Udacity Self-Driving Car Nanodegree
Project 2: Advanced Lane Finding

Hana Chew

This second project expands on the lane finding goal, but with more advanced principles which may be further tweaked to obtain a result that is a lot more robust. The project was once again programmed using a `Jupyter` Notebook, as it allows easy annotation to the code – and also makes it easier for diagnostics when for example, tweaking thresholds.

The goals / steps of this project are set out as follows:

- Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
- Apply a distortion correction to raw images.
- Use colour 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 centre.
- 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.

This report addresses each of the points that meet the above goals (further detailed in the project rubric), thus successfully completing the Write-up / README requirement. The `Jupyter` Notebook is programmed pretty much to match up with this report, and is divided into the same sections and numbered the same way for ease of reference.

1. Camera Calibration

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

This step is achieved by calling the following function at the start of the pipeline:

```
mtx, dist = cameraCalibration(saveImages='n')
```

The project includes a folder called `camera_cal/` that includes a collection of images taken of a chessboard with 9 x 6 inner corners. All the images are read, and the following steps are iterated on each of them.

The `cv2` function `calibrateCamera()` is used first. As inputs in addition to the image, this function requires two sets of points, `objpoints` and `imgpoints`, where the `objpoints` are the real-world 3D coordinates of the chessboard, assumed to be flat in a single `z=0` plane, and `imgpoints` are the corresponding points mapped out in the 2D image.

`cv2.findChessboardCorners()` is applied to a grayscale version of the image converted with `cv2.cvtColor(img,cv2.COLOR_RGB2GRAY)`, and if the corners are found, these corners represent coordinates to append to the `imgpoints` array. Each of these imgpoints correspond to the fixed coordinates of the chessboard given by `objp`, which is a 9 x 6 x 3 equal to `((0,0,0), (1,0,0), (2,0,0), ..., (7,5,0), (8,5,0))`. After each image, `objp` and `corners` are appended to `objpoints` and `imgpoints` respectively, giving a large number of datapoints on which to carry out the calibration. In case the calibration chessboard images with the points drawn on need to be saved, the flag `saveImages='y'` will use cv2.drawChessboardCorners to draw the chessboard corners on the original image, and save the output files into an output folder.

The function `cv2.calibrateCamera()` is then made use of to obtain the camera matrix `mtx`, and the distortion coefficient array `dist`. To test if the results obtained are correct, these are then used in conjunction with `cv2.undistort()` to obtain the following undistorted image of the chessboard (incidentally the corners were not able to be identified with `cv2.findChessboardCorners()`:

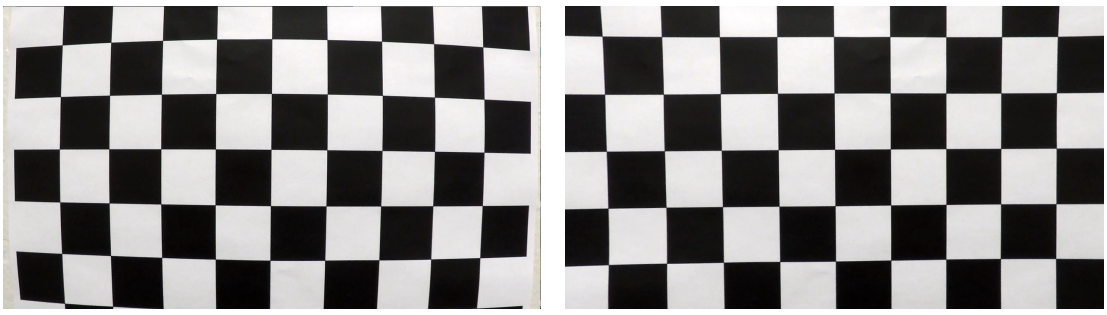


Figure 1: Chessboard in Raw Format (left) vs Undistorted (right)

2. Pipeline (on Static Images)

2.1. Distortion Correction

Distortion correction that was calculated via camera calibration has been correctly applied to each image. An example of a distortion corrected image should be included in the write-up (or saved to a folder) and submitted with the project.

Making use of the camera matrix (`mtx`) and the distortion coefficients (`dist`) obtained in the camera calibration step, the cv2 function

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

is used to correct the image. Even though the distortion with the camera is not so great, the difference is still noticeable especially at the edges of the image, specifically on the car bonnet, which has reduced in size after being undistorted.

Figure 2: Difference between image in Raw Format (left) vs Undistorted (right)

2.2. Colour Transformations & Gradients

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

Various functions were written to enable the thresholding of different parameters in order to accurately detect the lane boundaries. Through a process of trial and error, minimum and maximum values for a number of different parameters were chosen, and together they worked together to obtain satisfactory results on all the test images given.

For each of the thresholding functions, a binary function `binaryThresh()` is applied, whereby the values of the pixels within the specified thresholds would return a value of 1, while all other pixels remain at 0.

Colour Transformations

The main lane detection is provided by the S-channel from a HLS transformation. This is itself tends to provide a very good starting base for the lane detection, especially with yellow lane lines.

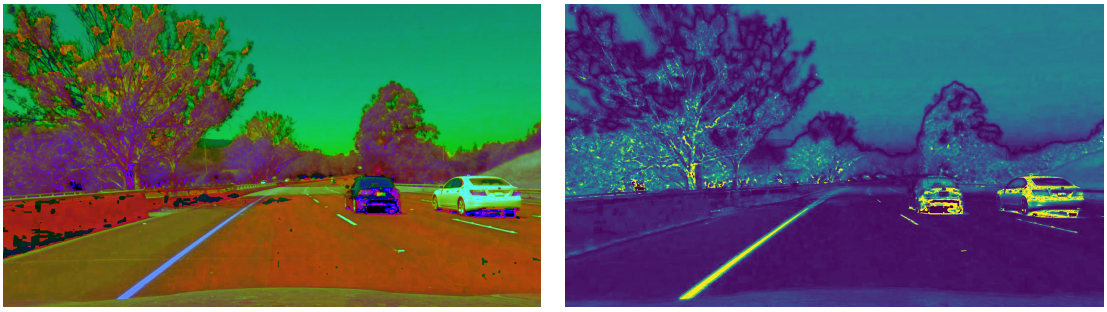


Figure 3: Image converted to HLS (left) and the S-channel isolated from said conversion (right)

The function `sBinary = ColorTh(img,threshold=(0,255))` converts the image into an HLS image, and then applies `binaryThresh()` on the S-channel of the image. Plotting the result, the figure below is obtained.

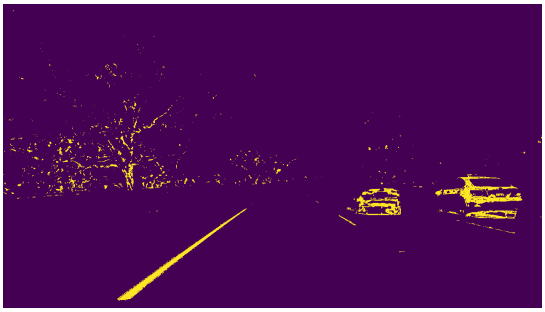


Figure 4: Binary Image of S-channel

As can be seen, the yellow lane marking is very clearly delineated, making the S-Channel a top choice for detecting these. The white lane markings however, tend to be a bit less pronounced, and other methods would need to be carried out in order to obtain these.

Other colour conversions (HSB, LAB) and channels were isolated to see if they would better detect the white lane lines, but in the end, the decision was made to just use the S-channel and the gradients in the next step – the best performing channel other than the S-channel was the R-channel in the RGB image, but these would not perform

Gradients

The next method used would be a combination of Sobel gradient transforms in order to try and obtain methods that can extract the white lane edges as well.

Function `absorbent` makes use of the `cv2.Sobel` function with the stated kernel size in order to obtain the absolute value of either $Sobel_x$ or $Sobel_y$– these are merely Sobel values in the x- and y-directions respectively – upon which `binaryThresh()` is then applied with the following threshold values to give:

```
gradx = absSobelTh(img, 'x', skernel=3, gradx_thr=(20,100))
grady = absSobelTh(img, 'y', skernel=3, grady_thr=(20,100))
```

The other values of interest would be the magnitude and directions of the $Sobel_x$ and $Sobel_y$. As usual, `binaryThresh()` is then applied on these calculated values with the kernel size and thresholds below in order to obtain the respective variables:

```
mag_binary = gradMagTh(img, skernel=3, mag_thr=(20,100))
dir_binary = dirTh(img, skernel=3, dir_thr=(0.7,1.3))
```

In order to extract the pertinent information, it would make sense that the points are only of interest if both the $Sobel_x$ and $Sobel_y$ are within their respective set thresholds, or both the magnitude and direction are within their thresholds, and these are then given by the variable:

```
allGradientsBin[((gradx==1)&(grady==1))|((mag_binary==1)&(dir_binary== 1))]=1
```

The resulting output from the process is a binary image given below.

Figure 5: Image converted to HLS (left) and the S-channel isolated from said conversion (right)

The right lane markings are a lot more noticeable in this image than with that obtained from the S-channel, and there is not a lot of other noise in the photo as it pertains to the road. The yellow lane lines are thinner and less obvious though, showing that it would be a good choice to combine these results.

Combined Effect

As mentioned, the images obtained from the colour transforms and the gradients are quite complementary, in that the combined information matrix would be quite useful in detecting the lane edges. As long as either the combined gradient thresholds or the colour threshold are met, it would be considered a pertinent point.

```
combined[((allGradientsBin == 1) | (sBinary == 1))] = 1
```

The thresholds and kernel sizes to obtain the various colour and gradient binaries were tweaked and modified by a series of trial and error, followed by visual checks, to see which obtained the best results on the test images provided. The thresholds as stated above seemed to hit the sweet spot for being robust – some thresholds were extremely good on some images, but performed poorly on the others. The results of the combination are given below:

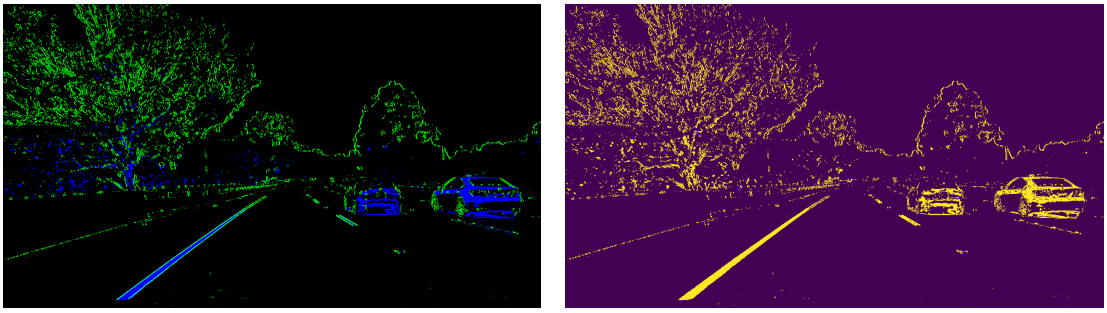


Figure 6: Combined Threshold Binary Images – Colour (blue), Gradients (green) and Combined (yellow).

The usage of these thresholds to produce binary images then provide a good basis for detecting the lane lines in the next steps. The background such as trees are picked up, but these points will eventually be filtered out in the following steps.

2.3. Perspective Transformation

Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

OpenCV function or other method has been used to correctly rectify each image to a "birds-eye view". Transformed images should be included in the write-up (or saved to a folder) and submitted with the project

---

The perspective transformation is called by a function coded to receive as inputs the image, and two sets of points.

```
perspectiveTransform(img,src,dst)
```

The two sets of points do not necessarily need to be provided – there are default values within the function of points which are obtained by selecting a trapezium for the lanes, and transforming the test images with said function.

The function makes use of the `cv2` function `getPerspectiveTransform()` and `warpPerspective()` to obtain the perspective transformation matrix and the warped birds'-eye view image. As outputs to the function, the transformed image as well as the inverse transformation matrix `Minv` (to be used later on to re-transform the image back after all the processing is done.

The following points were chosen based on the results obtained in Project 1, and also noting the possible presence of a car bonnet in the lower half of the frame (hence `h=0.95`). The important thing was that the transformed lane lines appeared parallel (and for the straight lanes, almost directly vertical).

src points	dst points
`( w/32 , 0.95*h )`	`( 150 , h )`
`( w/2-w/18 , 0.625*h)`	`( 150 , 0 )`
`( w/2+w/18 , 0.625*h)`	`( w-150 , 0 )`
`( 31*w/32 , 0.95*h )`	`( w-150 , h )`

The images below show the points from the left column mapped onto the source image, and how they are transformed to the points on the right column giving the destination image on the right.

Figure 7: Original distorted image (left) and the new birds-eye view (right)

2.4. Lane-Line Identification

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

Methods have been used to identify lane line pixels in the rectified binary image. The left and right line have been identified and fit with a curved functional form (e.g., spine or polynomial). Example images with line pixels identified and a fit over-plotted should be included in the write-up (or saved to a folder) and submitted with the project.

Applying all the above steps to the pipeline, the resulting image thus far is now given by Figure 8. By looking the image, it is easy to visually determine that there are two lines running approximately parallel that contain the lane areas, and the main point of this task is to only obtain lines within a certain margin left and right of this line.

Figure 8: Binary Transformed Image

Sliding Windows

Now in order to choose a suitable starting point for the search, the sum of all the points in the bottom half of the image's y-direction are summed, and plotted by the function `histPeaks()`. Figure 9 gives the histogram superimposed on the binary transformed image we have so far. The peaks of this histogram are then used as the starting point for the beginning of a search to obtain the points of relevance.

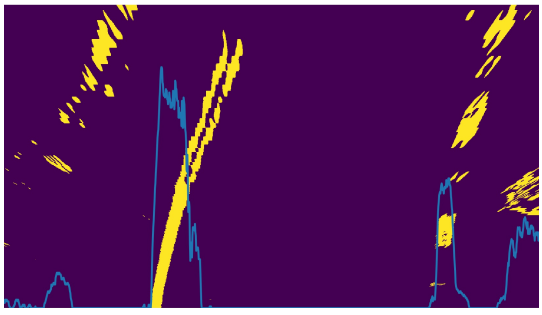


Figure 9: Histogram plotted on transformed Binary Image

For this image, the peaks are distinct and easily discernible, but paying close attention to the right hand side, one can see that there is a close second peak, which in some other cases could easily be mistaken for the lane marker. A margin of 50 pixels to the left and right of the image is thus introduced in which peaks found here are ruled out.

The principle behind this method is, using the histogram-found starting points, rectangular search windows are introduced in an iterative (in the case of this project, `nWindows=12`) and then stacked up on one another, shifting in the x-direction depending on where the points are found.

i. The search window is defined with a fixed height, and starting points (midpoints of the search window) determined by the peaks in the histogram. The `margin=70` is also defined.
ii. The margins and the height are added to the midpoints, to give the current search window.
iii. The indices of all points in the binary image that fall within this window are appended to lLaneIdx and rLaneIdx.
iv. If the number of points found in step iii is greater than minPix = 50, the midpoint of the next starting window. And the process is iterated from step ii until all the windows have covered the vertical distance of the image.

The iterative process above results in `lLaneIdx` and `rLaneIdx`, which may then be used on the binary image to obtain all the coordinates (`leftx`, `lefty`, `rightx`, `righty`) that are of interest for the right and left lane.

Making use of these coordinates, a best fit line of polynomial 2 may be obtained for the left lane, and the right lane. Note that the equation we are interested in here is $x = Ay^2 + By + C$ since the lanes tend to be vertical in our transformed image.

```
lCoeff = np.polyfit(lefty, leftx, 2)
rCoeff = np.polyfit(righty, rightx, 2)
```

Carrying this procedure out on the image results in the plot in Figure 10, where the effects from the shadows of the tree and other lane markers have now been discarded.

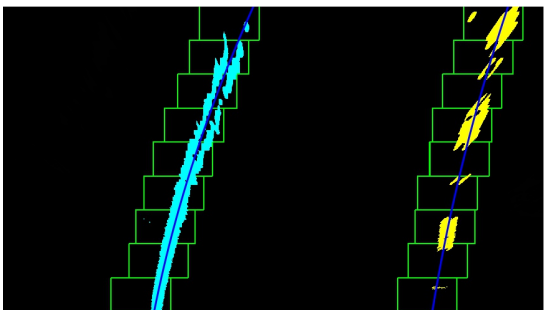


Figure 10: Sliding Windows - Best Fit Line using slidingWindows()

2.5. Radius of Curvature and Vehicle Position

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 centre.

The line of best fit for each of the lane line markers is given by $x = Ay^2 + By + C$. The coefficients given by A, B and C are the pixel coefficients.

The radius of curvature is calculated by the formula given in the lectures by `laneCurve()`:

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

This may be reduced to the following line of code in the `iPython` notebook:

```
rCurve = (1 + (2*A*h + B)**2)**(3/2) / np.abs(2*A)
```

If the values obtained from the coefficient are just plugged into the above, however, we would only obtain the results of the Radius in pixels. Two conversion factors are introduced, so that we may get the real-world curvature, which are then used to give the Curving Radius in meters.

```
A = (xmPerPix / (ymPerPix**2) ) * coeff[0]
   B = (xmPerPix / ymPerPix) * coeff[1]
```

The transformation values chosen are given by the following equation where h is the image height, and `imgLaneWidth` = 634 is the width between the average pixels obtained from the polynomial fittings of the test images:

`ymPerPix` = 30m (real-world) / 720 pixels (image)

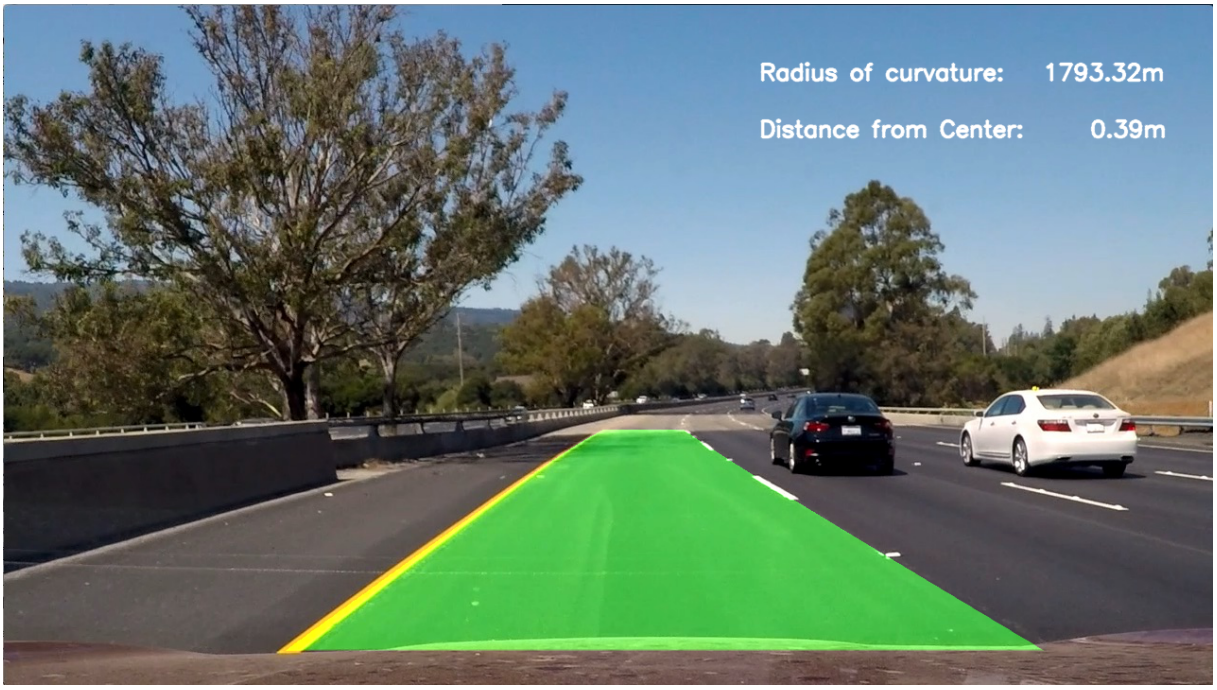
`xmPerPix` = 3.7m (real-world) / imgLaneWidth pixels (image)

Assuming the camera is mounted in the centre of the car, the centre of the image would represent the centre of the car. The distance of the centre of the car with respect to the centre of the lane is calculated by `findDist()` . This is done simply by calculating the midpoint between the calculated best fit lines obtained from in step 2.4 above at the base of the image (or maximum `h` from the top left hand corner) and finding the distance between that and the midpoint of the image (in meters).

The results are then output on the image as shown in the next section.

2.6. Example Image with result plotted back down onto the road

Section 2.6 of the `iPython` notebook details the steps taken to use `drawlane()` and the inverse transformation matrix `Minv` obtained from step 2.3, the perspective transform. The points obtained from the best fit lines from step 2.4 are used together with `cv2.fillPoly()` to plot a shape on a blank image (assumed to be the birds-eye view of the lane) and this polygon is then warped back using the inverse matrix `Minv` and `warpPerspective()` to give the following result.



A function `outputData()` is also defined in this section to display the calculated Radius of Curvature and the distance of the car from the centre of the lane.

All of the above steps are carried out by one of two functions. The function `processImage()` processes a simple image with only one result: that of the photo you see above. The other function `processImageDiagnostic(img)` includes a plot within the `iPython` notebook of a number of the important images from above, so the video pipeline below may be analysed statically frame by frame if necessary to check fit characteristics.

3. Pipeline (video)

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 described above are also used for the video, with the function `processImageDiagnostic()` providing a way in which the images can be analysed frame by frame so the shortcomings of the pipeline may be adjusted and tweaked for different lighting or contrast conditions.

3.1. Finding Lane Lines – Search from Prior

There is, however, one addition to the pipeline when it comes to a video. In order to find the lane lines for each new image, the sliding windows method is used. This iteration, however, is inefficient if it needs to be carried out for every frame. What's more, the results might end up being unreliable, because the car is not likely to suddenly veer off in a different direction between one frame and the next. This means, that it is reasonable to assume that the points of interest would be between a margin given by the lines of best fit in the previous frame.

The actual function `searchFromPrior()` itself is rather straightforward - the variables `lLaneIdx` and `rLaneIdx` are introduced and all points which fall within the specified margin to the left and right of previously calculated points by making use of `lCoeffPrev` and `rCoeffPrev` are considered points of interest.

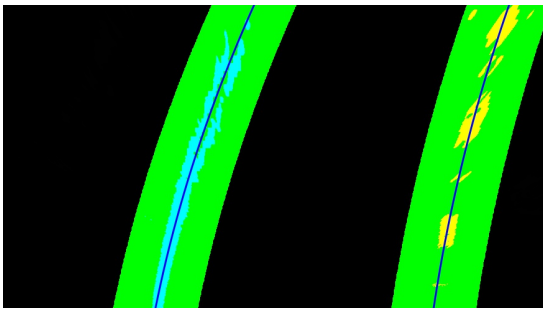


Figure 11: Area of Interest (green) and Best Fit Line using `searchFromPrior()`

The `leftx`, `lefty`, `rightx`, and `righty` points are then calculated in the same way as with `slidingWindows()`.

An additional function, `lineCheck()` is implemented, that checks the distance between the bottom-most x-points, and if the width of the lane does not fall between 550 and 800 pixels, the variable `switch` is activated. This function is used in `searchFromPrior()` and `slidingWindows()`, provided this is not the first frame of the image.

In order to successfully implement this `searchFromPrior()` function, two new global variables `lCoeffPrev` and `rCoeffPrev` are also introduced. The `processImage()` function checks whether `lCoeffPrev` exists in the global variables, because for the first frames or single image runs, one would always start with `slidingWindows()`.

When the `switch` variable is activated within `searchFromPrior()`, the results of the best fit would then be obtained via `slidingWindows()`. If this still results in `switch` being activated because the lane is still too narrow or wide, then the results from the previous frame are used for the line of best fit. Whenever the coefficients are obtained as a result of slidingWindows, the global variables lCoeffPrev and rCoeffPrev are updated with the values of lCoeff and rCoeff for use in the next frame.

The pipeline was successfully implemented on the video and there were no issues of the lanes being too large or too small. The video file is saved as "`OutputVideo.mp4`".

4. 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?

4.1. Problems / Issues faced in Implementation

The most tedious part of solving this project was tweaking the gains or the thresholds in order to obtain a best fit image. There were dozens of times that I thought the perfect balance had been reached with all the output images, and then when applied on the video, there would be certain skips and jumps and strange inter-crossing lane lines.

Even though this worked very well on the Project Video, it failed to perform on the Challenge Video due to the concrete divider, number of horizontal cuts or changes in contrast to the lane itself. The extra challenging video with its extra wavy lines would have been a disaster with the current pipeline.

4.2. Potential Shortcomings & Possible Improvements to Pipeline

There are a number of issues which could be addressed given more time, as the challenge videos were, simply put, a challenge, given these gains.

I. Optimisation of code so it runs more efficiently – the various functions are not completely optimised.

II. If the cars ahead are not within their lane markers, such as changing lanes or directly ahead, or really close, it could affect the lanes drawn, as the binary images show the cars really clearly.

III. The only sanity check included is the starting lane width. There could be other checks included to check whether the results make sense. For example, the radius of curvature could be assumed to be infinity on straight line roads if the coefficient of the squared term in the polynomial is small enough, among others.

IV. Shadows, concrete dividers, and really curvy roads are very difficult with the current state of the pipeline. Various other thresholds or tweaking could be done to improve this. Also, the pipeline makes use of a second order polynomial to fit the curves. While this is generally true, it could be modified to include a third order perhaps for extra curvy lane lines

V. The lines drawn on the lanes are currently only taken from the current (or if not available, the previous frame). Averaging the values across the last n frames could give a much smoother result, and also the accounting of missing frames could be increased from just one to 3 or 4 if necessary. Above could be solved by modifying code to include an averaging component or a weighting component with respect to the gradients or lines from previous frames (in a video) – this could for example be flagged by too drastic gradient changes from one frame to the next – and an averaging of the values could perhaps account for lane changing.

VI. If the car is changing lanes or drifts, the algorithm might not hold any longer

VII. The threshold values and various filters could be further optimised, but the possibilities are endless and the checks so far are only visual – perhaps another method to learn or see which would be the best value for different conditions.