Project 4 - Advanced Lane Finding

24th February 2017

Files in the Project

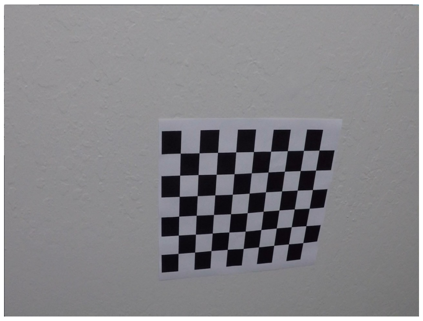
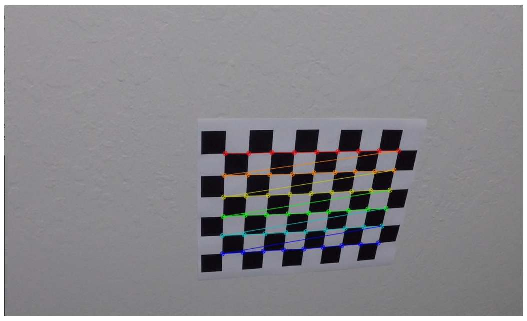
My project includes the following files:

- corner_finding.py contains a script that detects the corners on the chessboard
- image_gen.py contains the code to generate all of the images (binary, warped, etc..) throughout the project
- undistort.py contains the code to undistort the chessboard images.
- tracker.py contains code that keeps track of the previous results from different images.
- writeup_report.pdf summarizing the results
- lane_detecting.mp4 to show that the car made it around the track

Camera Calibration

1. **Have the camera matrix and distortion coefficients been computed correctly and checked on one of the calibration images as a test?**

   First, in the python file *corner_finding.py*, I used the **cv2.findChessboardCorners()** function to find the chessboard corners, and the **cv2.drawChessboardCorners()** function to draw them on the chessboard images. This can be seen in the figure below, where the left image is the original image and the right image has the dots drawn on the corners.

The code for undistorting the chessboard images is contained in the python file, *undistort.py*.

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()** (which can be found on line 36 in the file *corner_finding.py*) function. I applied this distortion correction to the test image using the **cv2.undistort()** (line 18 in *undistort.py*) function and obtained this result in Figure 1.

I also noticed that one of the chessboard images does not get corners drawn on it because the bottom of the image was cut off and so there were less corners in the image then we said there were in the computer code (as seen in Figure 2).

Pipeline (single images)

1. **Has the distortion correction been correctly applied to each image?**

   To demonstrate this step, I will describe how I apply the distortion correction to one of the test images, like in Figure 3

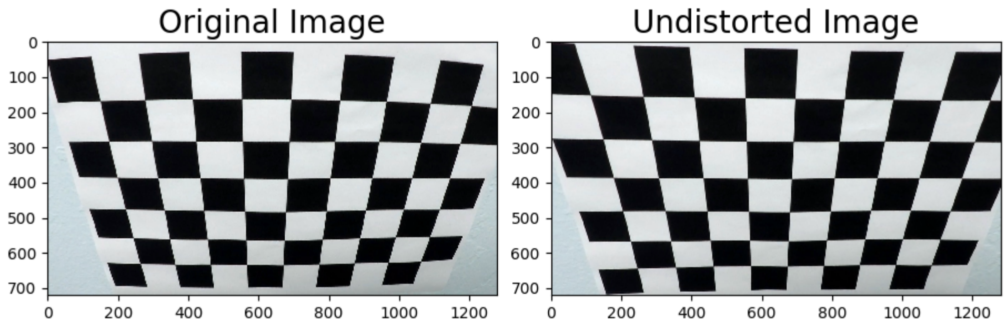


Figure 1: Showing a distorted image (left) vs an undistorted image (on the right)

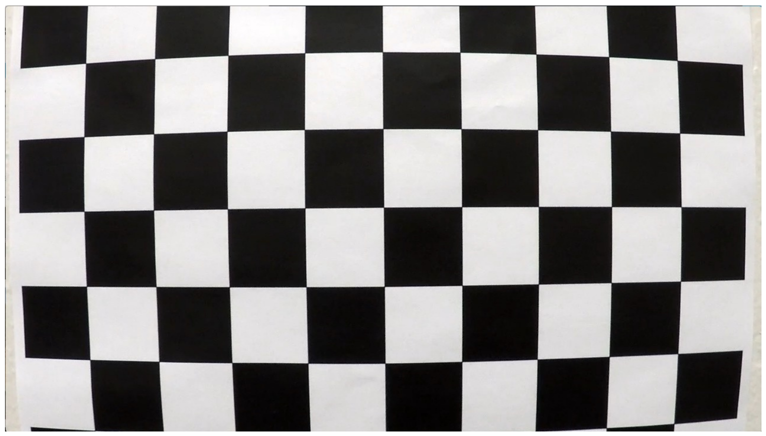


Figure 2: Corners could not be drawn on image because the amount of corners detected in the image is less then expected

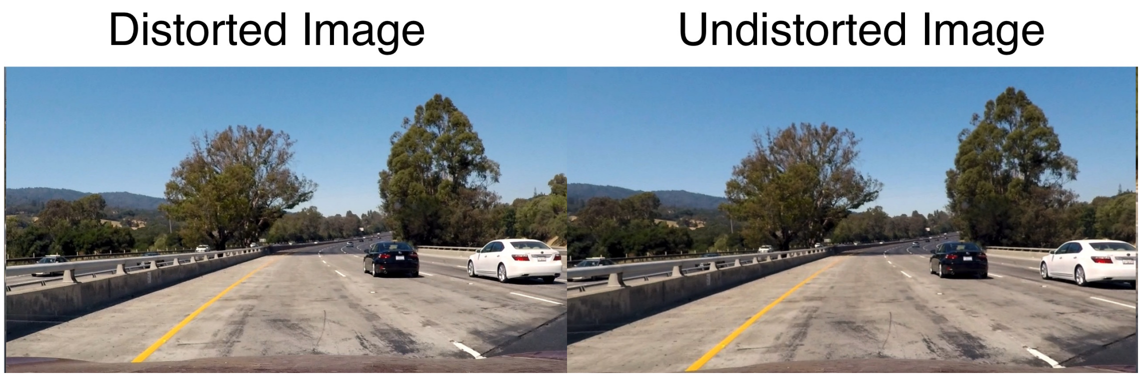


Figure 3: The left image is a distorted image and the right is an undistorted image

I applied the distortion correction by using the code lines 65-77 in the file *undistort.py*. I used a pickle to upload the mtx and dist values, which were obtained by calibrating the camera using the chessboards. Then I used the **cv2.undistort()** function on all of the test images provided. The camera used to take the test images was (I'm guessing) a good camera and so there was not that noticeable of a distortion in the test images. Although when you look at Figure 3, you can see that there was some distortion around the edges in the "distorted" photo. We know this because looking at the "undistorted" photo, we see that the car on the bottom-right of the image is now being cut off by the edge of the photo.

2. **Has a binary image been created using color transforms, gradients or other methods?**

   One of the binary images that was created can be seen in Figure 4

   This photo was created using different types of methods. I used a Sobel gradient in the x and y direction (which can be seen in *image_gen.py* on lines 14-32), which I had a threshold of (50, 255). I then decided to implement a color threshold (lines 71-94 in *image_gen.py*). I decided to change the image to and HLS and HSV color scheme (lines 73 & 83 in *image_gen.py*). After much trial and error I decided to extract the 'S' channel from the HLS (line 76 in *image_gen.py*) and the 'V' channel from the HSV (lines 86 in *image_gen.py*). I set the S-channel threshold

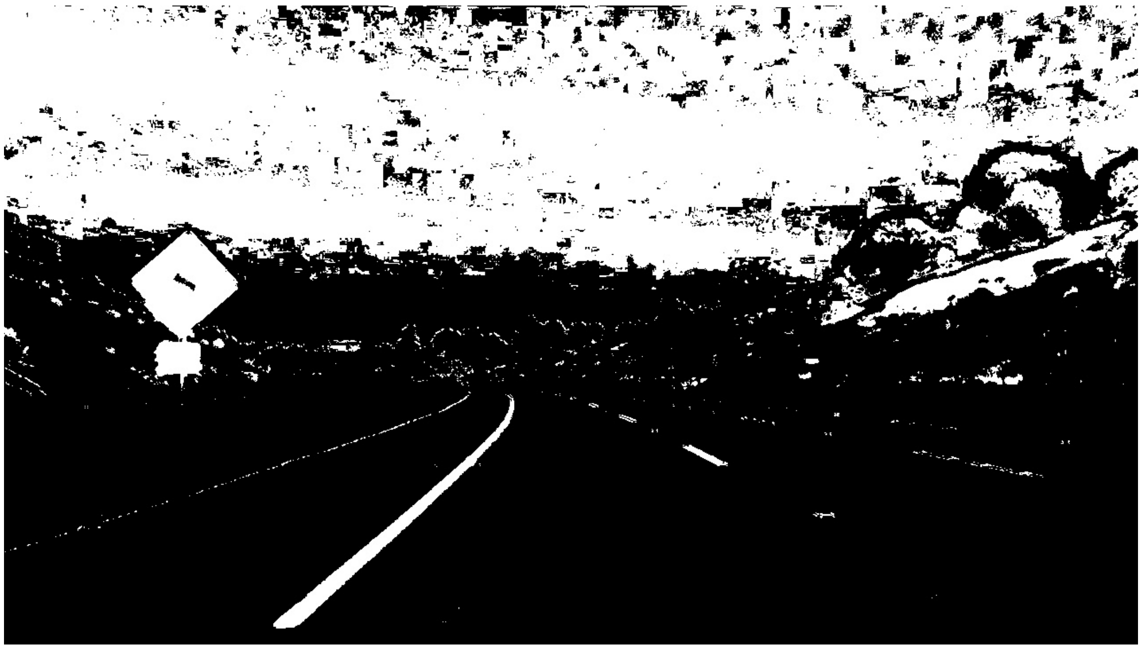


Figure 4: Binary image created by using color transforms and gradients

to (100, 255) and the V-channel threshold to (50, 255) (line 122 in *image_gen.py*). After a lot of experimenting with different combinations of thresholds and masks for the image, I found that using the Sobel gradient in the x and y direction, along with the color threshold gave the best output in my case.

3. **Has a perspective transform been applied to rectify the image?**

   The code for my perspective transform can be found on lines 131-148 in *image_gen.py*. This code takes the binary image that was just created (Figure 4), and then uses source and destination points that were predetermined (by experimentation) to make the perspective transform. The source points and destination points can be found on line 135 and 139, respectively, in *image_gen.py*.

4. **Have lane line pixels been identified in the rectified image and fit with a polynomial?**

   To determine where the pixels were in the warped image, I decided to use a convolution to detect the 'hot' pixels. I split the window into 9 vertical slices that the program would convolve in. Then I created a convolved signal that looked for the highest valued pixels in each vertical slice (lines 65 & 70 in *tracker.py*). The boxes appear on the lines as shown in Figure 5 and Figure 6

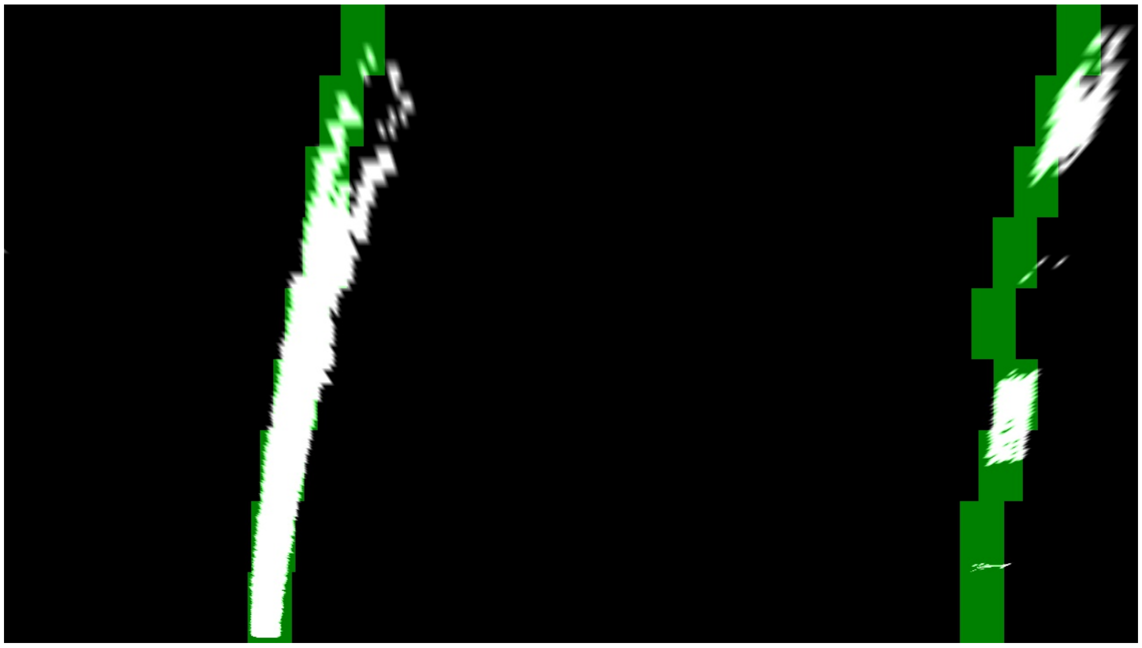


Figure 5: Example of boxes detecting the lanes

Then I decided that to fit a polynomial to the lanes, I would use the information from the boxes that were made from the convolution. To fit the lines, I decided to try and fit the polynomial so that the line would go right through the middle of the boxes. I figured this would provide the best estimation of where the lanes are, as long as the convolutional boxes outlined the lanes properly (this can be seen on lines 190-213 in *image_gen.py*). The polynomial fit can be seen in Figure 7.

5. **Having identified the lane lines, has the radius of curvature of the road been estimated? And the position of the vehicle with respect to center in the lane?**

   Yep sure did! As seen in Figure 8 and Figure 9

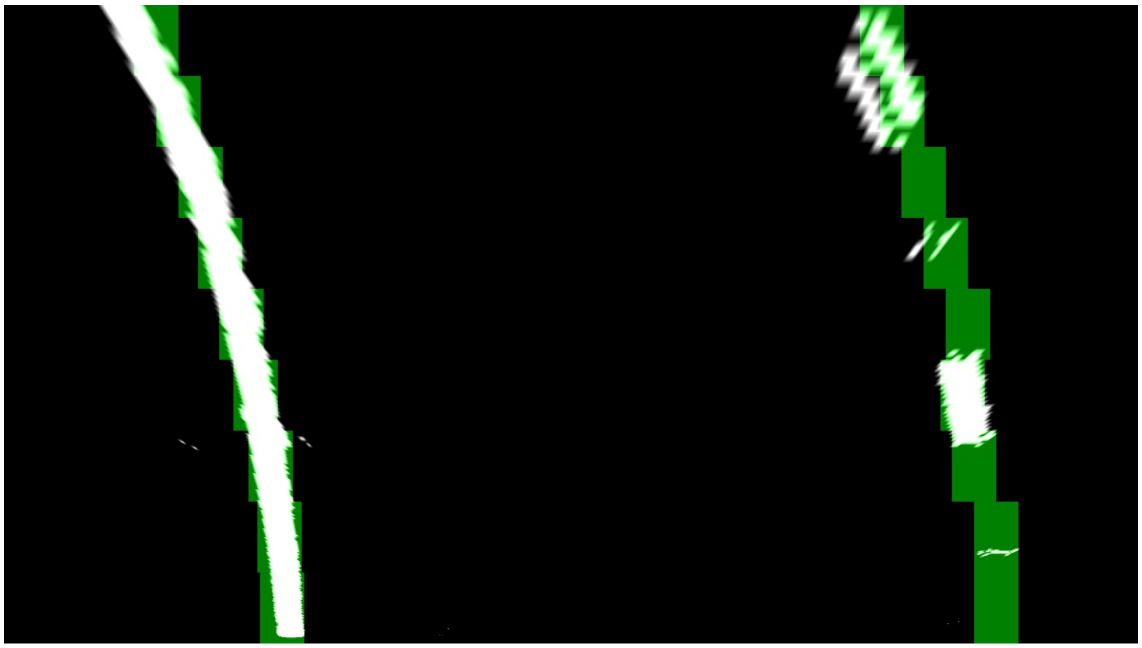


Figure 6: Example of boxes detecting the lanes

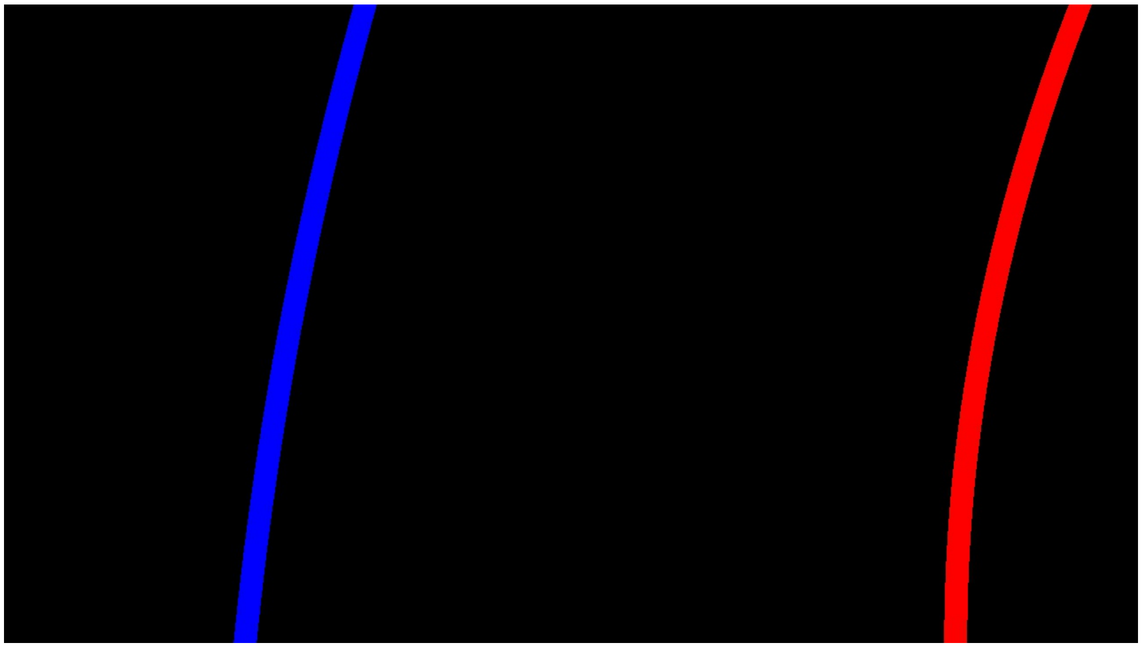


Figure 7: Showing polynomial that was fit to the lanes

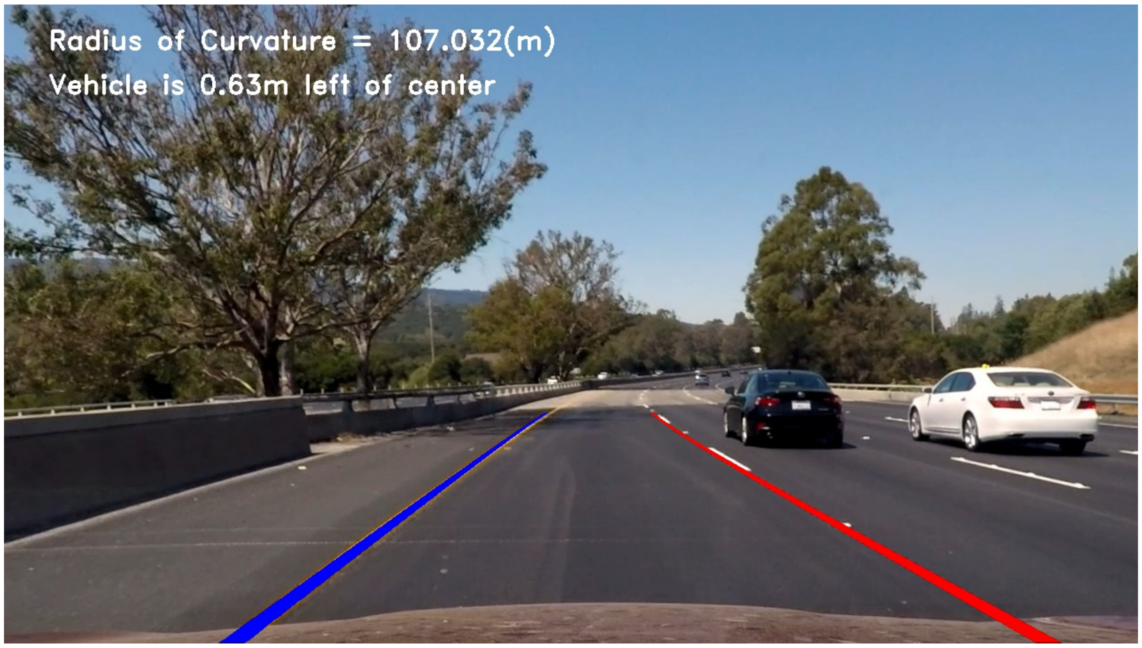

Radius of Curvature = 107.032(m)
Vehicle is 0.63m left of center

Figure 8: Showing the radius of curvature of the lane as well as the position of the vehicle

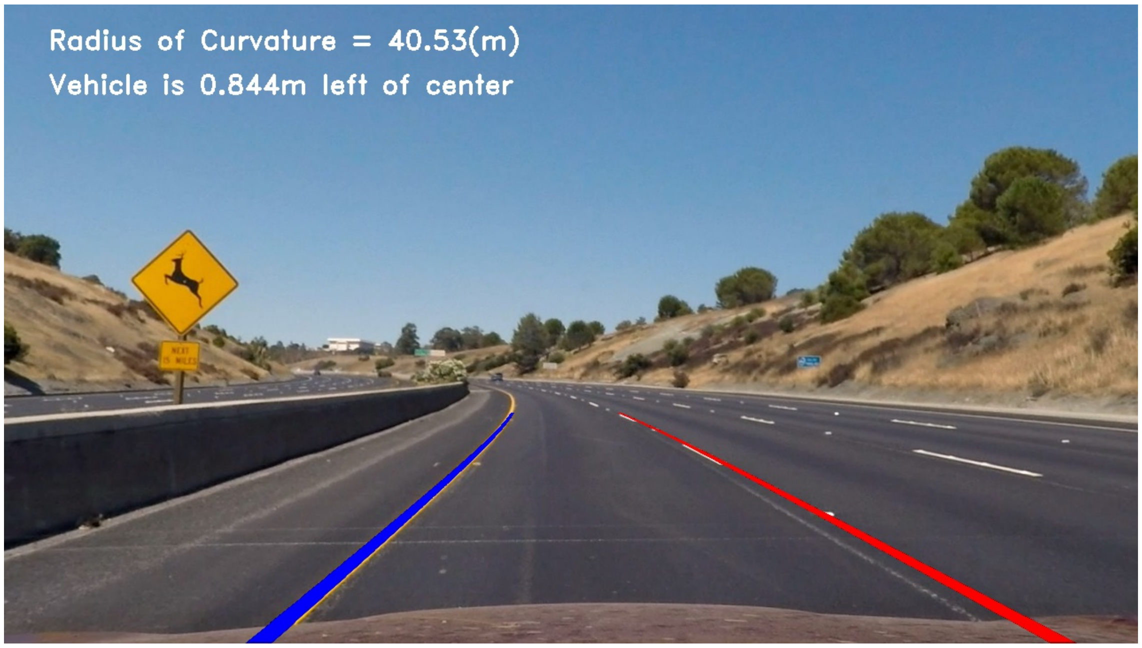

Radius of Curvature = 40.53(m)
Vehicle is 0.844m left of center

Figure 9: Showing the radius of curvature of the lane as well as the position of the vehicle

Pipeline (video)

Video is found by playing the lane_detecting.py

Discussion

I think that the techniques used in this project work relatively well. I think that it works well for white and yellow lanes to detect. I think if you had different colored lanes then it might cause an issue. I also think my pipeline falls short is determining where on the image the lane is. For example, I found the src and destination points by hard-coding them in. I think a more robust pipeline, could take in any size image, and then be able to use an algorithm or a ratio of some sort to find the most probable area that the lanes would be in. If more time was invested in this project, it would be interesting to see if you could use an convolutional neural network to determine the optimal area of the lane or just use it to completely map out the lane from the start.

Some areas I found difficult was trying to figure the threshold for the gradients. I had to do a lot of playing around because I wanted to decrease as much 'noise' as possible along the lane that was not the lane lines. The pipeline sometimes struggle with lighter pavement on the road, which I think could be fixed with playing around with different color and gradient thresholds. Also sometimes where the lane markings were very faint or were worn off on the road, I noticed that the pipeline struggle a bit, I think this could be fix by maybe extrapolating the line further down the road.

I also found that averaging out the measurements of where the lanes were over the past 15 images, really helped smooth out the tracking of the lanes. I also found my values for radius of curvature to be reasonable as well. As shown in Figure 8, the radius of curvature is higher then in Figure 9, because Figure 8 is much more straight, whereas Figure 9 has more of a bend which would decrease the radius of the circle.

I enjoyed doing this challenging project.