Advanced Lane Finding

By Seonman Kim

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.

Code Summary

I created 'Advance Line Finding-Final.ipynb' that includes all the functions and codes used to execute the lane finding algorithm and to create a video as an output.

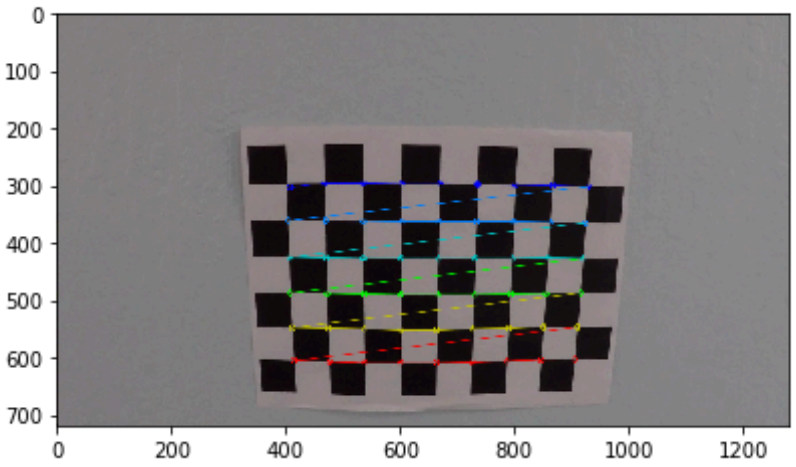
Camera Calibration

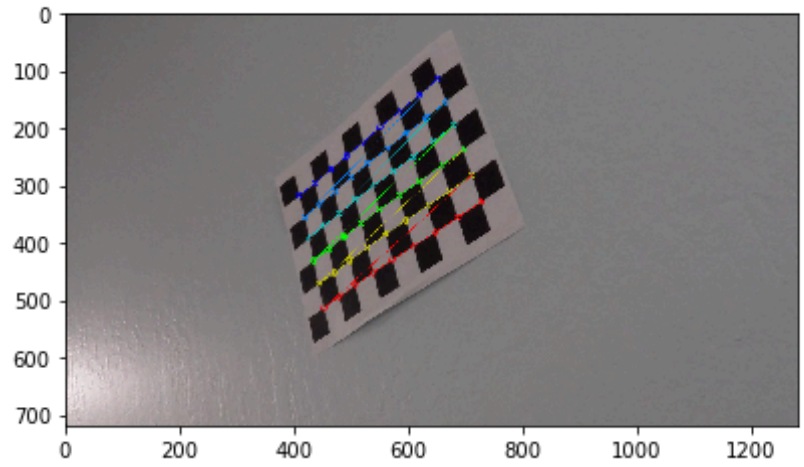
1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

The camera calibration is as follows:

- Load all the images used for calibration. ('./camera_cal/calibration*.jpg')
- For each image:
  - convert the image to GRAY color
    - GRAY = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
  - Find the chessboard corners
    - ret, corners = cv2.findChessboardCorners(gray, (nx, ny), None)
    - We save corner points (corners) for CameraCalibration later.
  - If ret == True, draw and display the corners
    - cv2.drawChessboardCorners(img, (nx, ny), corners, ret)
- Do camera calibration given object points and image points
  - ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, img_size,None,None)
- We save mtx (calibration matrix) and dist (distortion coefficients) for later use.

The following is a sample output of Camera calibration:



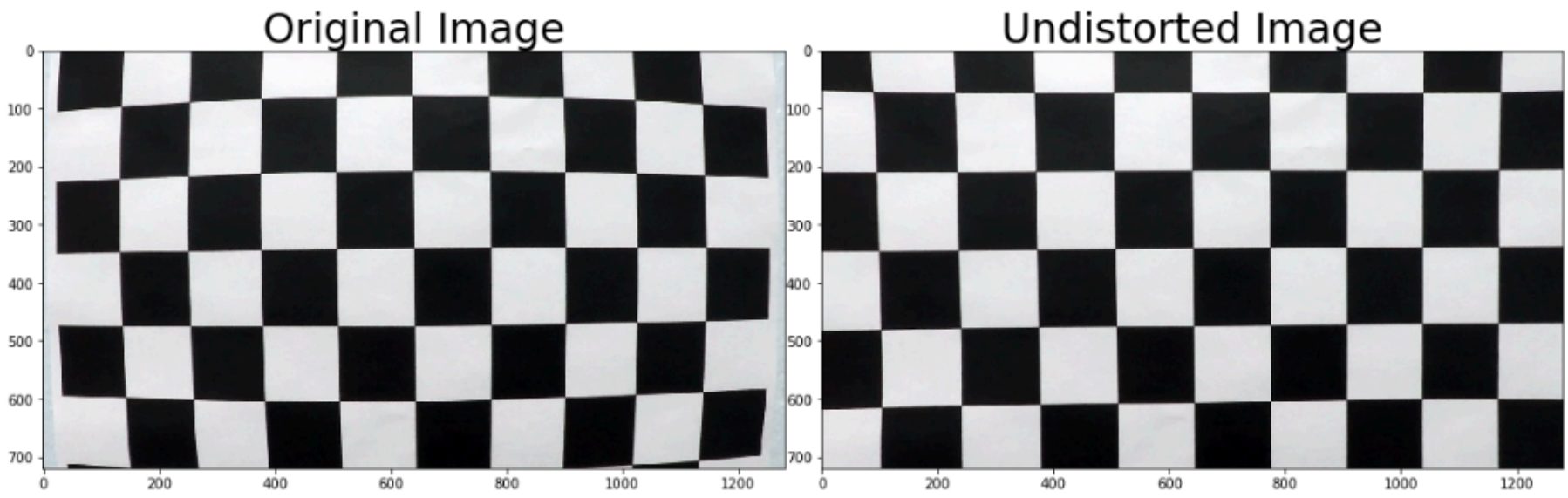


Undistort images

Using the camera calibration matrix and distortion coefficient obtained from above, we could undistort images using:

```
dst = cv2.undistort(img, mtx, dist, None, mtx)
```

The followings are the sample outputs:



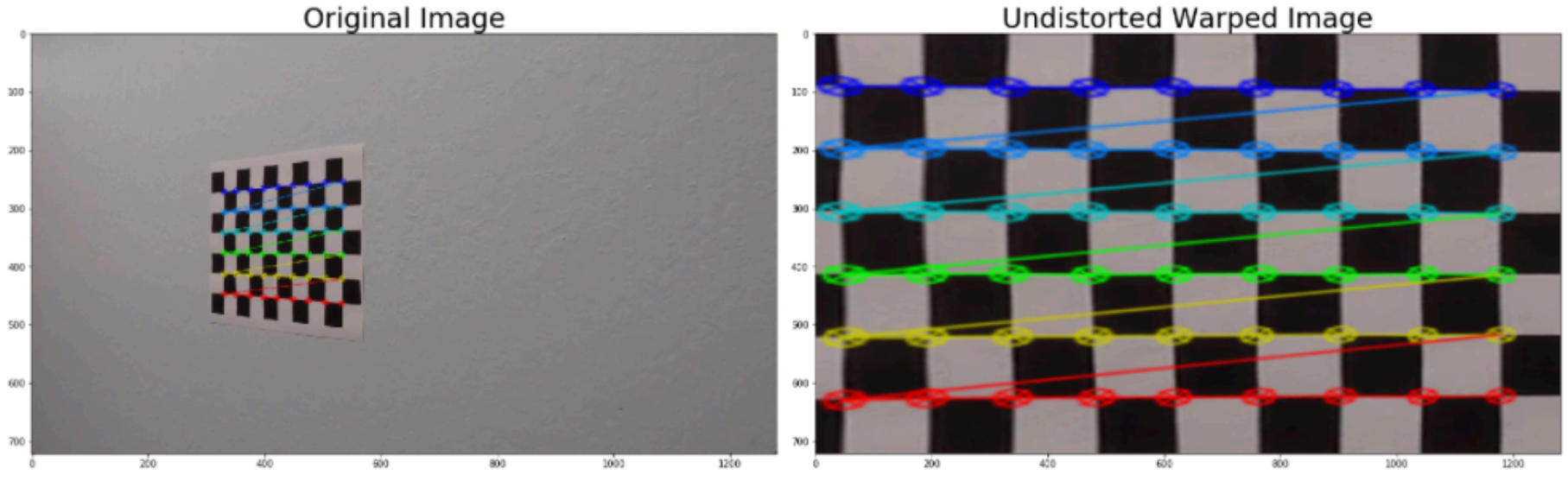
Warp images using perspective transformation

Warping the images is done by selecting a set of source and destination points, and then applying the points to `getPerspectiveTransform()` and `warpPerspective()` functions.

```
# Source and destination points
src = np.float32([corners[0], corners[nx-1], corners[-1], corners[-nx]])
dst = np.float32([[offset, offset], [img_size[0]-offset, offset],
                  [img_size[0]-offset, img_size[1]-offset],
                  [offset, img_size[1]-offset]])

# Given src and dst points, calculate the perspective transform matrix
M = cv2.getPerspectiveTransform(src, dst)
# Warp the image using OpenCV warpPerspective()
warped = cv2.warpPerspective(img, M, img_size)
```

Here is a sample output of warping with perspective transformation.



Pipeline (single images)

1. Distortion Correction

The first step in the processing pipeline was to undistort the input image using the camera calibration matrix and distortion coefficients that we obtained from the previous Camera Calibration step.

2. Perspective Transformation

I found that applying the perspective transform first gave better results and made it easier to see what the polynomial fits were trying to fit to.

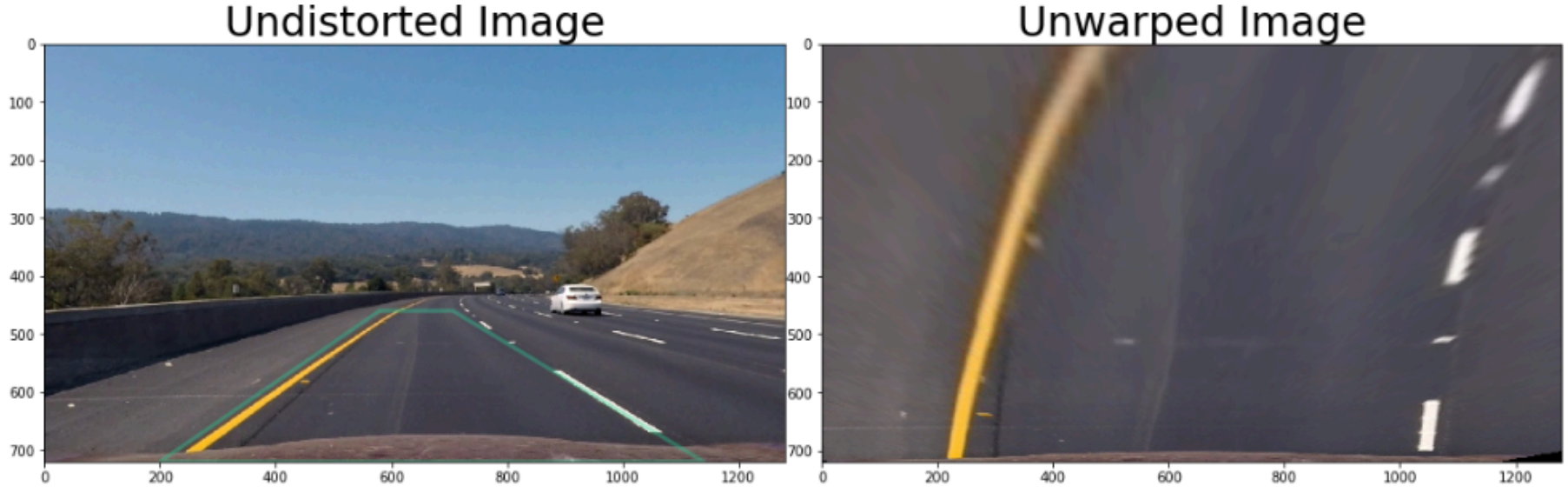
As described above, we have used `getPerspectiveTransform()` and `warpPerspective()` functions to perform perspective transformation. The following source and destination points were used:

```
src = np.float32([corners[0], corners[nx-1], corners[-1], corners[-nx]])
dst = np.float32([[offset, offset], [img_size[0]-offset, offset],
                  [img_size[0]-offset, img_size[1]-offset],
                  [offset, img_size[1]-offset]])
```

This resulted in the following source and destination points:

Source	Destination
580, 460	200, 100
200, 720	200, 720
706, 460	1040, 100
1140, 720	1040, 720

I verified that the 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. Gradients and color transformations for thresholded binary image

I used several different transformations to create the final binary_warped image.

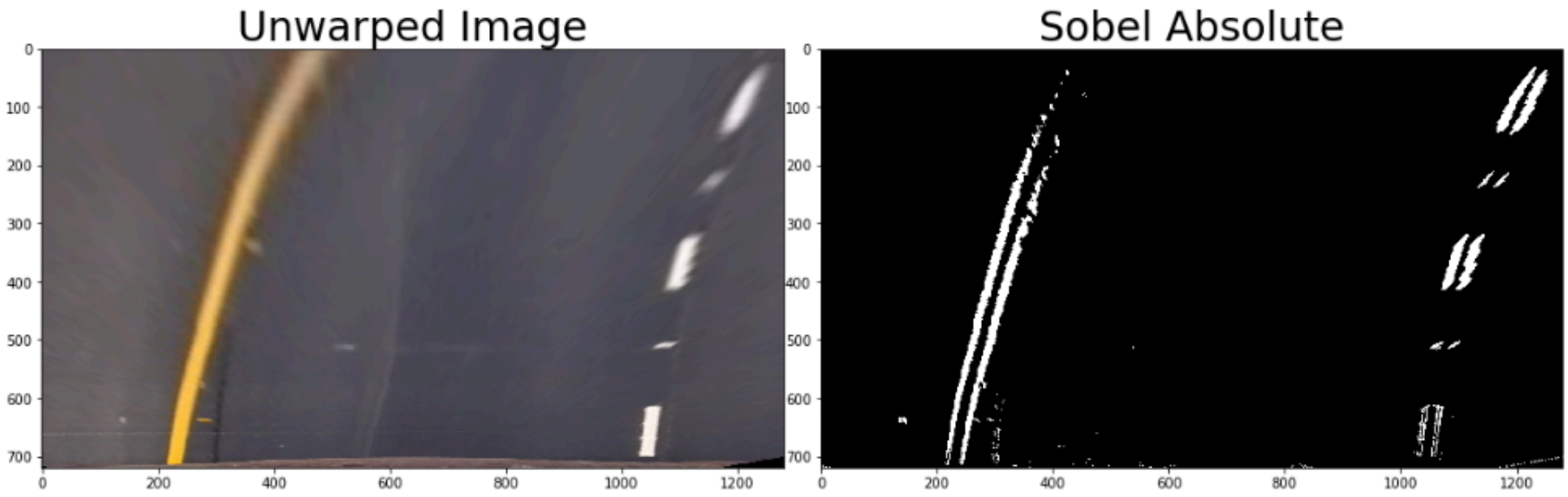
I used python library `ipywidgets` and `interact_manual` functions to interactively test multiple threshold values and find the paramters for the best output.

I found the following transformation and threshold made the best result.

Absolute Sobel X

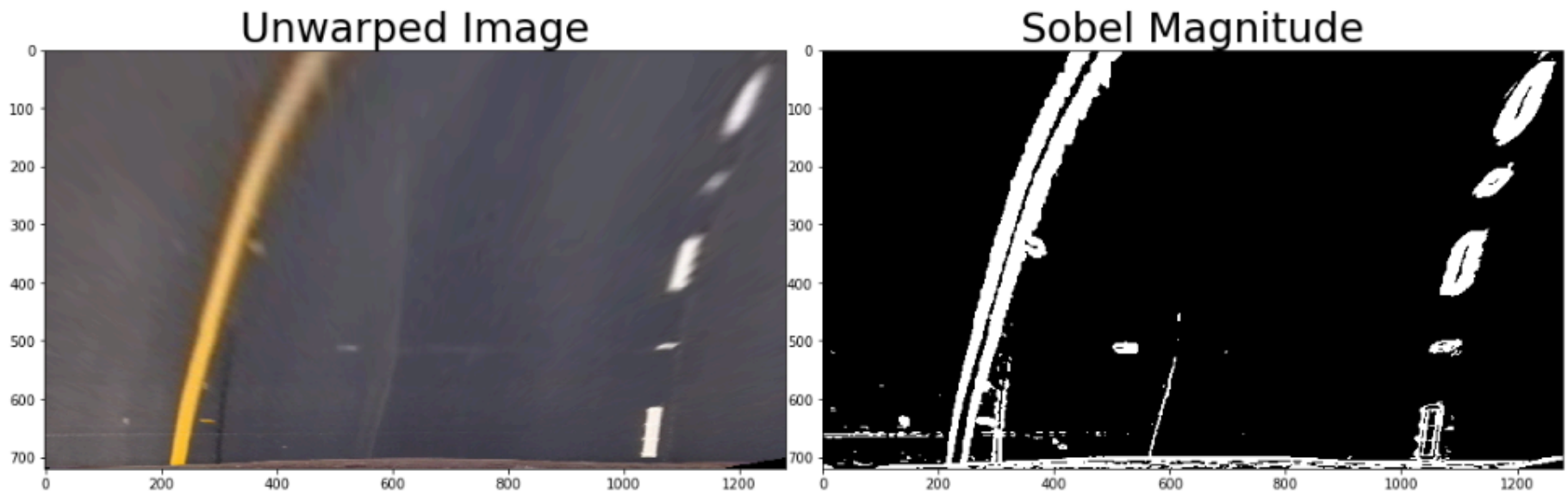
- Min threshold: 20

- Max threshold: 120



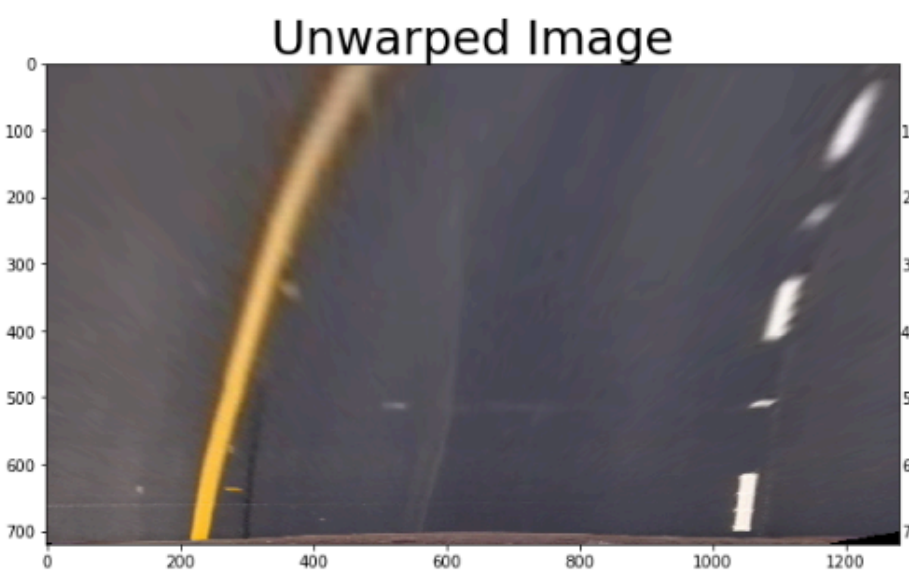
Magnitude of the gradient

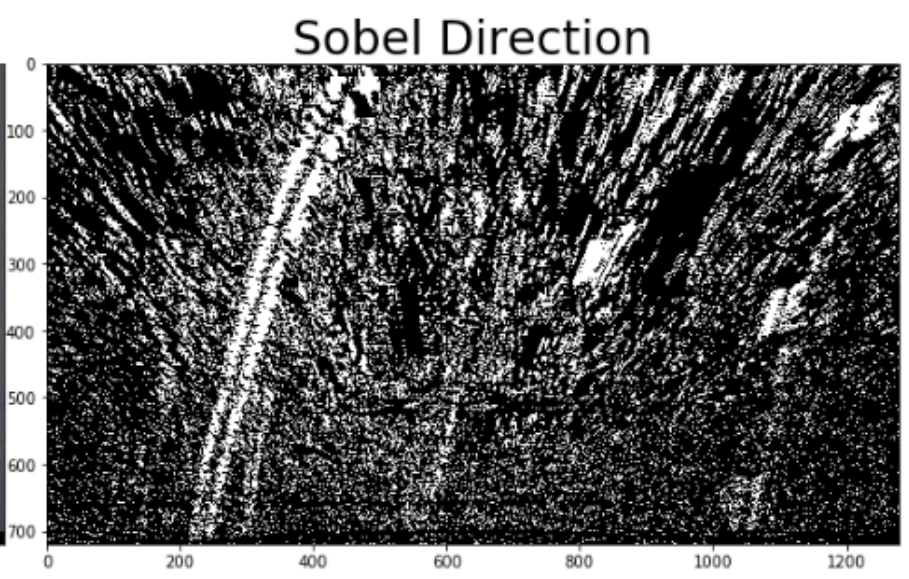
- Kernel size: 15
- Min threshold: 10
- Max threshold: 150



Sobel Direction

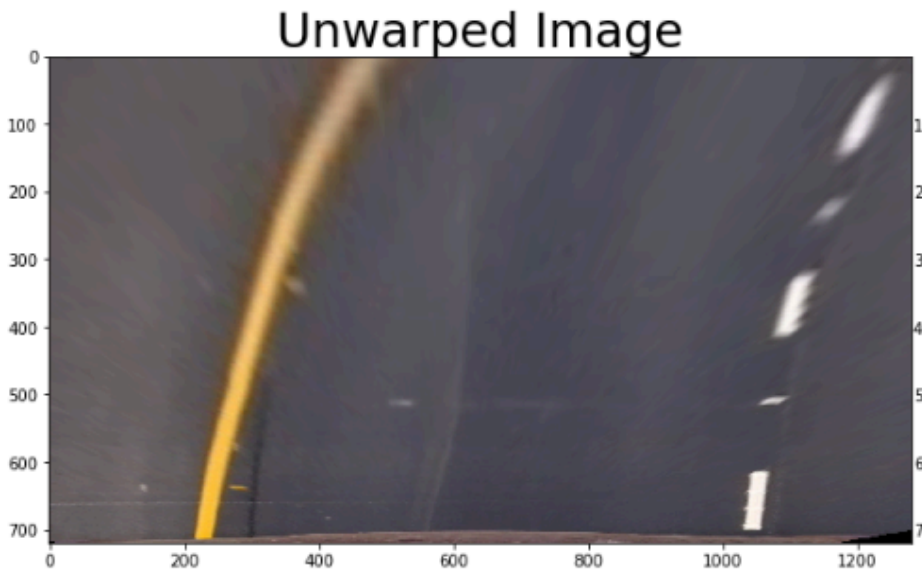
- Kernel size: 5
- Min threshold: 0.15
- Max threshold: 0.60

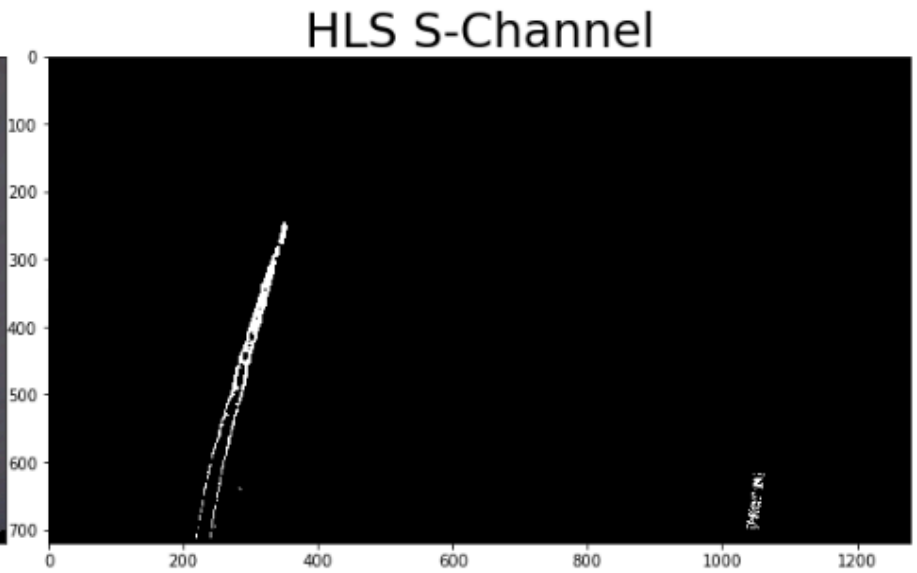




Color HLS S Channel

- Min threshold: 170
- Max threshold: 220





Combining the transforms

We didn't use abs_sobel_thresh on y axis, because it didn't help much.

We combined four transformations to create a binary_warped image, as follows:

```
    # Perspective Transform
    img_unwarp, M, Minv = unwarp(img_undistort, src, dst)

    gradx = abs_sobel_thresh(img_unwarp, orient='x', sobel_kernel=3, thresh=
(20, 120))
    # grady = abs_sobel_thresh(bird, orient='y', sobel_kernel=ksize, thresh=
(20, 100))
    mag_binary = mag_thresh(img_unwarp, sobel_kernel=15, mag_thresh=(10, 150
))
    dir_binary = dir_threshold(img_unwarp, sobel_kernel=5, thresh=(0.16, 0.6
1))

    # HLS S-Channel
    hls_s_binary = hls_sthresh(img_unwarp, (170, 220))

    binary_warped = np.zeros_like(dir_binary)
    # combined[((gradx == 1) & (grady == 1)) | ((mag_binary == 1) & (dir_bin
ary == 1))] = 1
    binary_warped[((gradx == 1)) | ((mag_binary == 1) & (dir_binary == 1)) |
(hls_s_binary == 1)] = 1
```

Sample outputs








4. Lane Line Curve Fitting

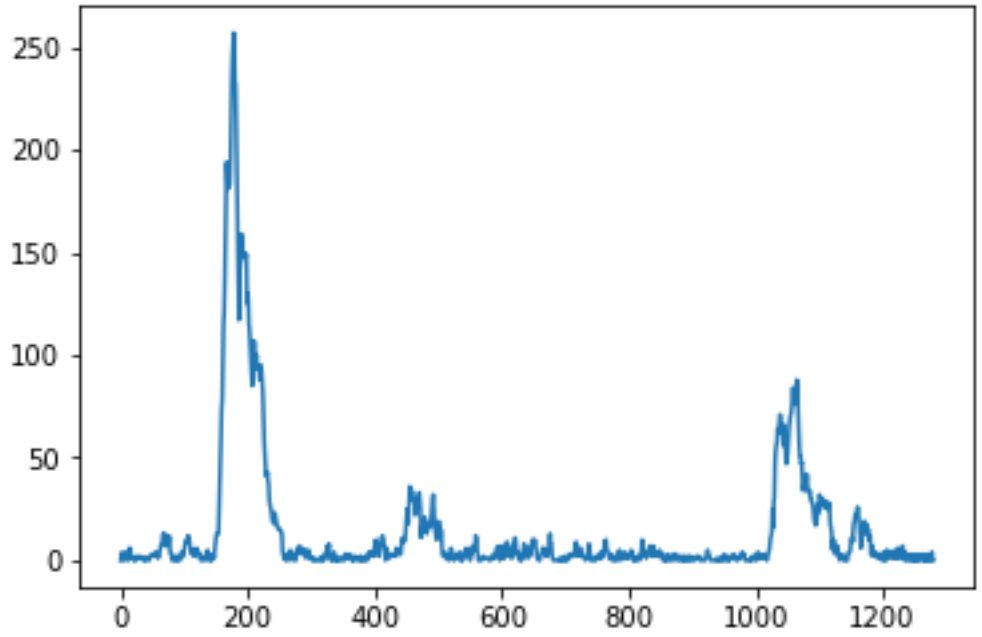
After creating the binary_warped image, the next step was to apply a histogram to the image to determine the peak locations of pixels in the image. Once the histogram was found, a sliding window technique was used to isolate the lane line pixels in the image.

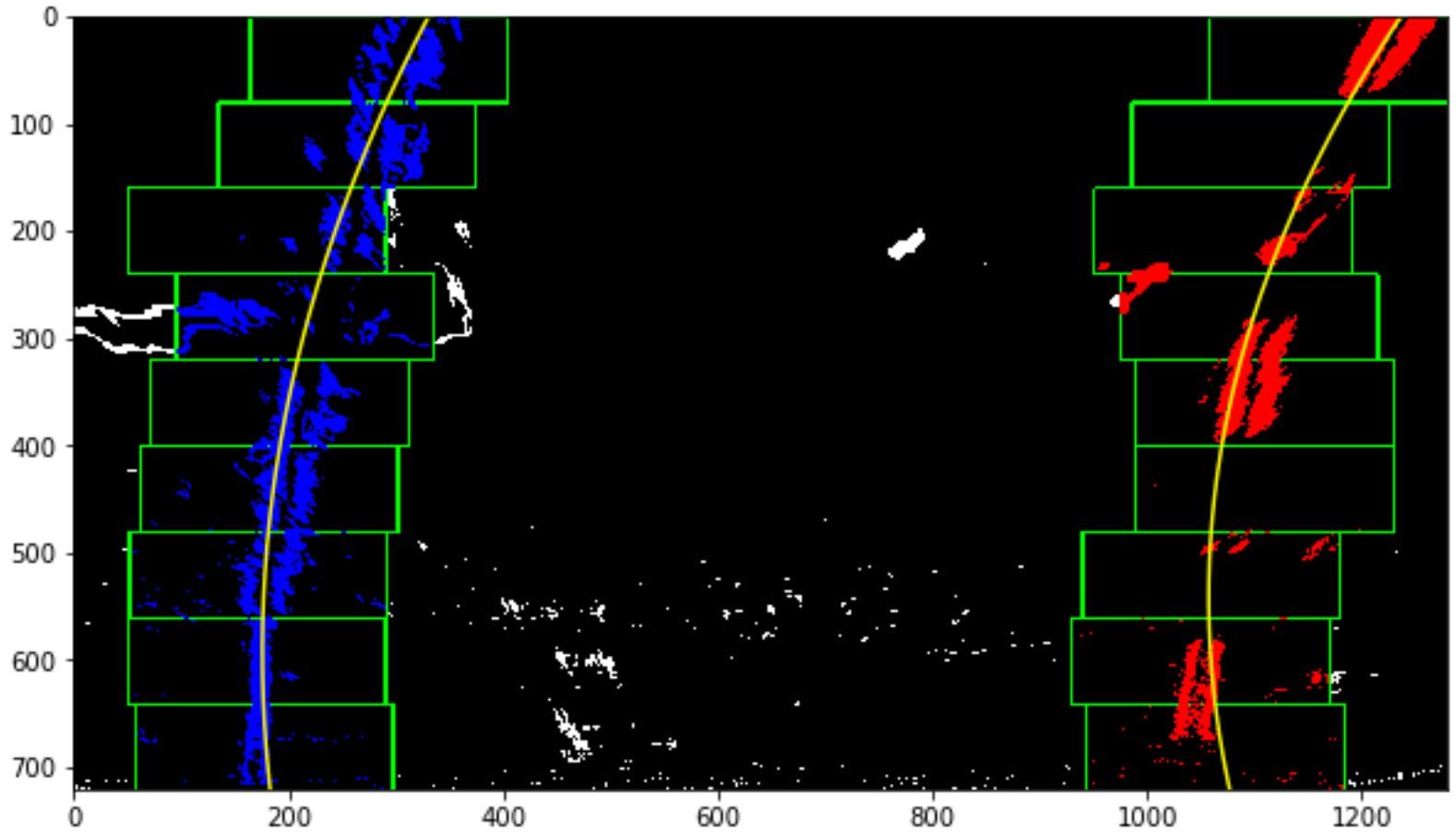
Histogram and Sliding Windows

This was a critical step in the code for determining the proper left and right lane lines. The sliding windows (rectangles) start with their center at the X locations were the histogram peaks occur. The sliding windows are then extended 120 pixels in the +/- X directions (margin = 120) and look for a minimum of 60 pixels. If this condition is met, then the next sliding window will be cetered at the average X location of these pixels, otherwise they will continue to center around the histogram peaks. Once the lane pixel indices and X,Y locations are determined, a 2nd order polynomial fit was used to fit lines to these pixels for both the left and right lane lines using the numpy polyfit function.

```
left_fit = np.polyfit(lefty, leftx, 2)
right_fit = np.polyfit(righty, rightx, 2)
```

An example of the histogram and the sliding window fit is shown below.





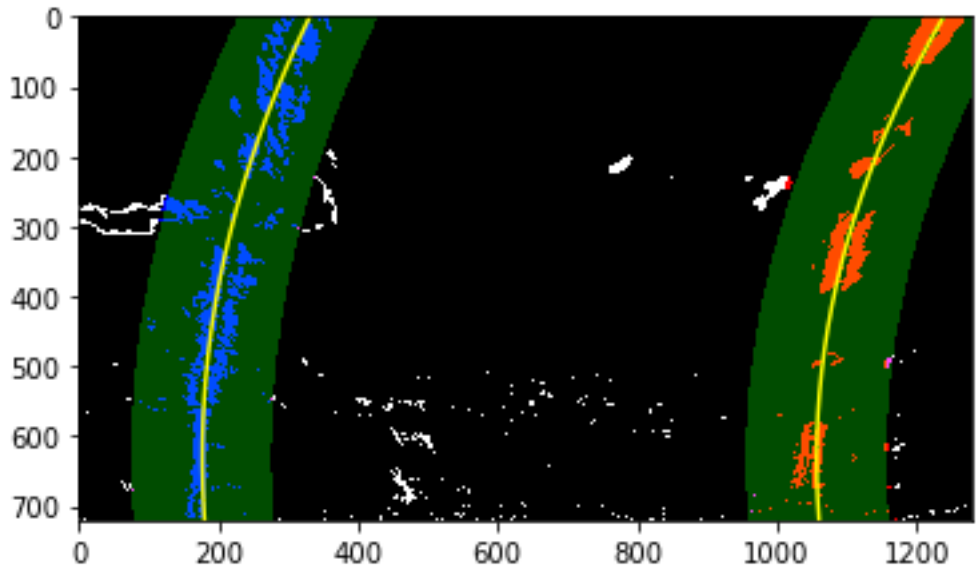
Still within the `sliding_windows()` function, the next steps are where I spent the most time and achieved the biggest gains in results. I used several different techniques to check that the determined polynomial fits for each lane line were realistic.

1. Limits to Polynomial Coefficients
    - I created several made up curves that were similar in curvature to what the lane lines would be. Using this information I limited the values that the squared and linear terms could take.
2. Confidence of left/right lanes
    - Based on the number of pixels found corresponding to each lane line, I implemented a “confidence” metric. To determine the optimal number of pixels found for the left and right lanes, I compared the number of pixels found during the relatively easy straight lines images with images that had high curvature, shadows and other road markings.
3. Lane Width Check
    - The distance lane width should not change drastically, it should constantly be around 800 pixels in my implementation. I applied a check for this and if the lane lines were too far apart or too close together, I would discard the information from the lane lines for that timestep if the condition was not met.
4. Averaging the Fits
    - I stored the fits from each images an applied an averaging (smoothing) over the most recent 20 images. Since the video is taken at 25 fps, this corresponds to smoothing of approximately 1 second of images. This helped to stablize the line fits and reduce the jitter that could be seen in the video.

Find Lines

`find_lines()` used the curve fits from the previous timestep to determine the new lines. Once the sliding window techqiue is performed in the first frame, the base of the left/right lanes nearest to the vehicle should not change greatly from frame to frame. Thus, the previous step's curve fit along with a margin was used to

determine the current timestep fits. An example of this technique is shown below with the search area around the polynomial curves highlighted in green:



Visualizing the lanes and calculating lane information

A function called rad_curve() draws the lane lines on the image, determines the radius of curvature of the lines and determines the car's position relative to the center of the lanes. The polynomial fits have the form:

$$f(y) = Ay^2 + By + C$$

From this equation, the radius of curvature can then be calculated at any point x of the function x=f(y) is given as follows:

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

The first and second derivatives of the second order polynomial used to fit the lane lines are:

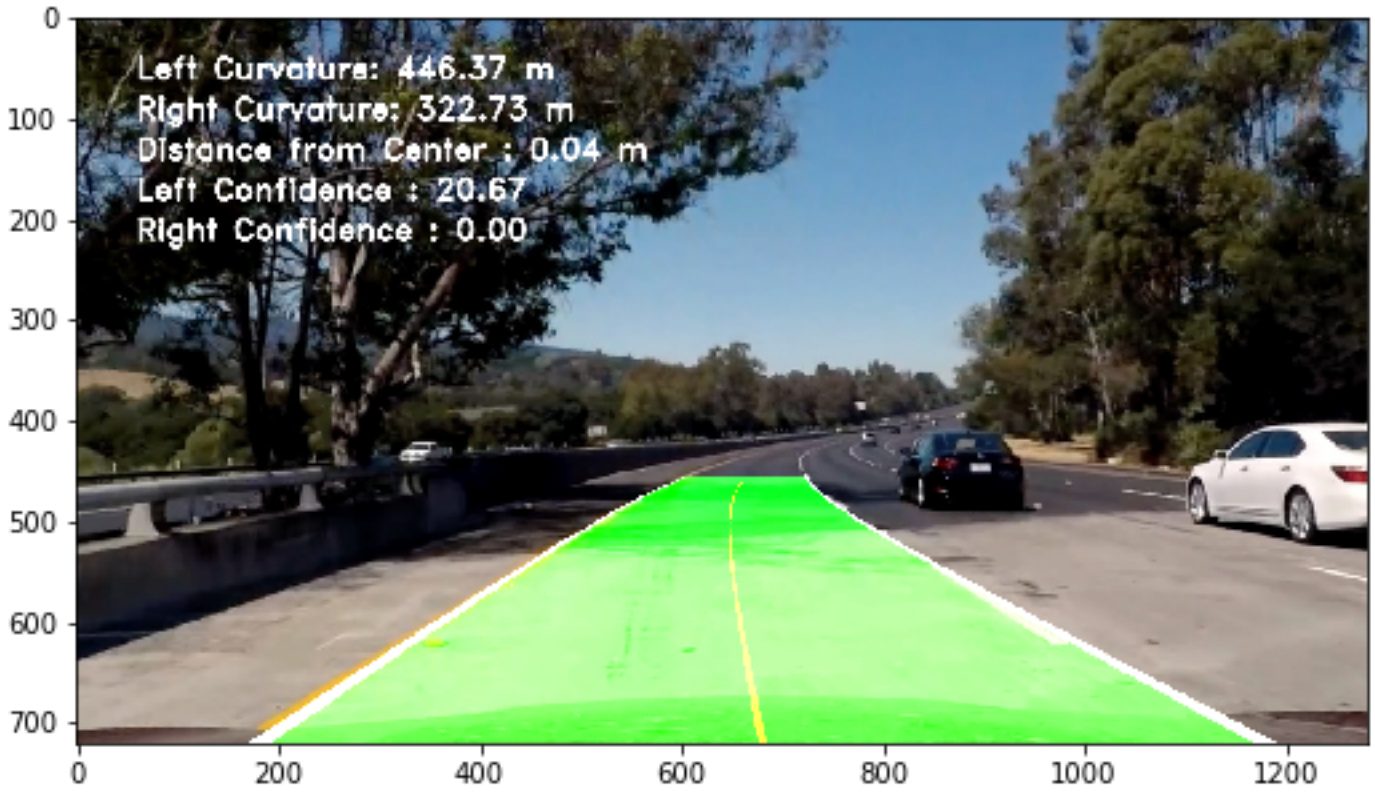
$$f'(y) = \frac{dx}{dy} = 2Ay + B$$

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

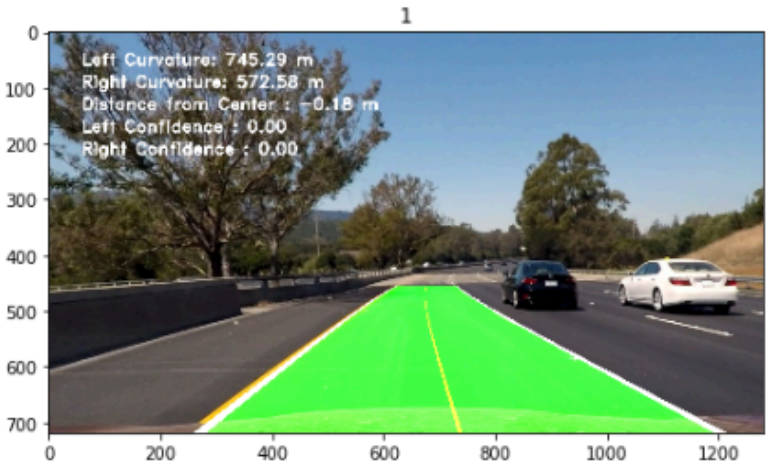
So, the equation for radius of curvature becomes:

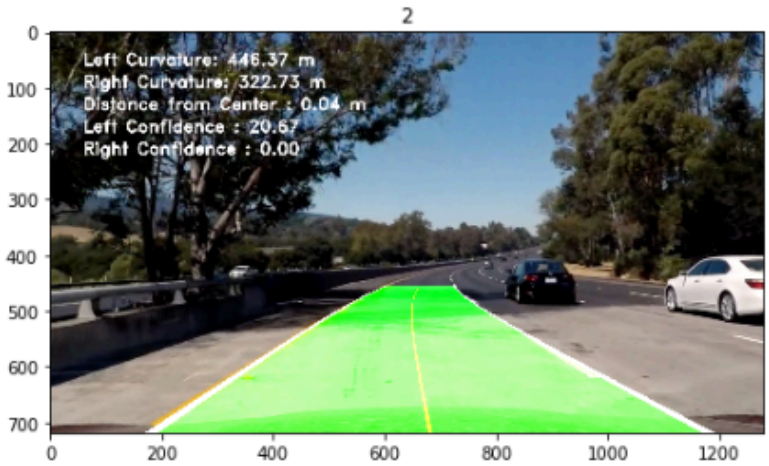
$$R_{curve} = \frac{(1+(2Ay+B)^2)3/2}{|2A|}$$

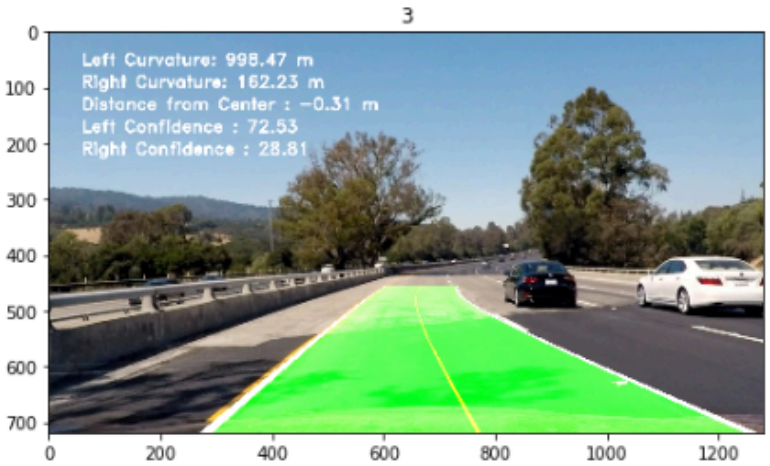
The following figure show the output.

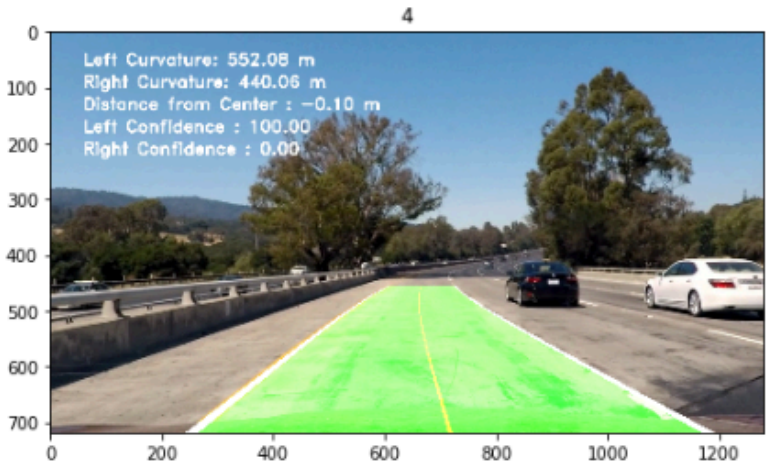


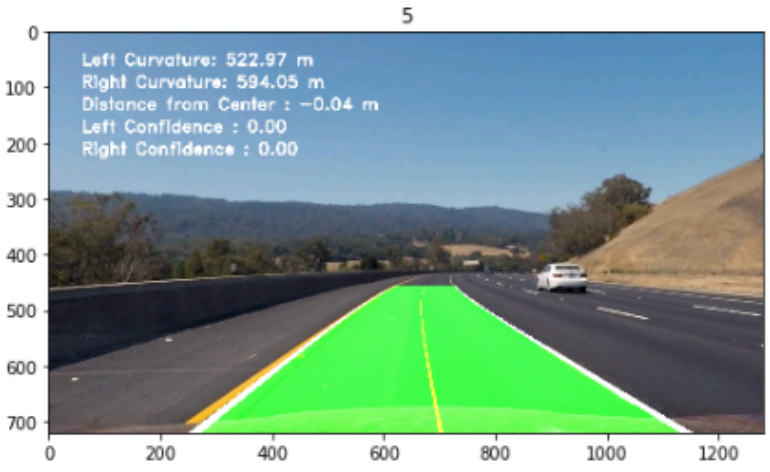
Sample output

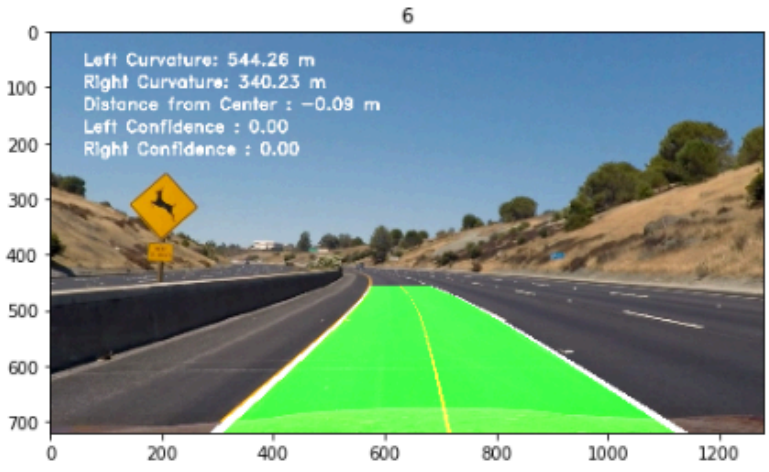












Pipeline Output (video)

Here's a link to my video result in Youtube. (https://youtu.be/fjo6dsGXLVk)

Discussion

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?

This project took relatively a large amount of time when it compares to other (deep learning) projects. The first camera calibration, undistorion and unwarpiong images were relative easy. But when it comes to color and sobel gradients transformation, I had to spend much time to find good threshold parameters. Luckily, I found ipywidgets library, and could do it interactively by manually setting the parameters. It was very helpful.

And, finding lines with sliding windows and curve coefficients were a little difficult to understand. I tried to refer the source code in the lecture, and it took much times.

Luckily, the output of the process pipeline was not bad, and I am glad on the final result.

In [ ]:

In [ ]: