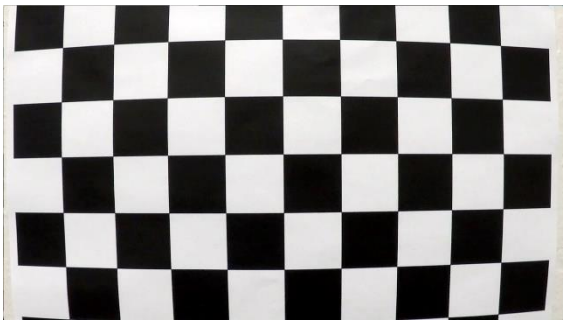
Lane Finding Writeup

Author: Jesse Prescott

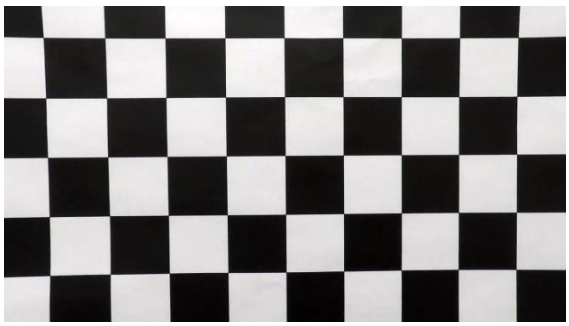
Below, I will address how I have achieved each rubric point individually. The final video can be found in the root directory in “output.mp4”.

Camera Calibration

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

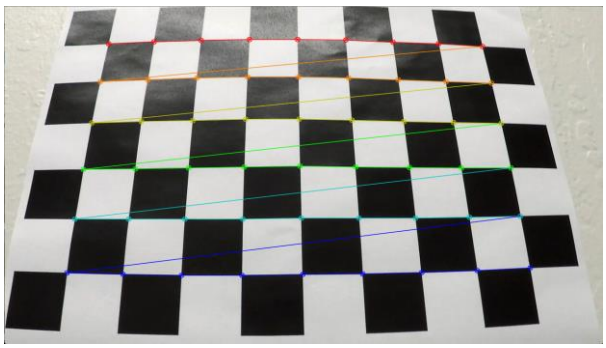


Before Calibration



After Calibration

To perform calibration, I first obtain at least 8 checkerboard images using the camera I want to calibrate. I then pass the location of the folder containing the images to the calibrate.py script. This script takes each image and uses the OpenCV function “findChessboardCorners” to locate the boundary between all of the individual checkers as shown:



```
# For each chessboard image.
for imageFile in images:

    # Load the image and convert to greyscale.
    image = cv2.imread(imageFile)
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Attempt to locate chessboard corners within image.
    ret, corners = cv2.findChessboardCorners(gray, chessboardSize, None)
```

Line 32 onwards in calibrate.py

I then append the pixel location of these points to a long list called “imgpoints”:

```
# Add new row of points.
objpoints.append(objp)

# Calculate sub pixel corner location.
cornersSubPixel = cv2.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)
imgpoints.append(cornersSubPixel)
```

Line 45 onwards in calibrate.py

As you can see, I am also using the OpenCV function “cornerSubPix”. This allows me to get a more accurate calibration by instead of defining (x, y) as integer values, but as sub-pixel positions. Once this has been done for every image, I then pass the points to the “calibrateCamera” function of OpenCV:

```
# Calibrate the camera.
ret, cameraMatrix, distortionCoefficients, rotationVectors, translationVectors = \
    cv2.calibrateCamera(objpoints, imgpoints, gray.shape[::-1], None, None)
```

Line 60 onwards in calibrate.py

Finally, I save the distortion and camera matrix as calculated by OpenCV and save it to a file called “calibrate.npy”. This allows me to load the calibration values another time, and not have to re-do the calibration:

```
np.savez(calibrationFile, cameraMatrix=cameraMatrix, distortionCoefficients=distort
ionCoefficients)
```

Line 68 onwards in calibrate.py

Pipeline

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

Before Calibration

After Calibration

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.

To create a binary threshold image, I used a combination of color transforms and the Sobel operator applied in the horizontal direction. First, I convert the undistorted image to the HLS color space and extract just the lightness and saturation channels:

```
1.  # Convert to HLS color space and seperate the lightness and saturation channels.
2.  hls = cv2.cvtColor(image, cv2.COLOR_BGR2HLS)
3.  l_channel = hls[:,:,1]
4.  s_channel = hls[:,:,2]
```

Line 43 onwards in pipeline.py

Next, I apply the sobel operator in the x direction to the lightness channel of the image we just extracted. This allows me to better find the road markings as they tend to be close to vertical:

```
1.  # Apply the sobel operator in the x direction.
2.  sobelx = cv2.Sobel(l_channel, cv2.CV_64F, 1, 0)
3.  abs_sobelx = np.absolute(sobelx)
4.  scaled_sobel = np.uint8(255*abs_sobelx/np.max(abs_sobelx))
```

Line 48 onwards in pipeline.py

I then apply a threshold to convert the Sobel-transformed image to a binary image. In this case, I only mark a pixel as white in the binary image of the Sobel value is between 20 and 100 inclusive:

```
1.  # Apply a threshold to the sobel output to create a binary image.
2.  sx_binary = np.zeros_like(scaled_sobel)
3.  sx_binary[(scaled_sobel >= sx_thresh[0]) & (scaled_sobel <= sx_thresh[1])] = 1
```

Line 53 onwards in pipeline.py

I then apply a similar threshold system to the saturation channel we extracted to create a separate binary image:

```
1.  # Apply a threshold to the saturation channel.
2.  s_binary = np.zeros_like(s_channel)
3.  s_binary[(s_channel >= s_thresh[0]) & (s_channel <= s_thresh[1])] = 1
```

Line 57 onwards in pipeline.py

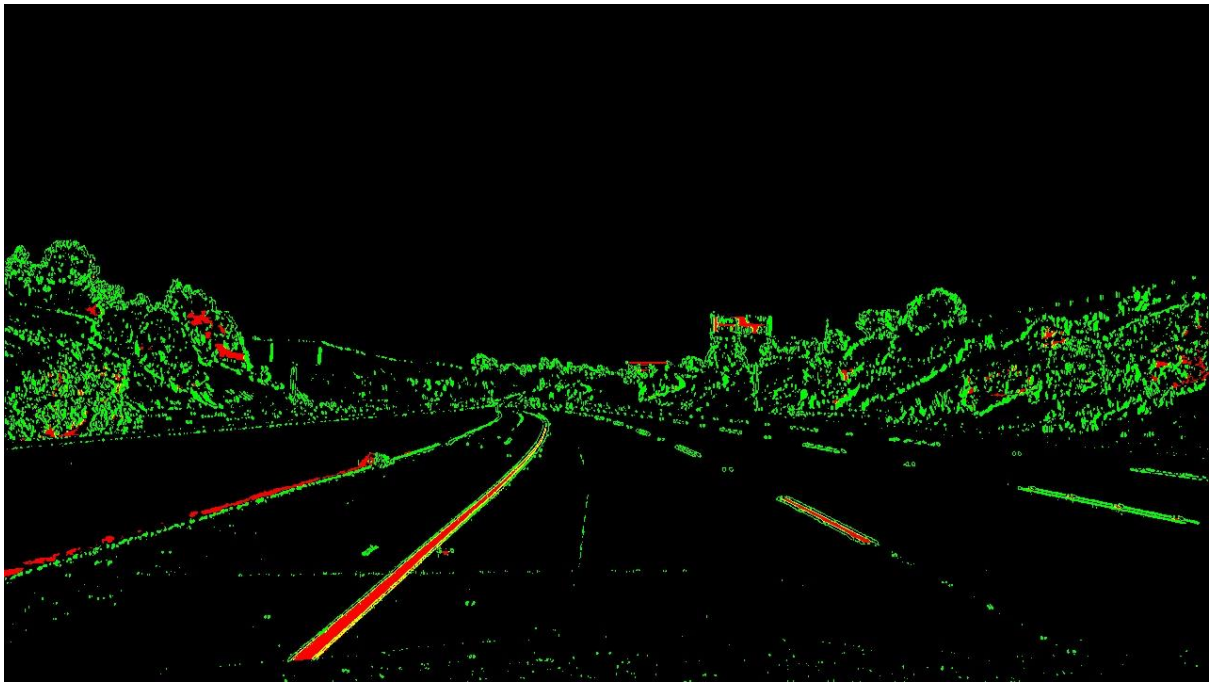
Finally, I take the two binary images we've created and combine them to make a single binary image using both techniques as shown below:

```
1.  # Stack each binary image together.
2.  combined_binary = np.zeros_like(sx_binary)
3.  combined_binary[(sx_binary == 1) | (s_binary == 1)] = 255
```

Line 61 onwards in pipeline.py

This creates a binary image like the following:

This image shows which parts of the two binary images are contributing to make the final image:



Here, the red is the binary image created using the Sobel operator on the lightness channel and the green is the binary image created by applying a threshold to the saturation channel.

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

First, I found an image where the ego car is travelling down a straight road. I then selected a portion of the straight road that represents the flat floor/ground. The road is assumed to be perfectly flat throughout.

I then use this trapezoid selected by the user and get OpenCV to calculate the transformation matrix that would turn it into a perfect top-down rectangle:

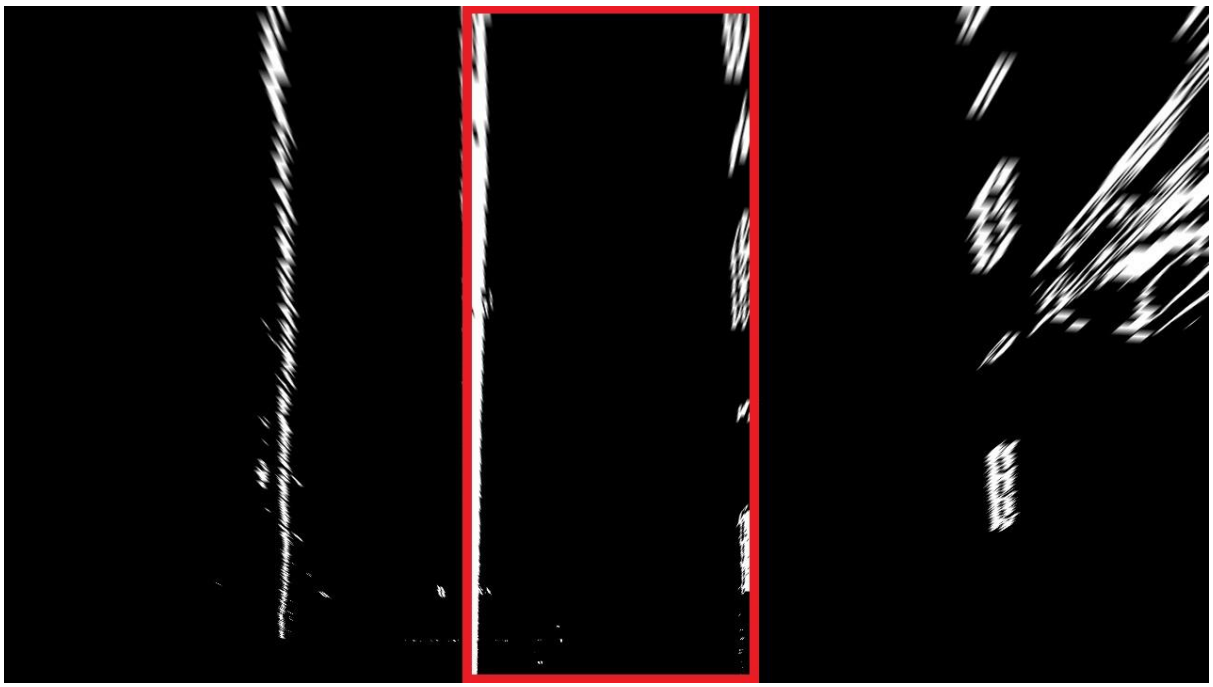
```
# Trapazoid as selected by the user.
tl = [297, 667]
tr = [1023, 665]
br = [679, 443]
bl = [605, 444]
trapazoid = np.float32([tl, tr, br, bl])

# Calculation of destination points for trapazoid.
height, width = image.shape
x_window_width = 350
imageVerticies = np.float32([[0+x_window_width, height], [width-
x_window_width, height], [width-x_window_width, 0], [0+x_window_width, 0]])

# Calculate perspective transform.
transformMatrix = cv2.getPerspectiveTransform(trapazoid, imageVerticies)
inverseTransformMatrix = cv2.getPerspectiveTransform(imageVerticies, trapazoid)
```

Line 88 onwards in pipeline.py

Finally, I use the OpenCV function “warpPerspective” to use the calculated matrix to actually transform the image:

```
1.  # Apply perspective transform.
2.  image = cv2.warpPerspective(image, transformMatrix, (width, height))
```

Line 104 onwards in pipeline.py



Transformed Image

The red box is the original trapezoid selected by the user now that it has been transformed.

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

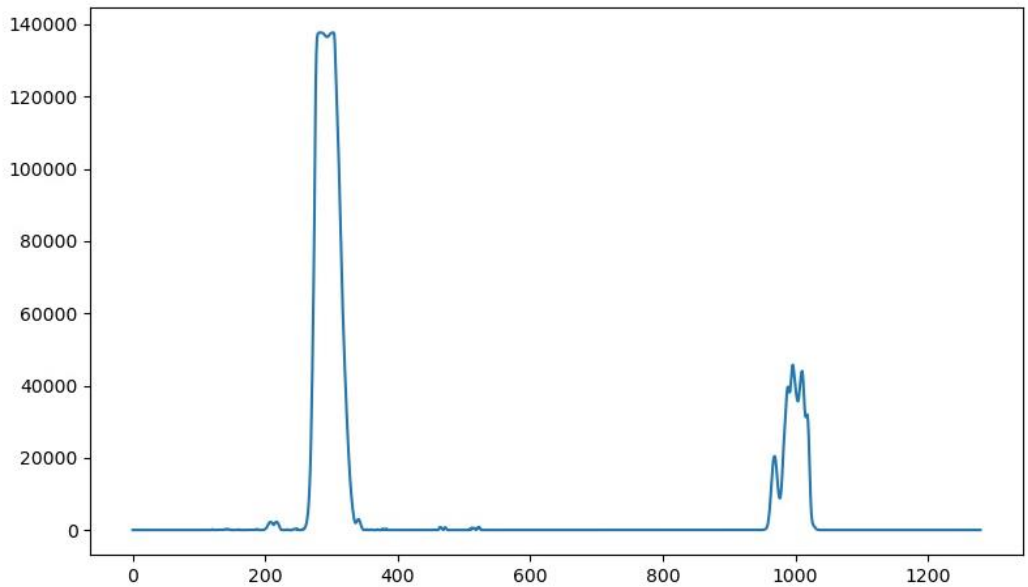
This portion consists of two methods. First a sliding window technique is used to find the initial location of the lane lines and fit the first polynomials. Once this has been done at least once, the polynomials from the previous frames can be used to create a margin to look for pixels to calculate the next polynomials. This increases speed of the pipeline and also stability as a few bad frames resulting in bad binary images do not cause drastically incorrect polynomials.

Sliding Window

First, I take the perspective-transformed binary image and calculate the histogram for the bottom half of the image. It's important to only calculate the histogram for a bottom portion of the image as we are trying to find where the lane lines closest to the ego car begin.

```
1.  # Get the bottom quarter of the image and create
2.  # a histogram.
3.  image_bottom = image[image.shape[0]//2:,:]
4.  histogram = np.sum(image_bottom, axis=0)
```

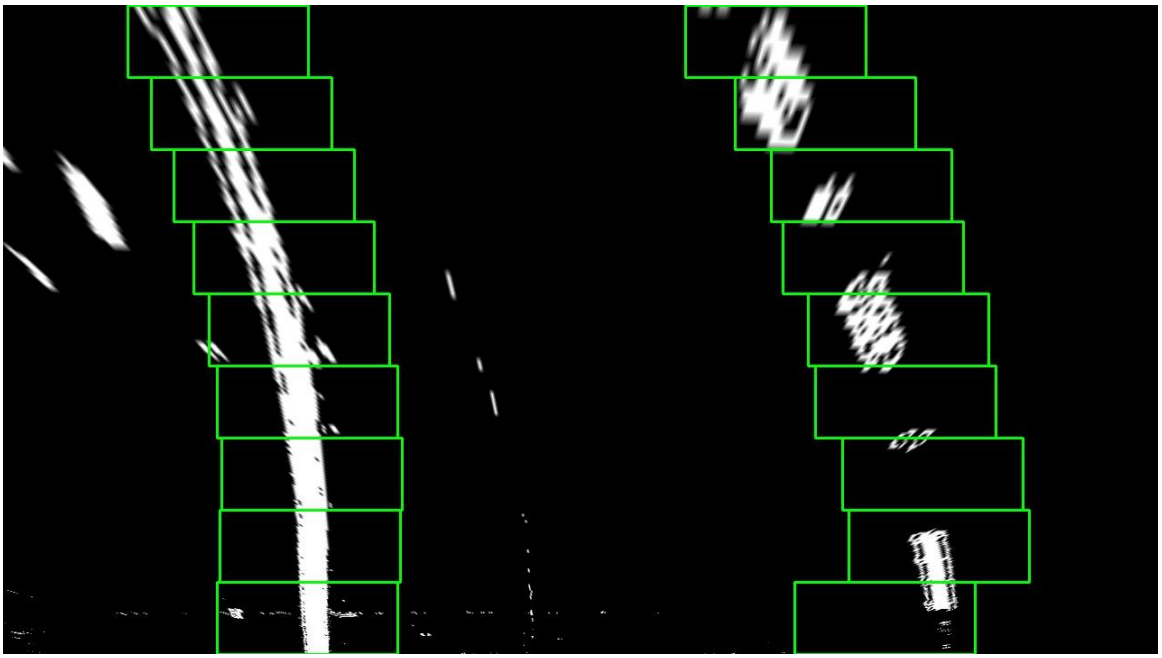
Line 131 onwards in pipeline.py



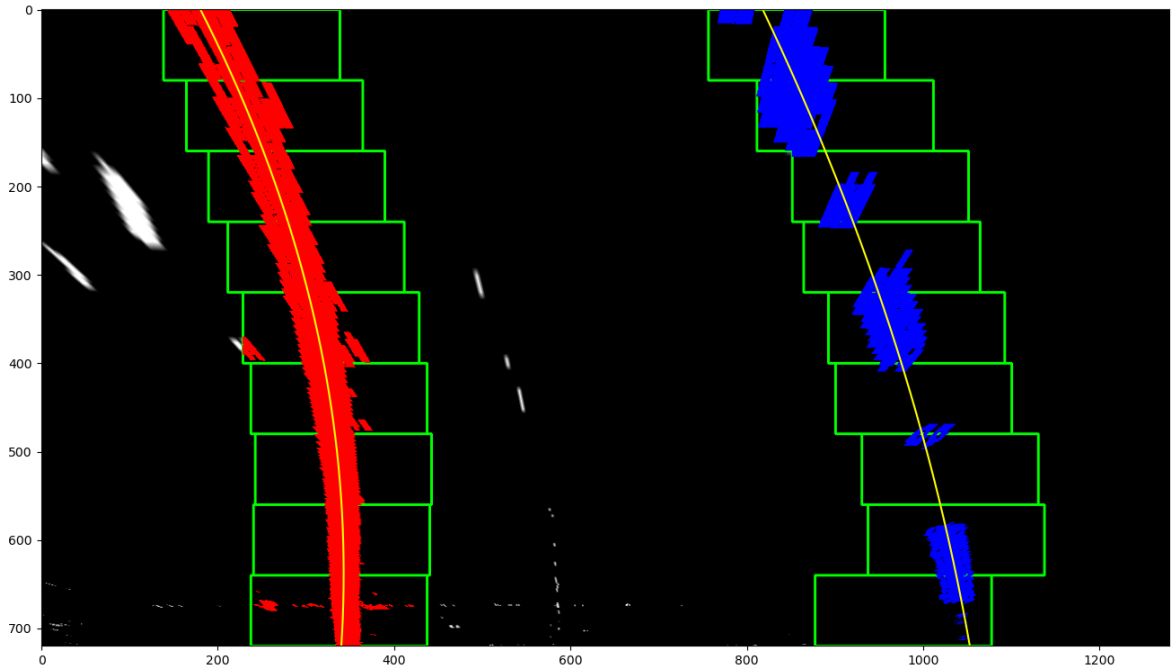
Next, I split the histogram in half and look for the peaks in the histogram on both sides. This gives me a good location to place my first sliding windows:

```
# Find the highest peaks on the left and righ half of the histogram.
histogram_midpoint = np.int(histogram.shape[0]/2)
leftx_base = np.argmax(histogram[:histogram_midpoint])
rightx_base = np.argmax(histogram[histogram_midpoint:]) + histogram_midpoint
```

Given I now have the starting positions for my sliding windows, I use the average position of all of the activated pixels within the sliding window to calculate the position of the next window as we travel upwards. This allows me to create sliding windows the follow the curve of the lane lines upwards as shown:



I then place all the (x, y) coordinates of the activated pixels that lie within the sliding windows and fit a polynomial through them. One for the left side and one for the right side.



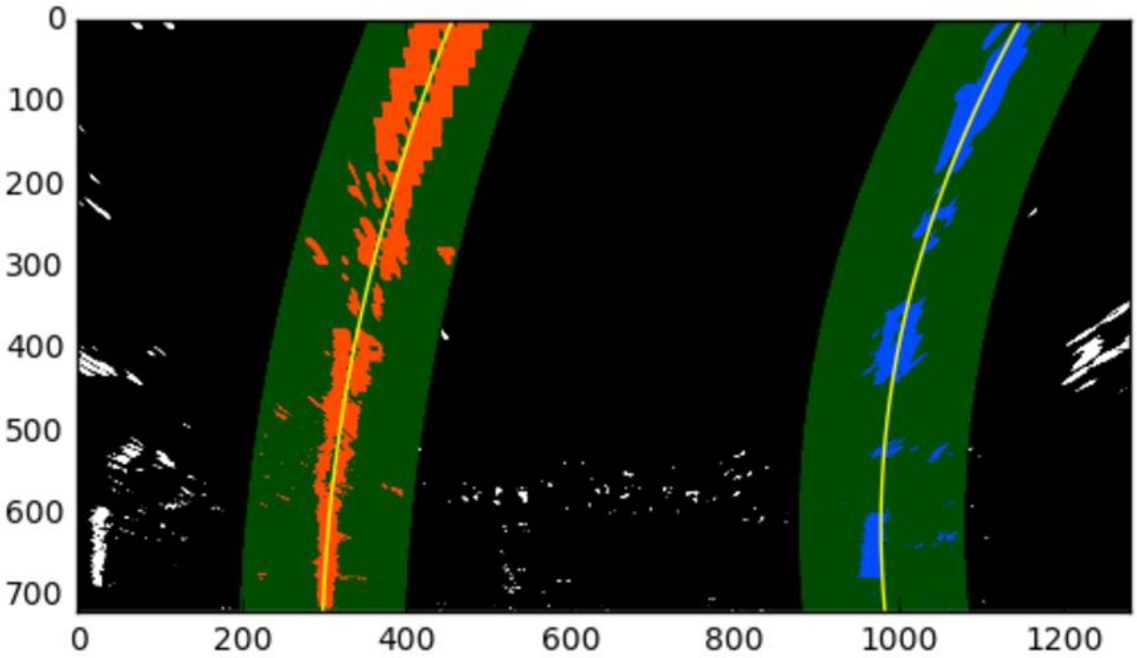
To fit the polynomials I use the Numpy function “polyfit”

```
# Fit a second order polynomial through our pixels.
left_fit = np.polyfit(lefty, leftx, 2)
right_fit = np.polyfit(righty, rightx, 2)
```

Line 208 onwards in pipeline.py

Polynomial Margin

Now that we have some initial polynomials, instead of using the entire sliding window process again on new frames we can use a margin around these polynomials to calculate new ones:



Any pixels that lie within the green margin of each polynomial are used to re-fit a new polynomial.

```
# Find activated pixels within a certain margin of our previous polynomials.
margin = 20
left_lane_pixels = ((nonzerox > (previous_poly_left[0]*(nonzeroy**2) + previous_poly_left[1]*nonzeroy +
        previous_poly_left[2] - margin)) & (nonzerox < (previous_poly_left[0]*(nonzeroy**2) +
        previous_poly_left[1]*nonzeroy + previous_poly_left[2] + margin)))
right_lane_pixels = ((nonzerox > (previous_poly_right[0]*(nonzeroy**2) + previous_poly_right[1]*nonzeroy +
                previous_poly_right[2] - margin)) & (nonzerox < (previous_poly_right[0]*(nonzeroy**2) +
                previous_poly_right[1]*nonzeroy + previous_poly_right[2] + margin)))
```

Line 219 onwards in pipeline.py

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.

To calculate the curvature of the lane, I calculate the curvature of each of our two polynomials using the following equation:

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

Where A and B are the coefficients of our 2nd order polynomial. I then take the mean of these two curvature values to give us the rough lane curvature.

```
1. left_curvature = ((1 + (2*left_fit[0]*y_eval*ym_per_pix + left_fit[1])**2)**1.5) /
   np.absolute(2*left_fit[0])
2. right_curvature = ((1 + (2*right_fit[0]*y_eval*ym_per_pix + right_fit[1])**2)**1.5)
    / np.absolute(2*right_fit[0])
3. curvature = int((left_curvature + right_curvature) / 2)
```

Line 282 onwards in pipeline.py

However, this gives us the curvature in pixel space. To convert to metres, we define a multiplier that represents the number of pixels in every metre in each direction. To calculate this multiplier, I roughly eyeball a 1mx1m grid knowing the standard lane width of an American road.

```
1. ym_per_pix = 400/720 # The meters per pixel in y dimension
2. xm_per_pix = 3.7/700 # The meters per pixel in x dimension
```

I then place the calculated curvature in metres as text to the image in the top right.

To calculate the ego car's offset from the centre of the lane, I simply calculate the midpoint of the lane by taking the mean of the x coordinates where the two polynomials intersect the bottom of the image. I then calculate how far away this midpoint is from the centre of the image. Finally, I multiply this with our multiplier to convert from pixels to metres.

```
1. # Calculate the cars offset from the lane centre.
2. left_point = left_fit_x[-1]
3. right_point = right_fit_x[-1]
4. lane_midpoint = (left_point + right_point) / 2
5. offset = round(((originalImage.shape[1]/2) - lane_midpoint) * xm_per_pix, 2)
```

Line 292 onwards in pipeline.py

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

To create this final image, I used the two polynomials as edges to create a polygon using the OpenCV "fillPoly" method. I then use the inverse perspective transform matrix to convert back from birds-eye view. Finally, I combine this with our original image as shown above.

```
1.  # Fill the region created by our polynomials points.
2.  cv2.fillPoly(mask_image_color, np.int_([points]), (0, 255, 0))
3.
4.  # Transform the masked image back from bird's eye view.
5.  mask_image_color = cv2.warpPerspective(mask_image_color, inverseTransformMatrix, (i
    mage.shape[1], image.shape[0]))
6.
7.  # Combine the masked image and the original image.
8.  final_image = cv2.addWeighted(originalImage, 1, mask_image_color, 0.3, 0)
```

Line 269 onwards in pipeline.py

7) 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!)

Please see the final video “output.mp4” in the root directory of this repository.

Discussion

8) 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?

Problems Faced

- At first, I struggled with calibration and the coefficients I were getting didn’t seem to be result in fixing the distortion very well. I eventually realised that because I told OpenCV the checkboard was 6x9 and not 9x6 it was actually picking up the checkerboards at 90 degrees. This resulted in the vertical edges of the image using the horizontal edge’s distortion coefficients and vice versa.

- Originally, I was using only the sliding window approach. This resulted in a few frames were the tarmac changed colour to cause the sliding windows to completely fail. No matter how much I tuned the binary image thresholds, I could not fix this issue. When I implemented the polynomial margin, this issue disappeared because it is now using the previous frames polynomial to help find the lane lines in a difficult image.

Limitations of Pipeline

- If the camera mounting location or angle were to change, the pipeline’s ROI and perspective transforms would be completely wrong. This might even lead to the pipeline not detecting any lanes at all.
- Bad weather such as rain would cause the binary image to detect little to no lane lines at all.
- Different cameras and lighting conditions will always result in having to tweak the threshold values for the binary image. It’s extremely hard to use computer vision techniques like this that can deal with all scenarios.

Improving the pipeline might be done by steering clear of computer vision techniques like this at all and moving to a machine learning approach.