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.

All code snippets that I pasted as image in this write up can be found in my Jupyter Notebook. I just took screenshots to show the code conveniently.

Camera Calibration

In order to calibrate our camera we used a few helpful functions that the OpenCV library provides us. These functions are, `cv2.findChessboardCorners()`, `cv2.calibrateCamera()`, `cv2.undistort()`.

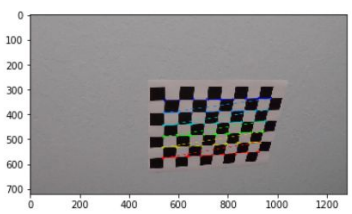
We take our distorted image of a chessboard and find the corners (where the black and white squares intersect) and apply the `findChessboardCorners()` function which outputs points that are represented in the 3D world and 2D image. We can then use these points to finally (`calibrateCamera()`) calibrate our camera and undistort (`undistort()`) images.

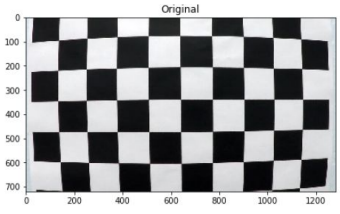
Below you will find a snippet of code that will grab our chessboard corners, inserted into calibrateCamera to get out coefficients, saved as a pickle object to be used, and then finally used to undistort our images.

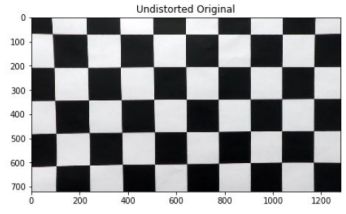
```
# Camera Calibration
# Initial code found here: https://github.com/udacity/CarND-
# and then modified to fit with our given chessboard which h

objp = np.zeros((6*9,3), np.float32)
objp[:,:2] = np.mgrid[0:9, 0:6].T.reshape(-1,2)

objpoints = [] # 3d points in real world space
imgpoints = [] # 2d points in image plane.

images = glob.glob('camera_cal/calibration*.jpg')

# Find chessboard corners to begin camera calibration.
for idx, fname in enumerate(images):
    img = cv2.imread(fname)
    img_size = (img.shape[1], img.shape[0])
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)


    ret, corners = cv2.findChessboardCorners(gray, (9,6), No


    if ret == True:
        objpoints.append(objp)
        imgpoints.append(corners)
        cv2.drawChessboardCorners(img, (9,6), corners, ret)
```

```
# now we want to save the calibration results as a pickle

ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, img_size, None, None)
dist_pickle = {}
dist_pickle['mtx'] = mtx
dist_pickle['dist'] = dist
with open('camera_cal/cal_pickle.p','wb') as output_file:
    pickle.dump(dist_pickle, output_file)
```

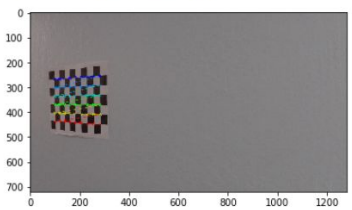
```
cal_images = glob.glob('camera_cal/' 'calibration*.jpg')
for cal_name in cal_images:
    f, ax = plt.subplots(1, 2, figsize=(15, 15))
    img = cv2.imread(cal_name)
    undist = cv2.undistort(img, mtx, dist, None, mtx)
    ax[0].imshow(img)
    ax[0].set_title('Original')
    ax[1].imshow(undist)
    ax[1].set_title('Undistorted Original')

plt.show()
```

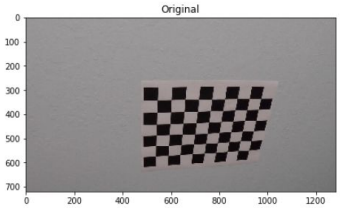


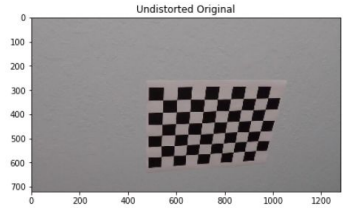




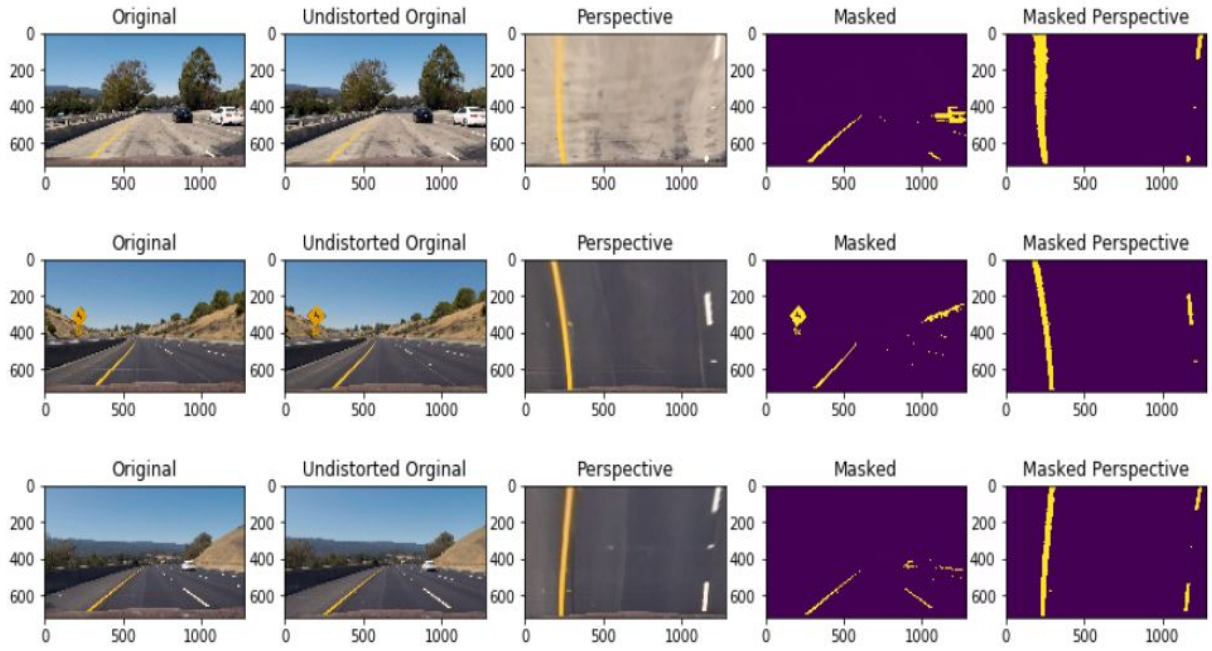
<Figure size 1080x1080 with 0 Axes>







Provide an example of a distortion-corrected image.



After we have a calibrated camera we can use its camera matrix and distortion coefficients we found from the chessboard to undistort “real” images. We must apply a camera undistortion if we want to be able to properly get real data out of the road to apply basic geometry such as finding the slopes and curves.

While the change between the original and undistorted images are small you can definitely tell the difference in between the images.

The above code to produce this series of images can be found in cell 5 of the notebook.

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 first step of correctly identifying the lanes is to produce a binary image. This is the most important step and took the longest time to fiddle with to get a good output. Code can be seen in cell 5. If the binary image produced is poor it greatly affects how the rest of the algorithm runs. This can be seen in the challenge videos where my algorithm fails because of the split color roads or where the sun changes the color (shade, etc). Nevertheless, here is what I chose to

do, I used a combination of color and gradient thresholds to generate the binary image. I first used abs_sobel_thresh and mag_thresh to get a reasonable estimation of the gradients of my lines. Then I grabbed the yellow and white values out of my image in HSV, HLS, and RGB color space. Above you can see the binary image it produces. We can see that it was a few weakness where it detects the car and street sign, maybe that's not a bad thing for future projects where we identify those objects. In this case it wasn't too bad as we warp our image perspective to get a birds eye view and we safely remove elements we don't need out.

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

```
src = np.float32([(300,720),(1100,720),(730,480),(580,480)])
dst = np.float32([(300,720),(1100,720),(1100,0),(300,0)])
M = cv2.getPerspectiveTransform(src, dst)
Minv = cv2.getPerspectiveTransform(dst, src)
```

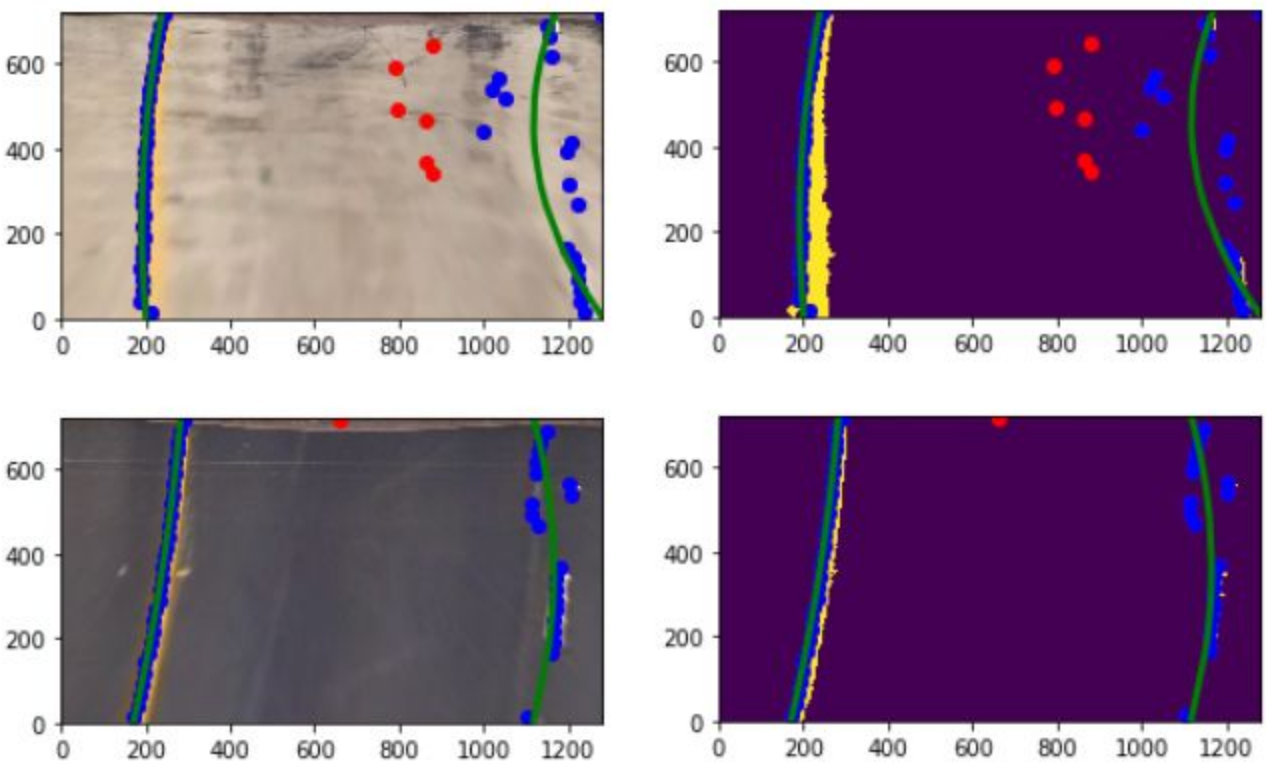
```
img = cv2.cvtColor(cv2.imread(fname), cv2.COLOR_BGR2RGB)
undist = cv2.undistort(img, mtx, dist, None, mtx)
birdseye = cv2.warpPerspective(undist, M, (img.shape[1], img.shape[0]), flags=cv2.INTER_LINEAR)
```

I hardcoded the points I probably should've used the more algorithmic approached instead of hardcoding the points in as shown in the example writeup_template.md but I found that these points work well enough for this project. What we are doing here is we are grabbing 4 points on the original image and then we define the 4 points in warped space and apply the getPerspectiveTransform function to give us our perspective transform M we can do the same to get our inverse also, Minv. We can then use our new transform points M and use the function warpPerspective to basically shift our image as if the camera was pointing from the top down.

Images can be seen in the above pipeline series of images.

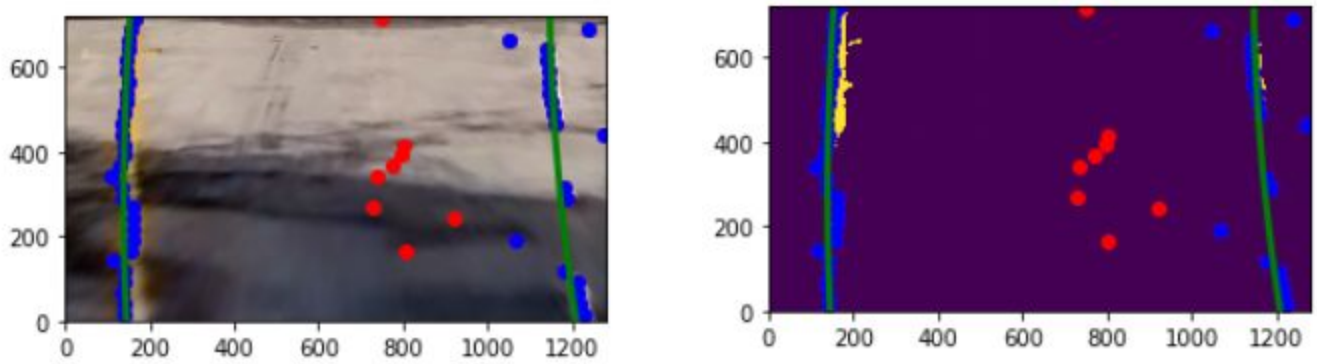
Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

The following code snippet is rather large so I won't paste a screenshot here but the code can be found in cell 12. It produces the following series of images. I used the concept of histogram peaks, sliding window, search from prior to identify our lane lines. We must perform a search over the entire frame to find the lane line because if we just look at brand new images we would never find the lane lines. So we start from the bottom of the image and slide upwards to find pixels that belong to a lane line.



You can see in the images above that there are both red points and blue points. The red points in the algorithm are considered the outliers that get get rejected. The reviewer of the last project suggested that I average my points and reject those that aren't considered in bounds. Here is the product of that suggestion.

The green line is then overlaid by getting an average of the blue points as you can see in the top image the blue points are not as uniform as the left line which results in a slight curve to the left. To fix that would require more tweaking in my lane identification algorithm.



Here is a good image where the sun and shade confuses the algorithm but because those points are rejected the green line that is drawn is okay.

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

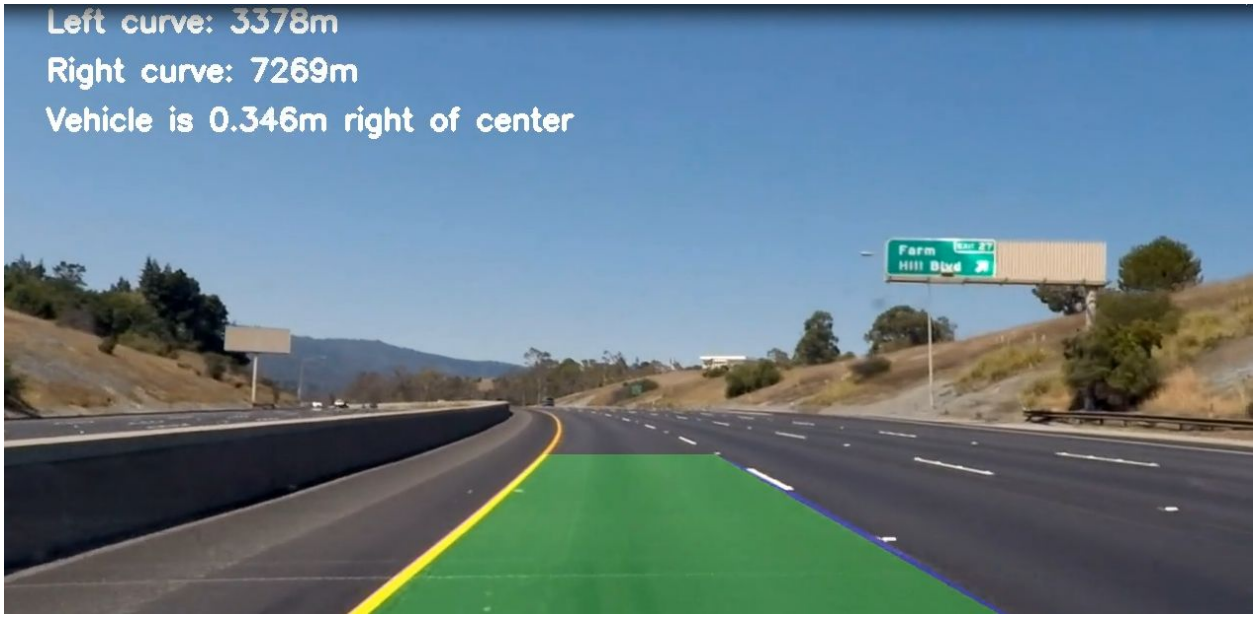
```
y_val = img.shape[1]
left_fit_cr = np.polyfit(left_y*ym_per_pix, left_x*xm_per_pix, 2)
right_fit_cr = np.polyfit(right_y*ym_per_pix, right_x*xm_per_pix, 2)
left_curve = ((1 + (2*left_fit_cr[0]*y_val + left_fit_cr[1])**2)**1.5) \
                            /np.absolute(2*left_fit_cr[0])
right_curve = ((1 + (2*right_fit_cr[0]*y_val + right_fit_cr[1])**2)**1.5) \
                                /np.absolute(2*right_fit_cr[0])

font = cv2.FONT_HERSHEY_SIMPLEX
cv2.putText(result, 'Left curve: %dm' % left_curve, (50,50), font, 1,(255,255,255),3)
cv2.putText(result, 'Right curve: %dm' % right_curve, (50,100), font, 1,(255,255,255),3)


lane_middle = left_fitx[0] + (right_fitx[0] - left_fitx[0])/2.0
deviation = (lane_middle - 640)*xm_per_pix
cv2.putText(result, 'Vehicle is %.3fm right of center' % deviation, (50,150), font, 1,(255,255,255),3)
```

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

To figure out the position of the vehicle with respect to the center we assume that the camera on the car is in the middle of the car. We can then guess the location of the car's deviation from the center lane by using the distance from the center of the image and the middle of the bottom of the two lane detected.

To measure the radius of the curvature we use np.polyfit which returns us a curve coefficient where we can then stick it back into the radius of curvature function found in the lesson to give us an estimation of the curve. I also took the hardcoded values of "ym_per_pix" and "xm_per_pix" directly from the lesson.

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



I ran the algorithm on all three provided videos. These processed videos can be seen in the home directory of my repo with “proc_” prepended to its name.

Link: https://github.com/idkvince/P2-Advance-Lane-Lines---SDCND/blob/master/proc_project_video.mp4

The algorithm works quite well in project_video.mp4

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?

The algorithm is very touchy. It requires a very controlled looking lane. Different shades of color and curve of the lane greatly affects the algorithm. I would never use this algorithm on an actual self driving car because there are too many variables that break it such as position of the sun, clouds, shade, color of the road, etc. (Literally everything affects it)

We can see that the algorithm fails in the first challenge video due to the road being split colored. My algorithm detects the middle of the lane as a lane line incorrectly, when the car goes under the underpass and shade gets in the way it responds incorrectly. This challenge video is quite eye-opening on the weaknesses of this pipeline.

We can see in the second challenge video because the lane curves and significantly more aggressive, lots of tree shade on the road, and the double yellow lines. The pipeline does not respond to these new changes at all. The lanes drawn is completely wrong!

While doing this project trying to produce the correct binary images was very tedious and if this step was done poorly… well the rest of the algorithm will not work. We can also see that the pipeline only works in a VERY controlled driving conditions any new conditions thrown in will absolutely break it. Our current way of finding lane lines will never be sufficient in the real world because the real world always changes and throws new things at you. My guess is that in future projects we let the machine learn these new conditions so we as the human would not have to create sophisticated if else trees to deal with these new conditions.

Works Cited

I used the following resources to help me with this project.

- The provided source code in the lessons plans and the udacity self driving car github.
- https://github.com/ndrplz/self-driving-car/
- https://github.com/priya-dwivedi/CarND
- https://www.math24.net/curvature-radius/