P4 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.
- Check the lane was correctly detected
- 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 Points

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

General approach

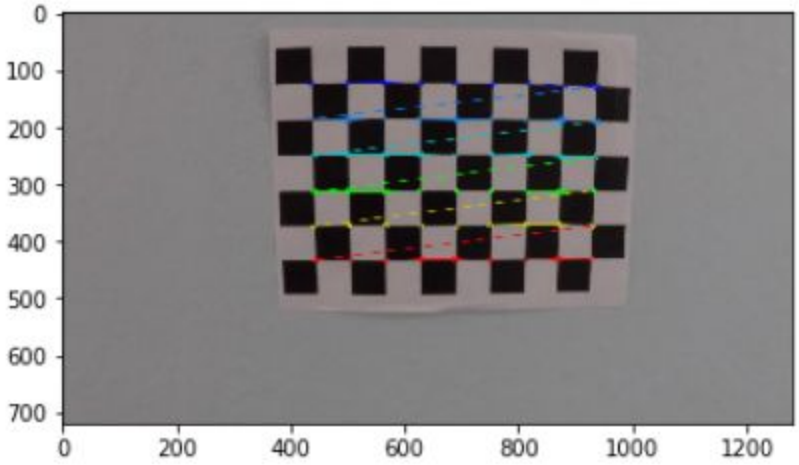
For this project, I decided to try a new approach: I created a iPython notebook to tweak all the parameters needed for the camera calibration and the perspective. I added a lot of visualization to be able to catch corner cases and find the parameters that best suited all the images.
After having found those best parameters, I used some of the code in this notebook to write my pipeline.
The advantage of this approach was that, in the pipeline, I only had to think about the logic of the lane detection, as I already knew that the parameters chosen would give me the best binary image.

Camera Calibration

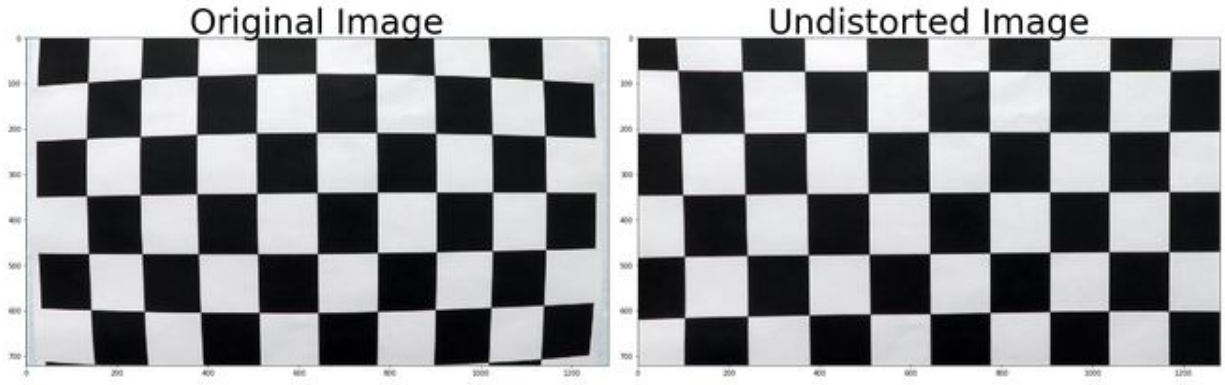
1. Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

The code for this step is contained in the 4th code cell of the IPython notebook - P4-pipeline.

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.
An example of the corner detection is:



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:

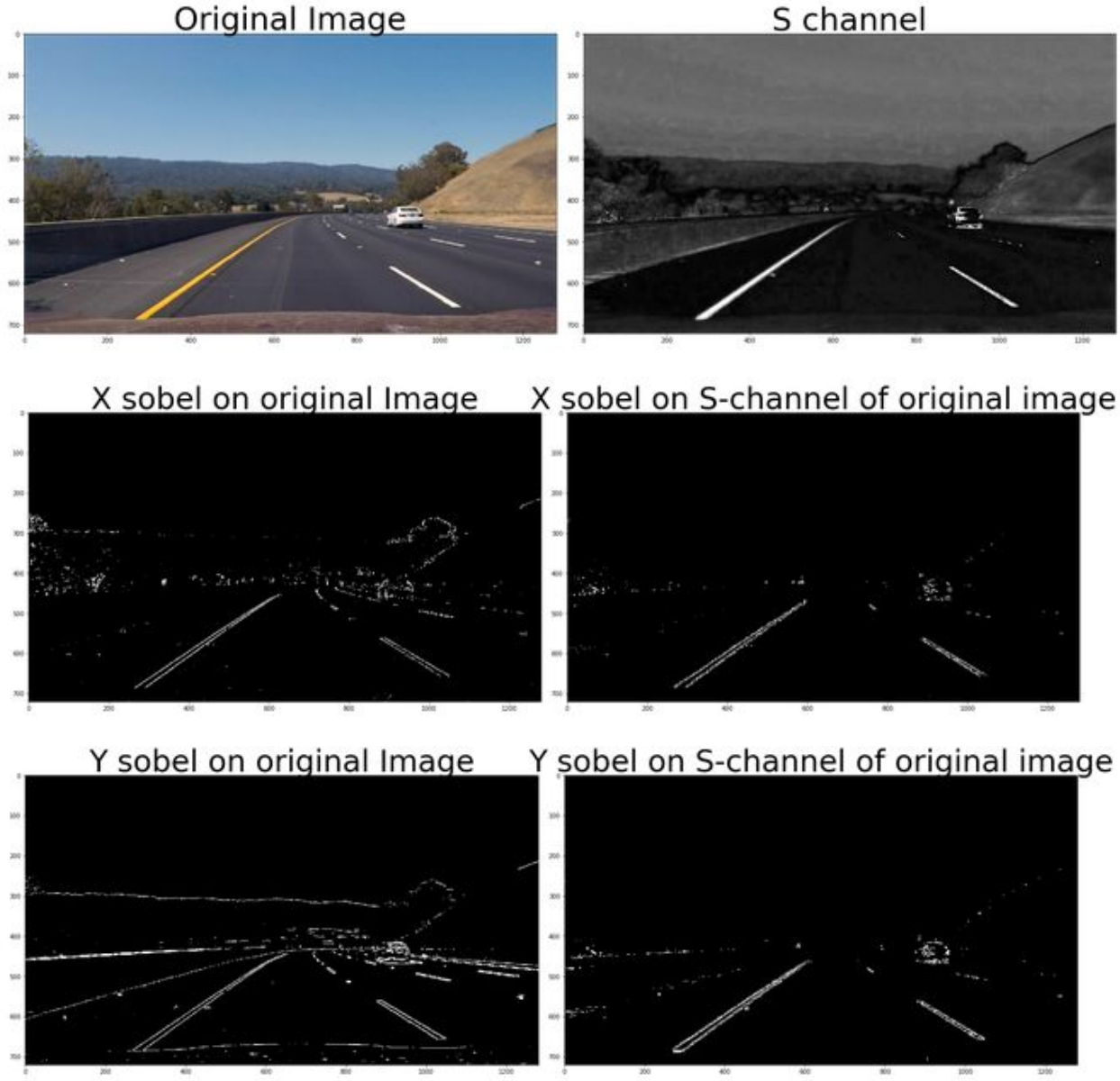


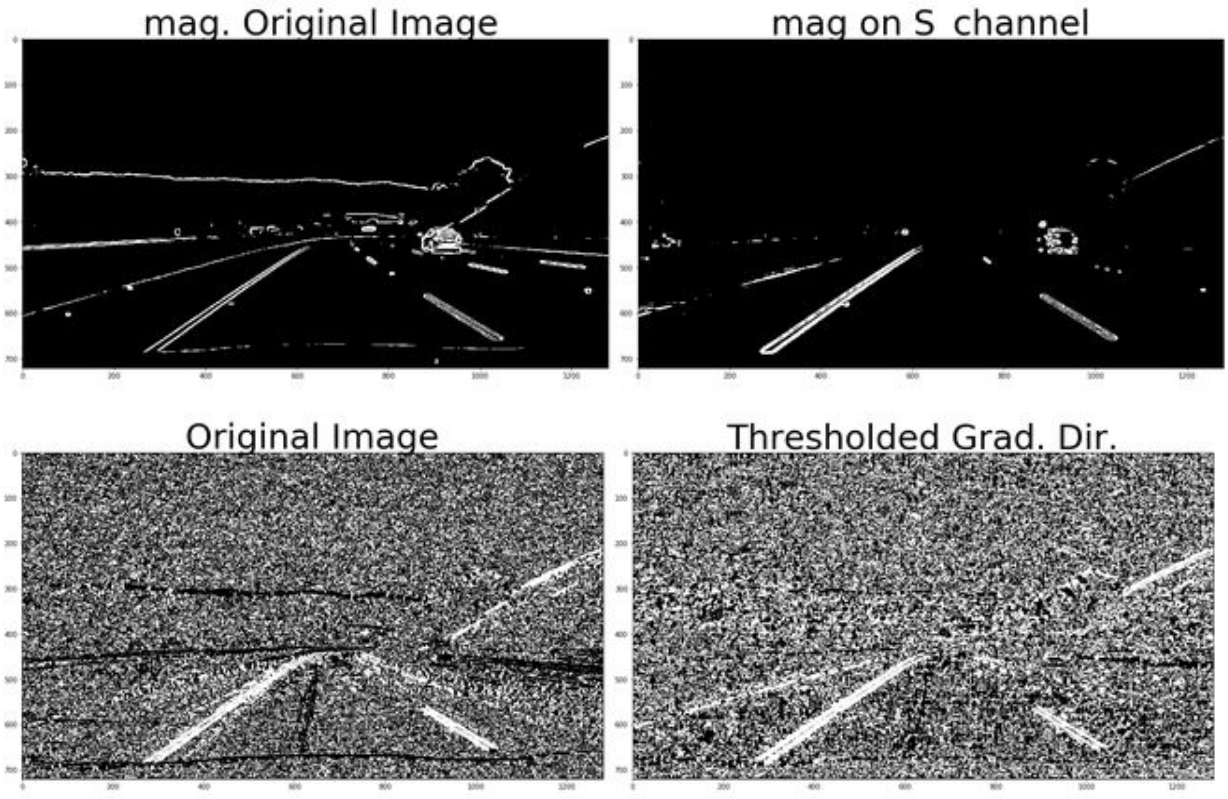
Single Image Pipeline

I wrote 3 methods to apply the sobel gradient, to apply a magnitude gradient and to apply a directional gradient.

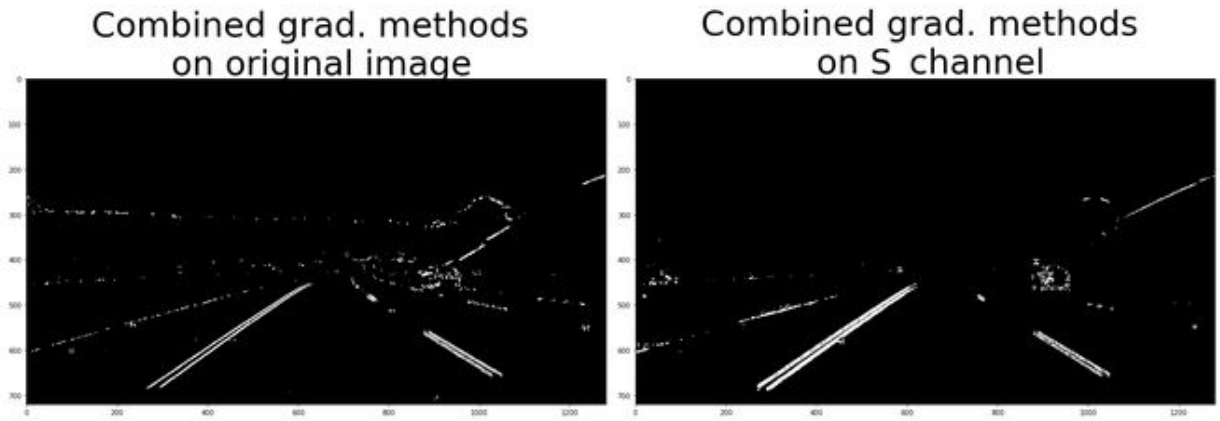
I chose one image from the test images provided, extracted the S channel, and applied the gradient functions to both the original image and the S channel image. I tuned the thresholds to get the best results on the S channel image.

Original image vs S channel image:

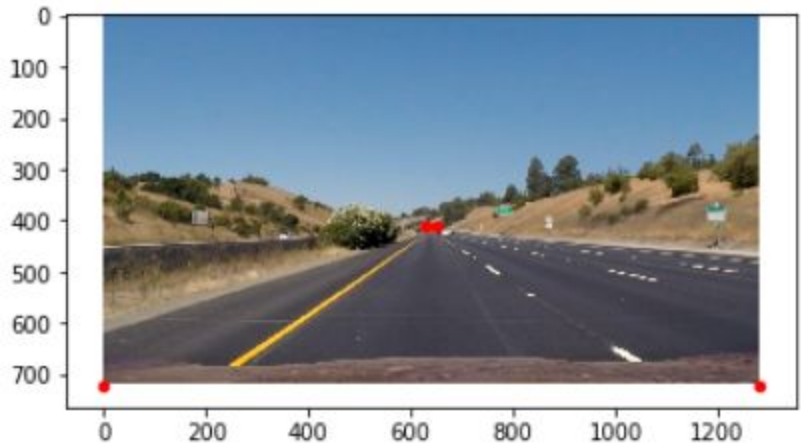




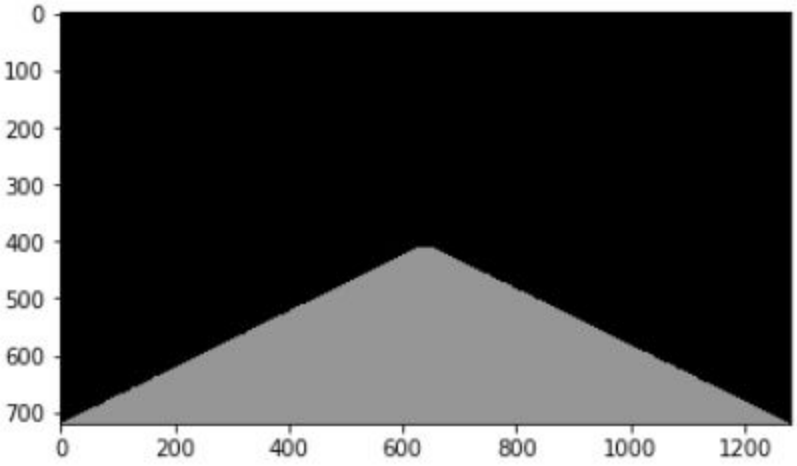
And finally compared the combination of those gradients images on the original image and on the S channel image.:



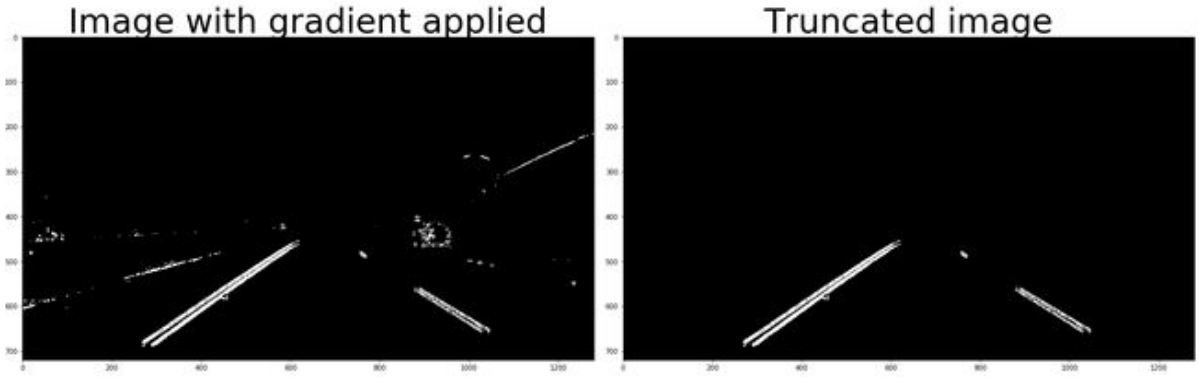
The next step was extracting the area of interest to remove any background noise.
As you can see on the following image, I chose some points on the image (after it had been distorted using the camera calibration).



The region we keep for the lane analysis is represented in grey in the next image. I used the openCV function fillPoly to mask the polygon of interest.



Applying the mask to the binary image I got from the gradient, I got:



We can clearly see the lanes in this image and there is almost no other points than the lanes.

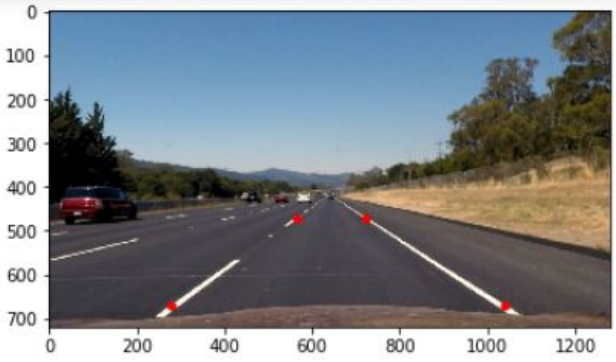
The next step was the perspective change.

For this, I used an image of a straight road and I rectified it using the camera calibration.

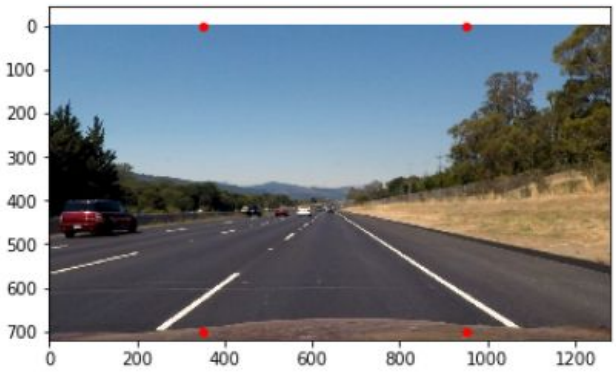
I selected 4 points on the lanes, 2 per lanes. The array representing those source points is:

```
x_array = [564,max_x - 557,280,max_x-240]
y_array = [470, 470,max_y-50, max_y-50 ]
```

The following image shows the 4 source points with the corresponding numpy array and the 4 destination points used to change the perspective.



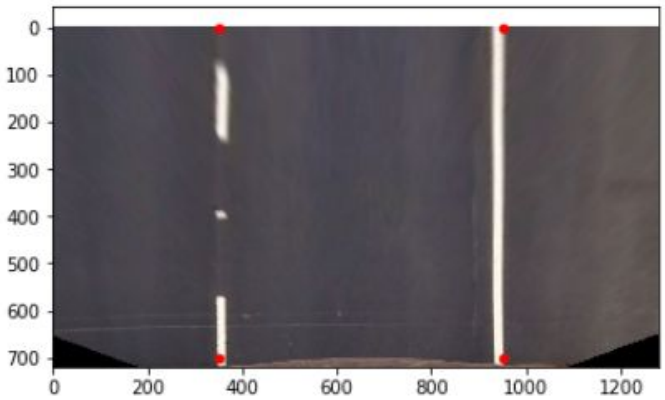
```
[[  564.   470.]
 [  723.   470.]
 [  280.   670.]
 [ 1040.   670.]]
```



```
[[ 350.    0.]
 [ 950.    0.]
 [ 350.  700.]
 [ 950.  700.]]
```

Then, I used the openCV function getPerspectiveTransform to get the transformation matrix and I applied the perspective transform to the selected image.

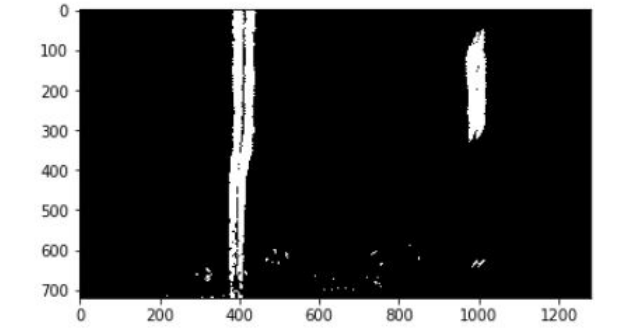
I had to carefully chose the source points to get a perfectly straight bird-view image.

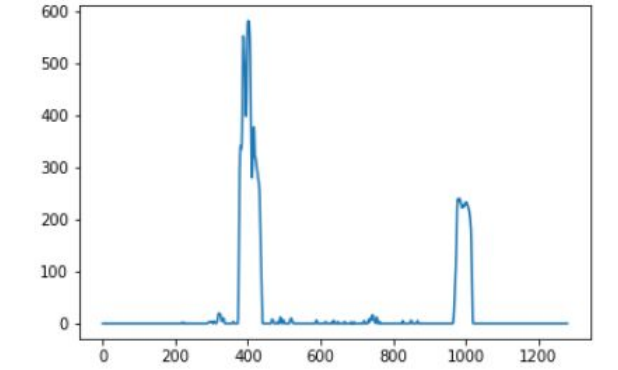


In order to test my sliding window detection method, I decided to use an image that looked a little more challenging. It was slightly curved, with some shadows that could perturb the detection.

The following 3 pictures show the original image, the bird-view image and the histogram on the lower half part of the bird-view image.

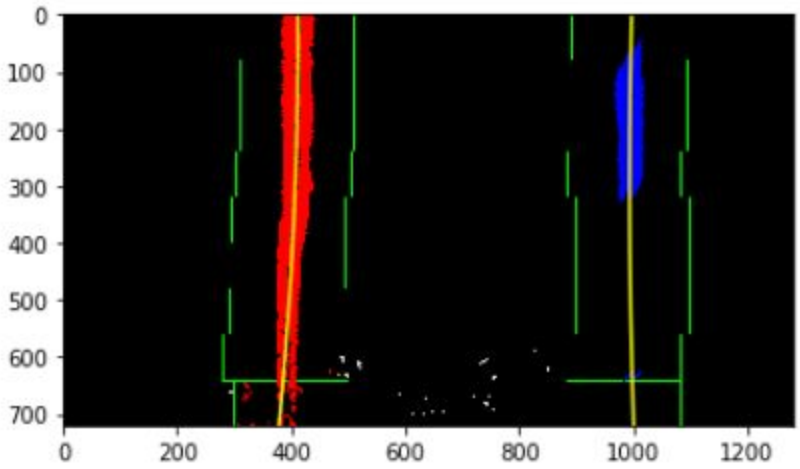

```
(720, 1280)
```





After implementing the sliding window detection algorithm as described in the lectures, I plotted the results of the detection superimposed with the windows used for the detection and the polynomial fit.

```
[ -7.27486607e-05   6.43001370e-03   4.11568132e+02]
[  3.96029992e-05  -2.35005234e-02   9.96914079e+02]
```



I calculated the curvature radius, first in pixel and then in meters, using the following conversion for pixel to meters: the full high of the image is roughly 30meters when the perspective transform is applied and the spacing between the right and the left lane is 3.7meters which is roughly 600px on my image.

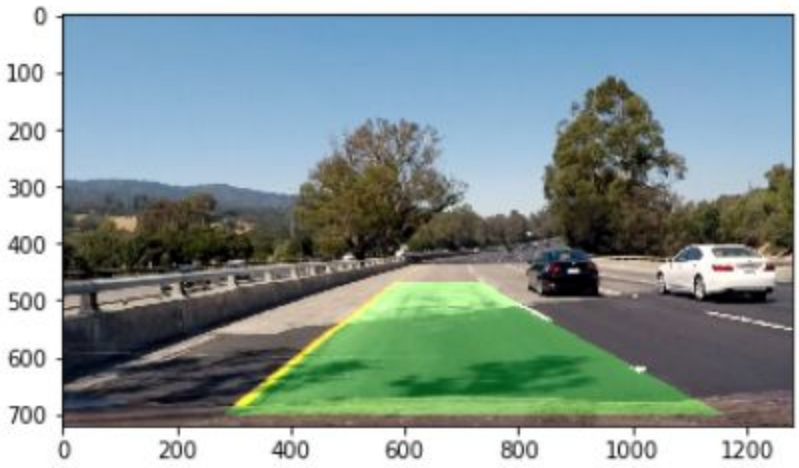
```
    ym_per_pix = 30/720 # meters per pixel in y dimension
    xm_per_pix = 3.7/600 # meters per pixel in x dimension
```

For the given image, the results are:

```
radius of curvature:
left:  6873.38594291 px right:  12635.69643 px
left:  2257.45408126 m right:  4146.87220042 m
[ 1935.  3554.]
```

I decided to round the number as the precision of the measurements here don't make sense, the distance being eye-balled.
Those number make sense, we can see that the lanes polyfit is almost straight which corrisponds to a large radius of curvature.

Using the inverse transformation matrix, I transformed a polygon defined by the fit found thanks to the sliding method back onto the original image.



I was now ready to switch to the video pipeline.

Video Pipeline

For the video pipeline, I decided to create some helper functions to make it easier to focus on the logic.
I also implemented a couple of checks on the lanes detected to ensure that the lanes looked good. We will discuss those checks in this section.

I implemented 2 types of lane search:

- The “first_time_lane_search” which uses the sliding window method
- The “smart_lane_search” that assumes the lanes were already found and the previous polynomial used to fit to the lane could be reused.

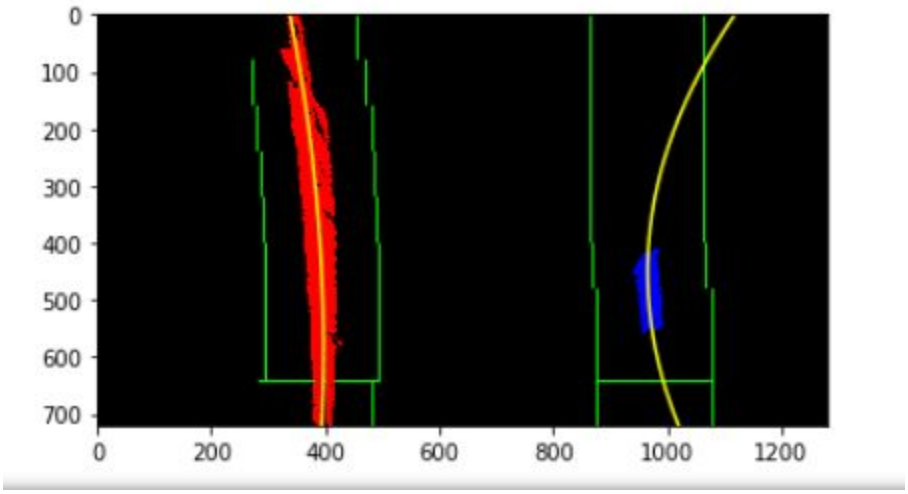
Both of those functions update the lanes (from a lane Class) with the most recent polynomial fit and others parameters that are defined in the class.

I then have 4 functions used to check if the lanes were correctly detected.

- Lanes_correctly_detected() is the high level wrapper. The logic in this function is as follows:
    - A first level check is applied to a lane to make sure that the left lane starts on the left of the image (and the right lane on the right).
      It also ensures that the lanes are still in their expected side of the image for y = 0.
      The radius of curvature for the lane is also supposed to be greater than 130m.
      This number was calculated thanks to the American road regulations directives regarding the minimum allowable radius of curvatures at different speeds.

- The distances between the lanes for y = 0 ; y = y_max and y = y_middle (middle of the bird_view image) must be roughly 600 pixels. If it's not the case, something went wrong in the detection
- If the radius of curvature is changing too much, it also means the detection can't be trusted.
  A caveat to this one is when we have a straight lane. The radius is a big number and depending on the inflection of the curve it can go from one frame to the other to a large positive number to a large negative number. In this case, the detection is still valid.

Typically, an example of a case where we want to mark the detection as invalid would be:



The fit for the right lane is clearly not good and that's mostly due to the lack of lane points. Using the previous lane fit and expecting that the polynamial fit for the next image proved to be working well.

Another helper function is used to update the lane after all the checks confirmed that the lane looked good or update the element of the lane class if the detection was not successful.

To apply all those functions to a frame from the video, we use the detect_lane() method. It takes only one argument as an input: the rgb image extracted from the video.
This method applies the camera calibration matrix, the gradient and the perspective transformation to the original image. It then calls the lane_search_and_check() method.
The idea behind calling another function was to be able to use recursion.
lane_search_and_check() has a counter and is the smart image detection doesn't work, it's going to recursively calls itself to try the sliding window detection.
In this method, all the lane detection checks are used and if a lane is not being detected, the previous polynomial fit is being used.
This function returns the result form the plot_lane() methods which uses the lane best_fit polynomial fit to draw the lanes on the image.

The best fit is calculated as a weighted average of the previous best fits and the current fit (respectively 40% and 60%).

Discussion

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?

Most of the implementation choices were discussed in the previous sections and examples were given.

The part that was the most important and tricky to get right was the logic to check that a lane was detected correctly.

I noticed that getting a bird-eye view of a longer portion of road would help to detect the lanes, especially when they are dashed lanes. However the drawback of choosing this approach was the distortion caused by the perspective change.
One idea would be to fit a polynomial on a bigger portion of the road (for instance 30meters ahead vs 15m) and only apply the inverse perspective transform to half of this image to prevent any distortion.

Compared to project1, using the S channel was very helpful and shadows on the road weren't an issue anymore.