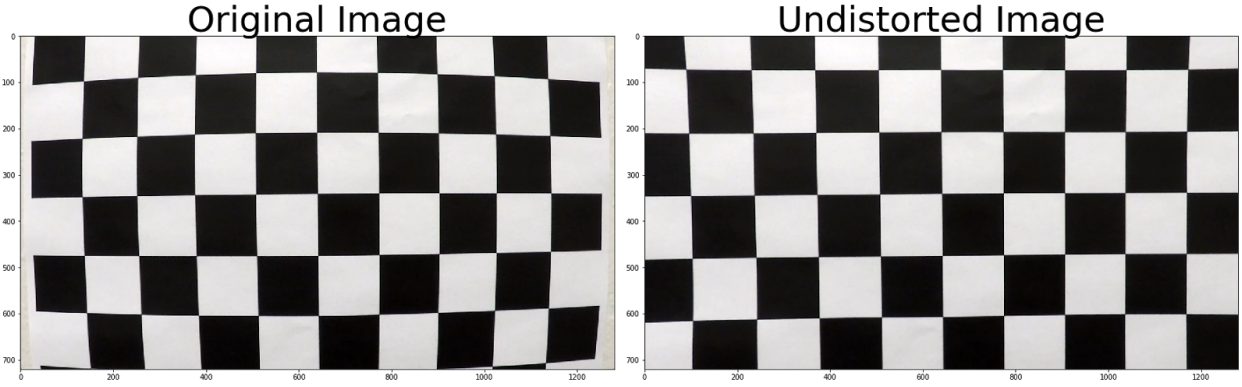
Step 1. Camera Calibration

I use the provided 9x6 chessboard images to calibrate the camera. With the aid of *cv2.findChessboardCorners()*, I can locate the (X, Y) coordinates of a list of points detected in the photos.



Separately, I create a list of (X, Y, 0) coordinates corresponding to a 9x6 grid to indicate the correct locations for these chessboard corners. The list looks like [[0., 0., 0.], [1., 0., 0.], [2., 0., 0], ..., [9, 0, 0], [0, 1, 0], [1, 1, 0], [2, 1, 0],...[9, 1, 0], … [9, 6, 0]]. I pass in these two lists of points to *cv2.calibrateCamera()* to get distortion coefficients. I use these coefficients to calibrate camera images with *cv2.undistort().*

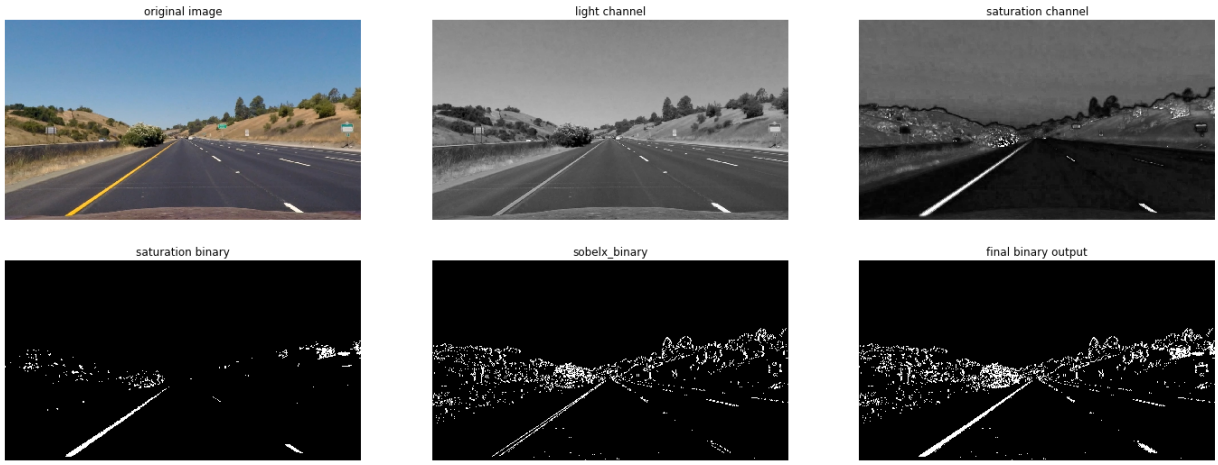
The result looks like this:



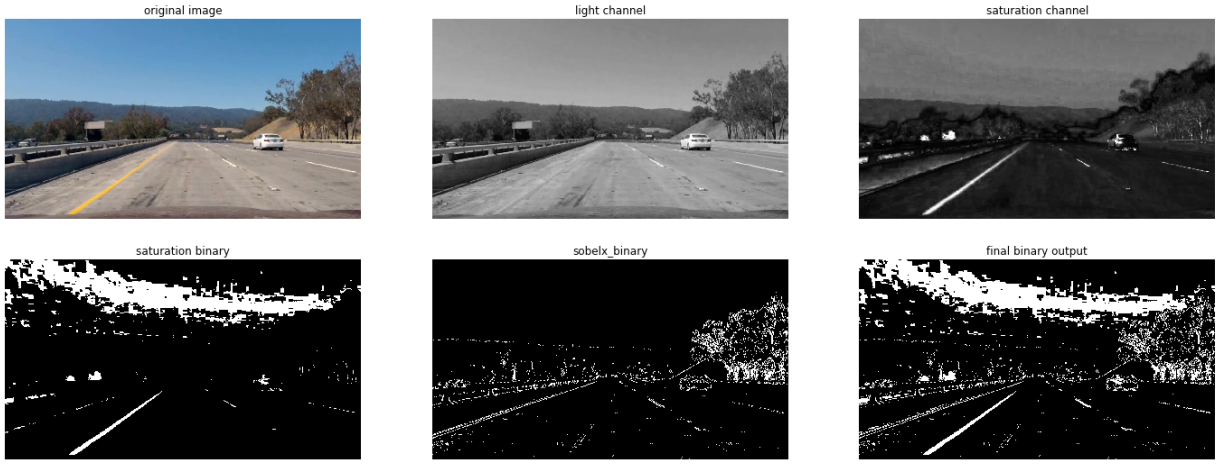
Applying the calibration on a road image:

The difference is minor. It is more obvious on upper left corner. I don't think this step affects the performance of the final output much.

Step 2. Create binary image



Example 1



Example 2

This step is to create a black-and-white binary images to single out the lane lines.

Performing on an undistorted image after step 1, I first convert the image to hue-saturation-light (hsl) scale, and get the light channel and saturation channel. The light channel sometimes fails to recognize the lane line in well lit areas (see example 2 above), while the saturation channel can usually sensitively identify the lane lines. I then convert the saturation channel to a binary image: any pixels with value above 120 will be 1 or else 0 (see lower left images “saturation binary” in the panels above). Note that the noise in other areas in the image isn’t that important to deal with, because after perspective transform (a later step) they will be out of the image frame.

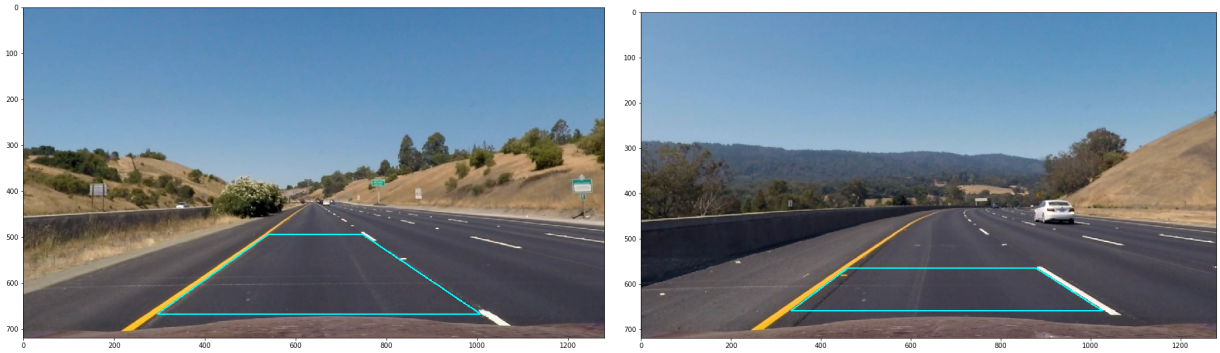
Then I apply the sobel operator on the x direction on the light channel. I then convert the output to binary (sobelx binary) by converting pixels between 20 and 100 to 1. Comparing to the saturation binary, the sobel operator is better picking up white lane lines which usually doesn’t have high saturation but good contrast in lighting with the paved road.

I combined the saturation binary and sobelx binary with a ‘or’ condition: if a pixel is 1 in either images, then it would be 1 in the final output.

Step 3. Perspective transform

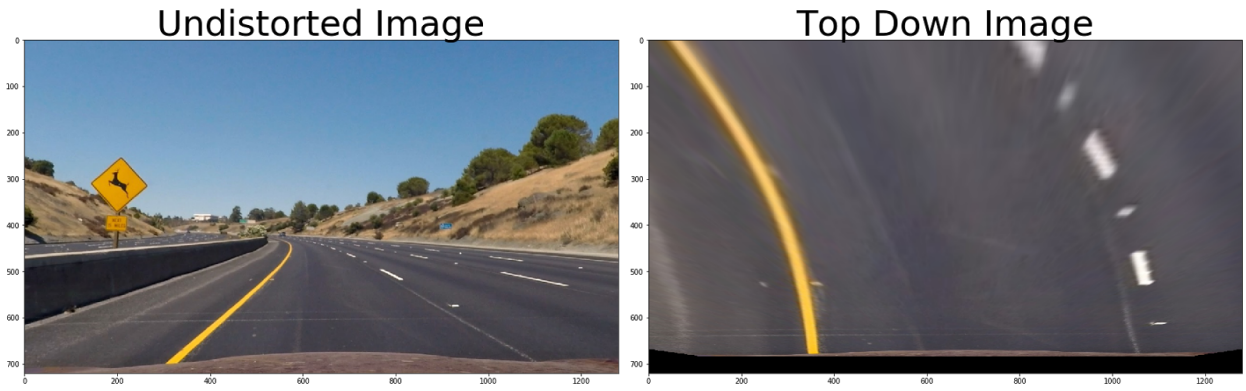
This step is to transform the lanes to a bird-eye view.

I manually identified an area that should be rectangles in a bird eye views and recorded coordinates of the four corners. I performed this on three test images:



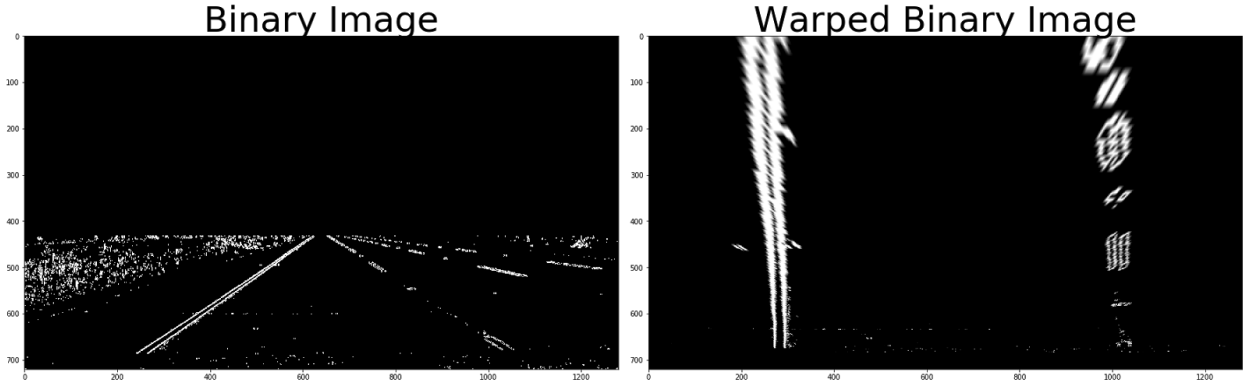
I also create a list of coordinates that correspond to a rectangle, fixing the bottom two corners and y coordinates of all corners of the polygons shown above. Then I used *cv2.getPerspectiveTransform* to get a transformation matrix for each of the three images. Then I averaged the three transformation matrices.

Applying this transformation matrix to test images, we get the following results:



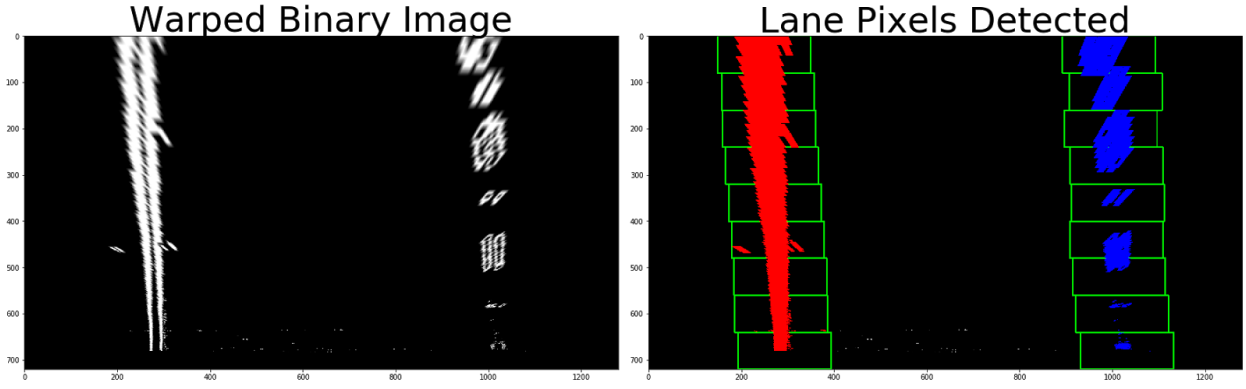
The lines are roughly parallel as expected.

So we apply this to the binary image we got in the previous step and get an warped binary image:



Step 4. Detect lane pixels

The goal of this step is to identify which pixels in the binary image are of lane lines.



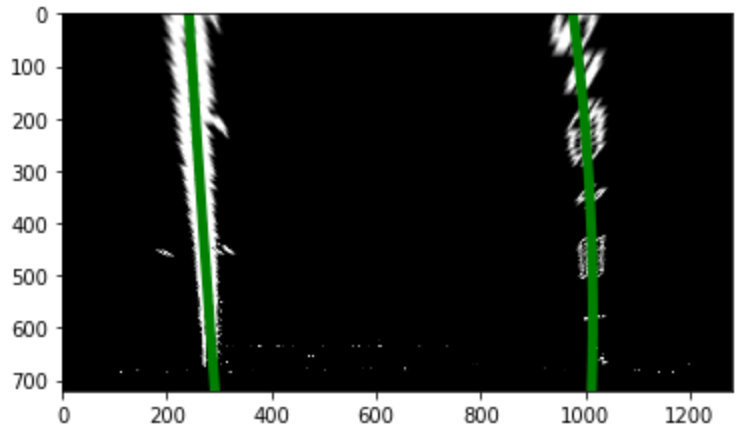
We did this using sliding windows. First we slice the image horizontally into a few strips with equal height. Starting from the bottom, we compute a histogram of the pixels and identify which two x axis positions have the densest pixels in the left and right half. We then draw a box around the center and identify all the pixels inside the box as lane pixels.

Moving on the the next strip, we reuse the two centers we have found in the previous step and box around them. Within the box, we find new centers at where the pixels are most densely when projecting to the X-axis. We will continue the search in the next strip using these new centers.

In practice, I use 4 windows instead of 9, because some dashed lane lines have long gaps in between, and sometimes throw the model off when a window is too narrow and find no pixels in the box.

The output of this step are two lists of points, corresponding to pixels in the left and right lane lines correspondingly.

Step 5. Fit lanes to polynomials



The goal of this step is to fit a second-degree polynomial (parabola) to the lane pixels. It is straightforward with the use of *np.polyfit()*. The template is X = A Y^2 + B Y + C. The output of this step tells us coefficients A, B and C for the left and right lanes.

Step 6. Calculate curvature and position of vehicle

To find the curvature and position of vehicle (with respect to the lane) in meters, we first need to convert pixels to meters. We know a lane is 3.7 meters wide and lane in image is roughly 30 meters long. They correspond to 720 and 700 pixels in the y and x directions respectively.

In the step above, we found X = A Y^2 + B Y + C in pixel unit. Now we need to know X * meter per pixel in X = A' (Y * meter per pixel in Y) ^2 + B' Y * meter per pixel in Y + C'. We see that:

A' = A / (meter per pixel in Y) ^ 2 * meter per pixel in X
B' = B / meter per pixel in Y * meter per pixel in X

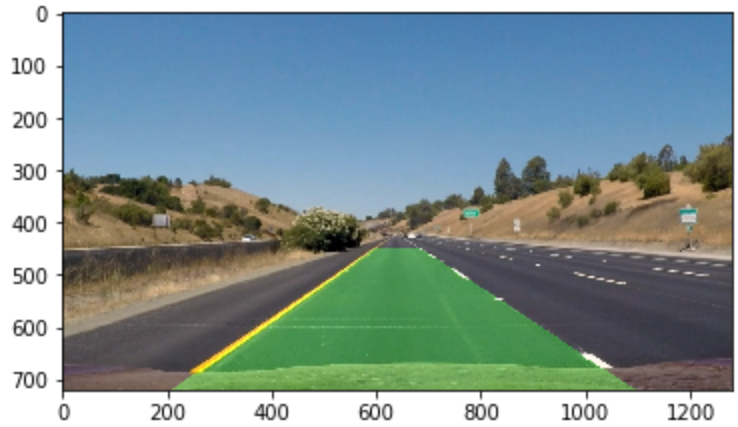
We can get the curvature in meters using the formula:

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

To find the position of the vehicle with respect to the lane, we need to know the x coordinates of the middle of the lane lines at the bottom. So we first find the two intercepts by plugging in y = image height (720) in our fitted polynomials. We then average the two intercepts to get the middle point. We know the camera is mounted at the middle of the vehicle in the x axis, which is image width (1280) divided by 2 which is 640. We subtract the lane middle point from the middle from the image to get the position of the vehicle. We compute all these using A', B' and C' to get a result in meter unit.

Step 6. Unwarp the detected lane boundaries back to the original image.

In the perspective transform step, we can also get the inverse matrix. We simply unwarp the two fitted polynomial lines back to the original image and fill the area in between for visualization purpose.



Discussion

The pipeline is not robust when tested on the harder challenge video. This is because that clip has frequent light and shadow changes, and part of the lane lines at times is not visible with grasses. If I have more time, I would want to use information from a previous frame to find lanes in the current frame. I will also use a dynamic threshold for the binary image creation: when the lighting condition is bad, I will assign lower threshold to detect pixels in the region where I found the lanes in the previous frame.

Acknowledgement

I reused some code provided by the Udacity course. I in addition used a script to translate the video into image frames based off this stackoverflow post.