PROJECT 4: ADVANCED LANE FINDING (MITHI SEVILLA, MARCH 18 2017)

This project includes the following files and folders:

`chessboard.py`	`camera_calibration.ipynb`	`pipeline.ipynb`	`project_video.mp4`
`birdseye.py`	`perspective_transform.ipynb`	`pipeline_debug.ipynb`	`camera_cal`
`lanefilter.py`	`gradient_and_color_thresholding.ipynb`	`pipeline_verbose.ipynb`	`output_images`
`curves.py`	`projecting_lanes.ipynb`	`project_video_output.mp4`	`test_images`
`helpers.py`	`calibration_data.p`	`project_video_verbose_output.mp4`	

OVERVIEW

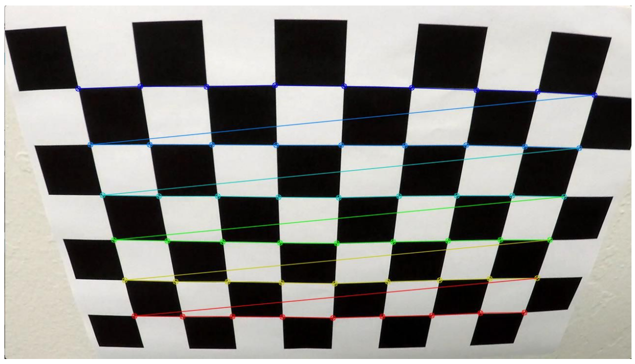
In this project, I have used computer vision techniques to identify lane boundaries and compute the estimate the radius of curvature given a frame of video of the road. To achieve this, the following steps are taken:

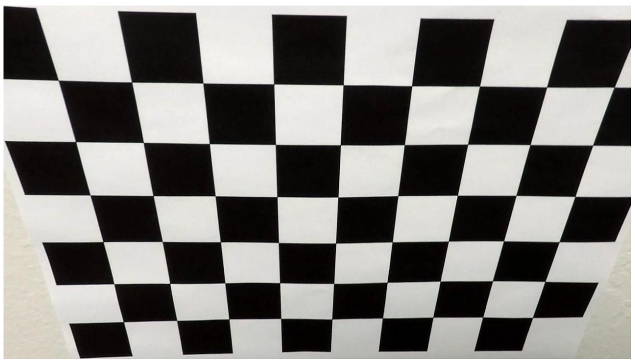
- Computed the camera calibration matrix and distortion coefficients of the camera lens used given a set of chessboard images taken by the same camera
- Used the aforementioned matrix and coefficient to correct the distortions given by the raw output from the camera
- Use color transforms, and sobel algorithm to create a thresholded binary image that has been filtered out of unnecessary information on the image
- Apply perspective transform to see a “birds-eye view” of the image as if looking from the sky
- Apply masking to get the region of interest, detect lane pixels,
- Determine the best fit curve for each lane the curvature of the lanes
- Project the lane boundaries back onto the undistorted image of the original view
- Output a visual display of the lane boundaries and other related information

CAMERA CALIBRATION AND DISTORTION CORRECTION

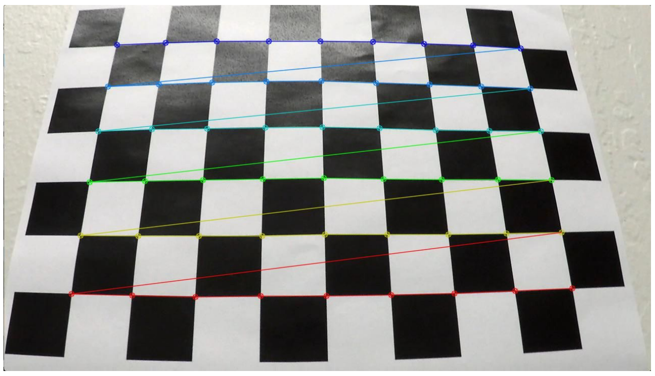
The transformation of a 3D object in the real world to a 2D image isn't perfect. We have to correct image distortion because it changes the apparent size, or shape of an object and more importantly it makes objects appear closer or farther away than they actually are.

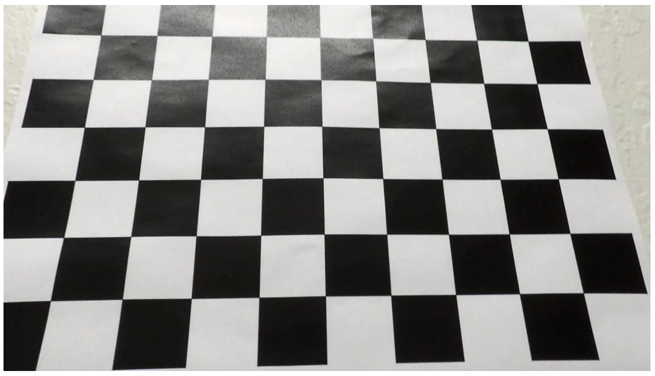
A chess board can be used because its regular high contrast patterns makes it easy to detect and measure distortions as we know how and undistorted chessboard looks like. If we have multiple pictures of the same chessboard against a flat surface from the camera, we can get the from the difference between the apparent size and shape of the images compared to what it should theoretically be. We can create a transform that map the distorted points to undistorted points, and use this transform to undistort any image, which will be discussed more later.





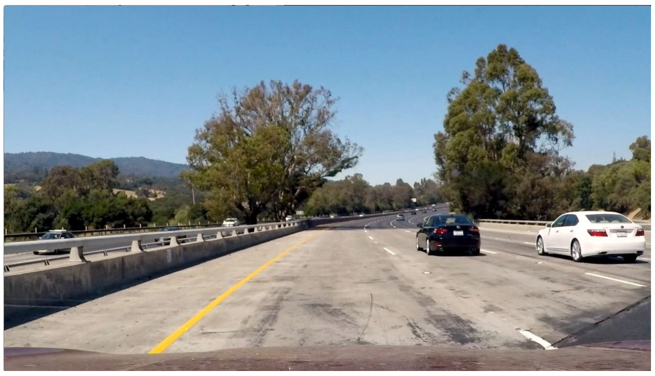
I have made a **ChessBoard** class specifically for this. It takes a path to a chessboard image, and the number of chessboard corners in each row and columns it expects to detect (in this case 9 and 6 respectively). It uses the opencv functions **findChessboardCorners()** and **drawChessboardCorners()**. For each of the twenty chessboard images provided in **camera_cal** folder, We can get the corners (if possible), and a give a set of points (**object_points**) that maps the corner points coordinates to the theoretical coordinates (IE **(0, 0, 0), (0, 1,0).... (8, 5, 0)**), so we can get the camera calibration parameters (**distortion_coefficients**, **camera_calibration_matrix**) that we can use to undistort any image. We use the built-in **calibrateCamera()** function of opencv for this. We feed this parameters to the built in opencv **undistort()** function for get the corrected image.



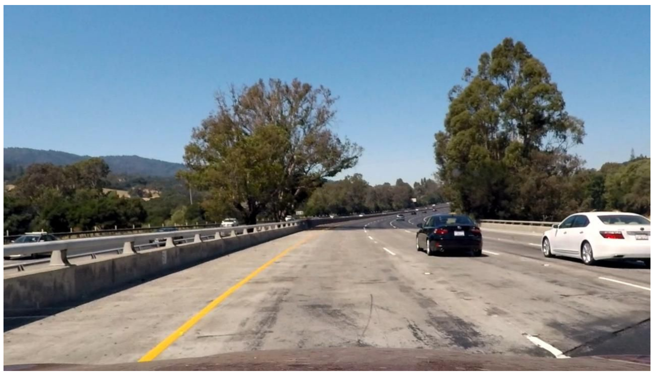


Check **chessboard.py** and **camera_calibration.ipynb** for the implementation details. We save the parameters to a pickle file **calibration_data.p** so we can use it later.

```
chessboards = []

for n in range(20):
  this_path = 'camera_cal/calibration' + str(n + 1) + '.jpg'
  chessboard = ChessBoard(i = n, path = this_path, nx = 9, ny = 6)
  chessboards.append(chessboard)

points, corners, shape = [], [], chessboards[0].dimensions

for chessboard in chessboards:
  if chessboard.has_corners:
    points.append(chessboard.object_points)
    corners.append(chessboard.corners)

r, matrix, distortion_coef, rv, tv = cv2.calibrateCamera(points, corners, shape, None, None)
```



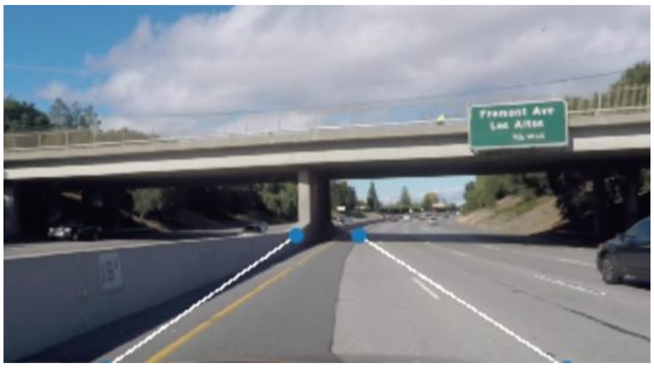
Raw Image

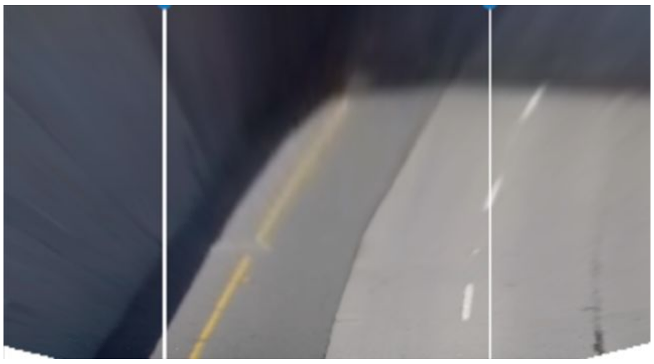


Undistorted Image

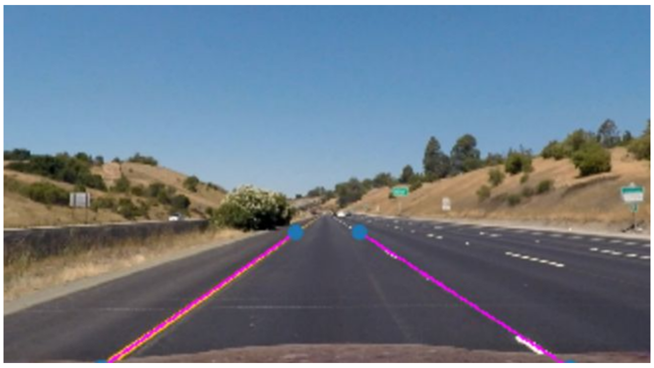
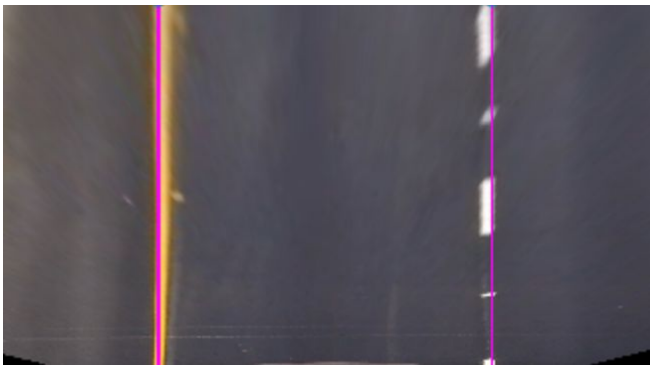
GETTING THE "SKY VIEW" WITH PERSPECTIVE_TRANSFORM

Given an undistorted image of the vehicle's perspective (*vehicle_view*) we can warp this image to output an image of another perspective such as a bird's eye view from the sky (*sky_view*) given a transformation matrix. We can derive transformation matrix (**warp_matrix**) by giving the pixel coordinates of points of the input image of one perspective (**source_points**) and the corresponding pixels coordinates of the output perspective (**destination_points**) using the function **getPerspectiveTransform()**. When we use this **warp_matrix** to output an image to another perspective from a given perspective using the **warpPerspective()** function. You can get the **inverse_warp_matrix** to get from *"sky_view"* to *"vehicle_view"* by switching the places of the source and destination points fed into the function **getPerspectiveTransform()**.





To get the **source_points** and **destination_points**, we get an image in *vehicle_view* of a road that we know have parallel lanes (not curved). So we know that the four **destination_points** will form a rectangle as the output of should have the lanes parallel.. I chose the **source_points** based on eyeballing the pixels on the lanes of the image in "*vehicle_view*" and adjusting based on output of the derived transformation matrix.

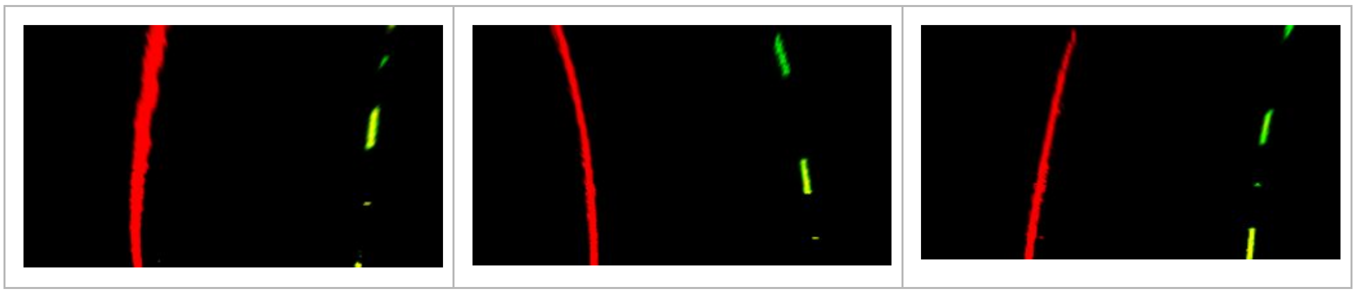
I've made a class **BirdsEye** specifically for this. It takes in **source_points**, **destination_points**, the **camera_calibration_matrix** and **distortion_coefficients**. You can use its **undistort()** function to output an undistorted image that makes use of the **camera_calibration_matrix** and **distortion_coefficients**. You can use it's **sky_view()** function to undistort and then warp the image to ar *"sky_view"* using the **warp_matrix** derived from the **source_points** and **destination_points**. The **BirdsEye** class also has a **project()** function that can be used to project *"sky_view"* lanes onto a "*vehicle_view*" image but more on that later.

```
source_points = [(580, 460), (205, 720), (1110, 720), (703, 460)]
dest_points = [(320, 0), (320, 720), (960, 720), (960, 0)]

birdsEye = BirdsEye(source_points, dest_points, calibration_matrix, distortion_coefficient)
undistorted_image = birdsEye.undistort(raw_image)
sky_view_image = birdsEye.sky_view(raw_image)
```

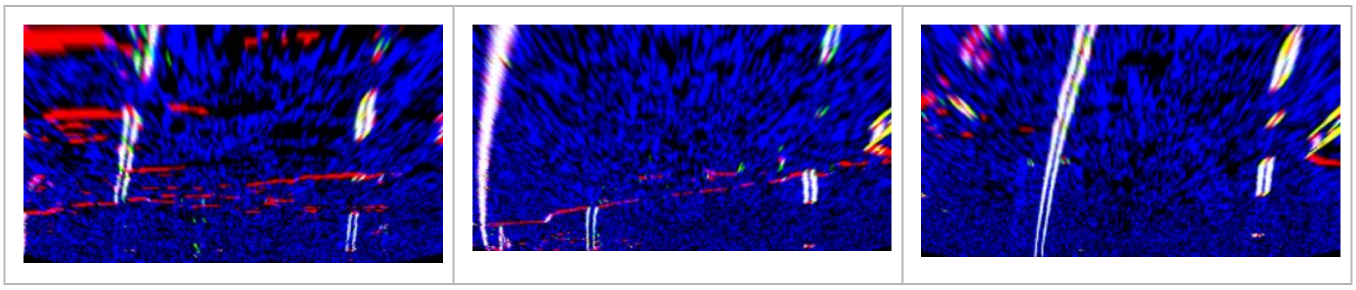
Check **birdseye.py** and **perspective_transform.ipynb** for the implementation details.

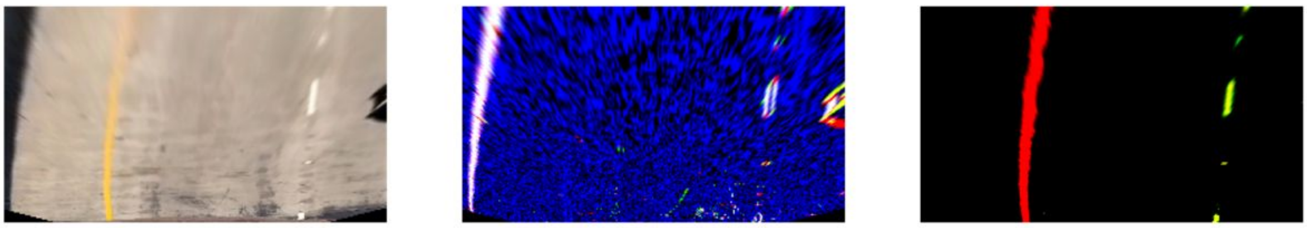
FILTERING LANE PIXELS: GRADIENT AND COLOR THRESHOLDING

I used a combination of the gradient thresholding and color thresholding to filter out unwanted pixels. For color thresholding, I needed to mask everything out except yellows and whites. I represented the colors in HSL format. The *Hue* value is its perceived color number representation based on combinations of red, green and blue, The *Saturation* value is the measure of how colorful or or how dull it is, and *Lightness* how closer to white the color is. The yellow lanes are nicely singled out by a combination of lightness and saturation above a certain value. The white lanes are singled out by having a really high lightness value regardless of the saturation and hue.



To do gradient thresholding we applied the sobel operator. Applying the sobel operator on an image is a way of taking the derivative (also known as the gradient) of the image in the x and the y directions. This derivative measures how fast certain value changes from one pixel location to another. I applied the sobel operator to the lightness value of the image. We used a combination of thresholding the gradient of the x component, the magnitude of the gradient, and the direction of the gradient. The sobel operator also takes a kernel parameter which changes the size of the region you apply the gradient to in an image. A low kernel size can have small noisy pixels but a high one can be prone to unwanted regions being part of of the output. I used a kernel of 5.
We want to weed out locations of gradient that don't have a large enough change in lightness. Thresholding the magnitude of the gradient as well as the x component of the gradient does a good job with that. We also only consider gradients of a particular orientation. A little above 0 degrees (0.7 in radians), and below 90 degrees (0.9 in radians). 0 implies horizontal lines and 90 implies vertical lines, and out lanes are in between them.

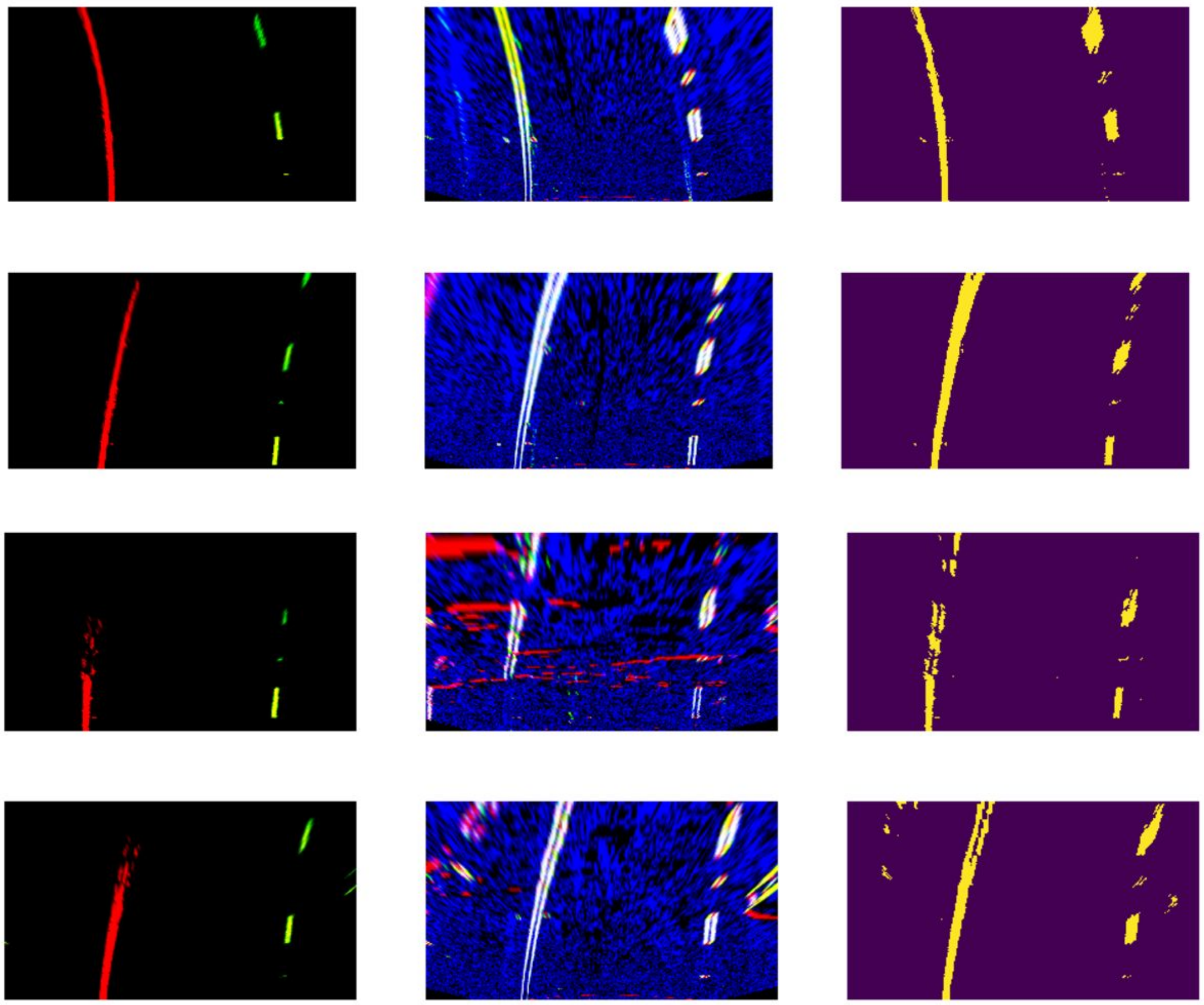




Adding both results of color and gradient thresholding filters out the lanes quite nicely. Although we get some extraneous values from the edges of the image due to shadows, so we apply masking afterwards.

```
p = { 'sat_thresh': 120, 'light_thresh': 40, 'light_thresh_agr': 205,
      'grad_thresh': (0.7, 1.4), 'mag_thresh': 40, 'x_thresh': 20 }

birdsEye = BirdsEye(source_points, dest_points, matrix, dist_coef)
laneFilter = LaneFilter(p)

undistorted_img = birdsEye.undistort(img)
binary = laneFilter.apply(undistorted_img)
masked_binary = np.logical_and(birdsEye.sky_view(binary), roi(binary))
sobel_img = birdsEye.sky_view(laneFilter.sobel_breakdown(img))
color_img = birdsEye.sky_view(laneFilter.color_breakdown(img))

show_images([color_img, sobel_img, masked_binary], per_row = 3, per_col = 1, W = 15, H = 5)
```

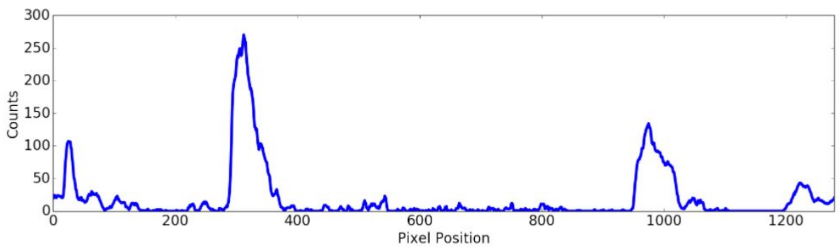
Check **`lanefilter.py`** and **`gradient_and_color_thresholding.ipynb`** for implementation details.



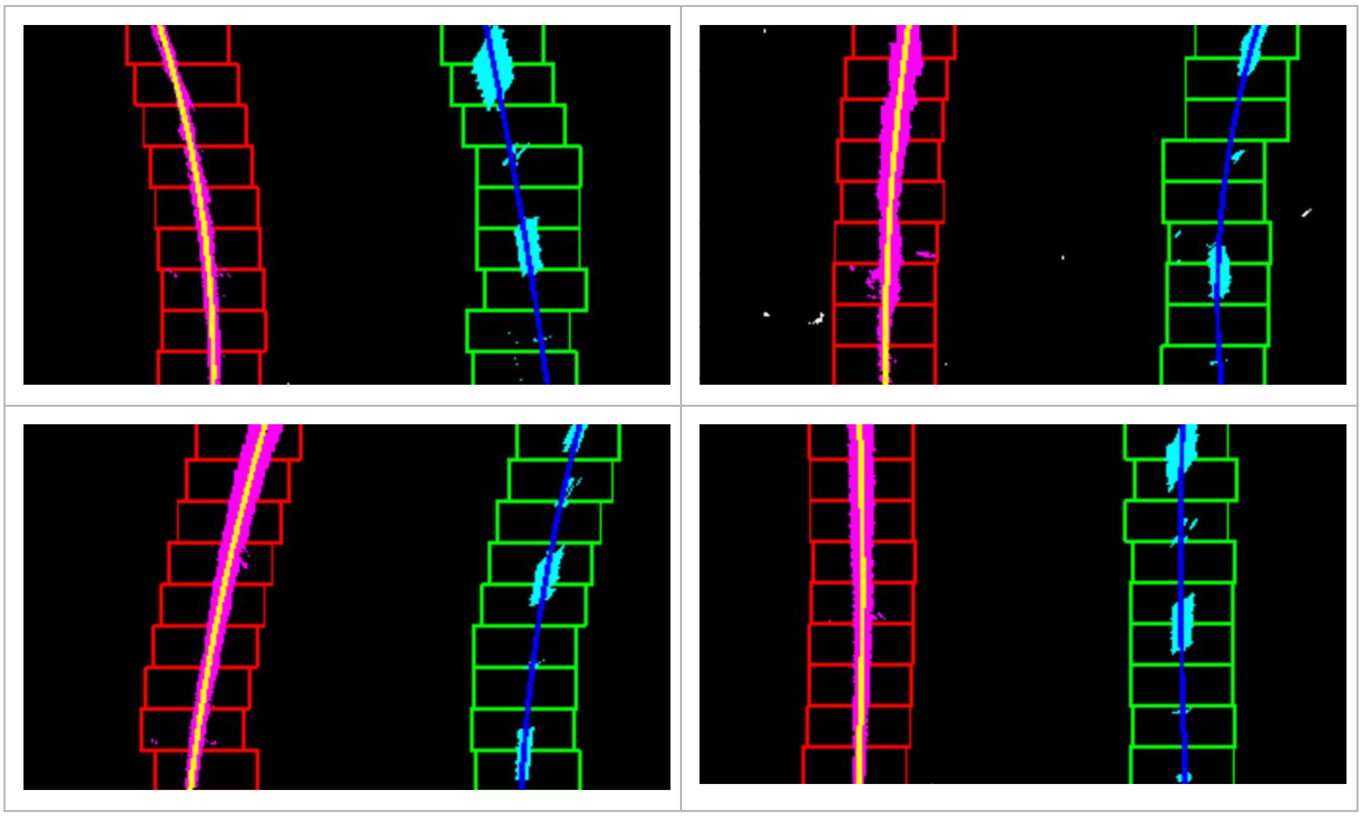
POLYNOMIAL FITTING, MEASURING CURVATURE AND VEHICLE POSITION

We can fit a curve for each lane with a second degree polynomial function **`x = y^2 + By + C.`** We have to find the coefficients for each lane [A, B, C]. We can use the built in function **`polyfit()`**, we feed it points and it outputs the coefficients of a polynomial of a specified degree of the curve that best fits the points fed. To decide which pixels are part of a lane we implement a basic algorithm as explained below.

I can take the histogram of all the columns of the lower half image and I will get graph with two peaks similar to the graph below (taken from the lectures). The prominent peaks of the histogram are good indicators of the x position of the base of the lane. So I use them as a starting point.



I can do a "sliding window algorithm". one window on top of the other that follows the lanes up the frame. The pixels inside a "window" are marked as "pixels of interest" and added to the list of points in the lane. We average the x values of these pixels to we know the base of the next window above. We can repeat this over and over until we get on top the lane. This way we have accumulated all the pixels that we are interested in that we will feed to our **`polyfit()`** function which spits out the coefficients of the 2nd degree polynomial.



Given the coefficients of the 2nd degree polynomial from **`polyfit(),`** use the following formula to compute for the radius of curvature of each lane.

$$f(y) = Ay^2 + By + C$$

$$R_{curve} = \frac{[1+(\frac{dx}{dy})^2]^{3/2}}{|\frac{d^2x}{dy^2}|}$$

```
def update_vehicle_position(self):
  y, mid, kl, kr = self.h, self.w / 2, self.left_fit_curve_pix, self.right_fit_curve_pix
  xl = kl[0] * (y**2) + kl[1]* y + kl[2]
  xr = kr[0] * (y**2) + kr[1]* y + kr[2]
  pix_pos = xl + (xr - xl) / 2
  self.vehicle_position = (pix_pos - mid) * self.kx
```

To estimate the vehicle position, I can calculate the lane width in pixels at the bottom of the image (closest to the camera). Say this was 1000 pixels wide. That implies that each pixel corresponds to 3.7m in real scale. Then I could compute the number of pixels the image center is offset from the lane center and interpolate the offset in m.

```
binary = laneFilter.apply(img)
masked = np.logical_and(birdsEye.sky_view(binary), roi(binary)).astype(np.uint8)
result = curves.fit(masked)

print("[real world] left best-fit curve parameters:", result['real_left_best_fit_curve'])
print("[real world] right best-fit curve parameters:", result['real_right_best_fit_curve'])

print("[pixel] left best-fit curve parameters:", result['pixel_left_best_fit_curve'])
print("[pixel] left best-fit curve parameters:", result['pixel_right_best_fit_curve'])

print("[left] current radius of curvature:", result['left_radius'], "m")
print("[right] current radius of curvature:", result['right_radius'], "m")
print("vehicle position:", result['vehicle_position_words'])
```

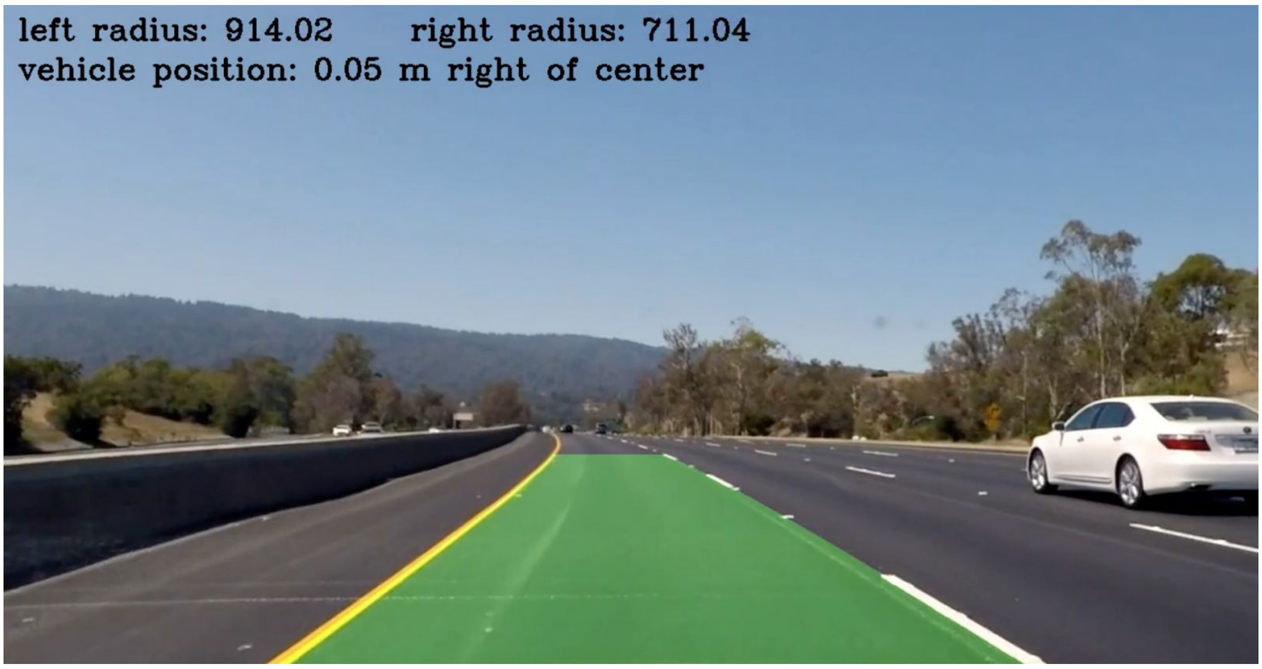
```
plt.imshow(result['image'])
```

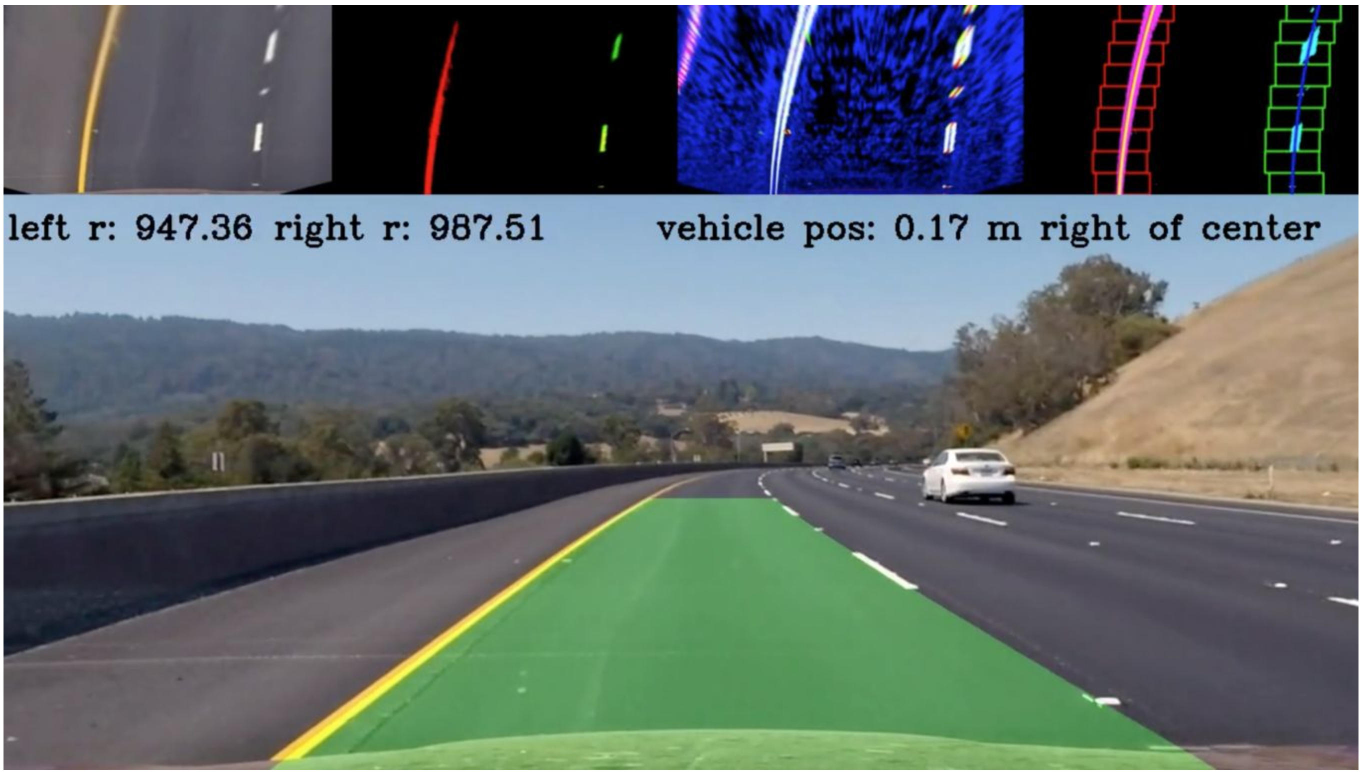
Check **`curve.py`** and **`fitting_curves.ipynb`** for implementation details.

PROJECTION, OVERALL PIPELINE AND VISUAL DISPLAY

Because we now have the lane line curves parameters we get the points of the curves and the **`fillPoly()`** function to draw the lane region onto an image. We can use the **`inverse_matrix`** computed earlier and combine this with a original undistorted image to project our measurement down the road of the image. We wrap this up nicely in a neat little function in **`birdsEye.project()`**

```
ground_img_with_projection = birdsEye.project(ground_img, binary, left_fit_curve, right_fit_curve)
```


left radius: 914.02 right radius: 711.04
vehicle position: 0.05 m right of center


left r: 947.36 right r: 987.51
vehicle pos: 0.17 m right of center

```
matrix = calibration_data['camera_matrix']
distortion_coef = calibration_data['distortion_coefficient']

source_points = [(580, 460), (205, 720), (1110, 720), (703, 460)]
destination_points = [(320, 0), (320, 720), (960, 720), (960, 0)]

p = { 'sat_thresh': 120, 'light_thresh': 40, 'light_thresh_agr': 205,
      'grad_thresh': (0.7, 1.4), 'mag_thresh': 40, 'x_thresh': 20 }

birdsEye = BirdsEye(source_points, destination_points, matrix, distortion_coef)
laneFilter = LaneFilter(p)
curves = Curves(number_of_windows = 9, margin = 100, minimum_pixels = 50,
                ym_per_pix = 30 / 720 , xm_per_pix = 3.7 / 700)

ground_img = birdsEye.undistort(img)
binary = laneFilter.apply(ground_img)
wb = np.logical_and(birdsEye.sky_view(binary), roi(binary)).astype(np.uint8)
result = curves.fit(wb)
ground_img_with_projection = birdsEye.project(ground_img, binary,
                            result['pixel_left_best_fit_curve'], result['pixel_right_best_fit_curve'])
```

Check `projecting_lanes.ipynb`, `pipeline.ipynb`, `pipeline_debug.ipynb` and `pipeline_verbose.ipynb` for implementation details.
Also, please check **project_video_output.mp4** and **project_video_verbose_output.mp4**

PROBLEMS ENCOUNTERED

I really found it difficult to get the right parameters for gradient thresholding which is very important to get a curve that makes sense. I think I still need to tweak to get a better radius of curvature estimates. I also think I could make a more clean readable implementation of the sliding window algorithm to get the pixels of interest for each lane line if I give it more thought.