Project 2: Advance Lane Line Finding Pipeline

By Devin Datt, Due: July 31st, 2018

Objection:

To build a pipeline in python that uses various advance vision techniques to draw the lane lines on and image and ultimately on video with offline processing. This project will build off some of the concepts from the previous Lane Line finding project 1. It will, however, use more advance techniques to fine tune the line detection algorithms to produce a much more clearer and richer visual experience to find lines accurately as best as possible with the tools we have learned thus far in the program.

I decided to breakdown my pipeline into modules to demonstrate each step and at times with its own 'calling' function so as to make it easier for testing and demoing the techniques on different steps of images. These steps have there own input and output comparisons I show at the end of the respective sections. These 'calling' functions then call the main pipeline section. At the end I run the full pipeline without any on screen output. I dump all results in data/output_images (for images processed) and data/videos (for video processed).

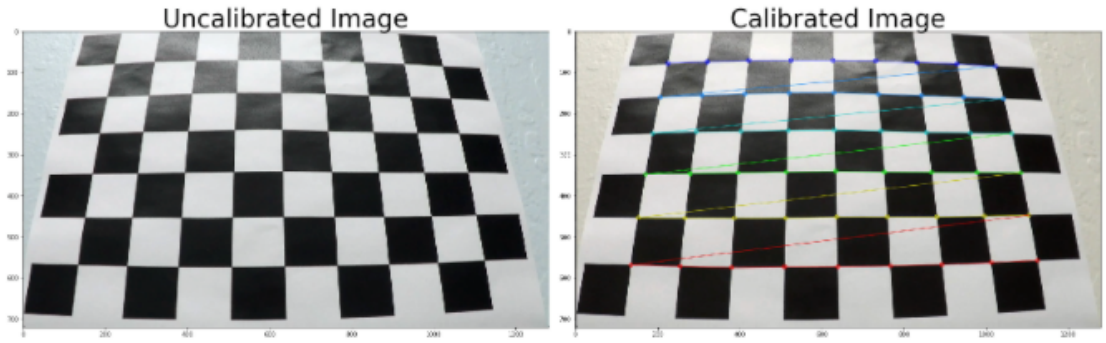
The pipeline we created follows this general process flow:

- 1) Compute the camera calibration matrix and distortion coefficients
- 2) Apply a distortion correction to raw images.
- 3) Use color transforms, gradients, etc.,
- 4) Apply a perspective transform to rectify binary image ("birds-eye view").
- 5) Detect lane pixels and fit to find the lane boundary.
- 6) Determine the curvature of the lane and vehicle position
- 7) Transpose the detected lane boundaries back onto the original image.
- 8) Output visual display of the lane boundaries and numerical stats
- 9) Run pipeline on videos

1) Camera calibration

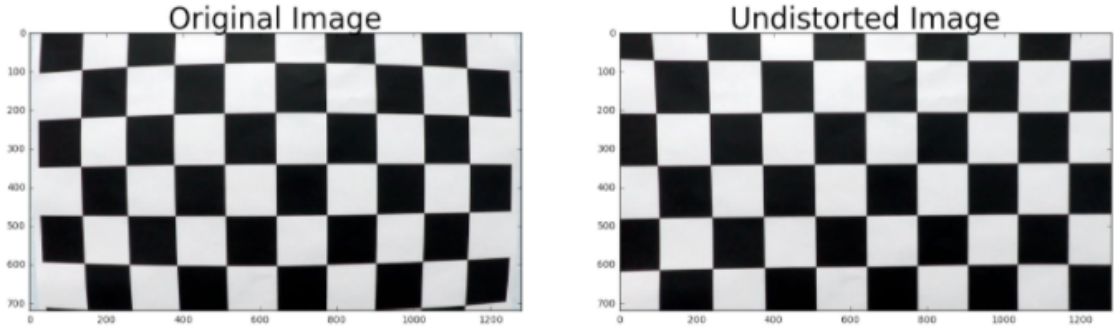
Here we are to compute the camera calibration matrix and distortion coefficients in a function called 'camera_calibration'. Note, we only need to compute these variables once, and then we can apply them to other subsequent undistorted image frames. We start by identifying the board dimensions, in our case it was a 9x6 squares wide. We then creating arrays (lines 3-4) to hold the **object_points** which just a frame work to hold the positions of the checkerboard corners when found, and **image_points** array to hold the position of the actual corners found.

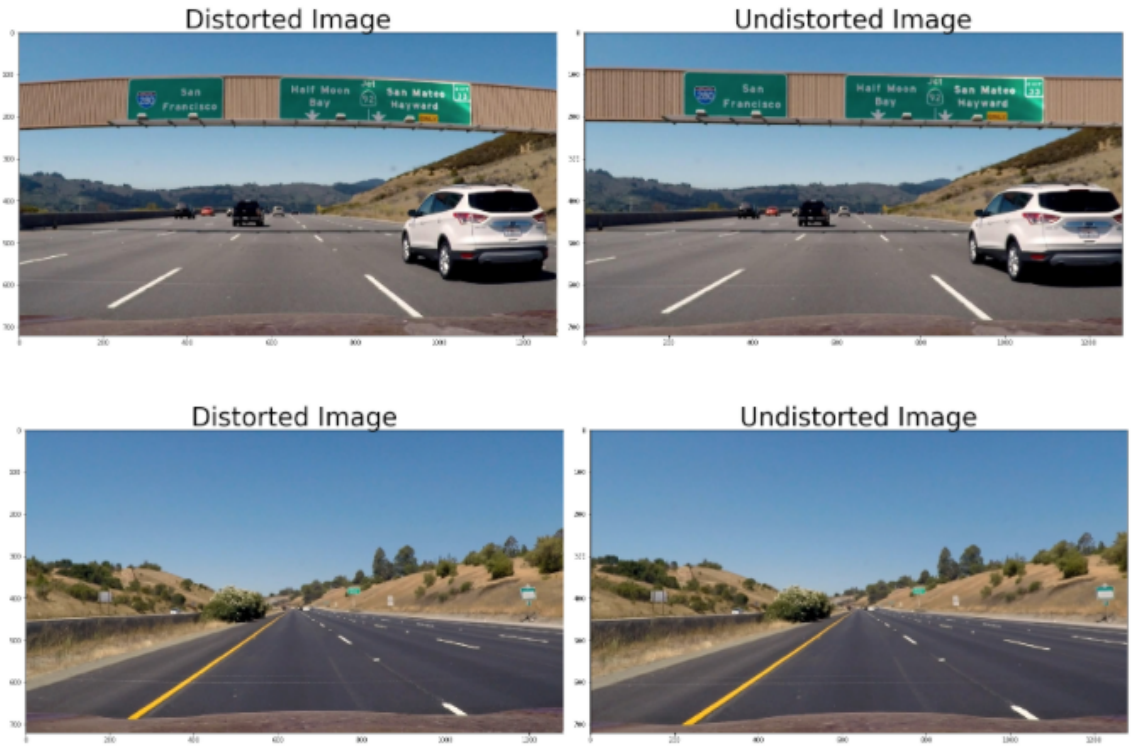
We convert the image to a grey scale and then run that grayscale image thru the ‘findChessboardCorners function (line 10). This returns the corners found and that is passed thru ‘drawChessboardCorners’ to draw connected lines on the passed original image. I then passed both calibration arrays along with the image dimensions (line 17). It takes in uncalibrated chessboard image and returns the distortion coefficients stored in a pickle file saved in the ‘camera_cal’ folder. This will be used later in the next step. The output of this step is the original chessboard with lines drawn attaching the corners:



2) Distortion correction

The next step is to create an undistortion function called ‘cal_undistort’ (code block #5) that will take in a distorted image and return that image undistorted. It does this by passing the ‘mtx & dist’ coefficients that we saved in our pickle file into the ‘cv2.undistort’ function (line 6). The resulting image is the original slightly stretched out. Here are some results:





It might be difficult to see the changes from the original to end result, but if you look closely to the edges of the undistorted image you will see some slight corrections in radial distortion.

3) Color/gradient Transformation

Great, now that we have our image undistorted, we will now begin to apply several steps to mutate and transform the image all with the goal to optimize our chances to isolate and detect lane lines on the road. We need this because if we

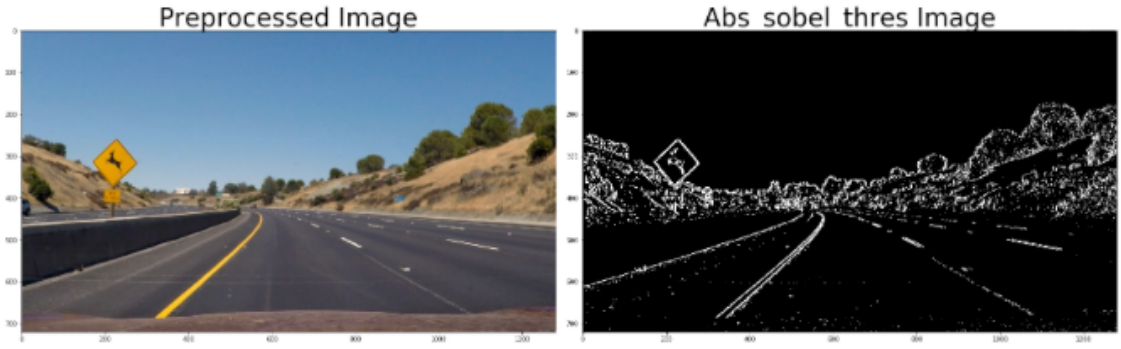
can use just cameras to detect where on the road it is and the path in front of the car we can successfully write intelligent and intuitive algorithms to negotiate our way down the road.

The next set of steps will be running through a combination of Color and Gradient transformations on our images. The objective here is to use color transforms, gradients, etc., to create a thresholded binary image to see what will produce images with lane lines illustrated the best. We find that applying sobel in the 'x' orientation seems to provide our best guess to find lane lines as they are mostly horizontal (ie. in the 'y' direction) which sobel in that direction will be limited in comparison.In general we are try to achieve a modification of our road image into something like this:

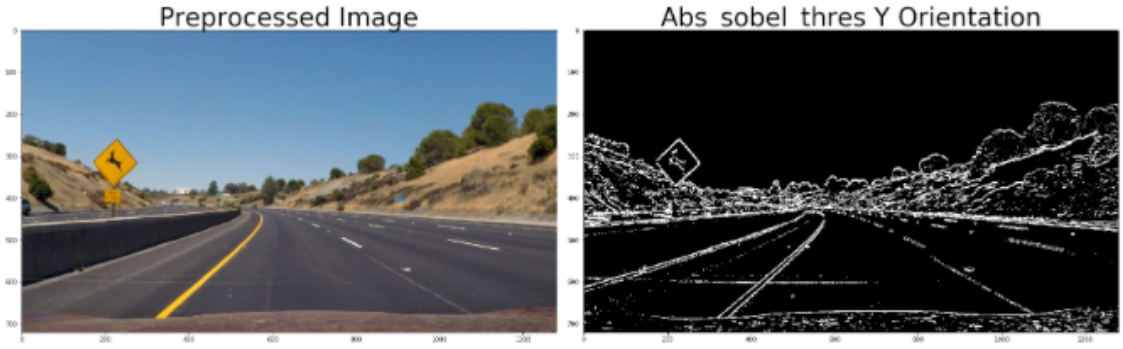
I figured this part of the pipeline was quite complicated as there are many types of colour and gradient functions/techniques each and each have there own set of thresholds and settings to optimize. As a result, I broke up this section into a multi tiered calling process. It starts with creating the code for each of the major colour and gradient functions:

- **abs_sobel_thresh** (lines 1-10) : takes in a coloured image and applies a sobel gradient in either the 'x' or 'y' direction. It does this by taking in an image and the orientation value for the gradient and applying it to a grayscaled image. After applying scaling and binary transformation we applied thresholds on the absolute sobel value (line 9). This function is good for picking up the vertical lane lines using sobel in the 'x' orientation. The following image is this function in the *'x' orientation with sobel_kernel =5 and threshold on binary limits of 20-150*.

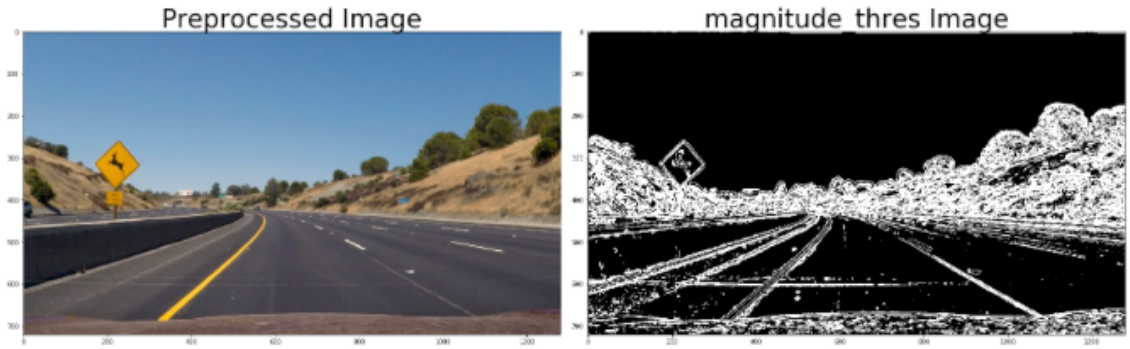
Absolute Sobel in X Direction



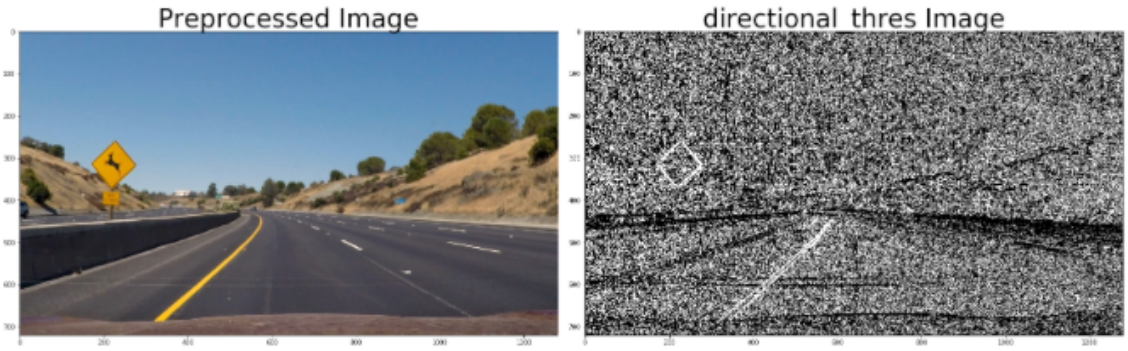
Absolute Sobel in Y Direction



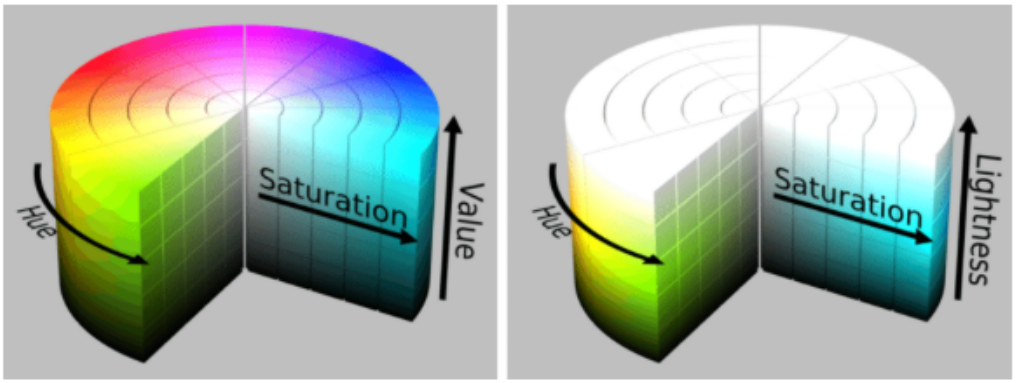
- **mag_thresh** (lines 11-20) : similar to ab_sobel_thresh transformation but applies sobel in both 'x' and 'y' and then takes the absolute value on that scaled binary analysis (line 15). This function further picks up vertical lines but allow picks up some horizontal items. The following image is this function taking the absolute *'x' & 'y' orientation values with sobel_kernel =9 and threshold on binary limits of 10-80*.



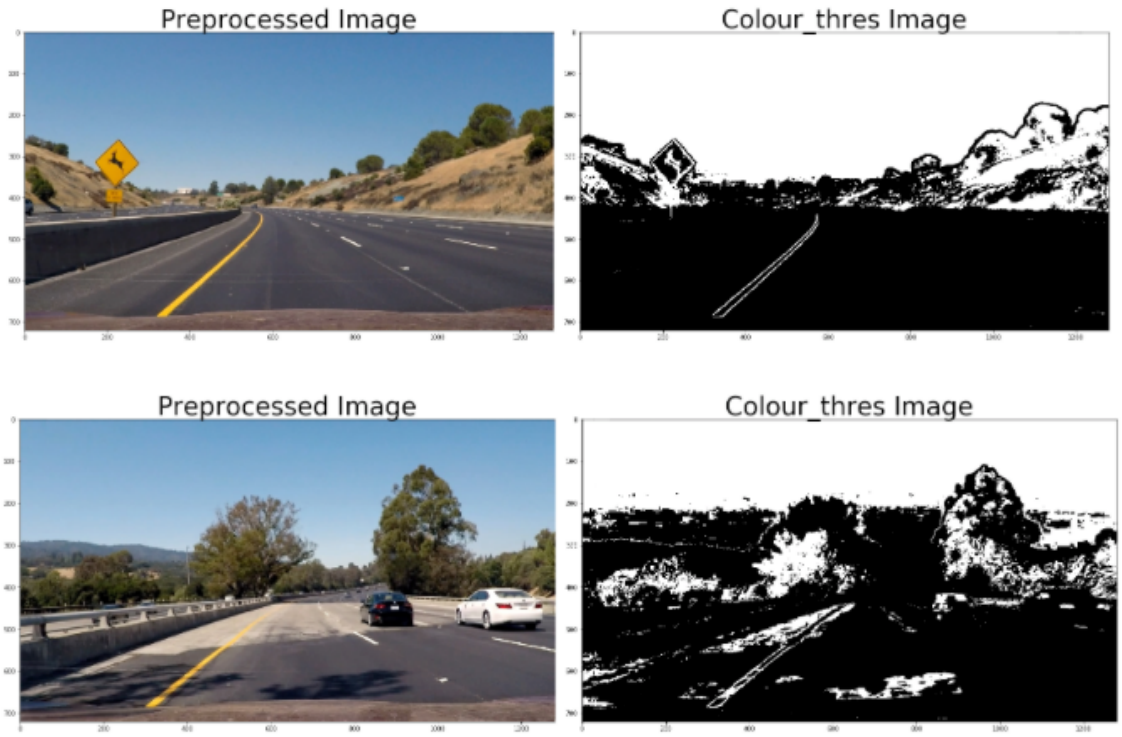
- **dir_thresh** (line 21-29) : this function also works similar to mag_thresh in both ‘x’ and ‘y’ sobel but this one further applies absolute arctan value of the sobel y/x ratio (line 35). It then applies threshold on those values. It seems to be only useful to pickup deep outlines of defined objects (solid lanes, street signs..etc). The following image is this function taking the absolute arctan y/x sobel directional values with sobel_kernel =3 and threshold on arctan values of (0.2 – π/3).



- color_thresh (line 30-45) : this function works only on the different colour spaces. It does this by converting image once to RGB to HLS and splitting out the ‘s-channel’ and another conversion to RGB to HSV and pulling out the ‘v-channel’.



- Seems like applying thresholds on the ‘s & v-channels’ produce some pixels that survive lines that are ‘yellow’ and covered by shadows. The following image is this function taking the colour values thresholds on s-channel (70-255) and v-channel (33-220).

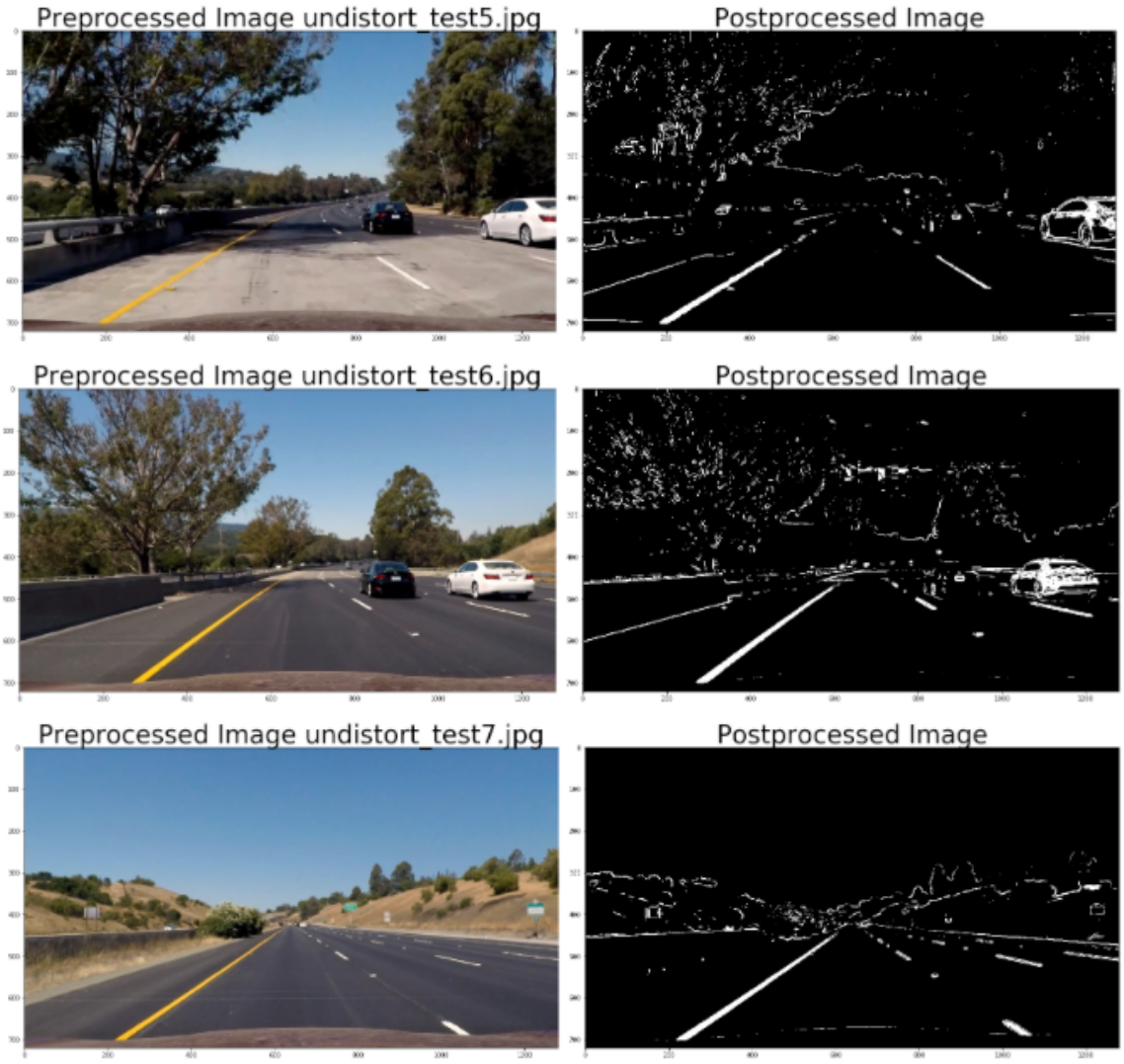


Now that we have our three (3) gradient and one (1) colour transformations created and tested we now pull them into one calling function (called 'color_gradient', code block #10) that takes an input of a undistorted preprocessed image and runs it through a series of these functions where we modify which ones are chosen and what thresholds are applied to each to see what get as an output of binary image that show distinct road lane lines with little noise.

One of the biggest issues in this part of the analysis is choosing which kernel size and threshold settings to use in each of the gradient and colour transformations. While I experimented with multiple permutations for several hours and got some respectable results I continually feel the possibility that there is still a group of settings that will produce some incremental improvement in showing the lane lines. I decided to settle on the following settings for my colour and gradient transformations:

```
gradx = abs_sobel_thresh(image, sobel_kernel=9, thresh=(70,100))
grady = abs_sobel_thresh(image,borient='y', sobel_kernel=5,
thresh=(20,255))
mag_binary = mag_thresh(image, sobel_kernel=9, mag_thresh=(80,210))
dir_binary = dir_thresh(image, sobel_kernel=17, arc_thresh=(0,np.pi/2))
c_binary = color_thresh(image, s_thresh=(120,255), v_thresh=(130,255))
```

With this group of settings I get the following results on my test images:



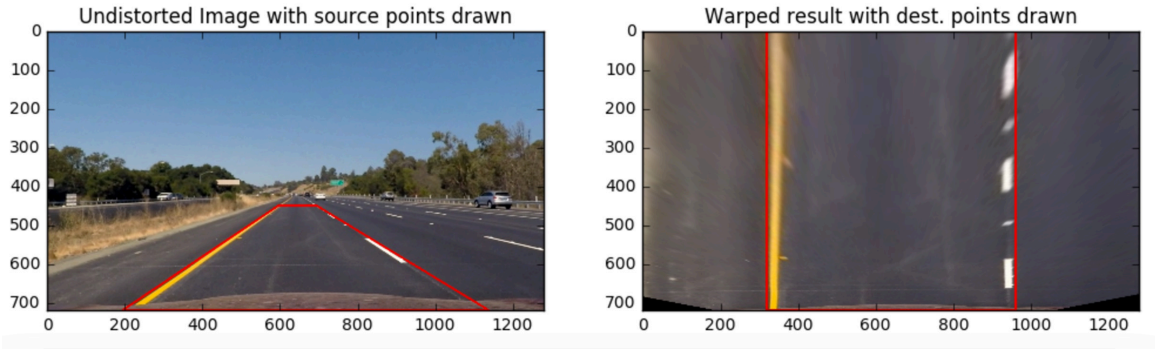
As you can see from these test images is that we can see the lane lines quite accurately even in conditions when the road switches in light, shading, or shadows. We don't have to worry about the external 'noise' that is produced beyond the lines as this will be filtered out in our next step when we do the perspective transformation.

4) Perspective transform

Here we want to see how a front facing camera image would look like from a 'birds eyeview'. So what looks like a trapezoidal shape in image would be roughly two parallel lines from above.

We will have the best results if we do the following:

- Assume the road in front of the car/camera is completely flat
- We choose an image of road that is general ‘straight’ with no curves



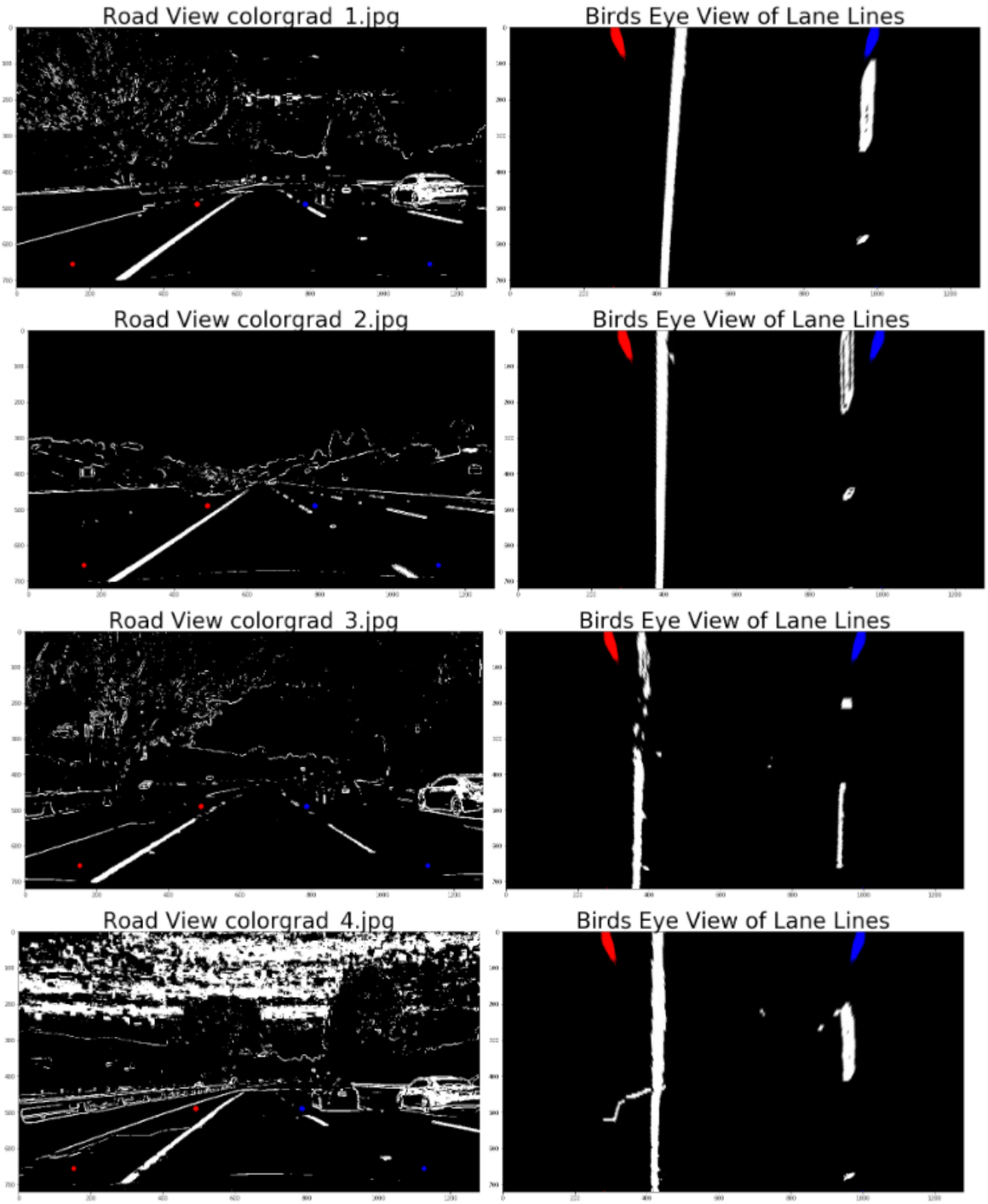
This should also be able to handle curves as this function is processing multiple frames a second. In other words, processing this step on each incremental frame will produce the ‘bending’ lines.

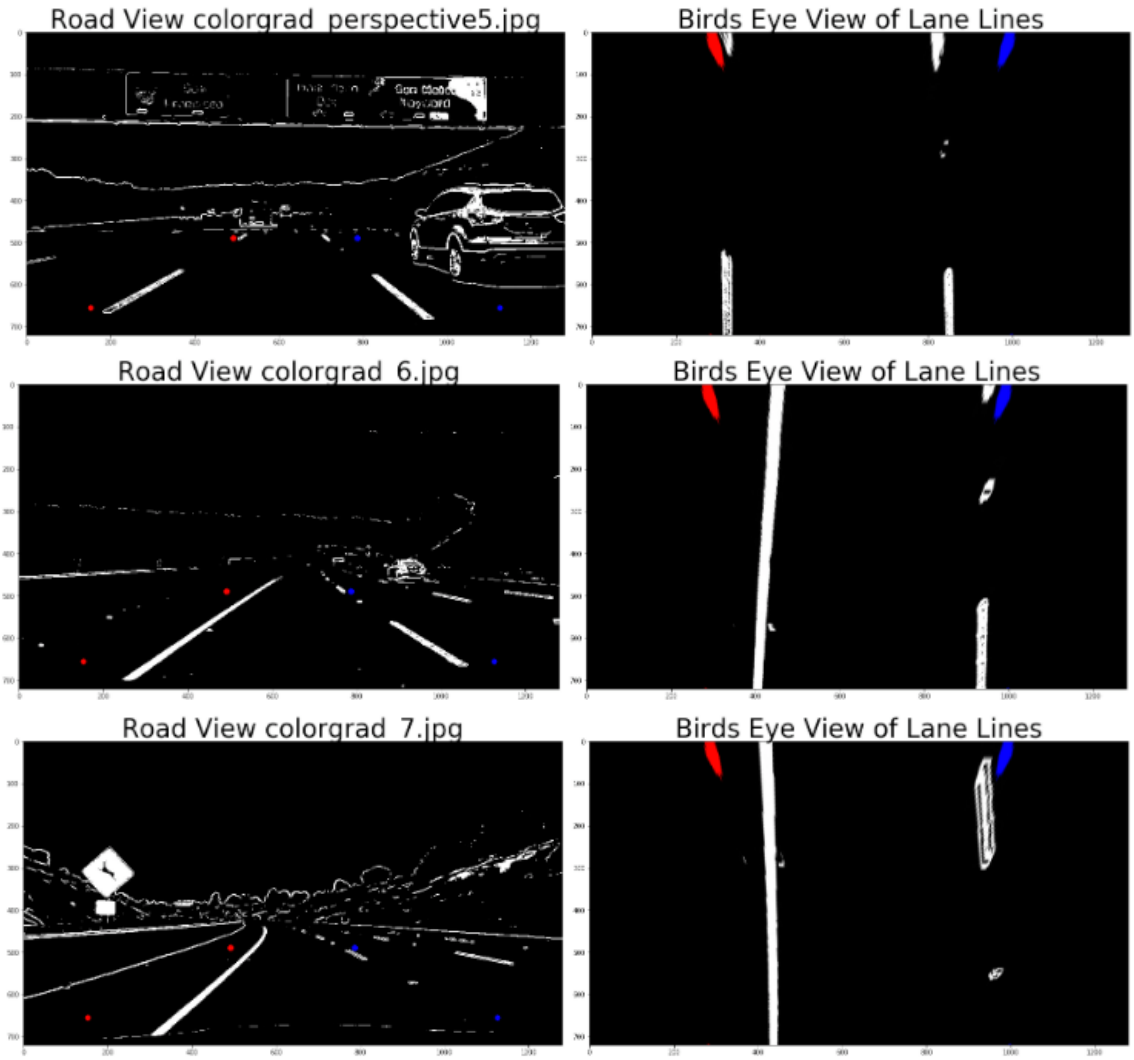
The process I choose to implement here is analyze our binary test images and find four points lying along the lines that, after perspective transform, I made the lines look straight and vertical from a bird's eye view perspective. This is in Step 4 ‘Perspective transformation’ in a code block #13 called ‘perspective_trans’ which takes in a binary image. It then creates some boundaries of a trapezoid in the middle of the image with these statistics (lines 5-8 in this function) :

```
bot_width = 0.77    #percent of bottom line of trapezoid
mid_width = 0.26    #percent of middle 'distant' line of trapezoid
height_pct = 0.68    #percent of trapezoid height
bottom_trim = 0.91 #percent from top of image to exclude car hood
offset = img_size[0]*0.25
```

This perspective function calls the ‘region_of_interest’ in code block #12 and passes it the binary image from the last step and array of settings for the vertices of trapezoid. It uses these parameters to mask out the region outlined in the trapezoid vertices and filters out the surrounding noise of the shoulder, scenery, signs…etc. Once this ‘region-of-interest’ is passed back to the main perspective function it then applies the ‘cv2.getPerspectiveTransform’ (lines 22-23) with the mapping ‘source’ and ‘destination’ points to get the transform matrix (M) and the inverse transform matrix (Minv) which will be used later. For now, as the last step of this function we pass the ‘region-of-interest’, image size, and the transform matrix (M) to produce a ‘warped’ image from ‘cv2.warpPerspective’ function (line 24). The resulting images are lane lines close as possible to parallel.

Note: I used 'red' and 'blue' dots below to illustrate the vertices of the trapezoid corners in the images. These dots were commented out during the actual running of the code on real images and video as this would add further noise and false pixels as lanes.





The challenge I found here was to make my perspective lane lines as parallel as possible even thou I was pulling transformations on images of a ‘straight road’ (ie. test 7 above).

5) Detecting Lave Lines

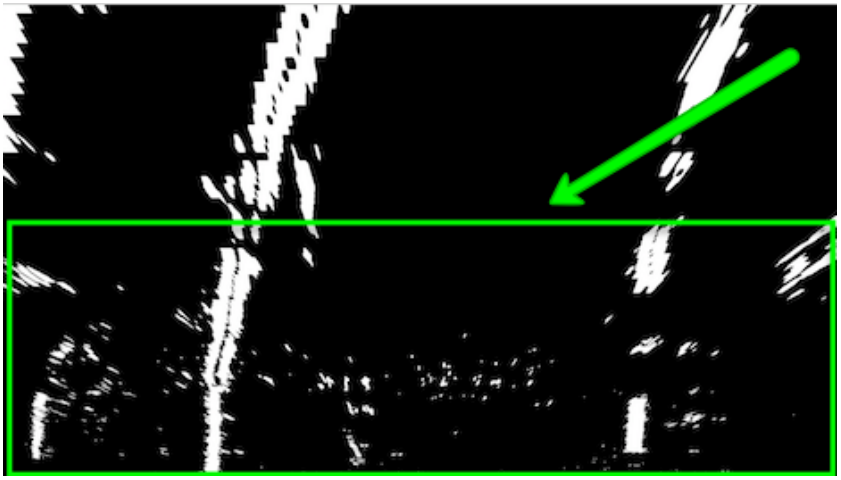
The next step is to now look at the perspective images (without marker dots) and decide which pixels in the frame are lane line pixels. There are several techniques

we can use to do this step: Histogram, Polynomial regions, and Centroid convolution searching methods.

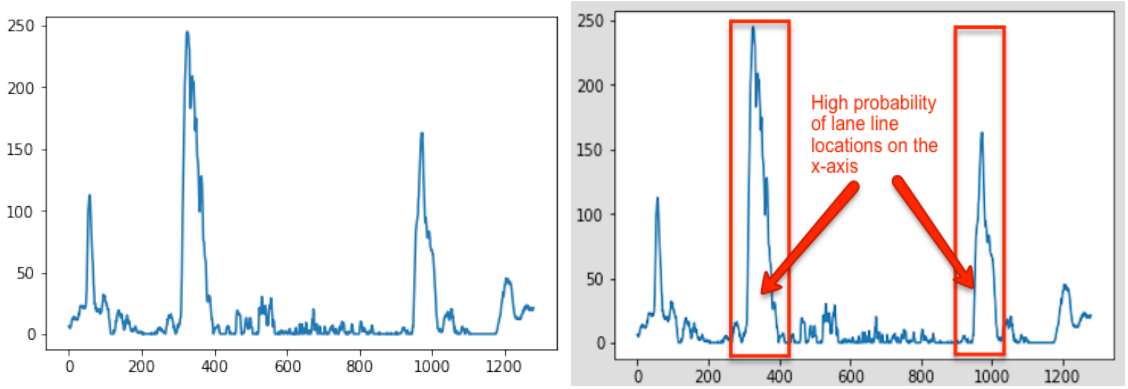
Histogram Sliding Window Method:

To start the process of finding the lane lines in a image frame, I used the function 'find_lane_pixels' (code block #15) that will systematically scan the image for white 'active' pixels.

I figure it was best to get a general starting point on the frame where to start the search. We can do this by searching the closest part of the screen to the car (lower half) which should give us the best chance to find the most accurate lines.



I used the Peaks in Histogram method to explicitly find which pixels are parts of the lines and which belong to the left line and which belong to the right line. This method simply adds up the places where the screen indicates 'white' or (1) from the threshold and perspective steps along the x-axis. This part of the screen is typically the lower half of the image as the car is mounted midway and looking out into the horizon. The results look something like this:



We then use these peaks as starting points as part of our effort in an effort to split the histogram into the individual lane lines (left and right) to find 'where' the lane lines are going (straight, bending left…etc).

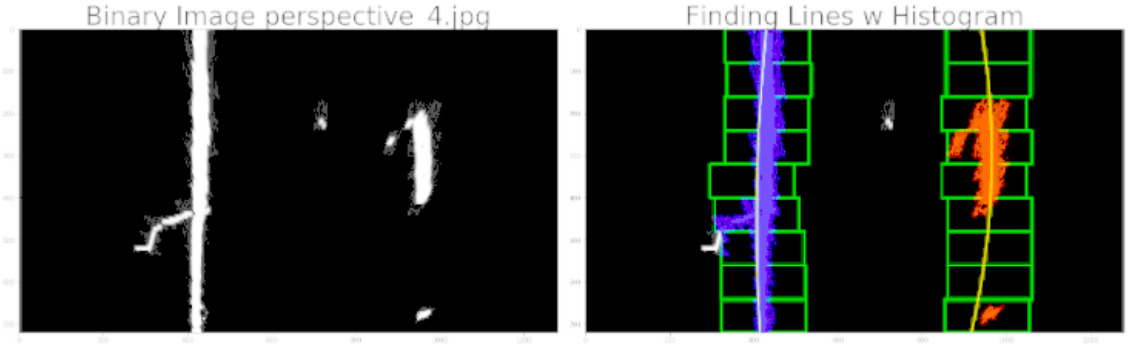
To do this we take our histogram previously found and split it down the middle on the x-axis. We then find the peaks of the 'left' and 'right' halves. We divide the screen into smaller window sections and iterate a search as 'active' pixel and re-centering our 'middle' of the valid lane line every certain amount of pixels. For my search I used the following hyper parameters coded within the function:

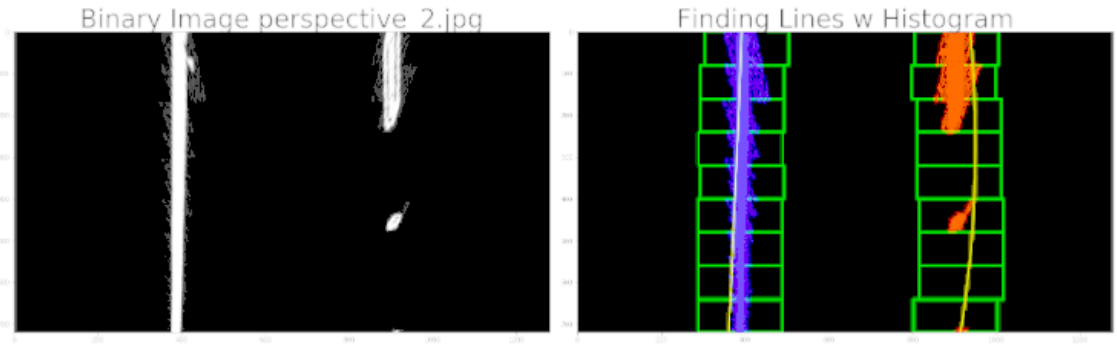
```
nwindows = 15  # Choose the number of sliding windows
margin = 100   # Set the width of the windows +/- margin
minpix = 70 # Set min number of pixels to recenter window
```

The result of this function will be four (4) sets of arrays (**`leftx, lefty, rightx, righty`**) with the coordinates for x & y for both left and right of all the active pixels found within sliding windows following the histogram starting point from the bottom to the top of the image.

The next step is to take those arrays of active pixel coordinates and pass them back to our 'fit_polynomial' (code block #16) to attempt to find the best polynomial line of 2nd order that would fit these x-y coordinates. It does this in first finding the polynomial coefficients in lines 3-4 and using a y-axis of same size as the height of the image.

Here I found some of my lines not to come out exactly parallel as the left lane seemed to be more dominate as it was a solid line and therefore provided more of a reliable measurement of the road lane line as oppose the right lane line which did its best to fit a polynomial with the little data it had at times:



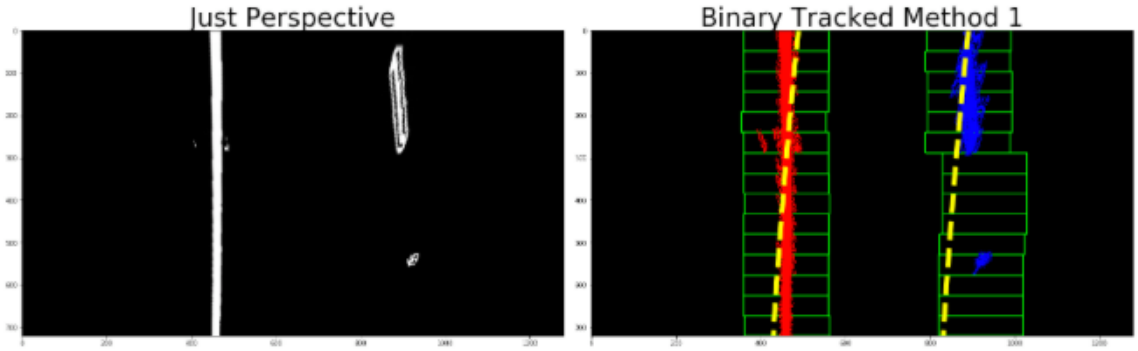


I found it might be best at some point in the code to 'force' the line with least amount of pixel info (ie. a dashed line) to take the polynomial curvature cofficients (A & B) and leave the 3rd component untouched as this was the y-intercept would still be needed as part of its line to intercept at its respective x-axis. I tried this code in my 'fit_polynomial' (line 5-8) but it only seem to work some of the time.

```
if len(lefty) >= len(righty):
    right_fit[0:2] = left_fit[0:2]

else:
    left_fit[0:2] = right_fit[0:2]
```

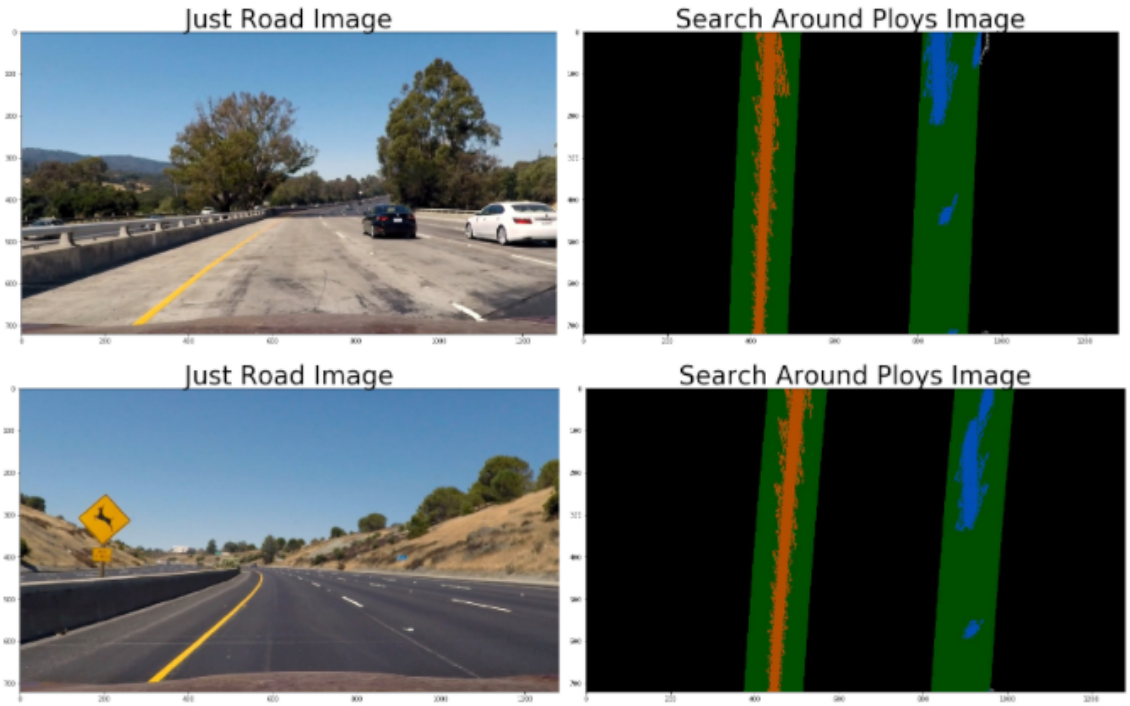
I got the



Search Around Polys Method:

An improvement on the previous method is once we searched the first couple of images and found our active pixels this should give us a rough estimate of where the lines are. As such, we shouldn't have to do another histogram/sliding window search for every next frame. We initially found our starting point for the lane lines with the histogram and fitted a 2nd order polynomial to best fit the activated parts of the image with left and right lane lines. Since the next image is only a fraction of a second later the lane lines shouldn't change much and should allow us to simply use a margin around our polynomial line to best capture most of the lane

lines for the next frame. We can do this by using the polynomials that were generated from our last step and draw boundary lines of width margin size on each size. This new boundary area around the polynomial (hence the name!) can limit our search area to find the likely existing active pixel lane lines. I did this with a function called 'search_around_poly' (code block #17). Depending on the margin width you use your margin of error around the 2nd polymonial would look something like this. I used a tight margin of 100 pixels below.

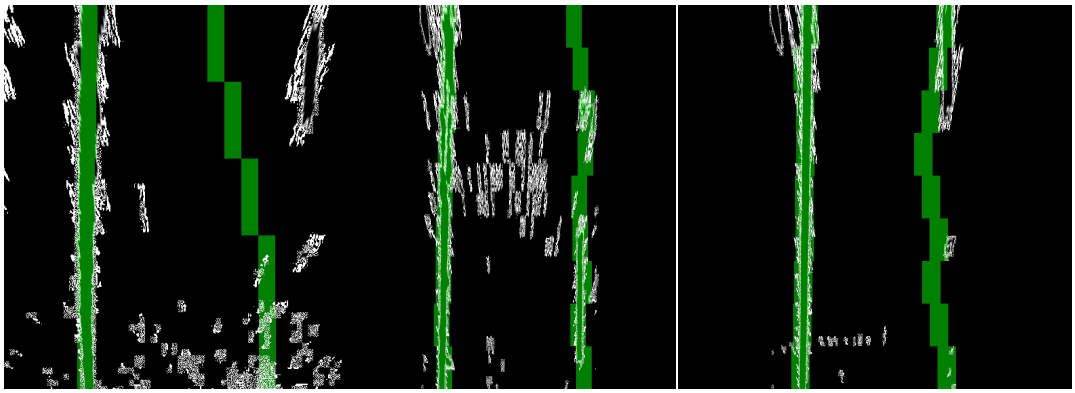


Once the boundaries (green area) is determined you can speed up your search with only your 'find pixels' to only search this limited area. Along with the resulting image, at the end of this function I chose to pass also the pixels found in left and right arrays (ie. leftx, lefty, rightx, righty). These arrays flow out right thru the pipeline and enter into the next process of 'draw_boundaries' (code block #17) .

Centroid Method:

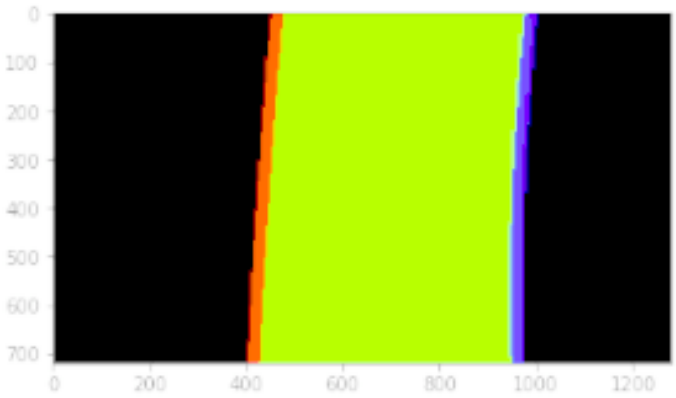
A third method we could use to find the active pixels is to use a convolutional search with a kernel size to 'sweep' across the image lines. If the signal of the convolution of the kernel and lane pixels is strong (maximum value) then we capture that center as a point and store it in an array for both left and right lane lines.

I ended up **not using** this technique due to technical issues. My polynomials were not calculated correctly for some reason and the lines were all over the place, as illustrated in some of my samples below:



6) Draw Boundaries/ Measure Curvature

Once we have our isolated the active pixels in the binary image with one of our detection methods above we then use this opportunity to take advantage of the image still in perspective warped format to draw some visual boundaries to indicate our functions accuracy. As mentioned before I passed the pixels we founded for left and right lanes out of our histogram/polynomial function which end up at inputs into this function 'draw_boundaries' (code block #18). I use these arrays to recalculate my left and right polynomials. The reason I did this is because when I use the code supplied in the course lesson I ran into 'array size mismatch' errors (lines 7, 11, 14, 17, 30) anytime I tried to plot or use 'ploy fit' functions with the y values (ploty or yvals) and my found x values (leftx or rightx). Not sure what this is the case but seems like like my 'y values' are to big for my x arrays. In either case, to create a band aid solution I simply 'macgyver'ed' the arrays to match. This allowed me to create the colour red & blue bands for the lanes and middle polynomial green filled section (in lines 25-29) This seemed to work surprisingly well, as it shows good accuracy on images.



Before leaving this function I decided to take advantage of the variables I have access to for radius and offset calculations. Therefore, to assist one of the next steps and to prevent from passing unnecessary variables I decided to also do the calculations of the Radius of curvature 'curverad' and camera offset 'center_diff' in this same code block on lines 30-35.

To measure the radius of our curve we will use the formula below and some of the measurements per pixel from our class function:

$$R_{curve} = \frac{(1+(2Ay+B)^2)^{3/2}}{|2A|}$$

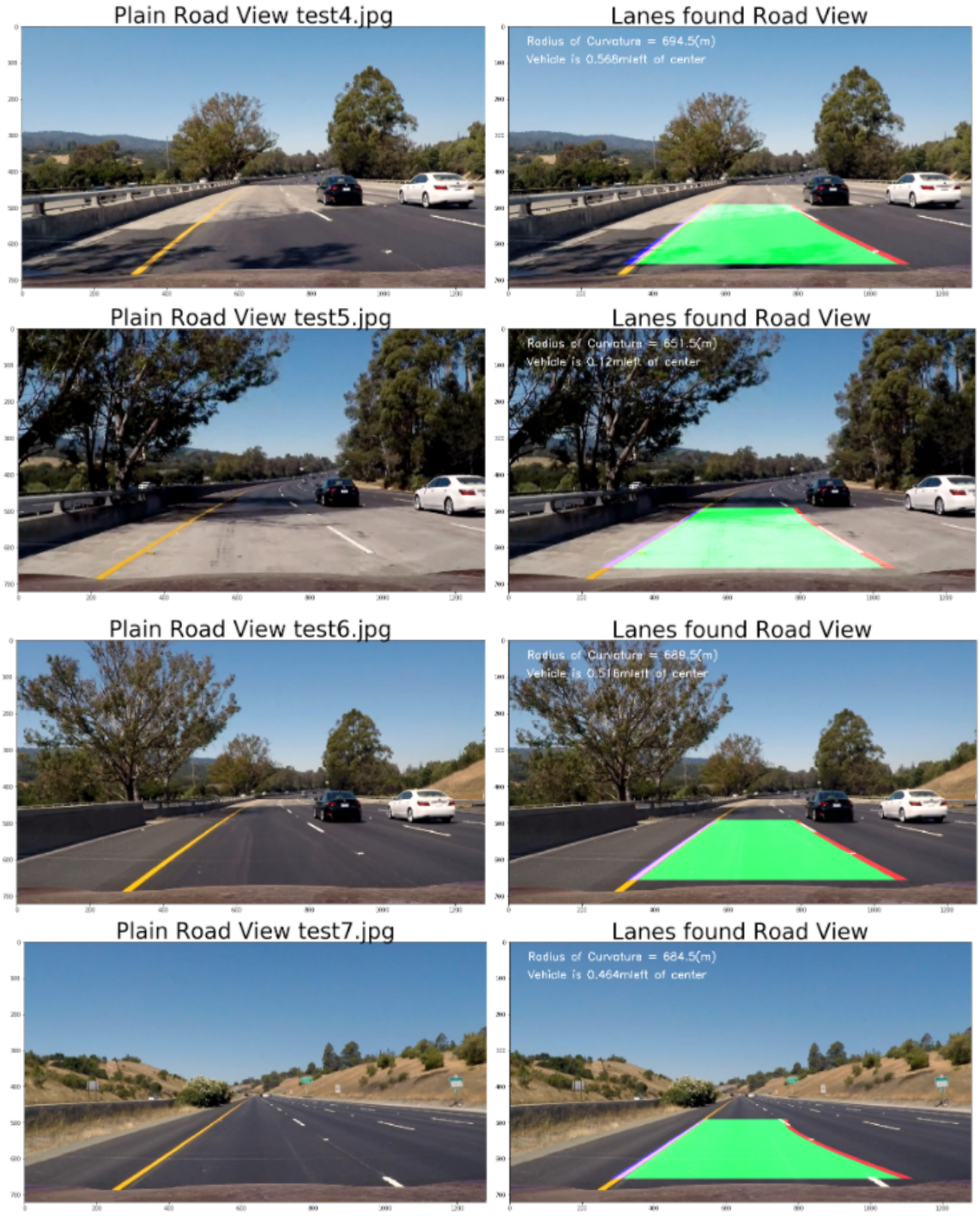
7) Warp Image Back on Image

Once all perspective view of the road is processed (ie. colour/gradient performed, active pixels found, boundary and visual effects added on image its time to project this 'back on the road'. We could do this by calling our same perspective transform function we used in step 4 but I felt there was a lot of special code in that step to handle getting vertices and transformation correct that is should stand alone by itself and also, using a separate function (ie. 'perspective_back, code block #19) gives a better understanding of the actual steps in the pipeline and also makes it more modular just incase we want to 'fork' this perspective_back transformation into another direction and combine it with new visualization or output conversions needed, ex. for a mobile application. This block is simple and basically runs our 'warpPerspective' function but this time with the inverse matrix coefficients (Minv). It then combines this with a 'addWeighted' function to combine the original road and our perspective binary road with coloured boundaries (lines 3-4). The output, in my opinion, for the first time gives you a sense that the info

superimposed on the road is informative and a much needed step in a long series of safety processes towards autonomous navigation.

8) Write Stats on Image

As with the last step and keeping the flow of the pipeline process theme, I separated out this function 'write_on_screen' (code block #20) to pass the curvature of the radius and offset statistics to write on the screen, lines (1-6). Output provided a dramatic effect:



9) Run Videos

I attempted the videos and I seemed to only have success with the 'easy' video as it probably had gentle non aggressive curvatures in the road lane lines. Output can be seen here → https://youtu.be/cZZkAalJ6fE

```
CPU times: user 12min 5s, sys: 1min 22s, total: 13min 28s Wall
time: 8min 38s
```

However, I didn't have any luck in completing the video process for the 'Challenge' or 'harder challenge' videos. The both failed at 28% and 27% respectively. Not sure what frame this is but both errors seem to be indicating the inability for my functions to provide 'non-empty' arrays to calculate the polynomials.

Error from "Challenge" video:

```
<ipython-input-228-5fbfc181d476> in search_around_poly(binary_warped, left_fit,
right_fit)
29
30     # Fit new polynomials
---> 31     left_fitx, right_fitx, ploty = fit_poly(binary_warped.shape,
 leftx, lefty, rightx, righty)
32
33     ## Visualization ##
<ipython-input-98-da6edfb70f20> in fit_poly(img_shap
e, leftx, lefty, rightx, righty)

1 def fit_poly(img_shape, leftx, lefty, rightx, righty):
2     # Fit a second order polynomial to each with np.polyfit() ###
----> 3     left_fit = np.polyfit(lefty, leftx, 2)
4     right_fit = np.polyfit(righty, rightx, 2)
5     # Generate x and y values for plotting
~/anaconda3/lib/python3.6/site-packages/numpy/lib/polynomial.py in polyfit(x, y,
 deg, rcond, full, w, cov)
553         raise TypeError("expected 1D vector for x")
554     if x.size == 0:
--> 555         raise TypeError("expected non-empty vector for x")
556     if y.ndim < 1 or y.ndim > 2:
557         raise TypeError("expected 1D or 2D array for y")

TypeError: expected non-empty vector for x
```

Error from "Harder Challenge" video:

```
<ipython-input-226-1029e05906b9> in fit_polynomial(binary_warped)
6
7     # Fit a second order polynomial to each using `np.polyfit`
----> 8     left_fit = np.polyfit(lefty, leftx, 2)
9     right_fit = np.polyfit(righty, rightx, 2)
10
~/anaconda3/lib/python3.6/site-packages/numpy/lib/polynomial.py in polyfit(x, y,
 deg, rcond, full, w, cov)
553         raise TypeError("expected 1D vector for x")
554     if x.size == 0:
--> 555         raise TypeError("expected non-empty vector for x"
)
556     if y.ndim < 1 or y.ndim > 2:
557         raise TypeError("expected 1D or 2D array for y")
TypeError: expected non-empty vector for x
```

Discussion

Here we will discuss some considerations of problems/issues faced, what could be improved about the algorithm/pipeline. Some of the problems/issues I faced in the implementation of this project were the following:

- 1) I have some issues getting the right type of format to save. I wasn't able to correctly output the proper binary files from colour and gradient processing my images. The images kept being outputed as black images with 'yellow' lines. This failed the next step as that function was expecting full binary 'black & white' images. I think it was resulted when I used a mixture of reading and writing files with cv2 and mpimg functions. Not sure why exact this works but using the same type of function on both the write and read ends doesn't seem to work all the time in my code.

- 2) Another issue I ran into is while the left lane was solid and seem to be give consistent results, the right 'dashed' line seems to throw my lane finding abilities out the window at times and produce what I call the 'fluttering sheet' affect where my lane tracking desparately attempted to track multiple misleading lines. I feel this could be remedied with some more time to examine my three lane line finding attempts with a reviewer to see what minor error I am making.
    - UPDATE: Was able to correct this effect with sanity check and to make my polynomials match to which ever one was more dominant w.r.t. the array that had the most ‘active’ pixels in it. Ie. More data points it most be a solid or shoulder lane line.

- 3) Another item that seem to be giving me errors is measuring to radius of curvature of my lanes. While the correct measurement should be in the order of 1000 meters my figures are giving a measurement of 10x to 100x times more which is clearly linked to the errors of lane finding and polynomial fitting calculations.
    - UPDATE: I was able to correct this error slightly with obtaining stronger curving polynomials. However, my radius now ranges from 10 m to 3000 m, which I assume is the ‘acceptable’ error given our limited tools to measure this accurately.

- One thing I noticed was with very small incorrect measurements in the polynomials calculation produced some extreme errors in my radius of curvature measurement. This would resulted at one point my radius was measured at 5000 km instead of a curvature in the neighbor of 1 km.

Some of the hypothetical cases that would cause the pipeline to fail are such scenarios: a) Dark underpasses, heavy shadowed roads, steep curved roads

This was obvious in the fact that my pipeline failed to run on the 'harder' videos indicating these frames contained some curves that my polynomials couldn't handle.

Clearly this pipeline is not meant for real-time application or home hardware, since to run my basic pipeline on 7 test images took a little over 6 seconds:

```
CPU times: user 5.65 s, sys: 441 ms, total: 6.09 s Wall time: 4.43 s
```

Which is wholly unacceptable if you consider on second of realtime SD video is 24 frames this work out to be ~ 21 seconds to process 1 second of video. Obviously we would need to use precompiled language that can be 10-100x improvements like C or C++.

Some things I could do if time, and skill set permitted to make it more robust would be to :

- To improve the results to obtain the best distinct road lane lines we would need to explore all (or most) of the possible permutations of the settings that go into the four colour and gradient transformations. This exploration could be possible to run a machine-learning/deep-learning algorithm to pick out the best and improve the results on every iteration.
- Another improvement that I didn't implement due to time was implement if the lane line pixels were within the standard US Highway regulation for distance between road lines.
- I would have reviewed most of the variables that I could store in a class but skill to implement was limited and I was already over due on completing this project.

END OF REPORT