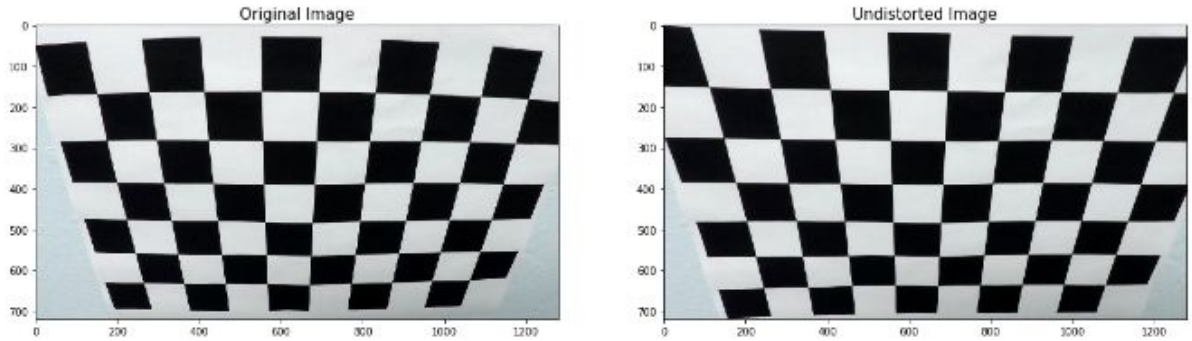
CarND Term 1 Project 4

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 first code cell of the IPython notebook located in "project4.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)

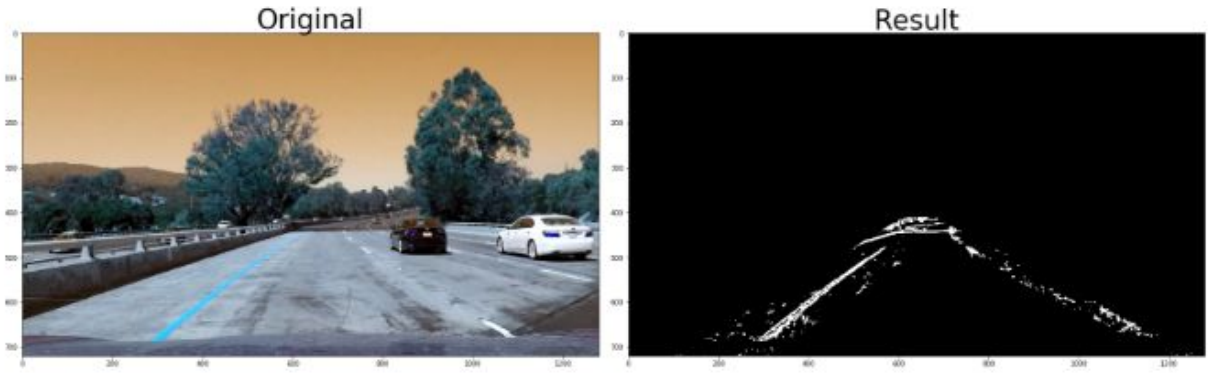
1. Provide an example of a distortion-corrected image.

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

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 in code cell 3. More specifically, I convolve the image with a gaussian to smooth it, convert to HLS and grayscale, then perform thresholding and apply sobel filters in the x and y directions. I then combine the outputs of these operations and apply a ROI mask.

Here's an example of my output for this step.



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()`, which appears at the top of cell block 4.

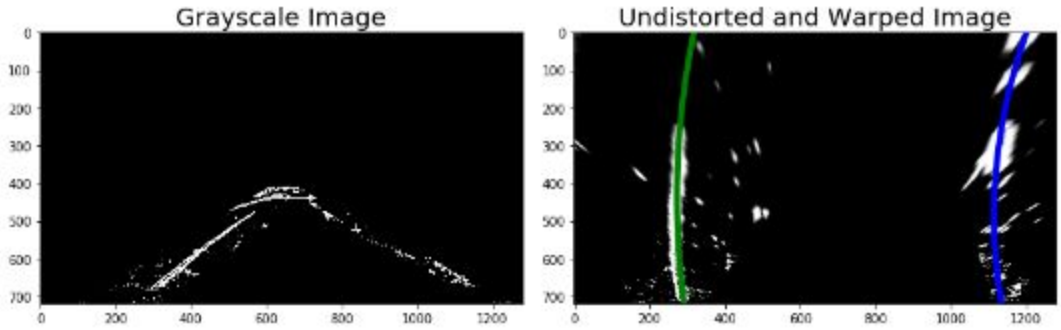
I chose the hardcode the source and destination points in the following manner:

```
    # offset from image corners to plot detected corners
    offset1 = 200 # offset for dst points x value
    offset2 = 0 # offset for dst points bottom y value
    offset3 = 0 # offset for dst points top y value


    # grab outer 4 detected corners in src
    src = np.float32(area_of_interest)


    # choose points that make the warping look nice in dst image
    dst = np.float32([[offset1, offset3],
                      [img_size[0]-offset1, offset3],
                      [img_size[0]-offset1, img_size[1]-offset2],
                      [offset1, img_size[1]-offset2]])
```

Outputs from this operation are:



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

After the perspective transform I begin by looking for peaks in the image histogram. This is done using multiple histogram windows. Once points to look for are identified their coordinates are added to an array that is then used to fit lines to. This 2nd order polynomial then defines the lines that are drawn onto the lane lines. The functions i defined for this are called find_peaks(),fit_lanes() and find_lanes().

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.

This is again in the 4th cell block of the notebook. It is carried out using the functions find_curvature() and find_position(). The radius of curvature was calculated using the 2nd derivative definition:

```
curverad = ((1 + (2*fit_cr[0]*y_eval*ym_per_pix + fit_cr[1])**2)**1.5) \

                    /np.absolute(2*fit_cr[0])
```

The position of the vehicle was then determined by taking the min and max position to the left and right of the center of the image where the y pixel value is greater than 700 (near the hood of the car). The position was then taken to be half the distance between these two points, and then converted to world coordinates using the scale factor 3.7/700.

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

This is again in code cell 4,using the draw_poly() function. 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 the video: https://youtu.be/S_SWPIEqrY8

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?

After the initial project submission I noticed I had missed one multiplicative term in the radius of curvature calculation that caused the radii to be much larger than they should have been. The above video on youtube is the output from the corrected code.
I found this pipeline still didn't do very great under very shady conditions or with weird lighting, the reason being that since I'm only using a thresholded saturation channel of

the HSL colorspace in conjunction with thresholded grayscale sobel gradients in the x and y directions and magnitude and direction of gradient in the same grayscale colorspace. I think if I had done more operations using just the saturation channel things could have improved, but since I wrote this code immediately after my initial arm surgery I was trying to move quickly and just get it completed.

Additionally, I found it quite difficult to accurately get a measure of the radius of curvature, but this appears to have been an issue of my head being in the clouds due to being on opiates after my surgery and I was just forgetting one multiplicative term on the radius of curvature calculation.

One thing I would have liked to have the chance to do would be test this pipeline when there are cars entering the lane that our car is in. Another test worthy case would be when the car our pipeline is on is behind a large vehicle such as a semi, which would make part of the outside lane line disappear while going around corners. This would require us to do something smarter like fit a model to the time series of fitted lane points and add inertia to the system. We may be able to accomplish such a thing by using a basic kalman filter (depending upon what our noise looks like), but by defining our state transition matrix as being more than a function of position, i.e. velocity and acceleration, we would inherently add inertia to the system and prevent detected lane lines from completely jumping across the screen between frames. This would not only smooth out the position of the detected lane lines, but it would also give us the ability to predict where the lane lines should be while they are occluded or even a few frames into the future. I believe this would be the next obvious place to go with such a pipeline, as using only the instantaneous information that we observe is a pretty flawed approach.

If we wanted to do something even better than the above approach which uses only position, and its subsequent derivatives from detected pixels in video frames, we could incorporate throttle angle and steering angle into the state transition matrix, which would create a much more complex model, but also has the potential to give us great robustness to visual occlusion as well as future prediction.

Another thing to consider is the severity of the distortion we are trying to compensate for. If we end up using a fisheye lens or any wide field of view lens it is very unlikely that we will get meaningful positional information as well as radii of curvature, since these will be very positionally dependent in the image, even after using the standard camera calibration models. If this is the case we will have to try to use a more exotic calibration method, such as the PTAM library, which uses the FOV distortion model described by Devernay and Faugeras in “Straight Lines Have To Be Straight, 2001”

Using this I have achieved reprojection errors FAR lower than using the standard radial and tangential calibration model which approximates a pinhole camera, and have successfully been able to perform calibrated measurement operations.

Ultimately the two largest difficulties to overcome with the approach we use in the project are:

1) Verifying that our camera calibration is sufficient for performing metric measurements. This shouldn't be much of an issue with the tools in use now operating on the Zhang approach provided our lens has a relatively narrow FoV, but will quickly become an issue as we increase the field of view.
2) Robustness to false detections and objects occluding our view of the lane lines.