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 (https://review.udacity.com/#!/rubrics/571/view) Points

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

Writeup / README

1. Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf. Here (https://github.com/udacity/CarND-Advanced-Lane-Lines/blob/master/writeup_template.md) is a template writeup for this project you can use as a guide and a starting point.

You're reading it!

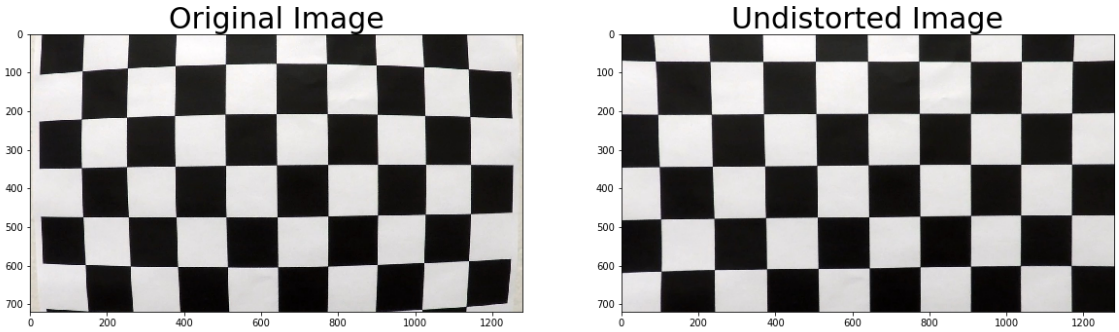
Camera Calibration

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

The code for this step is contained in the second code cell of the IPython notebook "code.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.

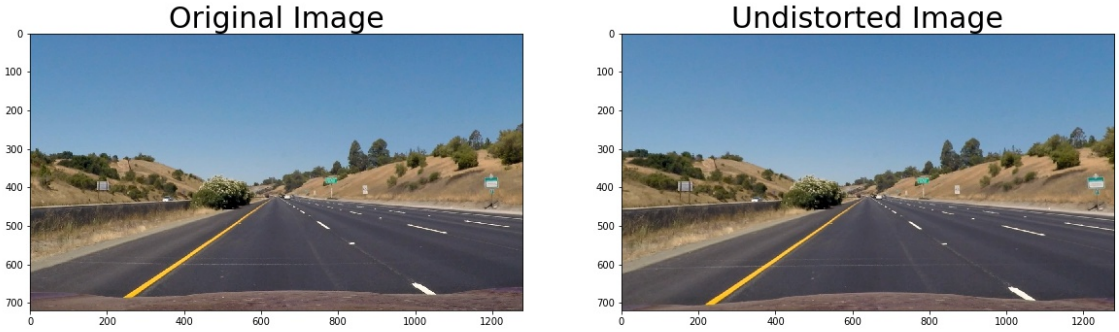
I then used the output `objpoints` and `imgpoints` to compute the camera calibration and distortion coefficients using the `cv2.calibrateCamera()` function. I applied this distortion correction to the test image using the `cv2.undistort()` function and obtained this result:



Pipeline (single images)

1. Provide an example of a distortion-corrected image.

By using the values obtained from previous camera calibration pipeline , I have defined a undistort() function which returns the distortion corrected image using cv2.undist() function. An example image is provided below

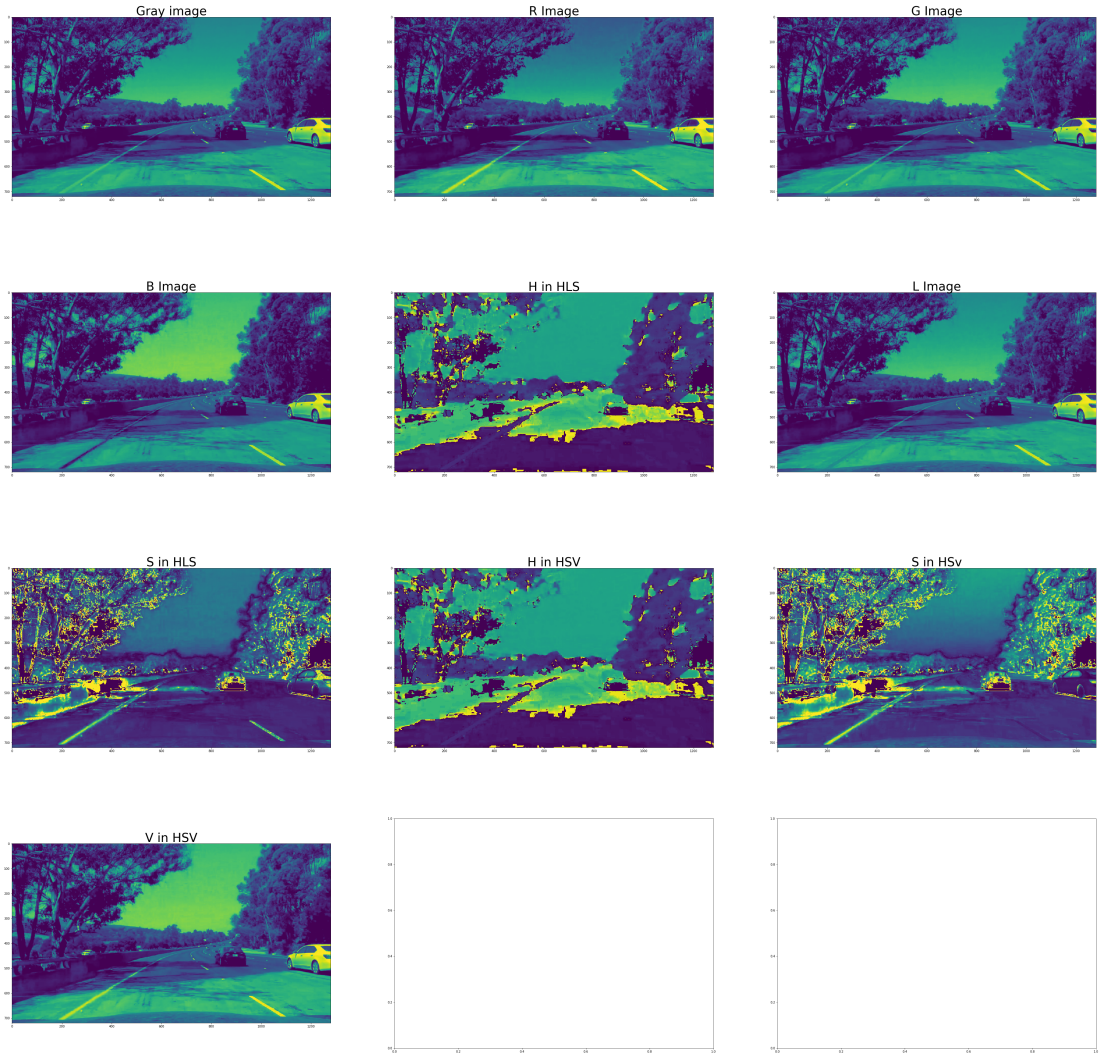


2. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

The code used to experiment with color and gradient thresholds can be found in cells [6] to [8].

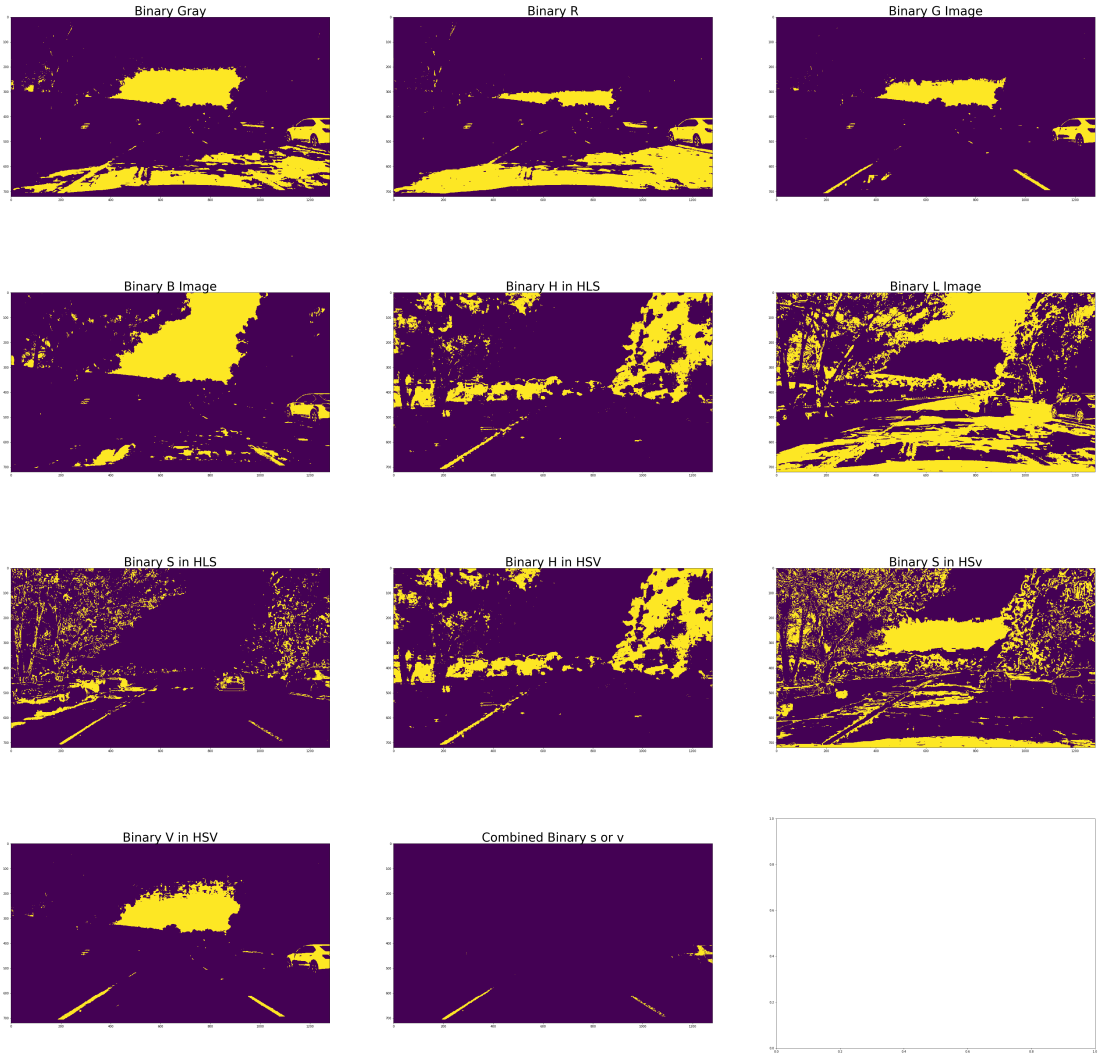
Color Spaces

First I have visualized the test image in gray, RGB, and then in HLS & HSV Color spaces. The following figure shows the test image in different color spaces. The code can be found in cell [6]



Color Thresholding

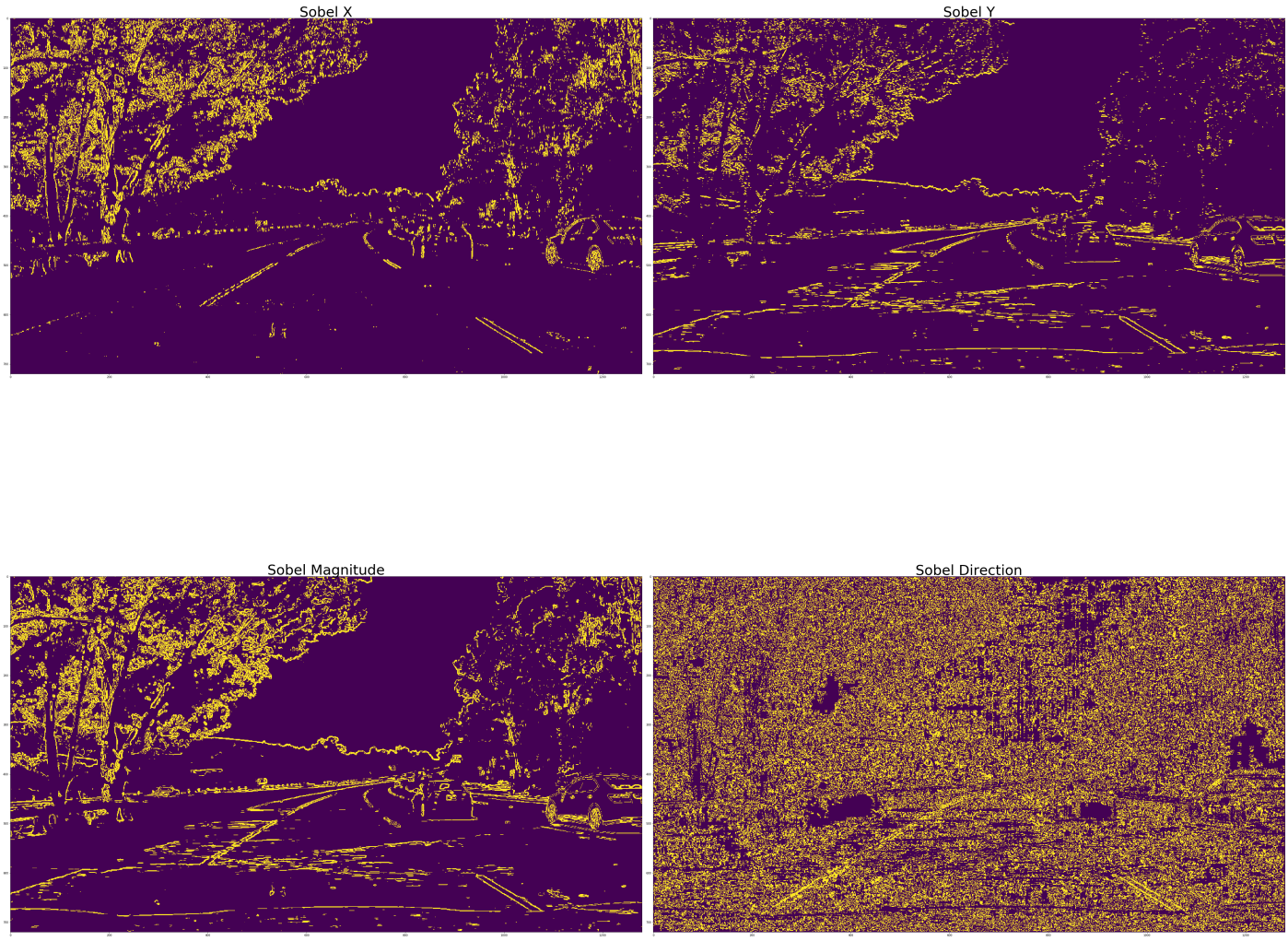
From the above figure I can see that lines are more visible in S region in HLS transformed image. Now I have applied thresholding to all the regions and the output images are shown below. The code can be found in cell [7]



After experimenting with different combinations, I found that a combination of binarized S in HLS and binarized v in HSV space gives more accurate result. The result of this combination can be seen in last image in the above figure where lane lines are clearly identified.

Gradient Thresholding

The code used to find Sobel gradients can be found in cell [8]. The following figure shows the results of applying different sobel gradients.



From above figure we can see that Sobelx is less noisy than all other gradients.

Combination of Color & Gradient Thresholding

I have used a combination of Sobelx and binary_v , binary_S_HLS in my final thresholding pipeline. This code can be found in cell [9]. The resulting binary image can be seen below.

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

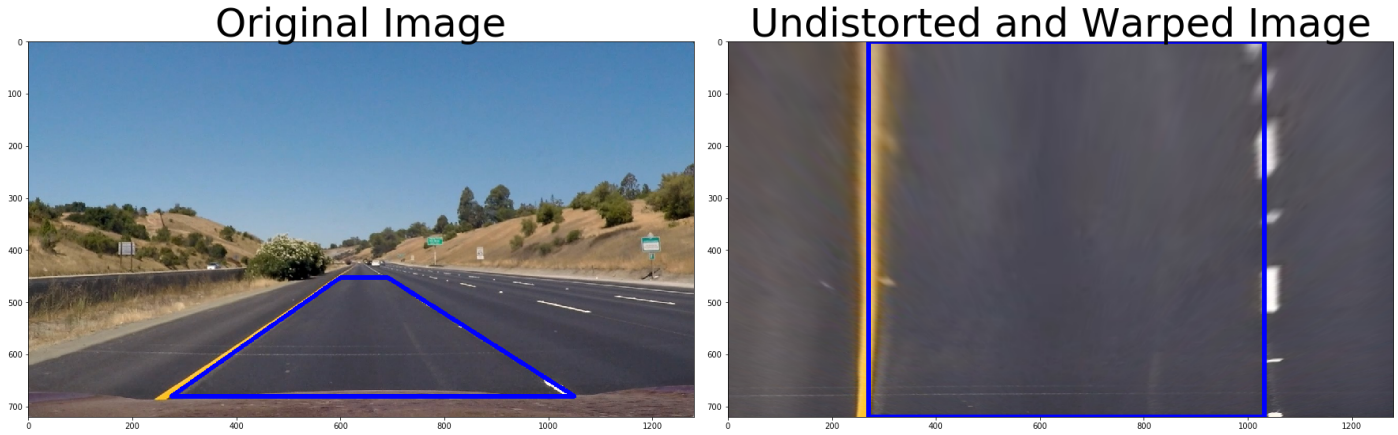
The code for my perspective transform(can be found in cell [5]) includes a function called `unwarp()`, which appears in the 5th code cell of the IPython notebook. The `unwarp()` function takes as inputs an image (`img`), and returns the perspective transform of the image. I chose the hardcode the source and destination points in the following manner:

```
src = np.float32([[600,452],[691,452],[275,680],[1050,680]])
dst = np.float32([[330,0],[1050,0],[270,img.shape[0]],[1060,img.shape[0]]])
```

This resulted in the following source and destination points:

Source	Destination
600, 452	330, 0
691, 452	1050, 0
275, 680	270, 720
1050, 680	1060,720

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.



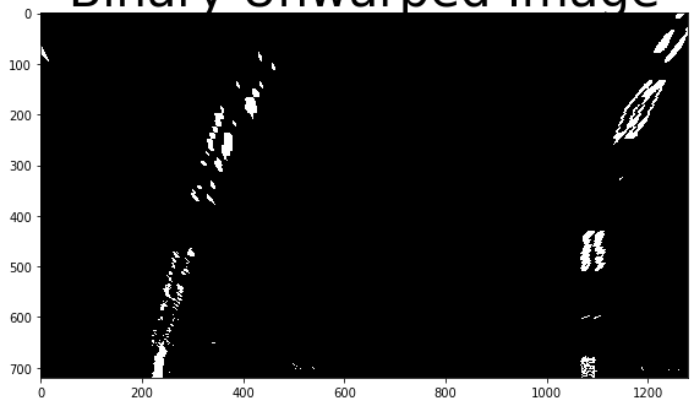
The final image after applying calibration, thresholding, and a perspective transform can be found below. We can see that the lane lines are clearly identified.

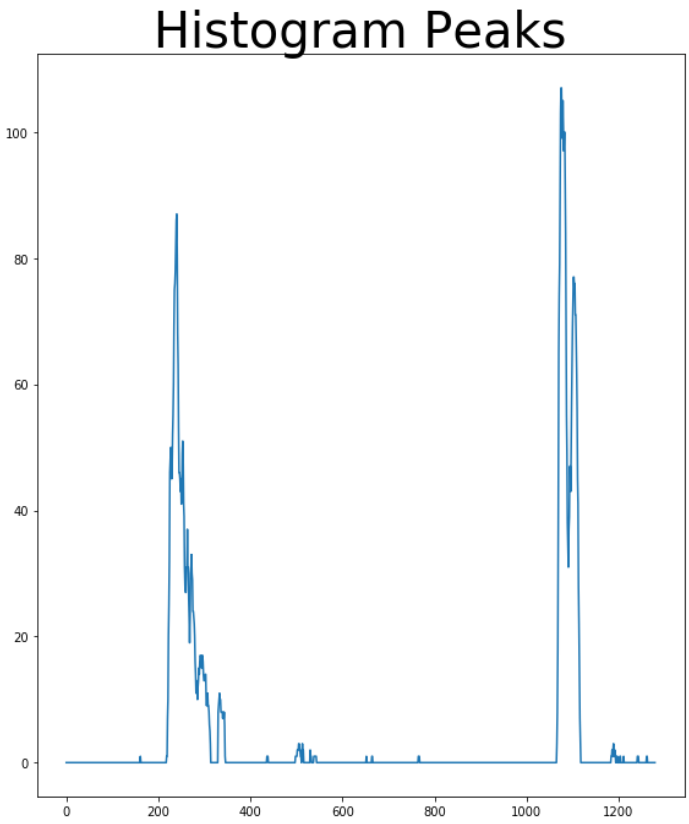
4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

The code for identifying and fitting lane lines can be found in code cells [10],[11],[12]. From the above binary image one can observe that lane lines are clearly identified. But we still need to decide which pixels are a part of the lines and we should be able to distinguish between left and right lane pixels. According udacity lesson 18, Ploting a histogram of the binary image is one potential solution for this.

Histrogram Peaks: Code cell[10]

Lane lines are likely to be mostly vertical nearest to the car. So just take the bottom half of the binary image and sum across image pixels vertically since the highest areas of vertical lines should be larger values. The resulting image can be seen below.



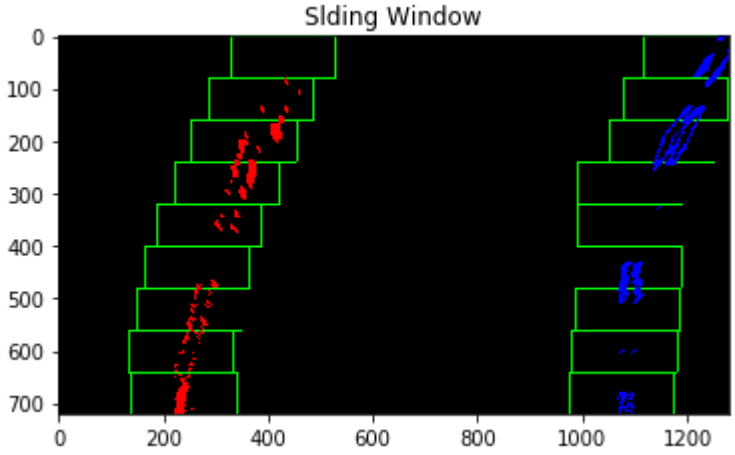


Sliding Window:

As seen in the above image we can use the two highest peaks to determine where the lines are using sliding window technique. The code can be found in code cell [11] in ipython notebook. I have used the same code that was given in the quiz. I have used find_lane_pixels() to identify pixels from each line and After finding the pixels from each line I have used np.polyfit() to fit a second order polynomial for left and right lanes to draw them on the image. I have defined a function `fit_polynomial()` which returns the output image with lines, left_fit, right_fit and left_fitx , right_fitx. Where left_fitx and right_fitx are found by fitting a curve

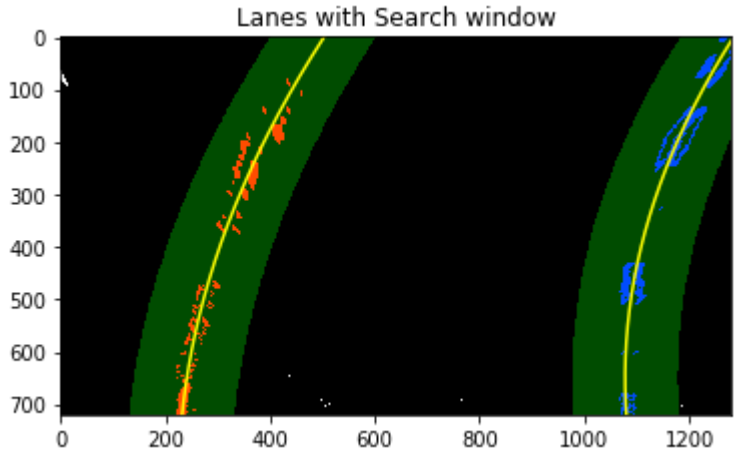
$$f(y) = Ay^2 + By + C$$

The sliding window example can be seen in the below figure.



Search Around Polynomial:

As mentioned in "Lesson 18:5:Findin the lines:Search from prior:" I have used customized region of interest which helps tracking the lanes through sharp curves and tricky conditions. Instead of blind searching again I have searched in a margin around the previous line position. In the below image the green shaded area shows where I searched for the lines. The code can be found in cell [12] of Ipython notebook. The fucntion search_around_poly returns warped image with lanes drawn and left lane fit and right lane fit X values.



5. 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 center.

The code to find the Radius of curvature can be found in cell [13] of Ipython notebook. By using the `left_fit, right_fit and ploty` values from the pervious `fit_polynomial()` function . The equation for radius of curvature is given below

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

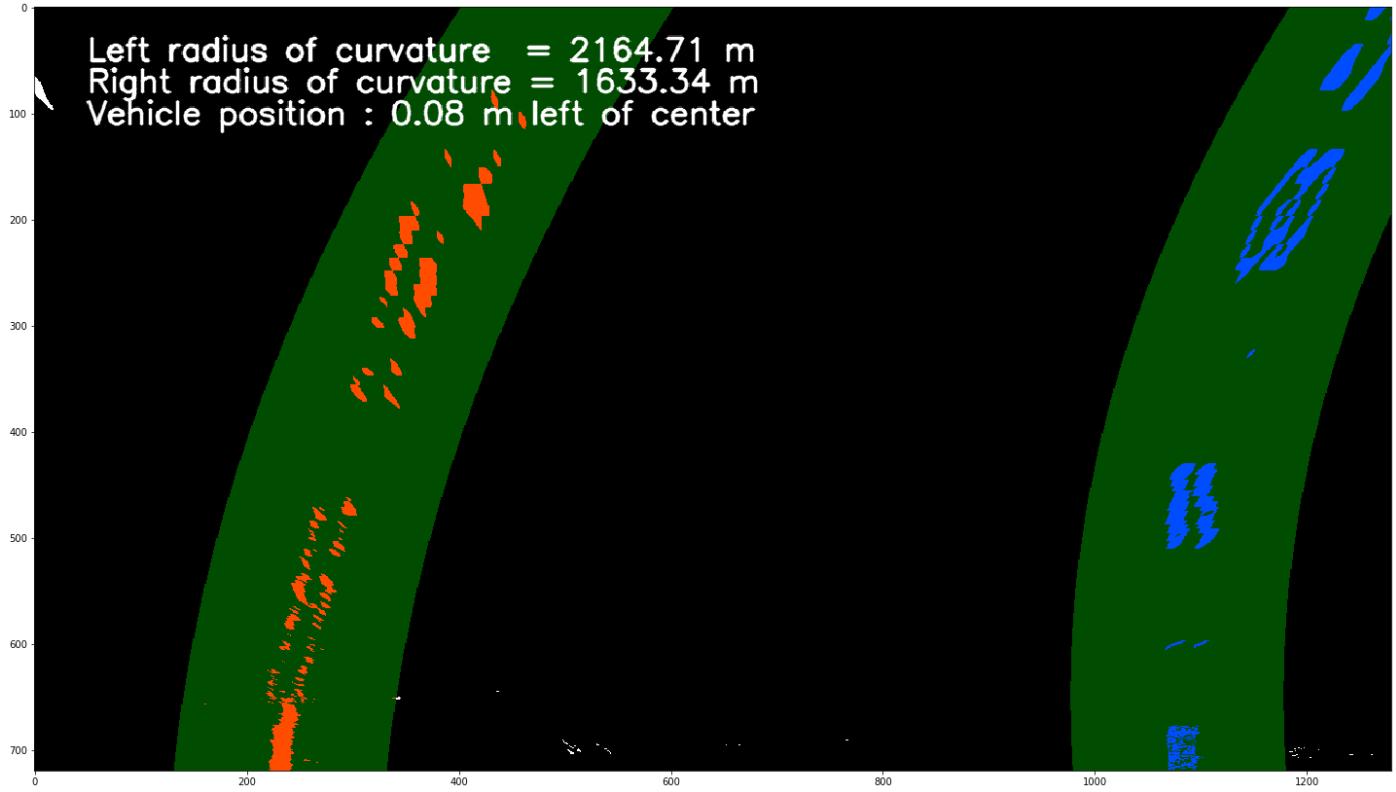
Image pixels are converted to meters by using below conversions

```
ym_per_pix = 30/720 # meters per pixel in y dimension
xm_per_pix = 3.7/700 # meters per pixel in x dimension
```

Vehicle position is Found by following code

```
center = (left_fitx[-1] + right_fitx[-1])//2
veh_pos=img.shape[1]//2
distance_from_Center = (veh_pos - middle)*xm_per_pix #Positive if on right, Negativ
e if on left
```

The test image annotated with radius of curvature can be seen below



6. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.

I implemented this step in code cell [14] in the function `process()`. Here is an example of my result on a test image:

Pipeline (video)

1. 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!).

Here's a link to my video result (./project_video_output.mp4)

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?

I have used a step by step approch and visualizing each step helped me to understand and debug the probelms easily. I have considered a few possible approaches to make my pipeline more robust.

1. Averaging the lines and selecting best fit to reduce sudden changes.
2. Dynamic thresholding of the images
3. Trying out other gradient threshold like Laplacian which improves lane detectio
n.