Writeup

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.

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

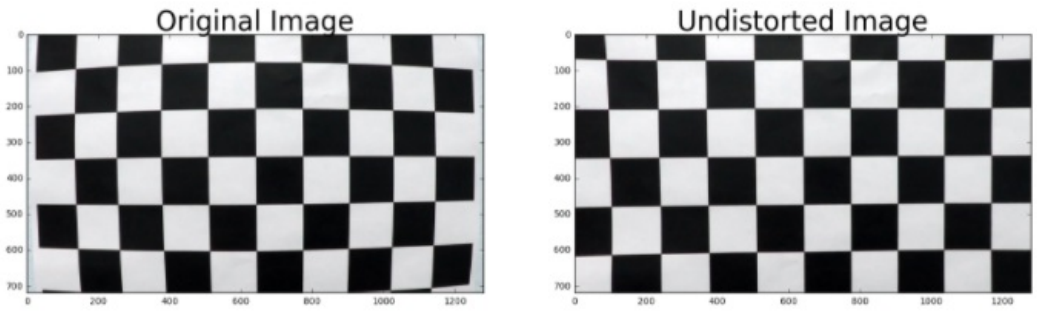
Camera Calibration

1. have the camera matrix and distortion coefficients been comuted correctly and checked on one of the calibartion images as a test

The code for this step is contained in the first code cell of the IPython notebook located in "./P4*Advanced*Lane_Finding.ipynb" .

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 result:



Pipeline (single images)

1. An example of a distortion-corrected image.

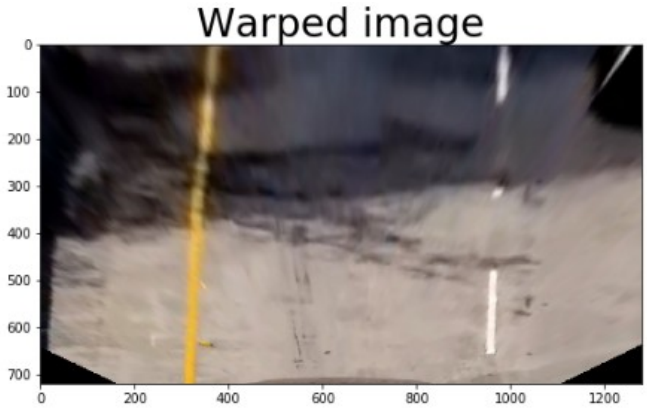
To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like this one:

2. Apply a Perspective Transform

the code for my perspective transform is includes a function called warp(),The function takes as inputs an image, as well as source (src) and destination (dst) points. I chose the hardcode the source and destination points in the following manner:

```
# Four source coordinates
src = np.float32([[753, 478],
                  [1092, 670],
                  [216, 688],
                  [549, 483]])

# Four desired coordinates
dst = np.float32([[997, 1],
                  [982, 712],
                  [316, 712],
                  [366, 1]])
```

I verified that my perspective transform was working as expected by drawing the src and dst points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.

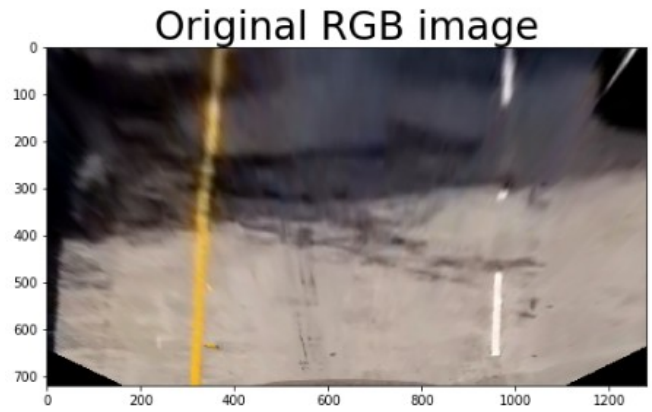


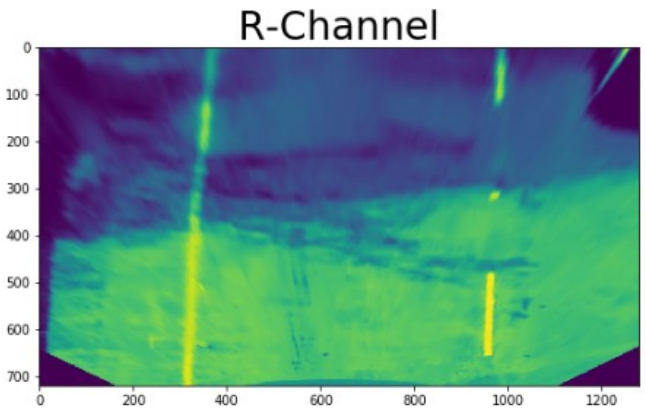
3. Create a Thresholded Binary Image

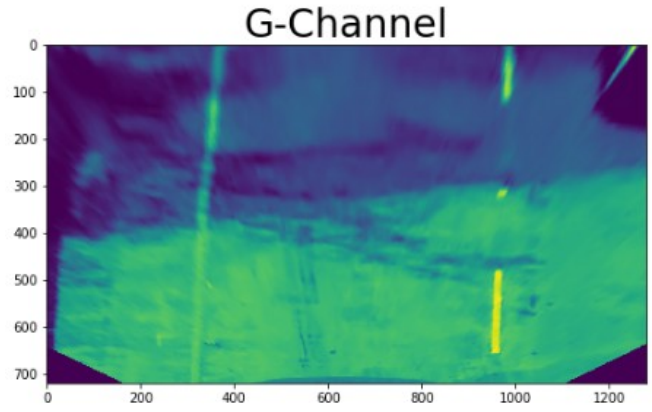
I used LAB B color channel and L color channel combination of HLS to generate a binary image. Here's an example of my output for this step.

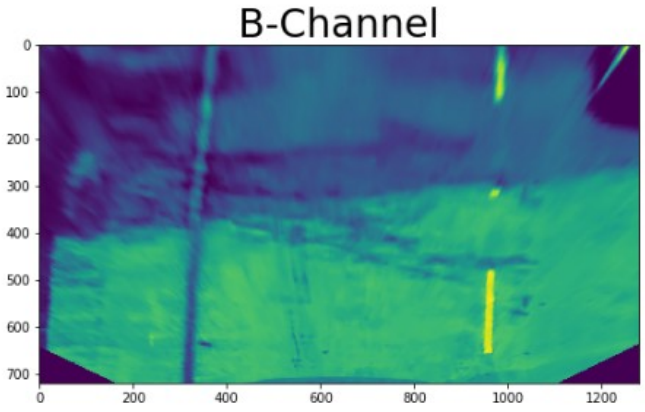
RGB color space:

RGB is red-green-blue color space. You can think of this as a 3D space, in this case a cube, where any color can be represented by a 3D coordinate of R, G, and B values. For example, white has the coordinate (255, 255, 255), which has the maximum value for red, green, and blue.



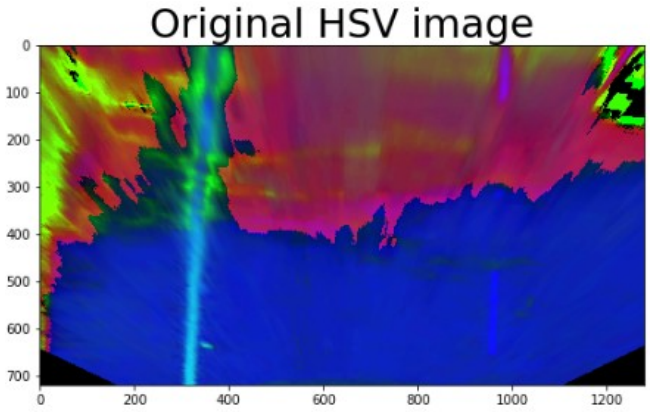


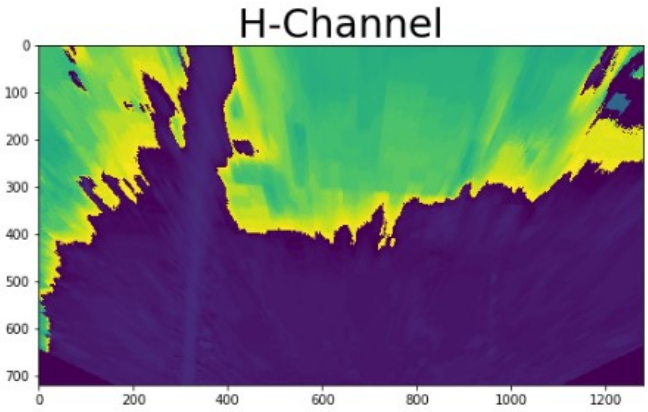


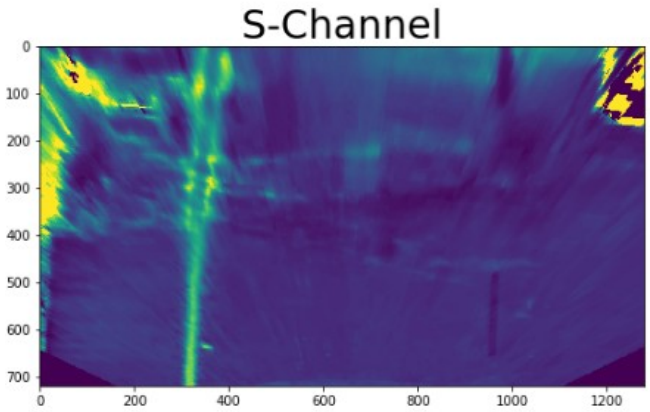


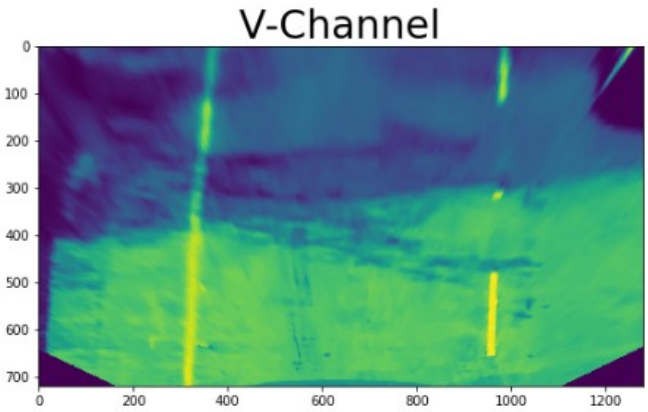
HSV color space:

HSV (Hue, Saturation, Value):The HSV representation models the way paints of different colors mix together, with the saturation dimension resembling various shades of brightly colored paint, and the value dimension resembling the mixture of those paints with varying amounts of black or white paint.



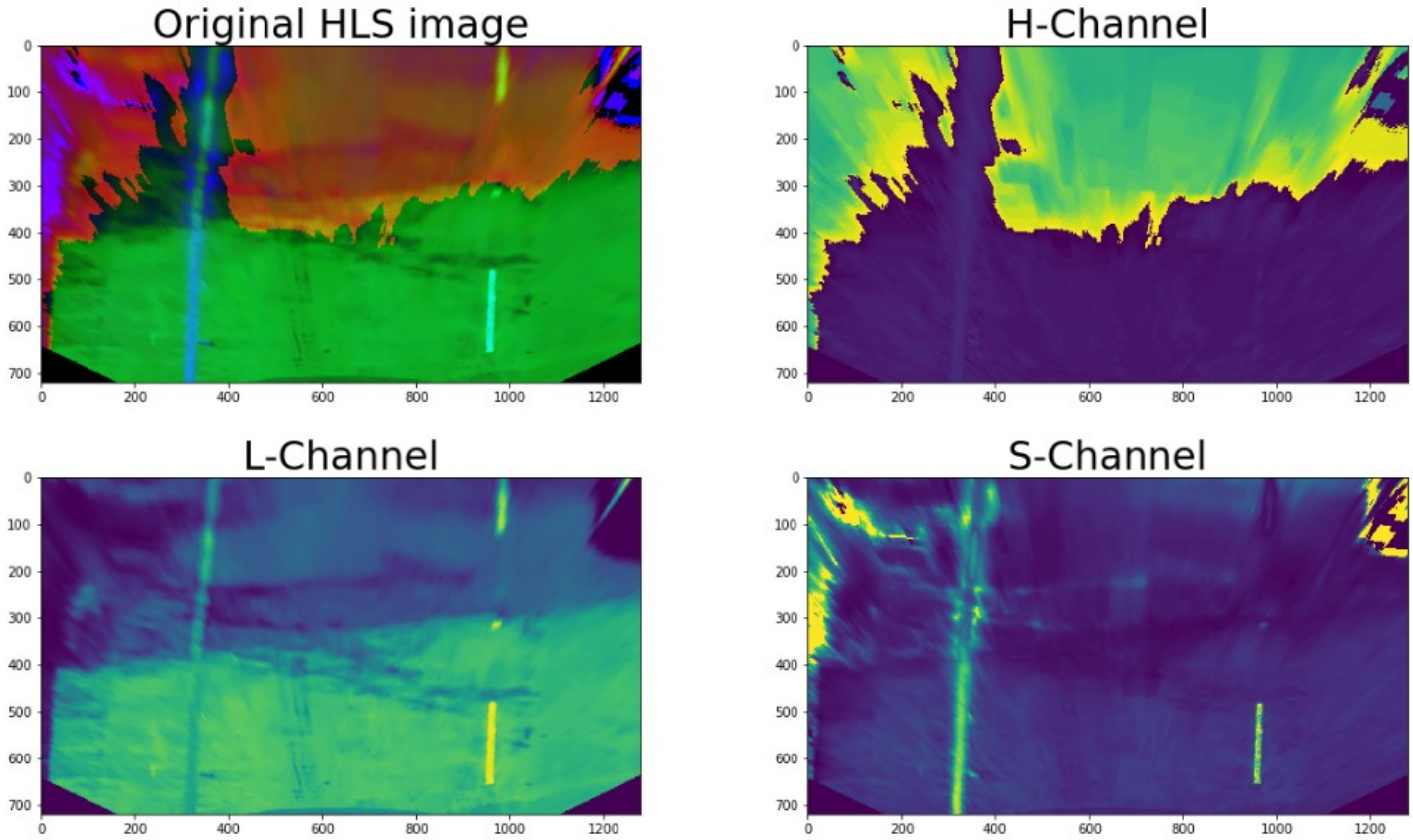






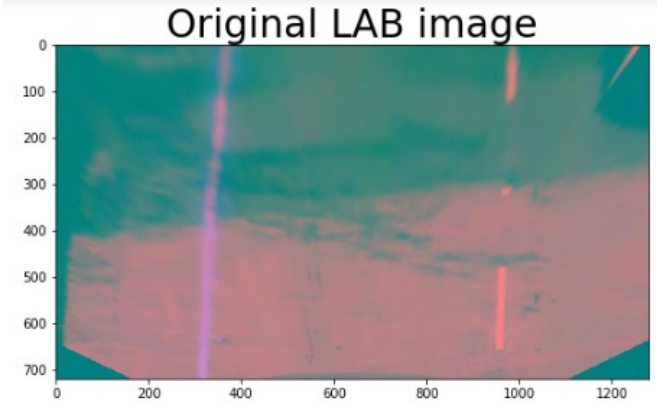
HLS color space:

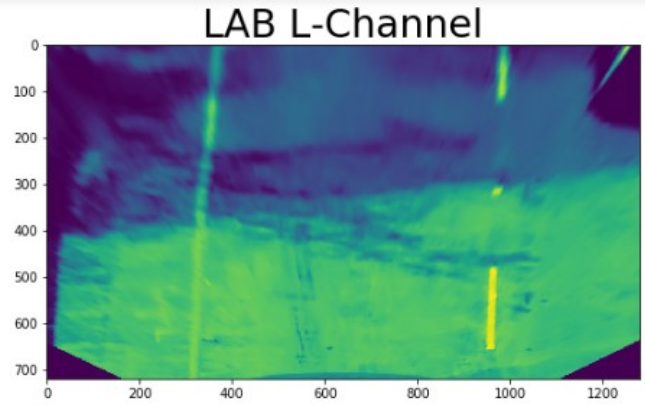
HSL (Hue, Saturation, Lightness):The HSL model attempts to resemble more perceptual color models such as NCS or Munsell, placing fully saturated colors around a circle at a lightness value of 1/2, where a lightness value of 0 or 1 is fully black or white, respectively.

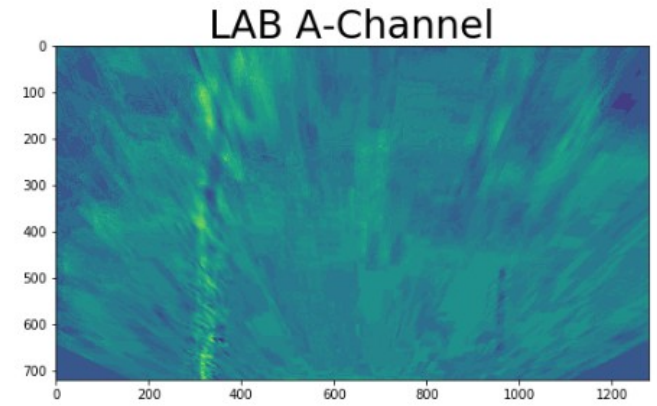


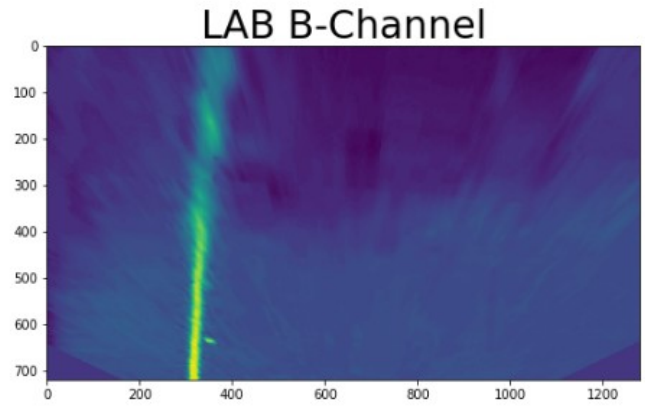
LAB color space:

The Lab color space describes mathematically all perceivable colors in the three dimensions L for lightness and a and b for the color opponents green–red and blue–yellow.



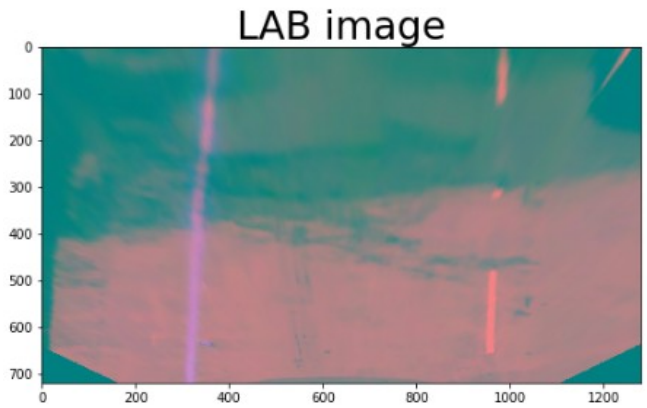


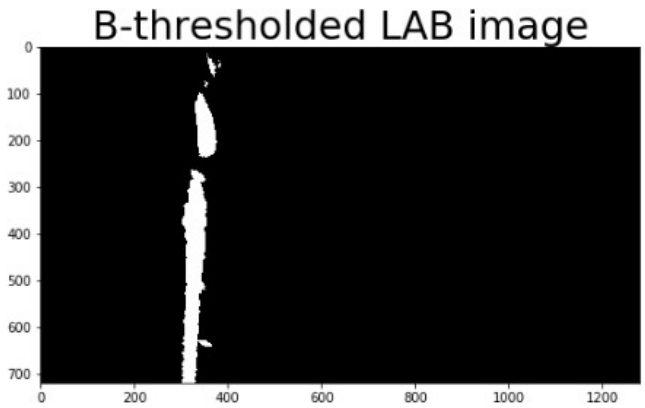




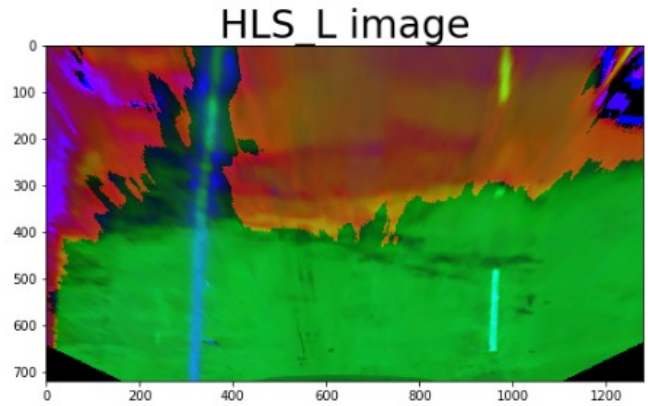
From the above pictures, we can see the B color channel of LAB, showing the obvious left lane line, while the L color channel of HLS shows obvious right lane line, so the combination of the two is used to generate binary images.

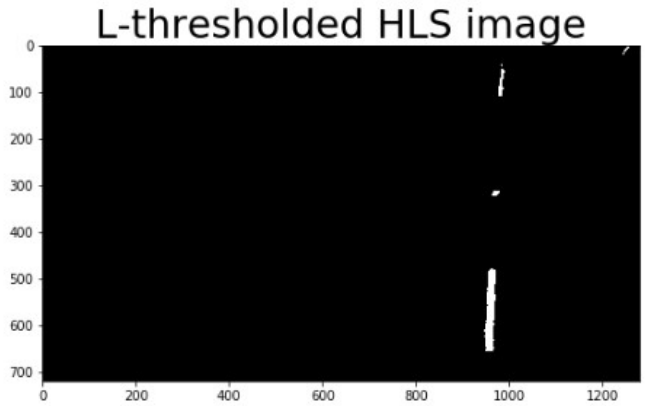
The B color channel of LAB:



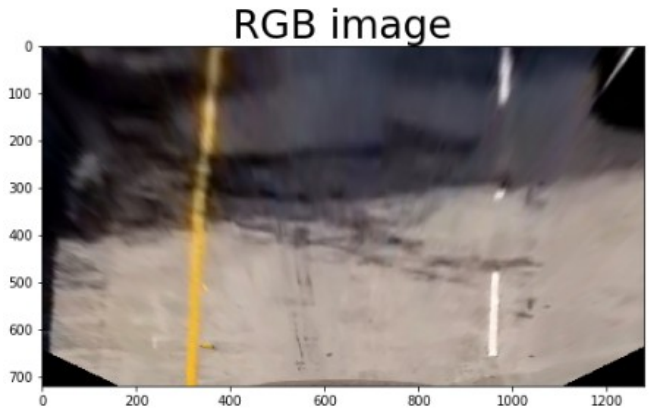


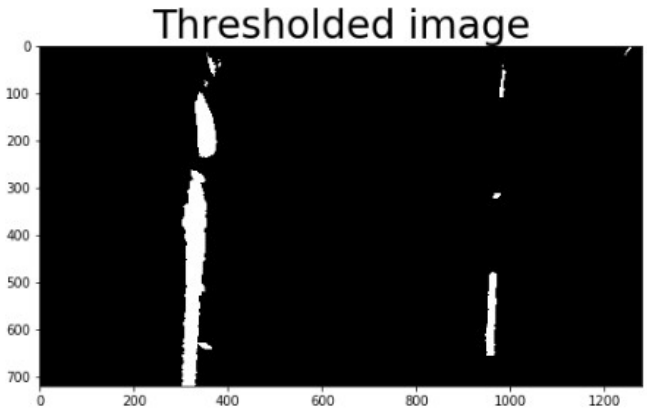
The L color channel of HLS:





Combined Two Color Space Threshold:

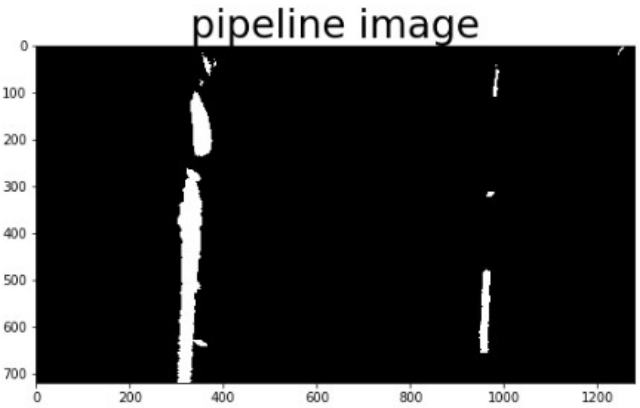




For this project using the above two color channel can be perfect to get around the lane line, Sobel calculation there will be some noise interference, but Sobel calculation in the face of complex effect will be better, this project will not display.

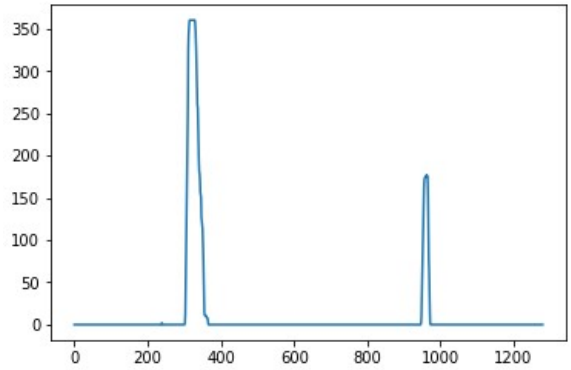
4. Create the Image Processing Pipeline Function

This function can directly calculate the final binary image we need:



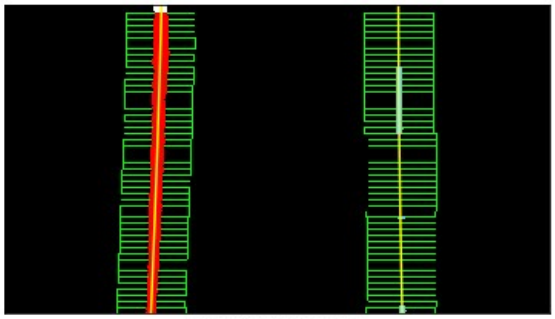
5. Sliding Window Search and Detect The Lane Lines

I first take a histogram along all the columns in the lower half of the image like this:



With this histogram I am adding up the pixel values along each column in the image. In my thresholded binary image, pixels are either 0 or 1, so the two most prominent peaks in this histogram will be good indicators of the x-position of the base of the lane lines. I can use that as a starting point for where to search for the lines. From that point, I can use a sliding window, placed around the line centers, to find and follow the lines up to the top of the frame.

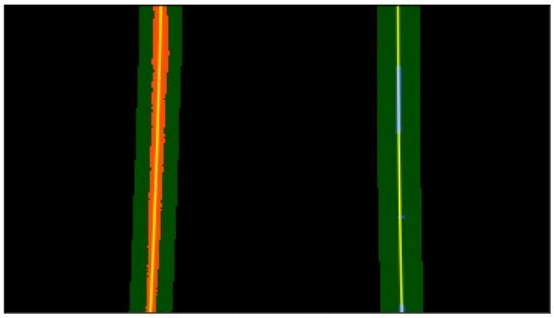
Original image



Sliding window

once you know where the lines are in one frame of video, you can do a highly targeted search for them in the next frame. This is equivalent to using a customized region of interest for each frame of video, and should help you track the lanes through sharp curves and tricky conditions.

Original image



Polyfit using previous fit

6. Measuring Curvature

step 1.The radius of curvature at any point x of the function x=f(y) is given as follows:

$$R_{curve} = \frac{[1 + (\frac{dx}{dy})^2]^{3/2}}{\left|\frac{d^2x}{dy^2}\right|}$$

In the case of the second order polynomial above, the first and second derivatives are:

$$f'(y) = \frac{dx}{dy} = 2Ay + B$$

$$f''(y) = \frac{d^2x}{dy^2} = 2A$$

So, our equation for radius of curvature becomes:

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

In the code, the specific equation is as follows:

`((1 + (2*left_fit_cr[0]*y_eval*ym_per_pix + left_fit_cr[1])**2)**1.5) / np.absolute(2*left_fit_cr[0])`

`left_fit_cr[0]`: on behalf of A. is the first coefficient (the y-squared coefficient) of the second order polynomial fit.

`left_fit_cr[1]`: on behalf of B. is the second coefficient.

`y_eval*ym_per_pix`: on behalf of y. `y_eval` is the y position within the image upon which the curvature calculation is based. `y_meters_per_pixel` is the factor used for converting from pixels to meters.

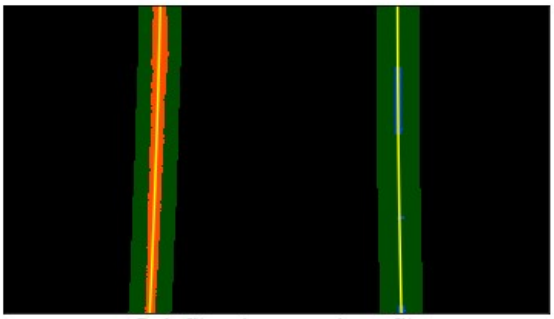
step 2.Then calculate the code of the vehicle in the center of the lane:

```
l_fit_x_int = l_fit[0]*h**2 + l_fit[1]*h + l_fit[2]

r_fit_x_int = r_fit[0]*h**2 + r_fit[1]*h + r_fit[2]

lane_center_position = (r_fit_x_int + l_fit_x_int) /2

center_dist = (car_position - lane_center_position) * xm_per_pix
```

`l_fit_x_int` and `r_fit_x_int` are the x-intercepts of the left and right fits.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).

```
Radius of curvature for example: 2719.57596707 m, 3226.04186033 m
Distance from lane center for example: 0.00313699671096 m
```

Original image



Polyfit using previous fit

7.Drawing the Lane Boundaries

First, create the draw*lane function: Draw the detected lane over the input image. Then create the write*data function: Write the lane curvature and vehicle position over the input image.

Original image

Drawing Lane and parameters

---

video_processor (video)

1.From the input to the picture direct output has processed the completed lane, curvature, vehicle position picture.

Here's a link to my video result

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?

Met a two color channel merging, validated test images file all images, including straight_lines2. JPG this image is all black, always thinking, finally found the problem, because in the value of the picture in more hours, after normalization, the value of the image is roughly 1.0, so a merger would see two lanes around the line, add judgment (if np. Max (img) > 180), let the pictures without normalized maximum value is less than 180, is to solve the problem.