Writeup Template

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) Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Writeup / README

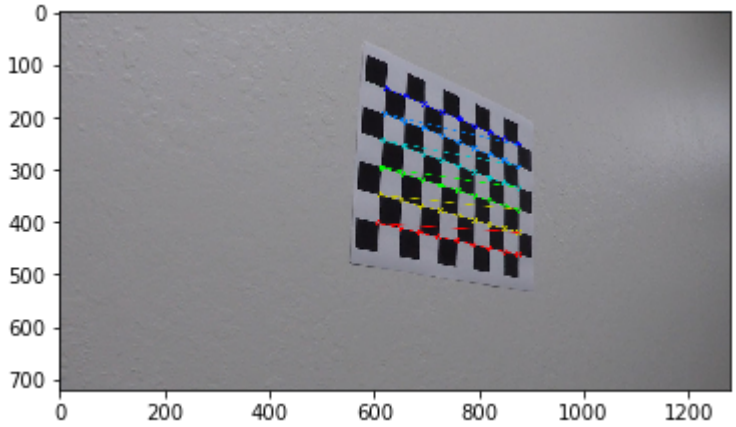
1. Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf. Here (https://github.com/udacity/CarND-Advanced-Lane-Lines/blob/master/writeup_template.md) is a template writeup for this project you can use as a guide and a starting point.

You're reading it!

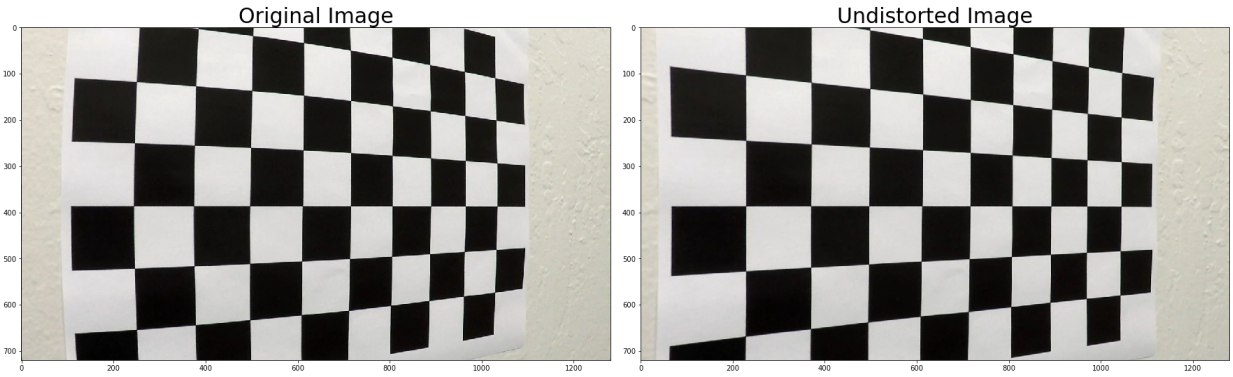
Camera Calibration

Calibration seems very straight forward. With the help of API from OpenCV library, such `(findChessboardCorners())`, `(drawChessboardCorners())`. Converting images to gray scale before processing is recommended for best results. I was able to loop through the 20 chess images to capture the drew points at each corner and created coordinates to properly mapping each distorted points to an undistorted image. One thing needs to keep in mind, not all chessboard image were usable if the corners identifed were less then the predefined x and y axis size. When it happened, the image needed to be dropped to avoid contaminating the undistortion result and moved on to the next one. There were total 3 images that were not usable because of this. With only 17 images, we still had enough datapoint to feed to `calibrateCamera()` from OpenCV to generate the undistorted coefficient and matix.

Here is one of the chessboard image used for drawChessboardCorners().



When comparing the original chessboard image with the undistorted chessboard image, you will notice the lines at the edge of image and around the corners are straightened instead of bending outward.



Pipeline (single images)

1. Distortion-correction

From the `calibrationCamera()`, the undistorted mapping and matrix is generated and apply to real road images. To demonstrate this step, I will describe how I apply the distortion correction to one of the test images like this one:

2. Color transforms, gradients thresholded

To find the best combination of color and gradient binary that would best described the lane lines in images, I experimented by adjusting the thresholds of each color threshold as well as their various combinations, with a goal trying to minimize the nosie around both white dotted lane and yellow solid lane as much as possible. For R, G, and S, they can be adjust to almost show only either white or yellow lanes. L suppose can be tuned to minimize areas of bright lights or dark shadows. But the best I can do still shows a big portion of light reflection form the road as shown in the pictue below. X gradient binary shows a definite line segments that are close to vertial which includes both yellow and white lane, while M magnitude binary shows too much outside of both lane lines which created a lot of noise.

Revised version:

I found that the color and gradient transform was done incorrectly previsouly. All color components should be "AND" together while all gradient components should be "AND" together as well. The result of color component should be "OR" with the result of gradient component as shown below.

Previously:

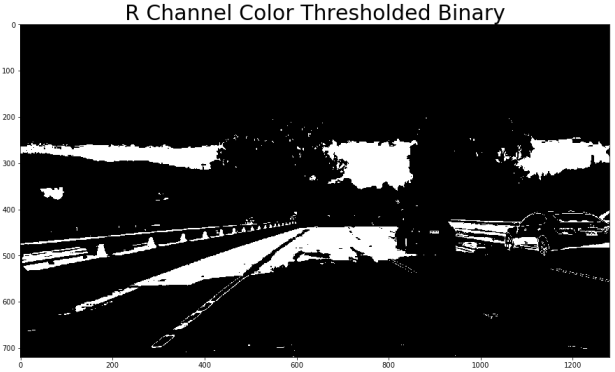
```
    combine_rgslm_binary[(r_binary==1) | (g_binary==1) | (s_binary==1) |
(l_binary==1) | (x_binary==1)] = 1
```

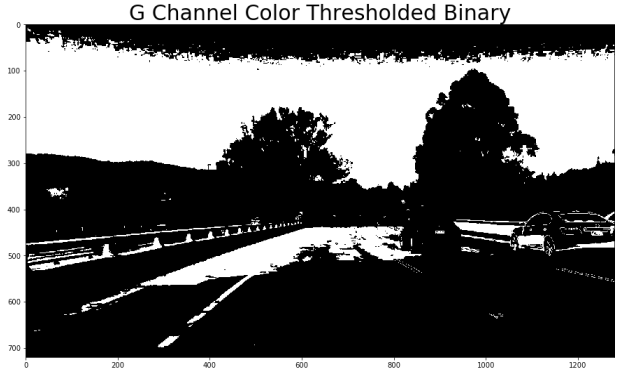
Modified:

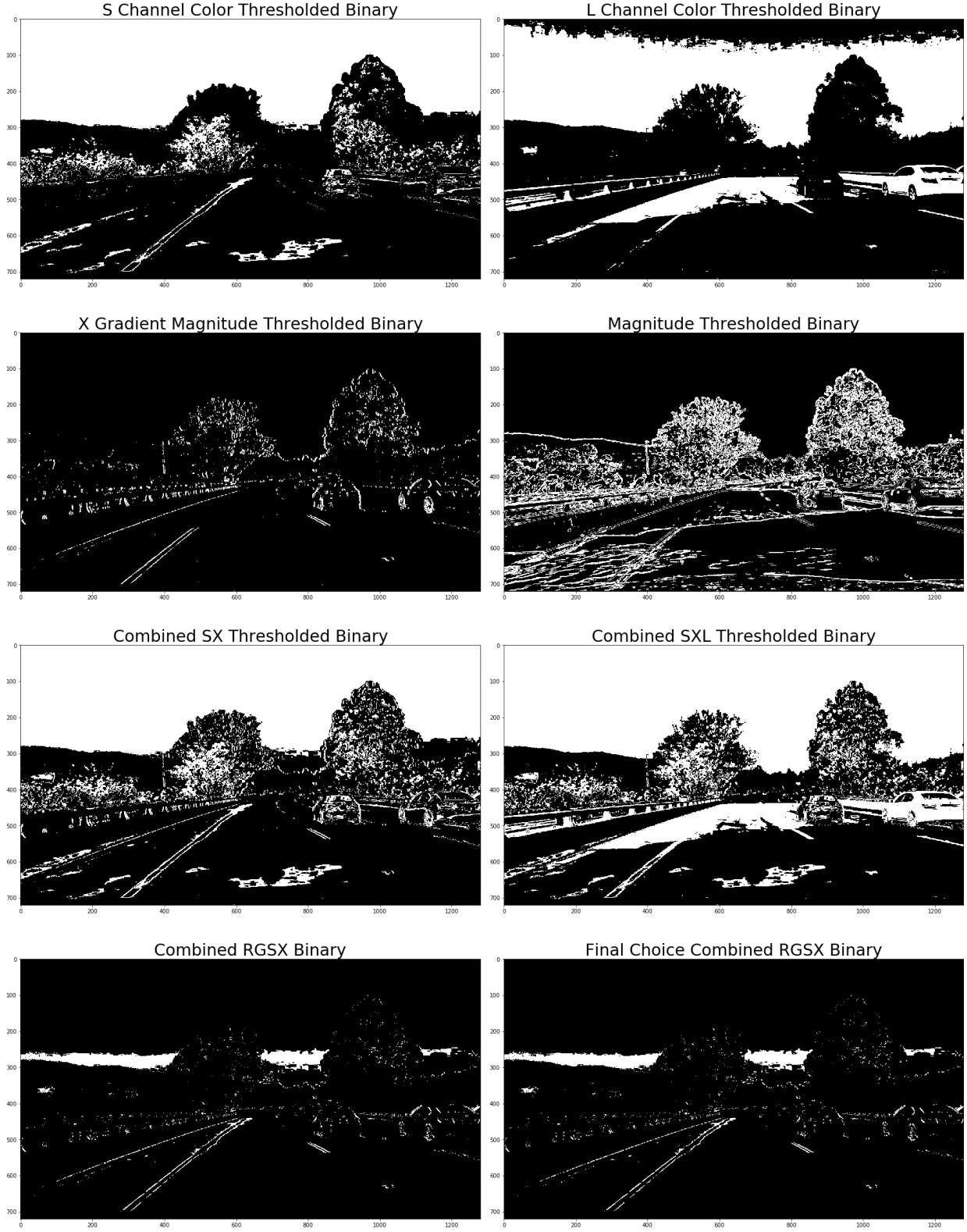
```
    combine_rgslxy_binary[((r_binary==1) & (g_binary==1) & (s_binary==1)
& (l_binary==1)) | ((x_binary==1) & (y_binary==1))] = 1
```

I keep the same thresholds as before as they produce the best results.

```
r_thresh    = (175, 225)  # best for white lane 195-255 RGB color space,
 R channel color thresholds
g_thresh    = (170, 225)  # best for white lane 195-255 RGB color space,
 G channel color thresholds
s_thresh    = (90, 155)   # best for Y lane 90-255 HSL color space, S cha
nnel color thresholds
l_thresh    = (165, 255)  # 205-255 best for eliminating shadows 150-165
 HSL color space, L channel color thresholds
x_thresh    = (40, 180)   # best for lane edges 40-180 Gray scale X gradi
ent thresholds
mag_thresh  = (70, 140)   # best for outlining Y and W lanes 70-140 magni
tude threshold for Sobel X and Y gradient
kernel_size = 5           # kernel-size for X gradient
```







alt text

With these output binary, both Combined SX and Combined RGSX color and graditent threshold seemmed to provided the best lane segment detection. Later on the pipeline, RGSX picked up a lot more noise compare to SX. As a result, I picked Combined SX binary as for my edge detection threshold.

Here's an example of the Combined SX binary comparing to Canny's edge detect as for comparison.

3.Perspective Transform

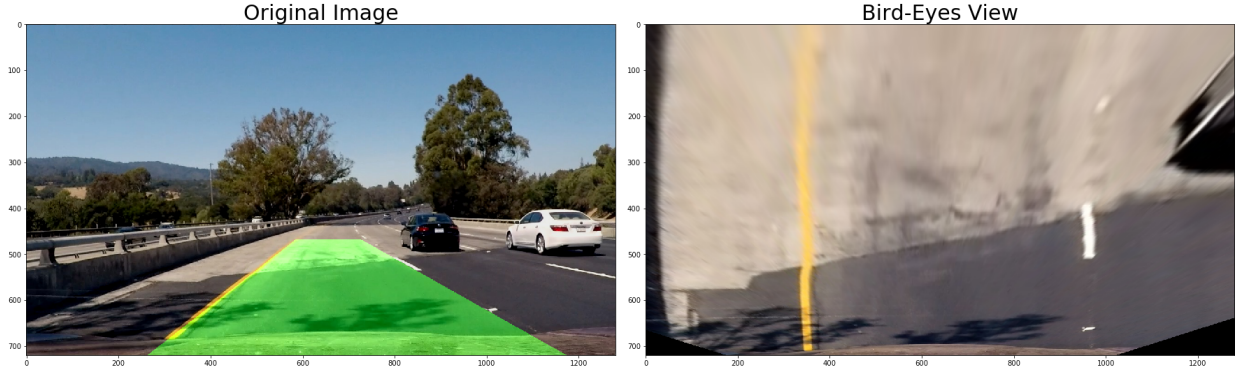
The key to generate a good perspective transform image, the choose of `src` and `dst` points from perspective mapping is crucial. I reused most of the helper functions defined in Project 1 in order to determine the position of the yellow and white lanes from each image frame using `HoughLinesP()`. With that, I was able to find the 4 coordinates that was best representing the lane area as the source points. Ideally, we wanted to show two nearly straight lines after prespective transform. As a result, I simply hard corded 4 coordinates from two straight lines in the image near 350 and 950 pixel on the x axis as destination points.

```
lineImage, coor = hough_lines(edges, rho, theta, 25, 25, 250)
print("coordinates", coor)

coordinates [(585, 468), (263, 720), (1173, 720), (727, 468)]

src = np.array([coor[0], coor[1], coor[2], coor[3]], np.float32)
dst = np.array([(350, 0), (350, 720), (950, 720), (950, 0)], np.float32)
```

This resulted in the following source and destination points:

| Source | Destination | | (vary from frames) | (constant) | |:----------------:|:----------:| | 585, 468 | 350, 0 | | 263, 720 | 350, 720 | | 1173, 720 | 950, 720 | | 727, 468 | 950, 0 |

I verified that my perspective transform was working as expected. Both the road image and the Combined SX binary showed similar straight lines which appeared to be parallel after prespective transform using the `src` and `dst` points.



alt text

4. Identifying lane-line pixels and fit their positions with a polynomial

After perspective transform on the binary image, I had a warped image from bird-eyes view. I took the sum of the warped binary as a histogram to determined the indices of each nonzero pixels in the image. The most dense portion of the histogram repsented the positions of the two lanes in the image. Taking the maximum number of pixels on the left side of histogram provided the starting position of the left lane, vice versa for the right lane.

```
# Take a histogram of the bottom half of the image
histogram = np.sum(binary_warped[y_startpoint:,:], axis=0)
# Find the peak of the left and right halves of the histogram
# These will be the starting point for the left and right lines
midpoint = np.int(histogram.shape[0]/2)
leftx_base = np.argmax(histogram[:midpoint])
rightx_base = np.argmax(histogram[midpoint:]) + midpoint
```

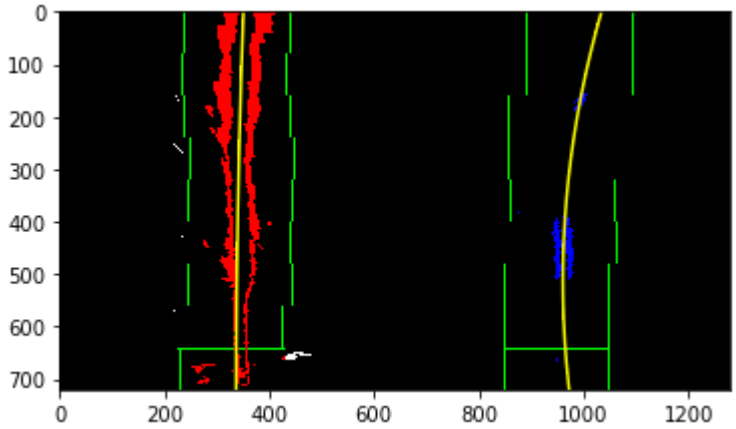
I then broke the image into 9 even windows in vertical direction, and setup a horizontal window from the left lane position (max # of pixel on the left plane) with a plus/minus `margin` as the equation to determine which dense area belongs to which side.

```
    for window in range(nwindows):
        # Identify window boundaries in x and y (and right and left)
        win_y_low = binary_warped.shape[0] - (window+1)*window_height
        win_y_high = binary_warped.shape[0] - window*window_height
        win_xleft_low = leftx_current - margin
        win_xleft_high = leftx_current + margin
        win_xright_low = rightx_current - margin
        win_xright_high = rightx_current + margin
        # Draw the windows on the visualization image
        cv2.rectangle(out_img,(win_xleft_low,win_y_low),(win_xleft_high,w
in_y_high),
        (0,255,0), 2)
        cv2.rectangle(out_img,(win_xright_low,win_y_low),(win_xright_high
,win_y_high),
        (0,255,0), 2)
        # Identify the nonzero pixels in x and y within the window
        good_left_inds = ((nonzeroy >= win_y_low) & (nonzeroy < win_y_hig
h) &
        (nonzerox >= win_xleft_low) &  (nonzerox < win_xleft_high)).nonze
ro()[0]
        good_right_inds = ((nonzeroy >= win_y_low) & (nonzeroy < win_y_hi
gh) &
        (nonzerox >= win_xright_low) &  (nonzerox < win_xright_high)).non
zero()[0]
        # Append these indices to the lists
        left_lane_inds.append(good_left_inds)
        right_lane_inds.append(good_right_inds)
        # If you found > minpix pixels, recenter next window on their mea
n position
        if len(good_left_inds) > minpix:
            leftx_current = np.int(np.mean(nonzerox[good_left_inds]))
        if len(good_right_inds) > minpix:
            rightx_current = np.int(np.mean(nonzerox[good_right_inds]))
```

Now that I had both left and right lanes indices, I applied them to `np.polyfit()` to fit a closet polynomial functions to repersent these two lane lines. I also fitted the scaled version of the closet polynomial for the real world, which would be needed when determining the radius of curvature.

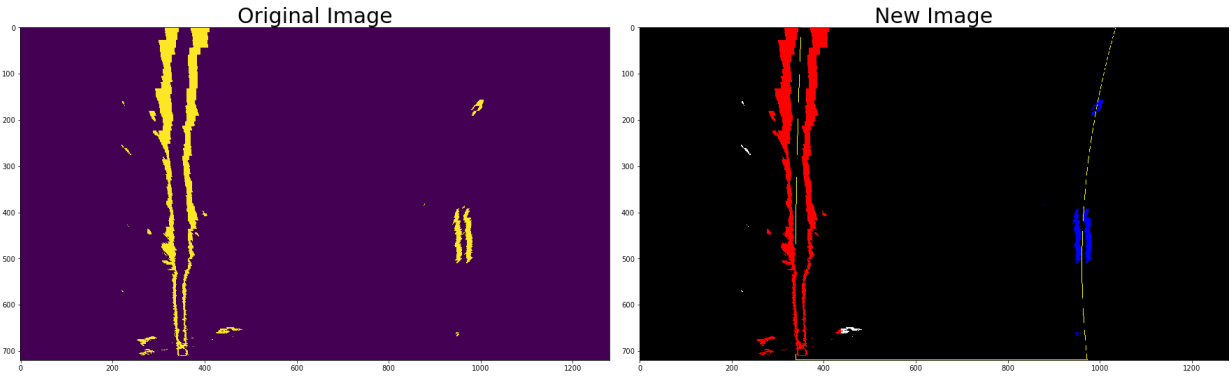
```
        # Fit a second order polynomial to each
        left_fit = np.polyfit(lefty, leftx, 2)
        right_fit = np.polyfit(righty, rightx, 2)
        # Fit a second order polynomial to each in real world scale
        left_fit_real = np.polyfit(lefty*ym_per_pix, leftx*xm_per_pix, 2)
        right_fit_real = np.polyfit(righty*ym_per_pix, rightx*xm_per_pix,
 2)
```

The sliding windows for the first fitted lines looked like this:



As we fitted the left and right lines from the first frame, the same equation can be used on finding the next fit lines with the following code:

```
    left_lane_inds = ((nonzerox > (left_fit[0]*(nonzeroy**2) + left_fit[1
]*nonzeroy + left_fit[2] - margin)) & \
                      (nonzerox < (left_fit[0]*(nonzeroy**2) + left_fit[1
]*nonzeroy + left_fit[2] + margin)))
    right_lane_inds = ((nonzerox > (right_fit[0]*(nonzeroy**2) + right_fi
t[1]*nonzeroy + right_fit[2] - margin)) & \
                       (nonzerox < (right_fit[0]*(nonzeroy**2) + right_fi
t[1]*nonzeroy + right_fit[2] + margin)))
```



5. Calculating radius of curvature of the lane and position of vehicle with respect to center of image

To find the radius of curvature in the real world, we needed to scale the equation from pixel to kilometer. As provided, 1 meter is equvilant to 30/720 pixel. The radius of curvature was determined by plugging y=image.shape[0]=720 in to the formula below.

```
    left_curverad = ((1 + (2*left_fit_real[0]*y_eval*ym_per_pix + left_fi
t_real[1])**2)**1.5) / np.absolute(2*left_fit_real[0])
    right_curverad = ((1 + (2*right_fit_real[0]*y_eval*ym_per_pix + right
_fit_real[1])**2)**1.5) / np.absolute(2*right_fit_real[0])
    avg_curverad = np.average([left_curverad, right_curverad])
```

The offset from the center of image was determined by subtracting the center between left land and right lane from the center of the image.

Revised version:

I found that I have been using the wrong values in my calculation of offset from center. I need to use fitted line coefficient in pixel world, calculate the different before center of the image to the center of the lane. Then, I can convert the result from pixel to real world using ym_per_pix (instead of xm_per_pix). The result apparently shows a big different. Now my offset are varying between 0.7 to 0.1 depends on the location of the vehicle.

Previously:

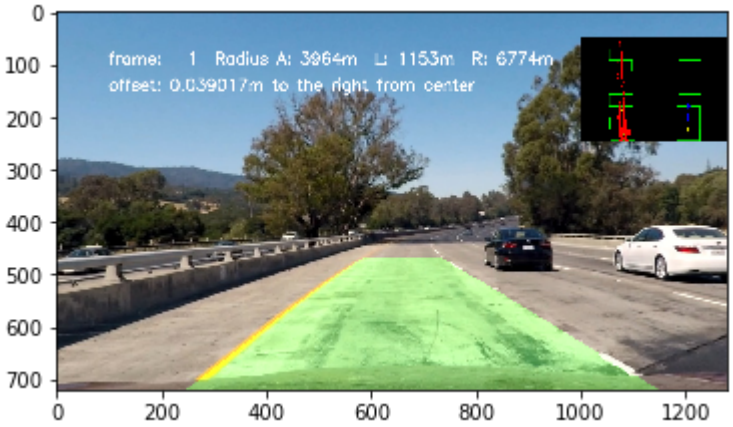
```
    y_eval = 720*ym_per_pix
    leftx  = left_fit_real[0]*y_eval**2 + left_fit_real[1]*y_eval + left_
fit_real[2]
    rightx = right_fit_real[0]*y_eval**2 + right_fit_real[1]*y_eval + rig
ht_fit_real[2]
    offset = (1280/2*xm_per_pix) - ((rightx-leftx)/2+leftx)

result:
curvature(average,     left,         right)
(2579.0534415931825, 4568.6489127738678, 589.45797041249739)
offset: 0.083448m to the right from center of image, left_fit_real=array
([  1.09441785e-04,  -5.35949521e-03,   1.85477283e+00]), right_fit_real=
array([  8.48508866e-04,  -3.62906570e-02,   5.46518591e+00])
```

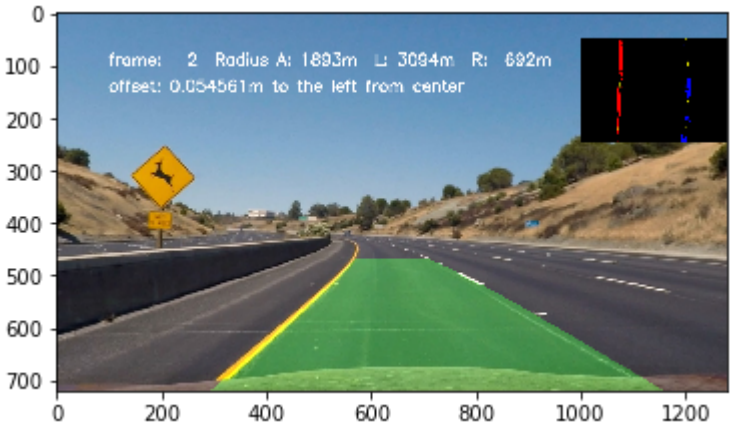
Modified:

```
def offsetfinding(left_fit, right_fit, ym_per_pix=30/720, xm_per_pix=3.7/
1280):
    y_eval = y_size-1
    leftx  = left_fit[0]*y_eval**2 + left_fit[1]*y_eval + left_fit[2]
    rightx = right_fit[0]*y_eval**2 + right_fit[1]*y_eval + right_fit[2]
    offset = (x_size/2) - ((rightx-leftx)/2 + leftx)
    offset_real_world = offset*ym_per_pix
    return offset_real_world

result:
curvature(average,     left,         right)
(7327.4337009033725, 3205.904294913229, 11448.963106893516)
offset: 0.400420m to the right from center of image, left_fit_real=array
([  1.59585080e-04,  -1.33765855e-01,   3.67577302e+02]), right_fit_real=
array([  4.39178922e-05,  -6.38598787e-02,   9.68532662e+02])
```

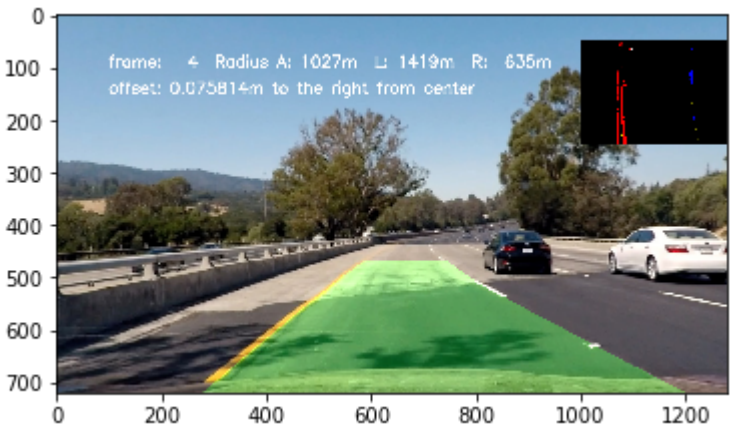
6. Result with Inv Perspective Transform fitted lines onto the road lane area

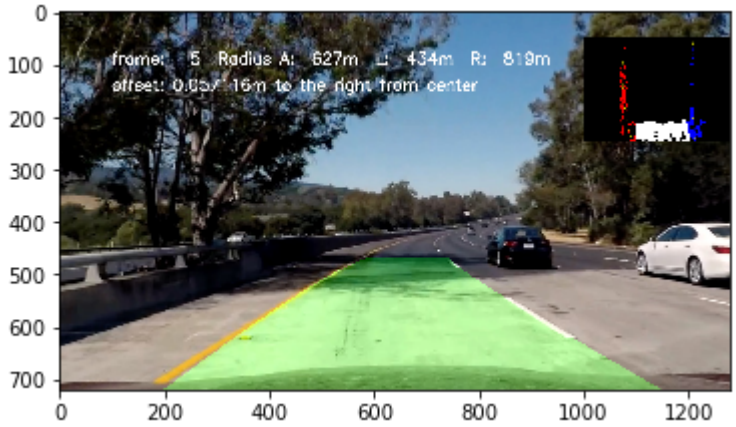
Once I had the fitted lines, I use `fillPoly()` to fill the area between lanes, and invert Perspective transform this area back onto the road image to overlay the tracked lane. To better illustrate the transform lane lines, I also overlapped the determined lane lines on the upper right corner of the reuslt frame, alone with the radius of curvature and offset to center on the same image. Here are the result images on each test images.

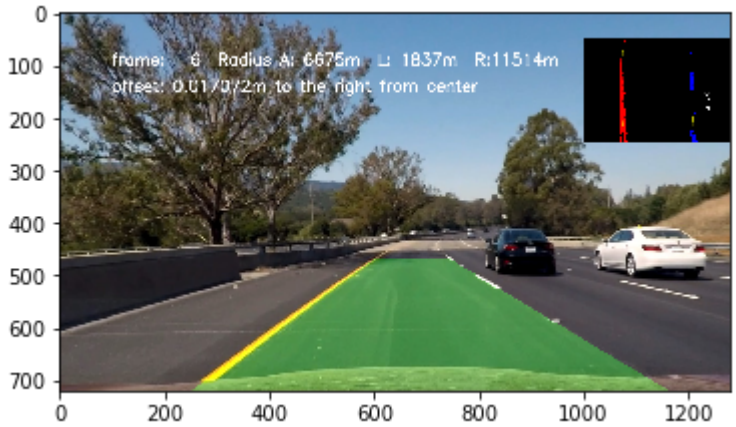

frame: 1 Radius A: 3964m L: 1153m R: 6774m
offset: 0.039017m to the right from center


frame: 2 Radius A: 1893m L: 3094m R: 692m
offset: 0.054561m to the left from center


frame: 3 Radius A: 2352m L: 3187m R: 1518m
offset: 0.028904m to the right from center


frame: 4 Radius A: 1027m L: 1419m R: 635m
offset: 0.075814m to the right from center





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 my video result (./test_video_outputs/project_video-saved.mp4)

I also tried the challenge videos, although the result were not as expected, it still worth sharing.

link to challenge video (./test_video_outputs/challenge_video-saved.mp4)

link to harder challenge video (./test_video_outputs/harder_challenger_video-saved.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?

As I worked on the project, the hardest part is not building the pipeline. The lecture had provided most of the instruction about what needed to be done from calibration, undistort image, applying color and gradient threshold, to perspective transform, and calculating the radius of curvature. But the hardest part seemed to be adding sanity checks to make sure the pipeline still worked robustly when a bad frame was provided and no valid lines were found. It took me a while to figure out right places to put sanity check and plan the exit route for bad frames, making sure None data structure be able to pass through the pipeline without causing a crash. Also, when it happened, how to reuse the previously good data to substitute the data from the current frame was the most time consuming. Yet, one by one, I saved the last good `coor` at HoughLineP when no line was found, and the `left_fit` and `right_fit` at slidingfit1() and sliding(), and reused them when a bad image frame persent. Sure the image was not as smooth and could be improved by increasing the history queue of the last good dataset. I will leave this as future improvement.

Also, another hurdle I encounter was tuning the threholds for various combined color and gradient binary. Combined SX seemed to work well with project_video4.mp4. However, it was not good enough for the other challenge videos. I tried exploring all other combinations, but none of them seem to be able to robustly filter out the bright lights poping in from frame to frame. I also tried adding another layer of filter using `region_of_interest()`. However, too much filtering may loose the flexibility of the pipeline usage and limit it to work only on specific images. The forum had discussion and suggestion about how to tune the threholds and combination, but none of the suggestion worked for me. The last combined RGSX seemed to do a little better, but still not good enough to filter certain bad frames in the middle of the video. I will also leave this as future improvement.

Again, this project is very challenging itself, while it does reveal a lot of interesting technic for finding lane lines.

Resubmission:

First bug was found in color and gradient transform. Fixed issue by ANDing all color binaries together only. Same for all gradient binaries only. Then ORing the result from color binary and gradient binary at the final step. The ANDing all the white dots/areas together to provide a clearer lane line segments, while leave the rest in black. Same for gradient binary, which provide a better definition on line segments.

Second bug was found in offetfinding(), where xm_per_pixel was used while the correct version used ym_per_pixel. The simpliest way to calculate is to use pixel world components to calculate the offset, then convert the pixel into meter by using ym_per_pixel.

Result of the tracker in project4_video-revised.mp4 had improved, as well as the calculated offset, which is within 0.7m to 0.1m.