**ADVANCED LANE FINDING**

**Camera Calibration**

The code for this step is contained in file camera\_calibration.py.

Cameras use curved lenses, due to which light rays bend a little or too high at the edges of the lenses. This distorts the images, making them a little or more bent than they usually are. This is radial distortion. Another is when they are not parallel to the imaging plane. This makes image looks tilted and the objects look closer or farther than they are. This is tangential distortion.

Therefore, the first step is to calculate the calibration matrix and the distortion coefficients of the camera which is used for finding lane lines. Each chessboard has 8x6 corners to detect in the image. I start by preparing object points. Object points are the 3D points of the corners of the real undistorted image, in x, y, z plane. I convert it to 2D by reshaping it to 2x2 matrix where z will become 0. Another I prepare is the Image points array, which is the empty 2D array to append corners of the distorted image.

An OpenCV function called cv2.findChessboradCorners() is used to detect the distorted chessboard corners. For each image, the detected corners are appended to the image points array and the copy of the prepared object points are appended to another array, since object points are same for all the images. The final array of the image points and the object points are fed into a function called cv2.calibrateCamera() to get the desired calibration matrix and distortion coefficients.

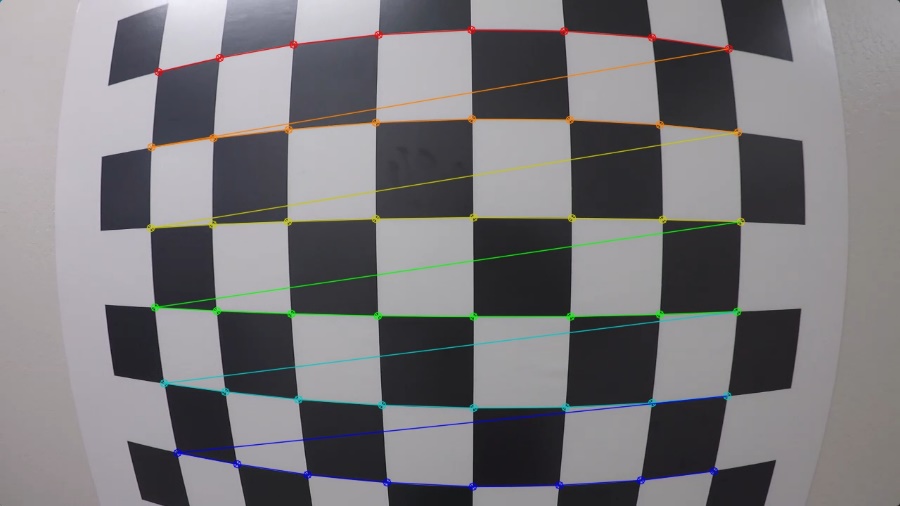


Fig 1: Chessboard Corners

And when I applied distortion correction to this image, I obtained the following result:

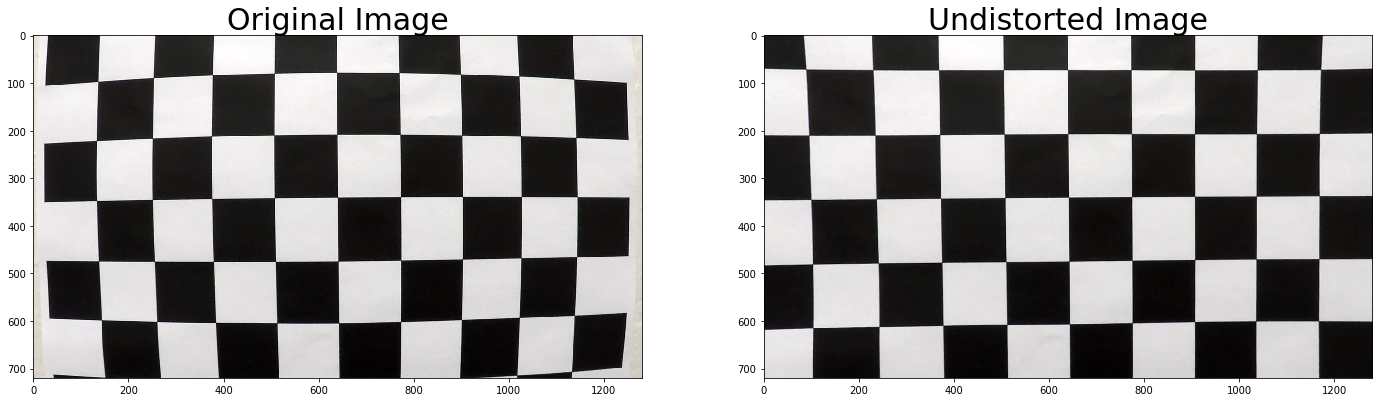


Fig 2: Comparison between original and undistorted image

**Pipeline**

**Undistort Image**

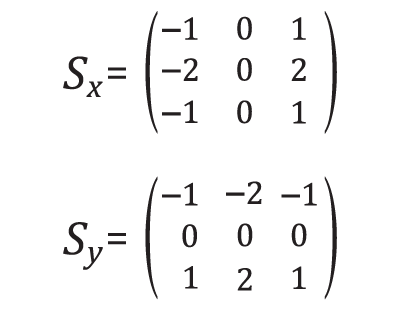
Now, to undistort the image or for distortion correction, the calibration matrix and the distortion coefficients are used in a function called cv2.undistort() to get the desired result. The result for one of the images is as shown below:



Fig 3: Comparison between original and undistorted image of a test image

**Threshold**

For finding the lane lines, how do we take advantage of the fact that lane lines are vertical? For this, we use gradients to detect steep edges. Gradients are the heart of canny edge detection algorithm. Canny edge algorithm is used to detect almost everything in image. Canny edge takes derivative of x and y to detect edges. In order to take derivative of x and y, Sobel operator is used, and it looks like below with kernel size = 3.



**How does the Sobel operator work?**

When you superimpose sobel operator on a region, you get the product of the matrix. If the sum of the resultant matrix is in x direction, then edges closer to vertical and if in y direction, then edges closer to horizontal.

Our goal is to find the yellow and white lane lines which runs vertically through the image. For this, we must create binary images within thresholds to select the pixels.

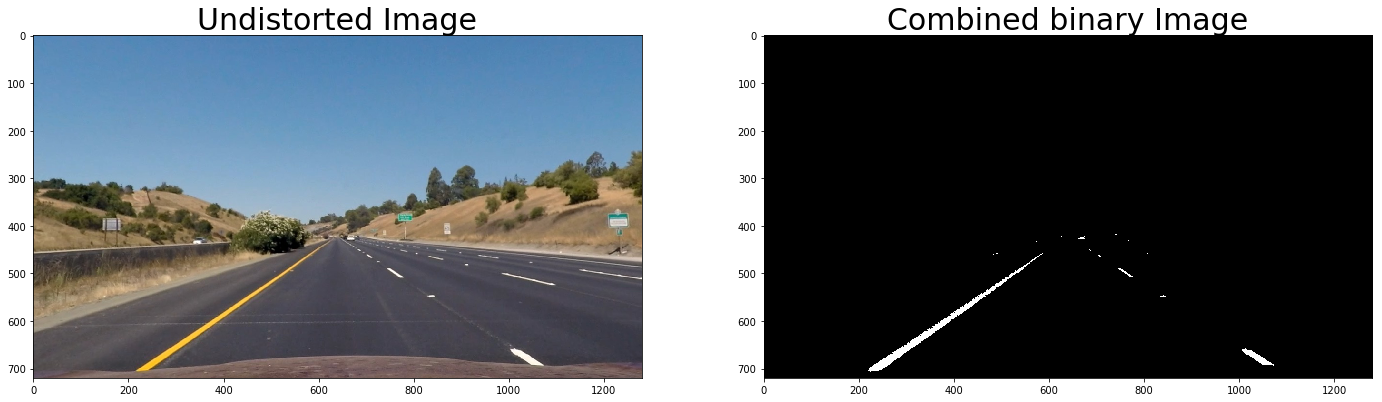
There are many combinations of the thresholds that can be used. There is gradient threshold which selects pixels based on the gradient strength. The gradient threshold comprises of x, y, magnitude and direction. And then there is color threshold where pixels are selected based on the color spaces, which comprises of RGB, HLS, HSV etc.,

Let me start with colour threshold which used to detect yellow and white lines at various degrees of light like shadows or light. After various experiments in RGB and HLS colour space, I chose to use the HLS colour space, because of the following reasons:

1. Hue: Hue is the colour irrespective of the brightness. This hue threshold allows me to select the distinct yellow and white colors for the lane lines.
2. Lightness: Lightness has low value when the colour is bright and high value when colour is faded. This threshold helps me in capturing the colour, at different lights, whether shadow or sunlight.
3. Saturation: Saturation has low value when colour is faded and high value when colour is bright. Saturation threshold helps to distinctly capture the lane lines irrespective of the colour.

Therefore, the combination of the right thresholds of hue, lightness and saturation has helped in the clear detection of lane lines under various environmental conditions.

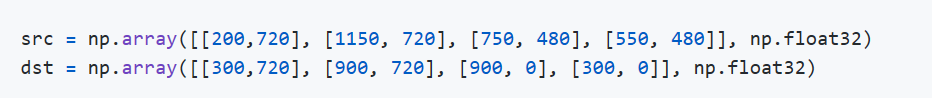
I used a combination of color and gradient thresholds to generate a binary image in file thresholding.py. Here is an example of my output for this step.

****

**Perspective Transform**

The use of perspective transform is to find the straight or curved lines, to make it look like in a top down approach, like that of maps, in a zoomed manner. This is called birds eyes view. For this, we select four points to perform a linear transformation perspective change, which is birds eye view. The four points are selected both in source image and for destination image, so that they can be mapped to get a bird’s eye view of the lane lines.

The code for perspective transform can be found in file perspective\_transform.py. We first obtain perspective matrix, M, from function getPerspectiveTransform. Then, we use warpPerspective() function which takes as inputs, an image (threshold\_image), as well as the hardcoded source (src) and destination (dst) points to get the warped image.



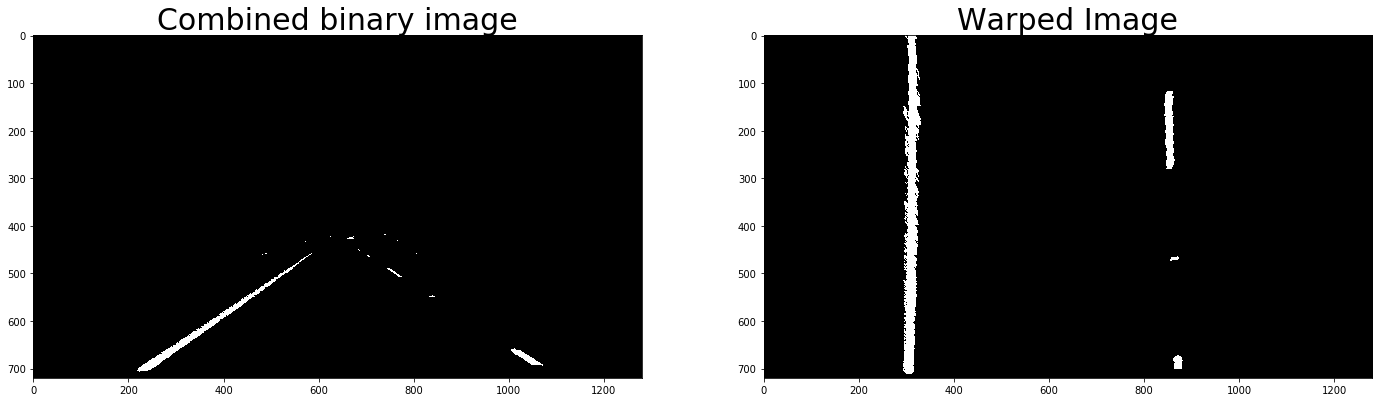
****

Fig 5: Comparison between original and undistorted image of a test image

**Finding lane line pixels**

**Histogram**

A histogram of the lower half of the image is taken (where lane lines are most likely to be vertical) to find the peak, which are nothing but the left and the right lanes. These peaks are considered as the starting point(x-positions) of the lane lines.

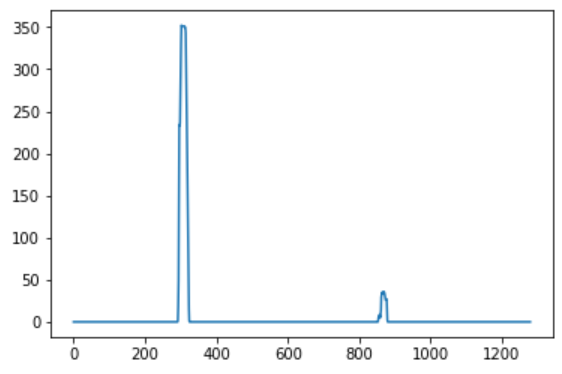


Fig 6: Histogram

**Sliding Window approach**

From the peaks, n number of sliding windows are formed up to the top of the frame with a margin of +/- 100 from starting points, meaning 100 from the left of the point and 100 from the right of the point. The activated pixels (non-zero x and y points), are extracted within the window. These x and y activated pixels are fit in a polynomial using np.polyfit() to get the left and right coefficients respectively. With the help of coefficients and the y value of the image, x values of left and right lane are calculated using the second order polynomial equation.

The code can be found in find\_lane\_pixels.py

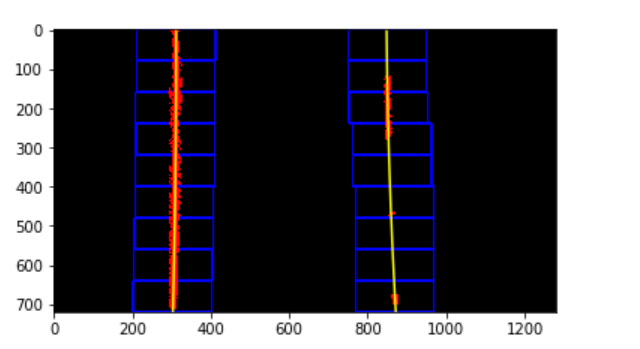


Fig 7: Sliding Window

**Lane Line Detection Using Previous Polynomial**

As the next frame arrives, it would be hectic to use the sliding window approach to calculate the left and the right lanes, instead we use the polynomial of the previous line. The coefficients of the previous line and the non-zero y values of the image help us get the new x and y pixel values of the image. These new pixels using np.polyfit() help us get the new coefficients of the lane lines. These new coefficients and y value of the image in turn gets us the new x and y values of the image.

The code can be found in find\_lane\_pixels.py under function find\_next\_lane\_line\_from\_prev\_poly().

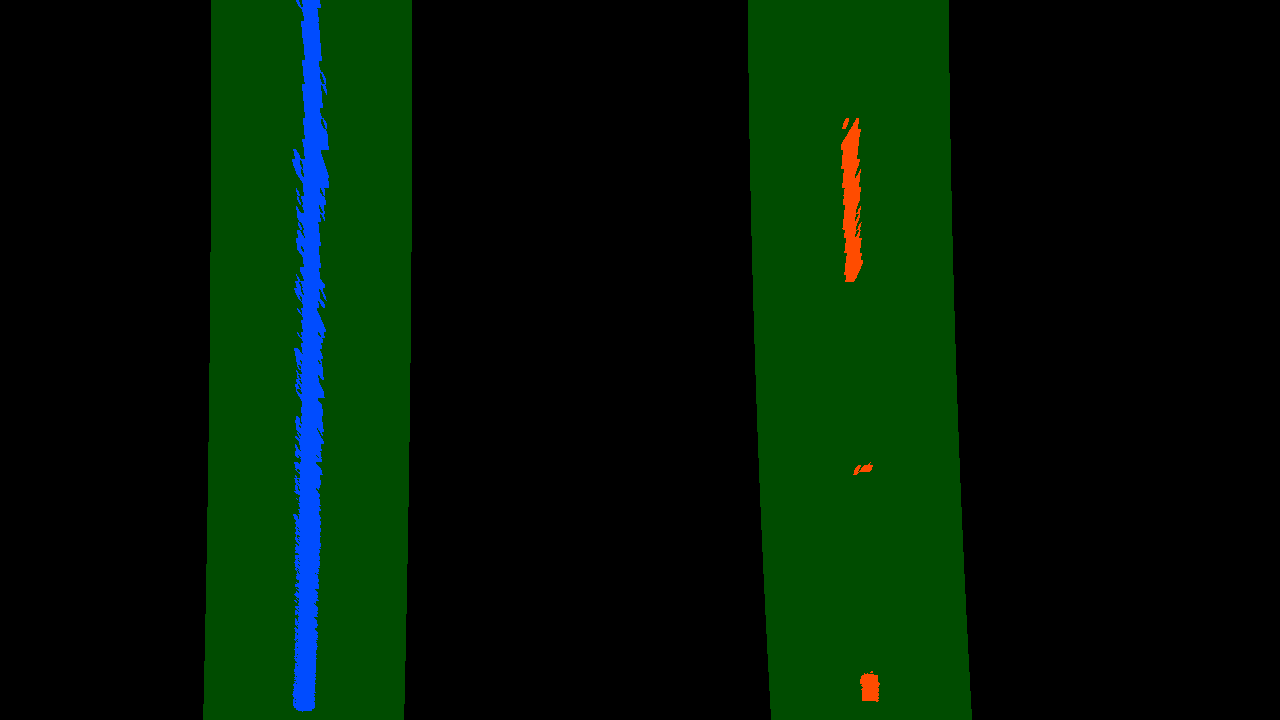
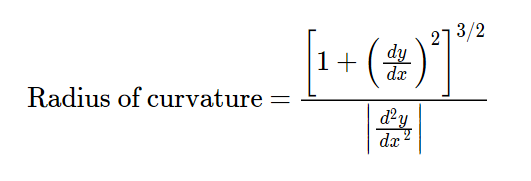


Fig 8: Next Frame Using Previous Polynomial

**Radius of Curvature**

The radius of curvature is calculated using the equation:



The y values increase from top to bottom of the image. Hence if the curvature must be calculated, the y values at the bottom of the image is considered, which will be ymax. In the real-world space, the real dimension of the lane is taken as inputs. According to the U.S. regulations, the minimum lane width should be 12 feet or 3.7 meters and let us take the length of the lane as 30 meters. Since our camera image is in pixels, I convert the pixels in meters as follows: ym\_per\_pix = 30/720 # meters per pixel in y dimension xm\_per\_pix = 3.7/700 # meters per pixel in x dimension

The coefficients are derived using np.polyfit() taking input as image y value and the x pixel values. Now that we have the coefficients, ymax value, the left and the right curvature is calculated using the radius of curvature formula.

The code can be found in radius\_of\_curvature.py

**Continuity of Lane Detection**

As my final video output ran for lane detection, the pipeline stopped at 83 percent indicating that there were no values present in the x-values list. That is when I realized that in between bad lines are detected when pipeline is processing from one frame to the next. Due to this the lane line polynomials are not formed, due to lack of line coefficients. To overcome this issue, I took the average of the coefficients of the lane lines of the newest five frames and passed it on to the function which calculates lane lines of next frames, instead of the just the coefficient the previous lane line.

**Final Video Output**