Writeup Template

You can use this file as a template for your writeup if you want to submit it as a markdown file, but feel free to use some other method and submit a pdf if you prefer.

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)](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.

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 from code cells 2, 3 and 4 of the IPython notebook located in "./ALF-1.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.

```
objp = np.zeros((5*9,3), np.float32)
```

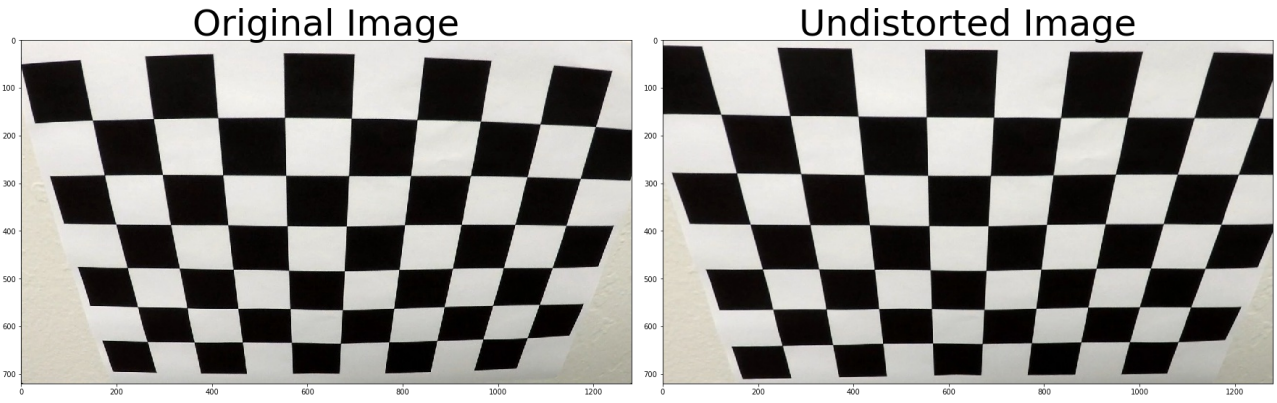
We iterated through 20 chessboard images to find their chessboard corners. Thus, `objp` is just a replicated array of coordinates, and `objpoints` will be appended with a copy of it every time I successfully detect all chessboard corners in the test image. `imgpoints` will be appended with the (x, y) pixel position of each of the corners in the image plane with each successful chessboard detection.

```
for idx, fname in enumerate(images):
    img = cv2.imread(fname)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Find the chessboard corners
    ret, corners = cv2.findChessboardCorners(gray, (9,5), None)

    # If found, add object points, image points
    if ret == True:
        objpoints.append(objp)
        imgpoints.append(corners)
```

When I need to undistort an image, we will use the output `objpoints` and `imgpoints` to compute the camera calibration and distortion coefficients using the `cv2.calibrateCamera()` function.

```
def cal_undistort(img, objpoints, imgpoints):
    ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints,
(img.shape[1], img.shape[0]),None,None)
    dst = cv2.undistort(img, mtx, dist, None, mtx)
    return dst
```

See the result below, a comparsion of the original vs distortion corrected image.



Pipeline (single images)

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

To demonstrate this step, here is an example of a distortion correction done to one of the test images, like this one. It may not be obvious in the first glance, but if you look carefully, the white lane on the bottom right hand side, near bonnet of the car is slightly shifted to the right due to the distortion correction.

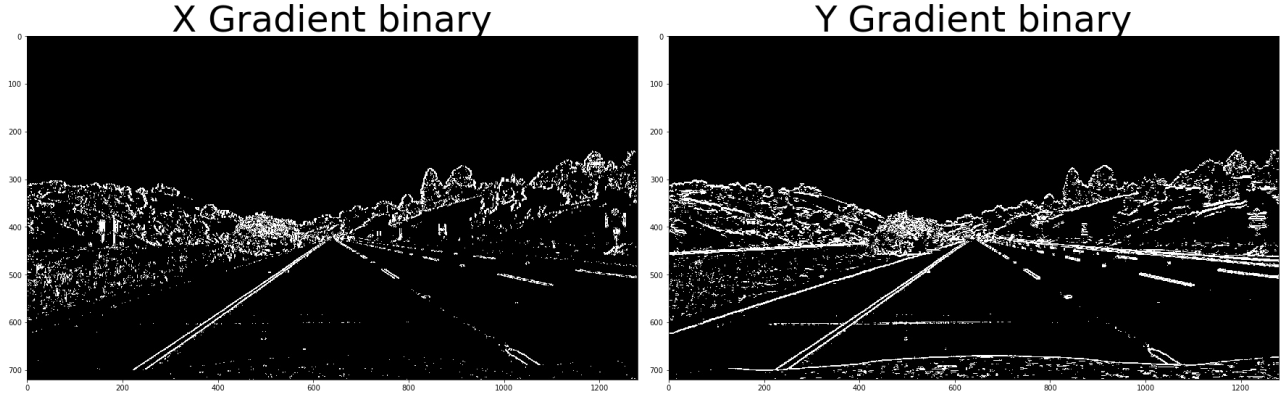
2. Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

For my image processing pipeline, I have decided to combine and use only binary outputs from gradient magnitude threshold, gradient directional threshold, saturation channel threshold and red channel threshold. The reason being is simply only these channels were able to display a good detection of the yellow and white lane lines throughout the dark and brightly lit road sections throughout the video recording.

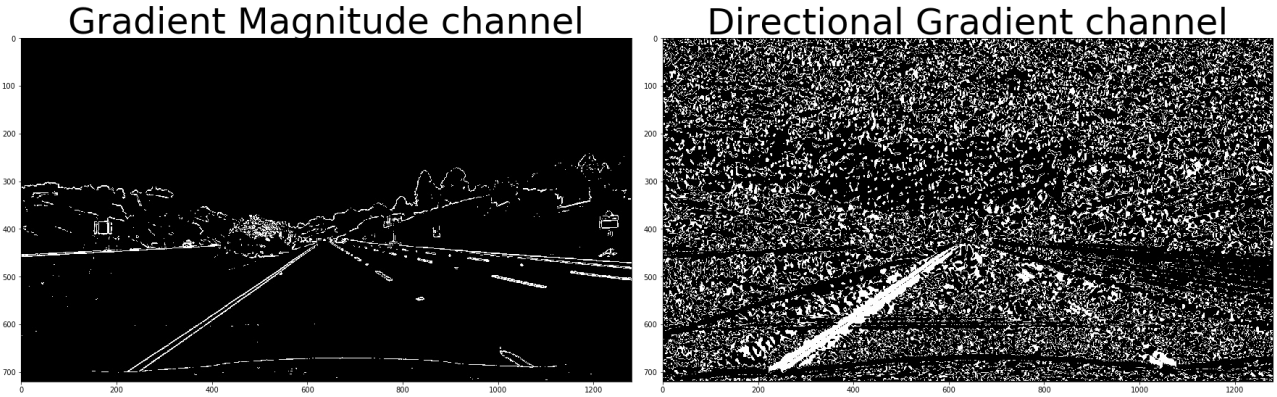
```
binary_result[((mag_binary == 255) & (dir_binary == 255)) | (S_binary == 255) & (R_binary == 255)] = 255
```

The actual working code can be found from code cell 7 to 12 in notebook ALF-1.ipynb). I used a combination of color and gradient thresholds to generate a binary image. Here's an example of the actual output for this step for ./test_images/straight_lines1.jpg.

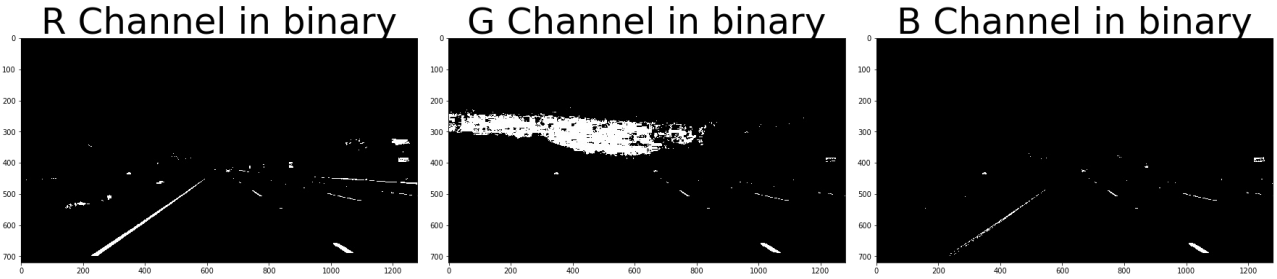
See Gradient X / Y binary output here:



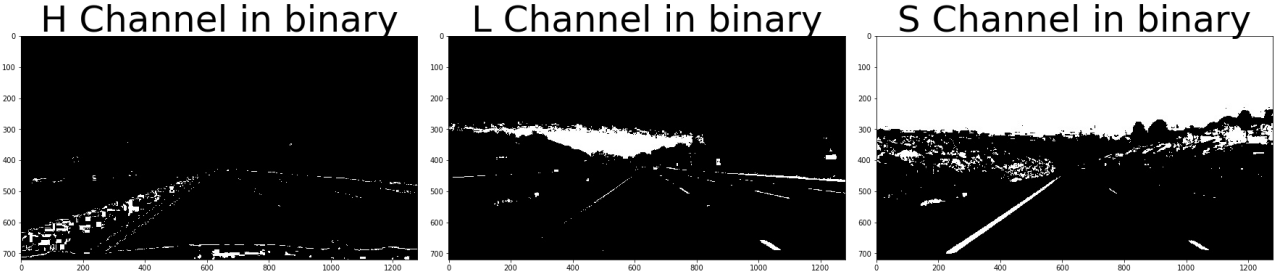
See Gradient directional / magnitude binary output here:



See RGB binary output here:



See HSL binary output here:



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 is located in code cell number 20. The function is called `transform_bird_eye_view(image, src, dst)`. I chose to hardcode the source and destination points in the following manner

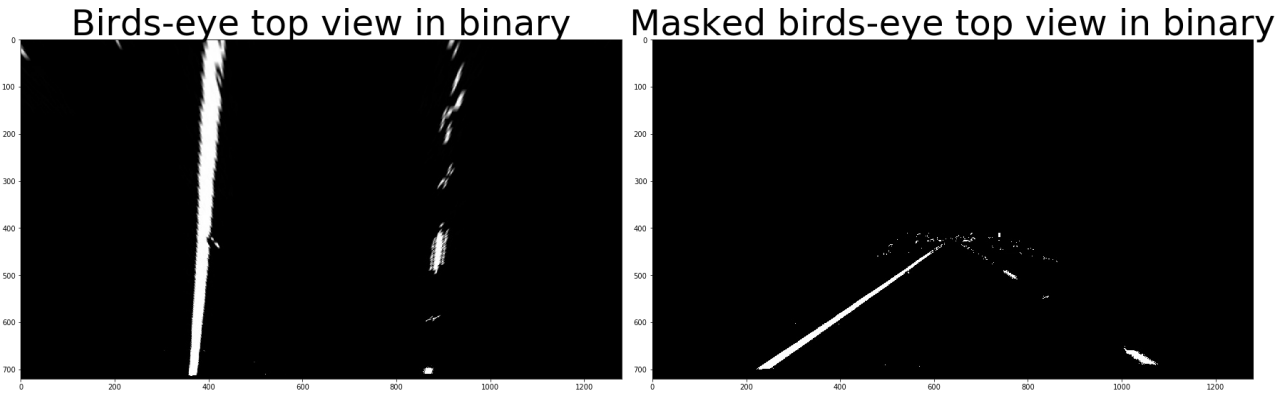
```
src = np.float32(
    [(140,720),(570,455),(700,455),(1280,720)])
dst = np.float32(
    [(330,720),(330,0),(960,0),(960,720)])
```

The source and destination points are:

Source	Destination
140,720	330,720
570,455	330,0
700,455	960,0
1280,720	960,720

Before transforming perspective, I also applied region mask (a quadrilateral) over the area of interest in the trying to reject any noisy pixels adjacent to the lane. See code cell 19 from the jupynter notebook. See the below image on the right.

In code cell 20, you will find the function `transform_bird_eye_view(image, src, dst)` that handles perspective transformation. 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. See the below for results:



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

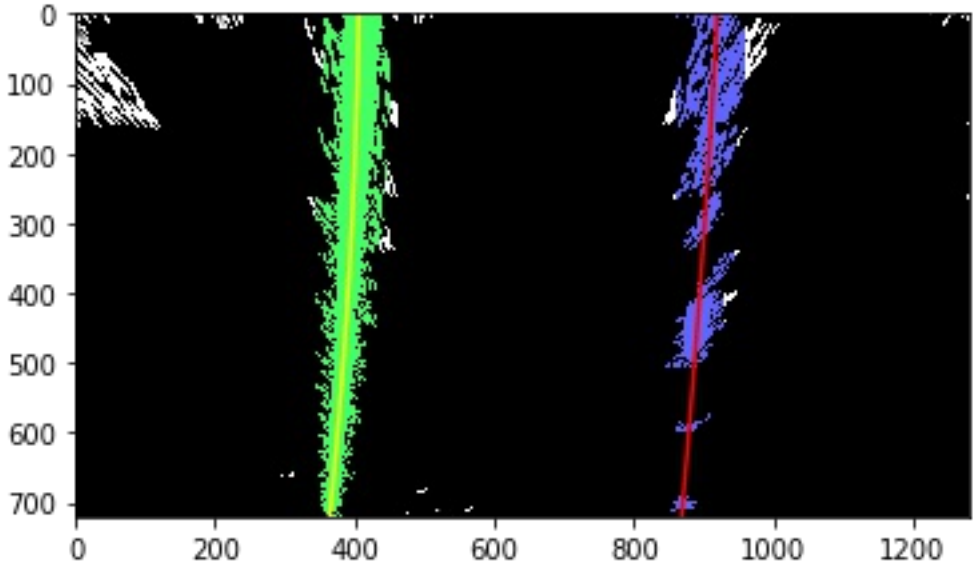
In code cell 23 the function `line_search_scanning(binary_warped)` take in a warped perspective transformed section of the lane line (a birds eye view of the lane lines section which is in front of the car).

The first step is to visualize the histogram of the image. See the below for example. The histogram will be used to observe and determine where the range / position of the left and right lane pixels.

Second step is to setup a sliding window process where there will be loop to iterate through the image from top to bottom for the left hand side and right of the image to find all the nonzero pixels

The final step is to apply a numpy polyfit function to fit a line through the pixels of left and right side of the image.

See the below image for the results of this:



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.

In code cell number 25, the function `def lane_curvature(ploty, fitx, xy, fit)` will take in output parameters from the function from the above section (4) and calculate the radius of curvature of the left and right lane.

The first step is to work out the actual x, y world space conversion. From the bird eyes view image produced from the above step, we can derive pixels space to metres in:

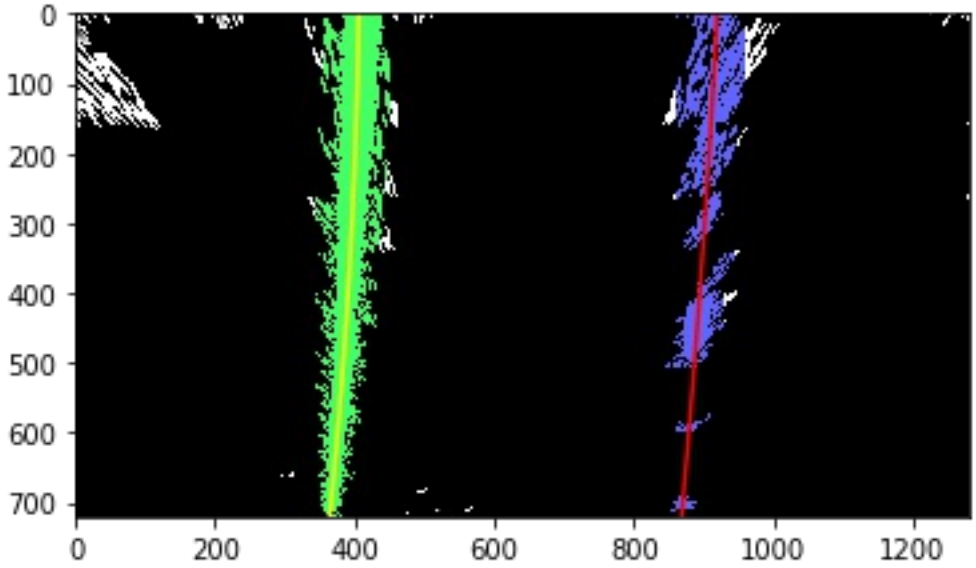
X axis: 3.7 metres approximately 500 pixels (as per US federal guidelines lanes are 12 feet apart)

Y axis: 25 metres approximately 720 pixlels (as per US federal guidelines lanes are 10 feet long and lanes are separated at 30 feet. Thus as measured from bird eye view image image4 (./output_images/bird-eye-top-view.png) and the green projected layer on the road image image5 (./output_images/lane_augmentation_project_output.png), it was measured to contain 2 white lane sections and 2 empty space sections, and this works out to be 80 feet, equivalent to 24.3 metres)

```
ym_per_pix = 24.3/720  # meters per pixel in y dimension
xm_per_pix = 3.7/500 # meters per pixel in x dimension
```

The next step is to use the xy array pixel points from the left and right lane to fit a new polynomial in the x,y world space by using and applying Xm and Ym here.

```
  leftx, rightx = xy[0][0], xy[1][0]
  lefty, righty = xy[0][1], xy[1][1]

  # Fit new polynomials to x,y in world space
  left_fit_cr = np.polyfit(lefty*ym_per_pix, leftx*xm_per_pix, 2)
  right_fit_cr = np.polyfit(righty*ym_per_pix, rightx*xm_per_pix, 2)
```



The final step is to calculate radius of the curve using the following formula.

R-curve = [(1 + (2Ay + B)^2)^(1.5)] / |2A|

The equivalent python code is:

```
  left_curve_metres = ((1 + (2*left_fit_cr[0]*y_eval*ym_per_pix +
left_fit_cr[1])**2)**1.5) / np.absolute(2*left_fit_cr[0])

  right_curve_metres = ((1 + (2*right_fit_cr[0]*y_eval*ym_per_pix +
right_fit_cr[1])**2)**1.5) / np.absolute(2*right_fit_cr[0])
```

For position of the vehicle with respect to the center of the lane. Please see code cells 27 and 28 for implementation.

As per the bird eyes view image [image4 (./output_images/bird-eye-top-view.png)](./output_images/bird-eye-top-view.png), is approximately the 500 pixel between the left and right lane. Therefore 500 pixels is equivalent to 3.7 metres (12 feet) as per US federal regulators guidelines on road lane marking design rules.

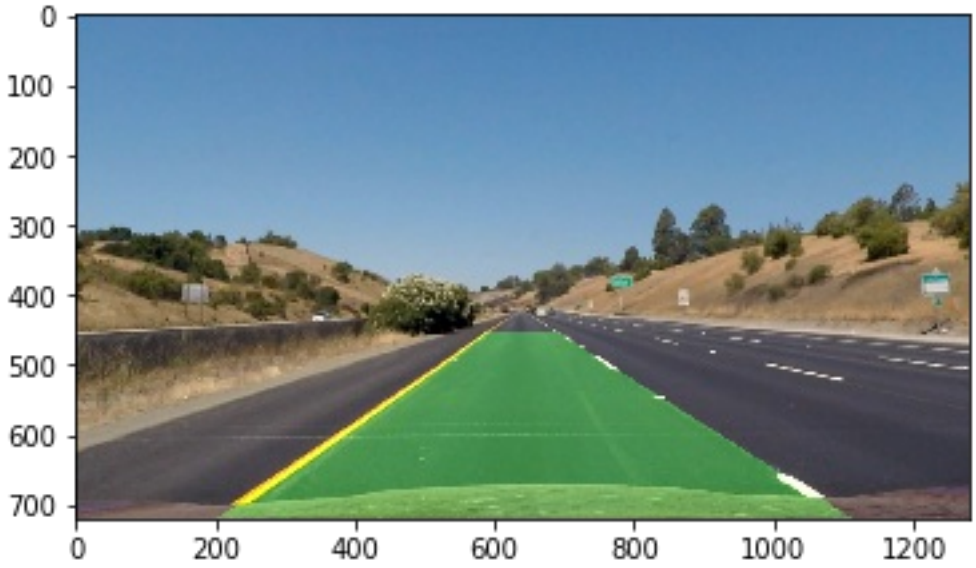
The calculation is:
vehicle offset from center = camera_pos - lane_centre_post

In python:

```
def vehicle_position(fitx):
    xm_per_pix = 3.7 / 500 # meters per pixel in x dimension

    left_x_vehicle, right_x_vehicle = fitx[0][-1], fitx[1][-1]
     # left_x_vehicle, right_x_vehicle = fitx[0].mean(), fitx[1].mean()

    # position of center lane
    x_pos_lane_center = (left_x_vehicle + right_x_vehicle) * 0.5

    # assuming the camera is located on the center of the vehicle so X position
is:
    x_pos_camera = 1280 * 0.5

    vehicle_offset_from_lane_center = x_pos_lane_center - x_pos_camera
    return vehicle_offset_from_lane_center * xm_per_pix
```

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 from code cells 1 to 31. See the below output from the image pipeline.



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!).

Processed video:
[video2 (./output_images/project_video_processed.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?

After completing the video processing pipeline I found the issue where only the brightly lit sections of the video will have the lanes failed to be detected by the pipeline. It was not obvious initially what the problem was, because going back to the image pipeline did not show any failure in brightly lit sections of road.

Eventually I found out VideoFileClips' fl_image function was returning RGB images, whereas the pipelines I wrote expected BGR channels, thus everything worked fine except in brightly lit image conditions.

There could be many situations that can easily throw off the pipeline and failing. Extreme ends of lighting conditions can put off the pipeline. Currently the image thresholds are set specifically for this particular video, so I would suggest dynamic and variable image channel selection will be required to handle difficult light conditions.