Carissa Chan

Final Writeup: Advanced Lane Finding Project

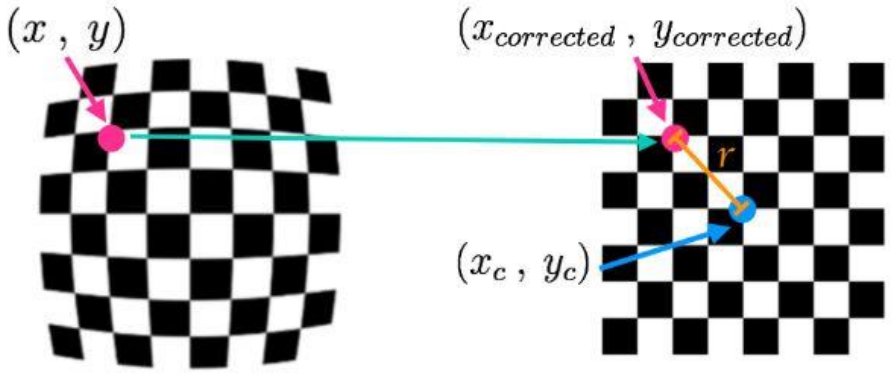
The goals / steps of this project are the following:

- 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.

Camera Calibration

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

Simply put, distortion correction translates 3d objects in the real world space. 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. The distortion coefficients reflect the radial or tangential distortion relative to the image center for various angled lenses. They will be plugged into the correction formula to translate a real-world coordinate to its respective location on the corrected image plane. I applied this distortion correction to the test image using the `cv2.undistort()` function and obtained this result:

$$x_{distorted} = x_{ideal}(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

$$y_{distorted} = y_{ideal}(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

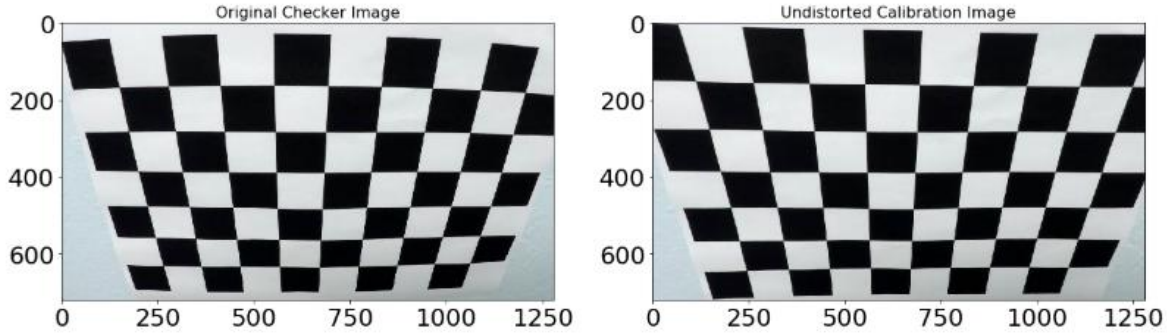
Radial distortion correction.

$$x_{corrected} = x + [2p_1 xy + p_2(r^2 + 2x^2)]$$

$$y_{corrected} = y + [p_1(r^2 + 2y^2) + 2p_2 xy]$$

Tangential distortion correction.

To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like the one seen below.



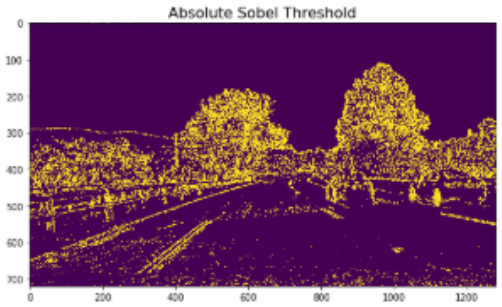
2. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

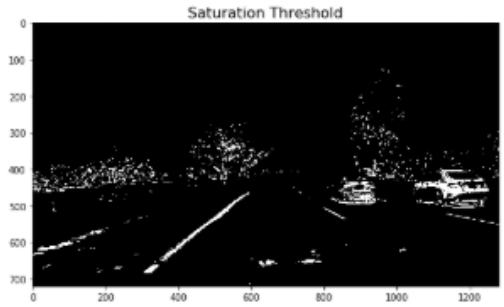
I used a combination of color and gradient thresholds to generate a binary image (thresholding steps at lines # through # in `another_file.py`). After a series of trial and errors, I picked only a subset of the thresholds, namely gradient, directional, R/G channels, and saturation range from the HLS colorspace to process the image as applying all thresholds had filtered out too few pixels for certain test images.

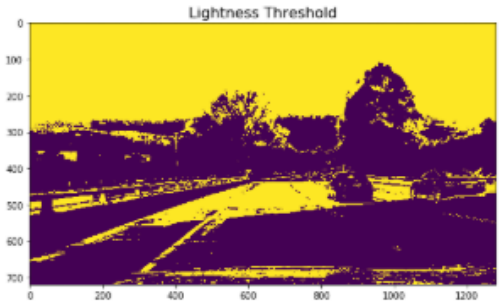
The gradient threshold represents changes in the color intensity. Assuming the lanes generally contrast from the pavement on both sides, it was more advantageous to look at the gradient along the horizontal axis. Therefore, when defining the gradients, we are interested in two aspects: how sensitive our algorithm is to intensity fluctuations and in which direction can we capture these changes such that it is unique to the target feature.

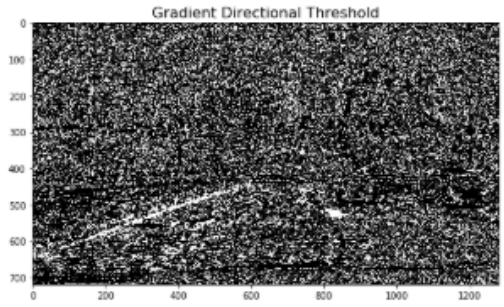
Looking at the individual binary outputs, one can see the saturation threshold was the best at isolating the lane lines. From adjusting and toggling the lightness threshold, I noticed that while it removed a significant amount of white noise within the lane of test5.jpg, it would almost wipe out most of the left line for test4. Therefore, I wanted to choose constraints that were more generic in different environments.

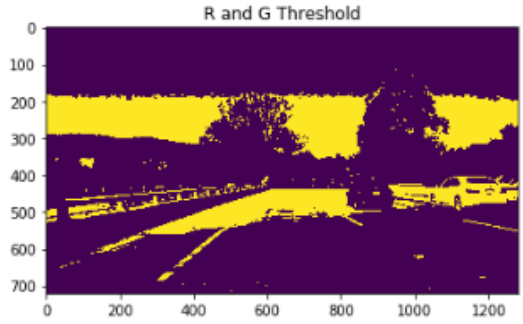
Color Threshold Transformations





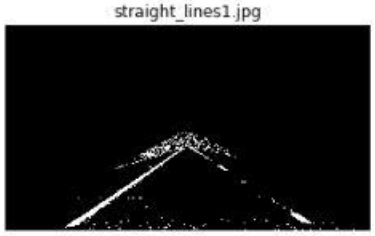




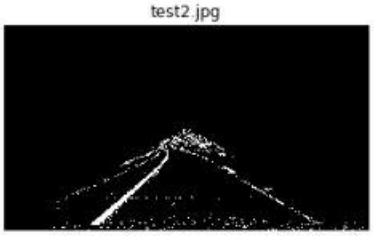


Binary Output of Color Threshold Transformations

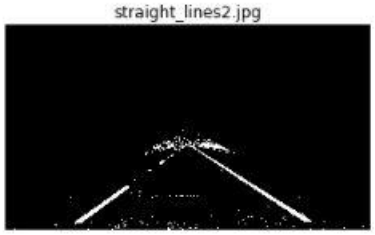
straight_lines1.jpg



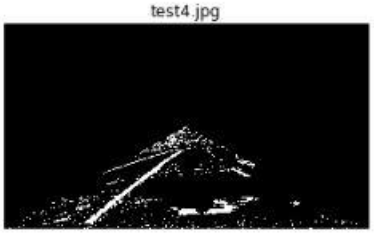
test2.jpg



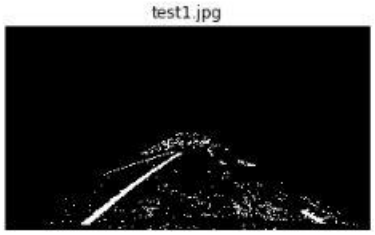
straight_lines2.jpg



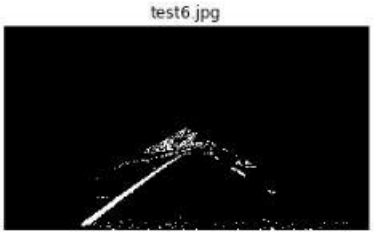
test4.jpg



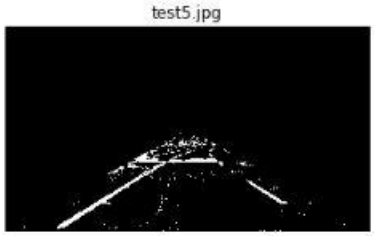
test1.jpg



test6.jpg

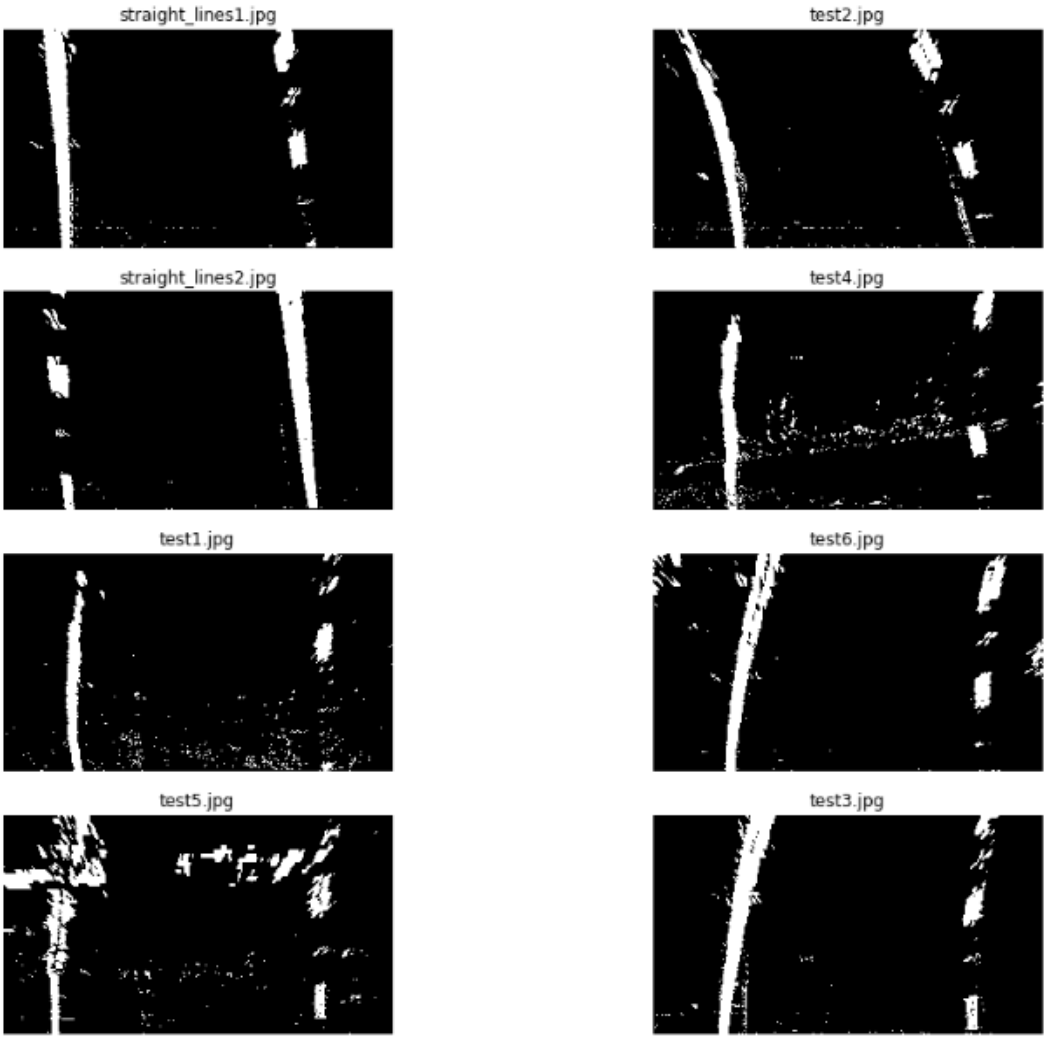


test5.jpg



test3.jpg

Binary Warped Color Threshold Transformations



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

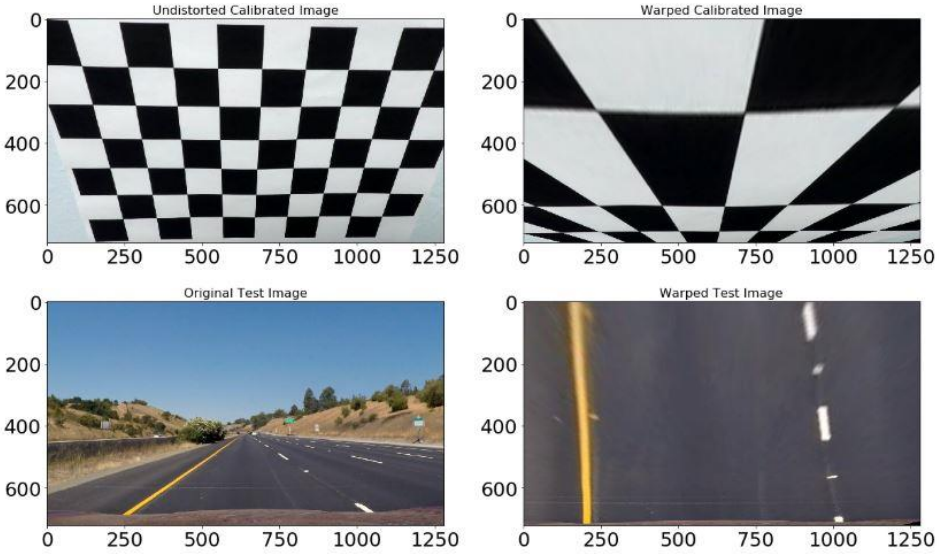
The code for my perspective transform includes a function called `warp()`. I started by eye-balling the source and destination points based on a plotted test image. These matrices are then passed to getPerspectiveTransform() , where the coordinates of the target region from the front view will be translated to those on a top-down view. The inverse matrix will also be computed as that will be later used when the left and right lane indices from the polynomials fitted aerially need to be converted back to the driver's perspective.

```
src_coordinates = np.float32(
  [[280,  700],  # Bottom left
   [595,  460],  # Top left
   [725,  460],  # Top right
   [1125, 700]]) # Bottom right

  dst_coordinates = np.float32(
  [[250,  720],  # Bottom left
   [250,    0],  # Top left
   [1065,   0],  # Top right
   [1065, 720]]) # Bottom right
```

def warp():

…

```
    # Compute the inverse perspective transfor also by swapping the input parameters
    Minv = cv2.getPerspectiveTransform(dst_coordinates, src_coordinates)

    # Create warped image - uses linear interpolation
    warped = cv2.warpPerspective(img, M, img_size, flags=cv2.INTER_LINEAR)
```



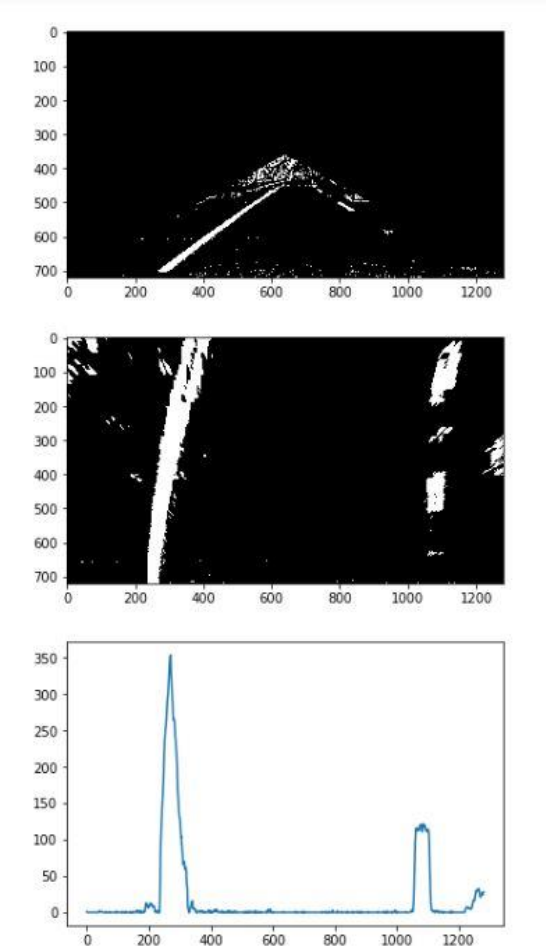
def fill_lane():

…

```
    # Warp the blank back to original image space using inverse perspective matrix (Minv)
    newwarp = cv2.warpPerspective(color_warp, Minv, (img.shape[1], img.shape[0]))
```

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

After applying the various color and gradient thresholds stated above, a histogram was generated for the bottom half of the binary image along the vertical (axis = 0) as this is how you can easily detect peaks where lanes exist.

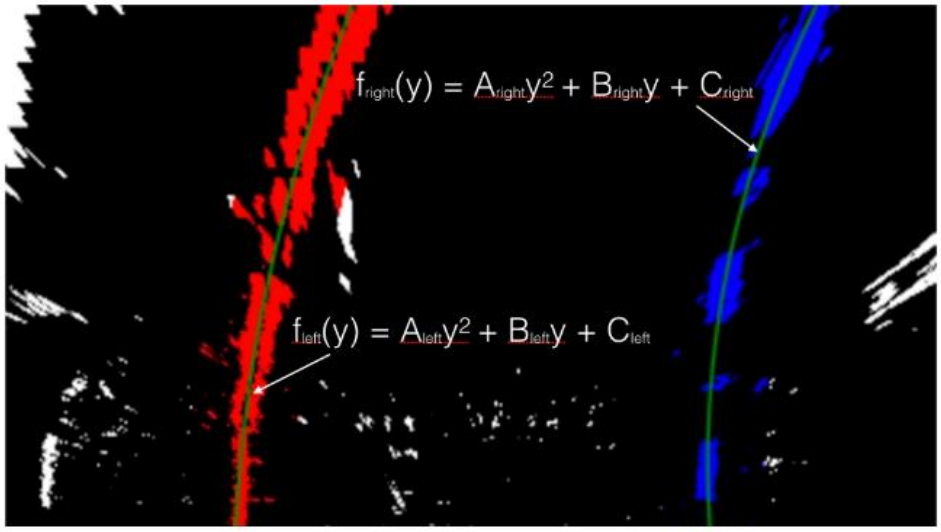


The location where these peaks occur gives us a starting point for where to search for “activated” or nonzero lane pixels. These points will be known as the leftx_base and rightx_base. A margin is created to define the window around the base values for which nonzero pixel indices should be extracted. When found, they should be stored in their respective good_left_inds and good_right_inds arrays. Based on the indices identified, an average of the indices for each side will be calculated and used to update the leftx_current and rightx_current values, which were initially set to leftx_base and rightx_base. This shifts the search window laterally as the program iterates down n windows. Once we have all the (x,y) pairs of nonzero pixels within each window, they need to be concatenated and then sorted into separate x and y indinces for the left and right lanes, leftx, lefty, rightx, and righty. Now, the most important step is fitting a second order polynomial to these points and then converting from pixel to real-world space units. A second fit is then performed on pixels extracted within a margin defined around the previous curve.

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 curvature is calculated using the following equations in the function curvature_radius(). The arguments required are passed from the previous find_lane_pixels(), where all the left and right points (left_fitx and right_fitx) were generated from the fitted polynomials. The curvature will be found about the lowest point of the image, hence "y_eval = np.max(ploty)".

The last reading on the display below is the lateral offset between the image's and car's center, which is assumed to be the center of the detected lane. Again, this is taken from the bottom of the image, explaining why the last element is extracted in "car_pos = (left_fitx[-1] + right_fitx[-1])/2".



In the case of the second order polynomial above, the first and second derivatives are:

$$f'(y) = \frac{dx}{dy} = 2Ay + B$$

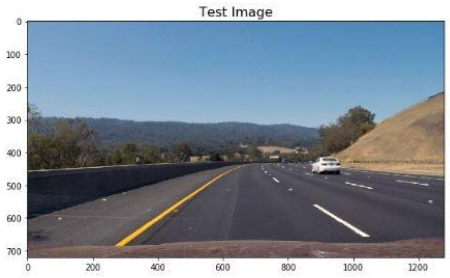
$$f''(y) = \frac{d^2x}{dy^2} = 2A$$

So, our equation for radius of curvature becomes:

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

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

Last but not least, you have the last function of the pipeline fill_lane()



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?

1. There are portions of the code that should be more dynamic. a. source and destination points: Instead of hard cording the coordinates, perhaps the region should be defined relative to the image size.
2. More efficient lane pixel search could have been implemented: Instead of performing a search and/or transformation on the full scope of the image, a class Line could be implemented to store. Each line object may contain critical parameters such as the left_fitx and right_fitx points. This will allow for a more targeted search in the next frame by only detecting activated pixels +/- a certain margin from the previous set of extrapolated points.
3. Certain thresholds worked very well *alone* in different environments: Rather than always combining the thresholds, it may be better to implement a set of thresholds depending on the output of the nonzerox and nonzeroy arrays. If we find there are too few pixels extracted for either side, we may implement a recursion or switch case scenario that will implement a different set of thresholds until a minimum number extracted is met.
4. The curvature could be more accurate: If this were a windy road meandering more erratically within the target frame, the curvature calculated at the bottom of the image would be quite inaccurate.
5. Could use a regression model to update coefficients and trajectory predictions: Maybe we can go one step further of updating parameters such as distortion when there are changes in the environmental conditions that may impact the vision system. Additionally, we can “fill in the blanks” for very obscured lane markings using a prediction algorithm that has been trained to a diverse image dataset.