Due date 19 April 2018

Table Of Contents

Project Overview:

In this project the objective is to find and mark lane lines on a road in a series of test still images , then two further test videos and then one further Challenge video.

The program functions by taking a single image or extracting a single image from a video and passing these images through a series of steps called a ***"pipeline".*** This pipeline does various steps such as converting to grayscale and using mathematical techniques such a Hough line detection to create composite images that consist of the original image or video with new lines overlaid to mark the detect road lanes. Ultimately these detected lines are extrapolated to generate a clear and unbroken line marking the left and right lanes.

The code used to generate the results are provided in Jupyter Notebook Format and the code runs in the same environment to allow results to be verified by assessors.

The write up here is to explain and elaborate on the many steps taken , issues that arose, solutions and partial solutions that were implemented and to present a summary of the results and steps that were taken getting to the point where results could be generated.

The requirement to process test still images and the two videos were met and most of the relevant work was done in a function called *draw_lines.*

The challenge videos were not processed completely and many problems were encountered with this. Some partial solutions were implemented. The collection of partial solutions and attempts at solutions were put aside in one section called draft_draw_lines and this function is not called by any other and does not effect the rest of the program. It is there to show the work done in attempting to resolve various issues the are explained in this report.

The positive results are clearly shown in the Jupyter Notebook results and illustrated by images and screenshots of videos in this report. This serves two purposes, to illustrate the work done and to facilitate the quick review of the work if an assessor wishes to only verify that the work was carried out and results were achieved.

Describe Pipeline:

The pipeline is similar to the lesson and very similar to the youtube tutorial as follows

In the function process_image the following steps are done
1 The colour image is converted to a grayscale image
2 A Gaussian blur is applied to the grayscale image
3 Canny edge detection is applied to the image
4 The dimension (shape) are extracted from the image
5 A region of interest is applied to the image
6 Hough line finding algorithm is applied to the Region of Interest in a function called hough lines
7 Lines returned from the Hough function are overlaid on the original image

Hough_lines is an important function and in turn calls *draw lines. Draw_lines* is where most of the additions and modifications happen because it is at the point of writing the lines that we wish to decide are the lines left or right lane and to create an average extrapolated line based on the other lines.

The above is done by following the lesson and the youtube Questions and Answers video and that gets you to the point where you have red lines making each white road marking separately. Also the video suggested using the slope to determine if a line was the left lane or right lane and suggested some formula to use, which were in fact used in the project as submitted.

There follows a series of sections outlining various attempt to illustrate how I worked through the problems and some of the solutions and attempted solutions.

Attempt one, "Joining Endpoints" :

The first idea attempted was to try to join each line return from hough_lines together. I imagined each red line from the Hough process to be joined to the next line further up the screen. Similar to connecting each road marking end point to the next road marking. I thought this would give some element of curve in the line, perhaps 3 lines with 3 connecting lines potentially at different slopes and this would help with bends in the road.

This attempt is in function *draft_draw_lines* and is marked
ATTEMPT 1: JOINING ENDPOINTS
OUTPUTS in ZIP File: joining_endpoints_image and joining_endpoints_video

Comments:
Well this is so bad its funny but there is a few things I learned here worth noting..... Firstly this picture is obviously in California and there may well be an alien invasion with UFO's in the sky shooting down laser beams and that's a scenario that perhaps we should not discount immediately but ….. I went with the idea that my lines were going all over the place.

The good news is that I did manage to extract many end points and you can see that many lines connect to the end point of the road markings. So that was a partial success. Obviously the end points are connected to the top of the image because I made some error in the algorithm somewhere which may have been easy to fix.

This approach was abandoned for a good reason that is explained by the two lines connecting to the image origin in the top left. This happened because the algorithm was a loop and in the first iteration of the loop used default values of (0,0) so the first left and first right lines connect to the top left. At the time of writing the code I was happy to come back to that but it actually turned out to teach me something useful. The point connecting to the top left was the first line in the left or right line arrays. This was important because as you can see the first lines connected are not at the top of the image or top of the region of interest.

Once I understood the above, I realised that the lines found were not sorted going from top to bottom of the image and this idea of connecting endpoints required the first line in the array to be

the first line toward the top of the region of interest and the next line the next road marking further down the image etc.

I considered a sorting algorithm to sort the points from top to bottom of the image so I could connect endpoints more easily. I decided to try something more straight forward instead.

Attempt 2 , Connecting Highest Point to Lowest Point Left and Right Side:

This attempt is marked in *draft_draw_lines* as follows:
ATTEMPT: CONNECTING HIGHEST POINT TO LOWEST POINT LEFT AND RIGHT SIDE
OUTPUT IN ZIP: Highest_lowest_point [video], Highest_point_lowest_point_image , first_ROI_low_high [video] , highest_point_lowest_point_challenge_image

This attempt was the simplest thing I tried and gave something that worked to a basic level. The idea was simple. I created a loop with some variables to store the highest and lowest values in the y-axis for the left and right side. As the loop iterated if eventually found the highest and lowest points on each side. These points were connected with lines. I use blue and green for the overlays so I can see what is happening better. Sample image below.

Comments :
This is not bad. On the left side with a continuous line there is a blue line overlaying it from top to bottom. That's OK. On the right side the line goes from the highest to the lowest but in this case the lowest point is half way up the screen leaving a big gap at the near end of the lane.

I could have improved this by calculating the slope of the line and extending the line to edge of the image. Looking ahead to the Challenge video with a curve in the road I realised this approach was too simplistic see image below. The image shows the region of interest is high and too the left so the first line is in the trees and the lowest point is halfway up the road. Even fixing the ROI this approach would not make good allowance for the curve in the road. So I decided to try another approach. The video is actually kinda funny to watch and the error in the ROI is explained later but basically I am accidentally using the dimension from the first test image in the notebook and not the dimensions of the video. The image below is from the video "*first_ROI_high_low"*

Attempt 3, as per youtube video, averaging slopes , centre points and extrapolating lines:

This attempt is marked in *draft_draw_lines* as follows:
ATTEMPT: FOLLOWING YOUTUBE VIDEO, AVERAGE CENTER POINT AND AVERAGE SLOPE
OUTPUT: Challenge , Challenge_ROI_marked, Challenge_One_Second

Why didn't I do this in the first place?..... Because I didn't quite understand what the instructor was talking about so I tried to simplify it. Also I thought the connecting endpoints may allow for some degree of curve in the road. It took a while for me to understand this more sophisticated approach. From the point of view of learning I needed to work up to this point and that is a reflection of my abilities in Python at that moment in time. For example lines like the following I would not have easily come up with myself as I am not familiar enough with *numpy* library.

r_center=np.divide(np.sum(rc,axis=0),len(rc))

At this point it is important to note that there were some small problems getting the pipeline to work on the still image and the first two videos.

There follow an exploration of the initial development of the pipeline and then the issues and attempts at resolution for the challenge

Applying the Pipeline to the still test images and first 2 videos *solidWhiteRight* and *solid_Yellow_left*.

Recap of existing pipeline a time of tests:
At the time of the successful tests and as the notebook files are presented for submission the pipeline consisted of

- An image is read in from a file , as an image or part of a video
- The image or video single frame is converted to a grayscale image
- A Gaussian Blur is applied
- Canny edge detection is applied
- The image dimensions are extracted
- A Region of Interest is applied
- Hough transformation is applied to the image to generate lines
- Hough transform function calls a draw_lines
- Draw_lines adds lines to the images and is where the lanes are marked

The function draw lines is where the bulk of the work was done and where most of the problems were in the challenge section.

A second similar function exists and is not used called draft_draw_lines and this is where the numerous attempts were made to address the issues separately while keeping one functiong draw_Lines function for the report.

Draw_lines in detail:

The function Draw_lines contains the code to draw the lines found in the Hough transform and also draw the extrapolated lines.

As per the Q+A youtube video a for loop steps through tan array called lines and calculates slopes and centre points and adds them to arrays one set for left and right. This is more or less straight from the video the left and right decision was suggested not demonstrated so a little of my own work as snippet is as follows

```
rm = [] # should I put in intialisation?
lm = []
rc = []
lc = []
# steps through lines and gets a slope and center for each
# slope greater than one left side less than one right side
for line in lines:
    for x1,y1,x2,y2 in line:
        color=[255, 0, 0]
        cv2.line(img, (x1, y1), (x2, y2), color, thickness)
        slope  = (y2-y1) / (x2-x1)#had this wrong oops slope  = (x2-x1) / (y2-y1)
#iterate between each point
        center = [ (x2+x1)/ 2 , (y2+y1) / 2 ] #calculate slopes and centers
        if slope > 0.5 and slope < 4 : # and slope < 5   :
            lm.append(slope)
```

Of note in this section is the formula for a slope,,

```
slope  = (y2-y1) / (x2-x1)
```

I literally had the X's and Y's mixed up. This caused a problem illustrated below.

These lines are directly from the video and I would not have come up with *np.sum(rc,axis=0)* on my own at least not in time for the deadline. These lines calculated the average slope and average centre. I probably would have used a 2 dimensional array for the point and averaged them with a for loop rather than the np.sum function .

```
r_slope=np.sum(rm)/len(rm)
l_slope=np.sum(lm)/len(lm)  # note to self....how do I throw out exptreme value ??

r_center=np.divide(np.sum(rc,axis=0),len(rc))
l_center=np.divide(np.sum(lc,axis=0),len(lc))
```

The next section calculates the top and bottom points for the extrapolated lines using the equation of a line. This was describe in the video and I wrote the code myself

```
yMiddle = int( np.size(img,0)*0.6)  # 0.5 was too high
yBottom = int(np.size(img,0))
xMiddle = int ( ( r_slope * r_center[0] - r_center[1] + yMiddle )/ r_slope )
xBottom = int (  ( r_slope * r_center[0] - r_center[1] + yBottom )/ r_slope    )

color=[0,255, 0]
thickness=10
cv2.line(img, (xMiddle, yMiddle), (xBottom, yBottom), color, thickness)
```

So all the above worked with some debugging and tweaking. The formula for the slope was initially wrong and effect the overall line slope which took a while for me to figure out and the Region of Interest was adjusted to give better results. See the series of images below.

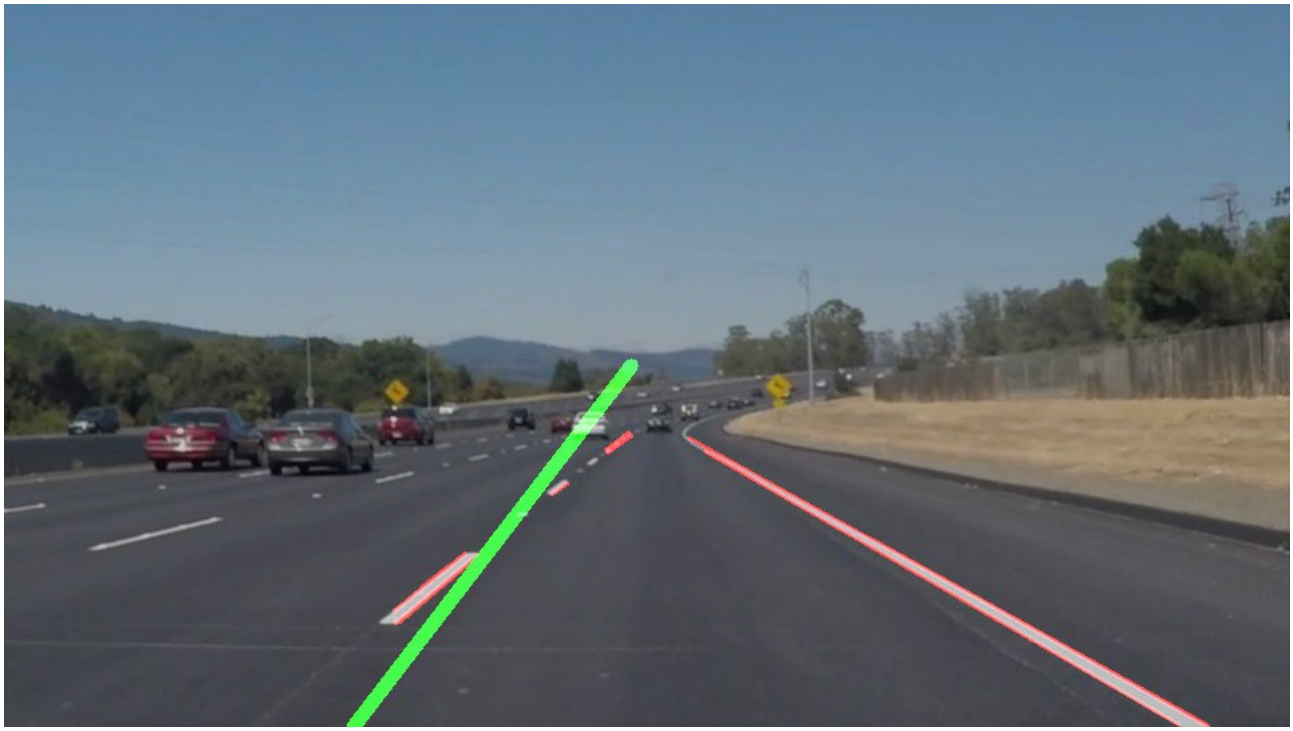


Image File: "left_line_too_long_and_wrong_slope"

The above image shows the results of the pipeline after above code set up but with some problems, only the left side is shown as I was trying to get one working then two. Thisimage shows the extralated line is extrending too high and at the wrong angle. I adjusted the region of interest as shown below.

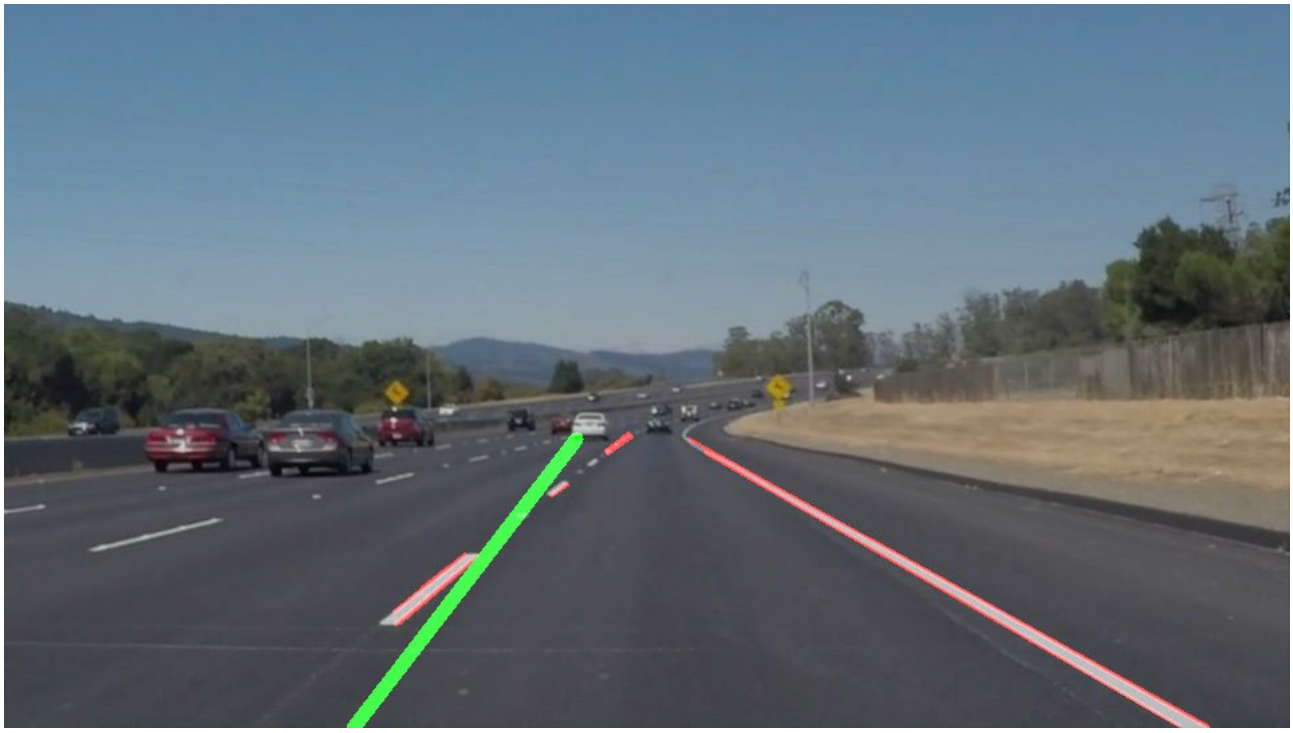


Image File: "left_side_right_size_wrong_angle"

The above image shows the region of interest has been adjusted so now the line is about the right length but the angle is still wrong.

Image File: "left_line_OK"

This image shows the left line drawn correctly after the mistake in the slope formula was fixed. I had the X's and Y's mixed up and once that was fixed the line was drawn well.

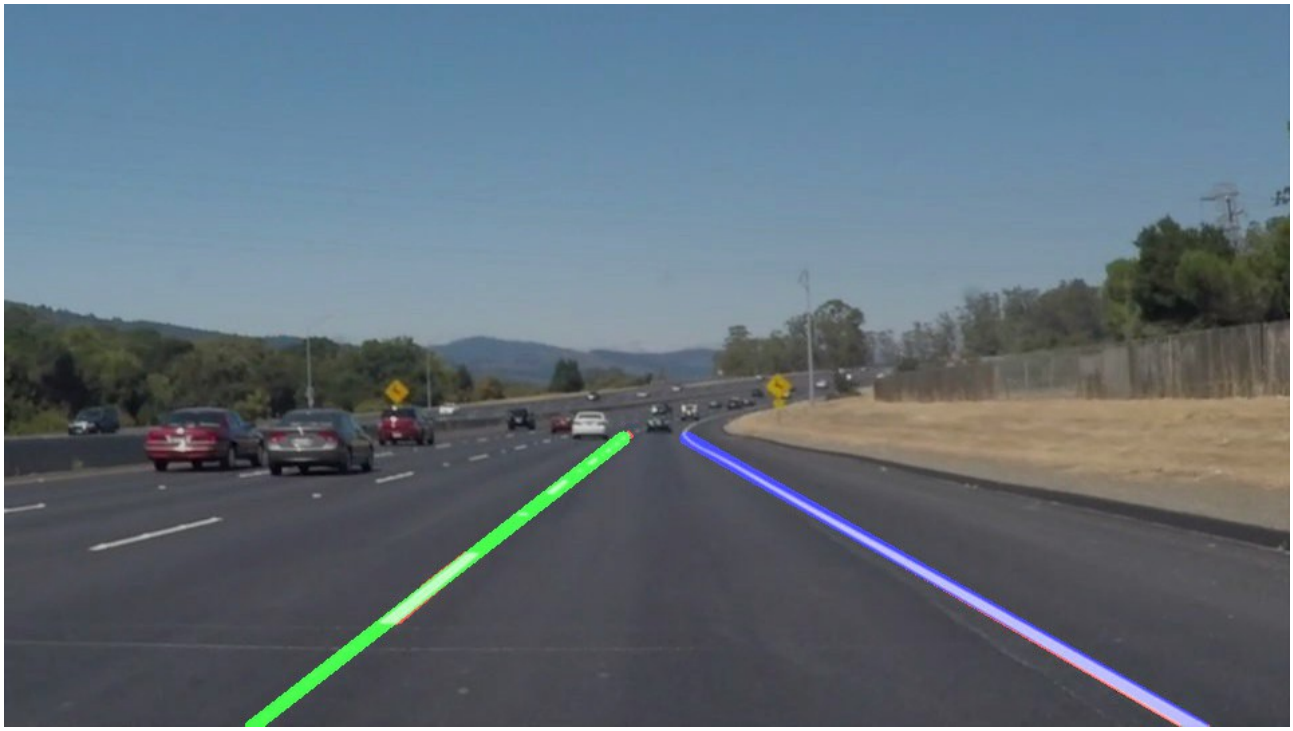


Image Files: "both_lines_OK"

Once the left side was done I copied and adjust the code for the right side and that worked fine. I was also happy to keep the different colours as you can make out areas in red where the extrapolate line is not covering a line produced by the Hough function.

Applying the pipeline to videos:

The pipeline worked well on the videos solidYellowLeft.mp4 and solidWhiteRight.mp4 as shown below.

Video solidYellowLeft.mp4

Video solidWhiteRight.mp4

The Challenge, the problems begin....

Upon proceeding to the challenge the pipeline starts to generate many errors and crashes. This attempt implements the ideas of separating left from right lines by positive or negative slopes, averaging centre points and averaging slopes to create a formula for a line. Then drawing a line that will go from the top of the region of interest to the edge of the image based on the formula of the line for each side left and right sides.

This attempt also contains many comments and commented out sections. These were numerous experiments to try to deal with various problems that I will outline below. The final version of *draw_lines* is based on this draft version with all the comments and extra sections of code removed to make something that works and is more readable.

This attempt contains many work arounds in relation to many errors I was getting.

Basically once I moved the region of interest I started to get lots or errors around data types . There is a separate section on the Region of Interests below as this became important in the attempt to reduce problems.

Some issues and workarounds/solutions include

Errors Float to Ints:
I got some problems with errors saying *floats* were given when *ints* were needed so I recast them for example
*yMiddleRight = **int**(np.size(img,0)*0.6)*

This was done many times and seemed to fix many errors but these errors only arose with the challenge video

Errors such as *Nan* "not a number":
It seemed likely that some outlying values were causing some kind of crash or out of bounds errors. This was hard to understand because it didn't happen until I moved the region of interest and even the first region of interest which had many very spurious values with lines going everywhere is no throw these errors.

This line:

*xMiddleRight = (l_slope * l_center[0] - l_center[1] + yMiddleRight)/ l_slope*

was causing the trouble so I looked up what a *Nan* was and then checked for it and substituted a default value for it if the number was a *Nan.* I used lines like this in 213 to 234 which use the *Python math.isnan()* function

```
if math.isnan(xMiddleRight): #math.isnan(x)
    xMiddleRight = 100
```

```
else :
   xMiddleRight = int(xMiddleRight)
```

This seemed to partially work as the error stopped but gave a different errors somewhere else and this is the start of me really moving the problem around as opposed to fixing it.

Errors about scalar values:

Currently the program will crash at the challenge with this error generating only a short video

I got many errors that I could not resolve around scalar to index. I looked it up and it is to do with index's in arrays, but I couldn't see the problem and I got a suggestion to check for no content in the array from the slack group. I tried a few things to address this. Such as the following.

Default values:
I tried to but in some default values to initialise the arrays so they would not contain nothing
lines 142 -145 had the declarations I added a default value of 0,0 to them but it did not fix the problem

Range Clipping:
I tried to check if the values in the arrays were below zero to clip them to zero and above an arbitrary value to clip them at a high point. Lines 175-183

```
if r_center[0] < 0 : r_center[0] = 0
   if r_center[0] < 700 : r_center[0] = 700
   if r_center[1] < 0 : r_center[1] = 0
   if r_center[1] > 700 : r_center[1] = 700

   if l_center[0] < 0 : l_center[0] = 0  # had r here woops
   if l_center[0] < 700 : l_center[0] = 700
   if l_center[1] < 0 : l_center[1] = 0
   if l_center[1] > 700 : l_center[1] = 700
```

This also did not get rid of the error, I really thought that would solve the problem

Try, Except:
I thought I could skip the troublesome values using a try statement , so I would try an equation and if it didn't work to skip that value. This partially worked it seemed to generate a file quickly skipping from 0% to 100% in a few steps but the file was less than one second of video. It seems with all the skipping a functioning MP4 file was not created.

At that point I was running out of ideas I hoped to get something useful at that point. But if you quickly repeated open the file it plays for a split second and not even one line was drawn so that approach didn't work at all. The Region of Interest is drawn into this video as I was experimenting with that at the time. It might actually has caused a problem but I didn't consider that at the time. This was the try statement. Line 212

```
    try:
   xMiddleRight = ( l_slope * l_center[0] - l_center[1] + yMiddleRight )/ l_slope
    except:
       xMiddleRight = 100
```

Print Statements:
I frequently used print statements to check what was going on. Lines point, slope values, almost everything at some point I printed out to check my understanding usually with a note to myself as to why I was doing it, for example.

```
 print("Second time suspected rounding?? r_slope " , r_slope , " l_slope " , l_slope )
```

Finally for draft_draw_lines:
I made a copy of the function and renamed the copy ***draw_lines*** and called the original ***daft_draw_lines*** I took out everything that was not working from the function and took out lots of the comments. The result is a function draw_lines that works well on the first 2 videos but only works for split seconds of subclips in the challenge. What happens is when a spurious value is found it causes a crash and the longest video create with valid lines lasting only a couple of seconds.

The function *draft_drawlines* can be run but it is never called so it does not interfere with the rest of the problem and I include it so you can see what I was trying.

Function "process_image_challenge":

The pipeline for the challenge is different to the other videos. I was having problems with the Region of Interest and wanted to experiment without breaking the other sections so I created a new pipeline that is called "process_image_challenge" and is very similar to process_image.

The main difference is there is a section where I extract the define the region of interest In lines 34 to 41 I define the vertices eg

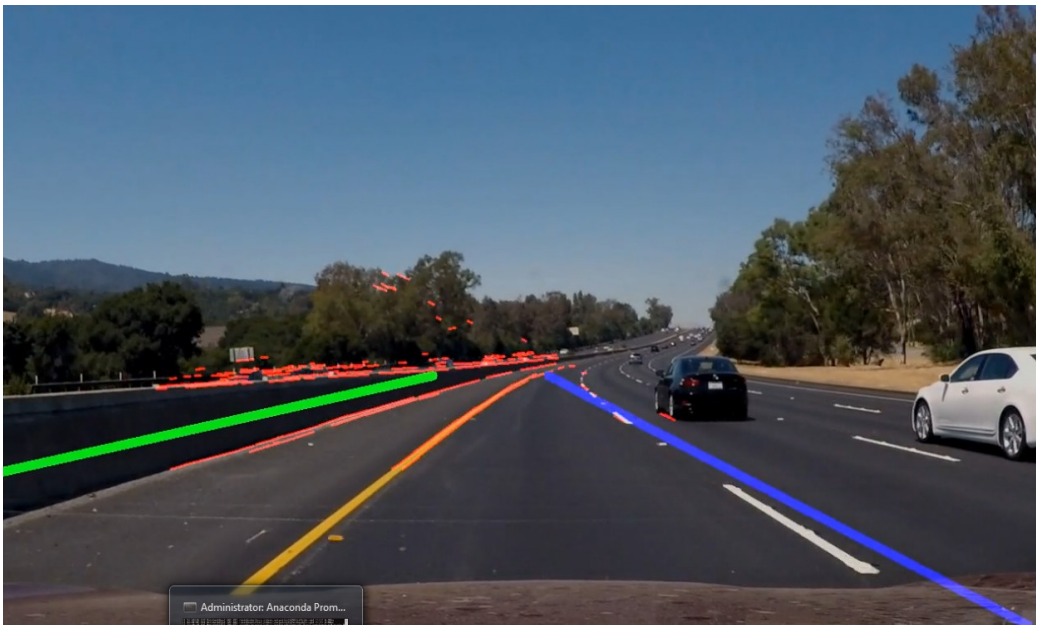
*topLeftX = int(imshape[1] * 0.45)*

and then drawlines 51 to 55

image = cv2.line(image, (topLeftX,topLeftY) , (bottomLeftX,bottomLeftY),color,thickness)

It is important to note that when the lines were drawn into the image I commented out the section that continued to process the images. That is to say I drew in the Region of Interest and returned. I did not continue with processing the image. This was because the process kept crashing and I hoped that by perfecting the region of interest I would prevent the crashes.

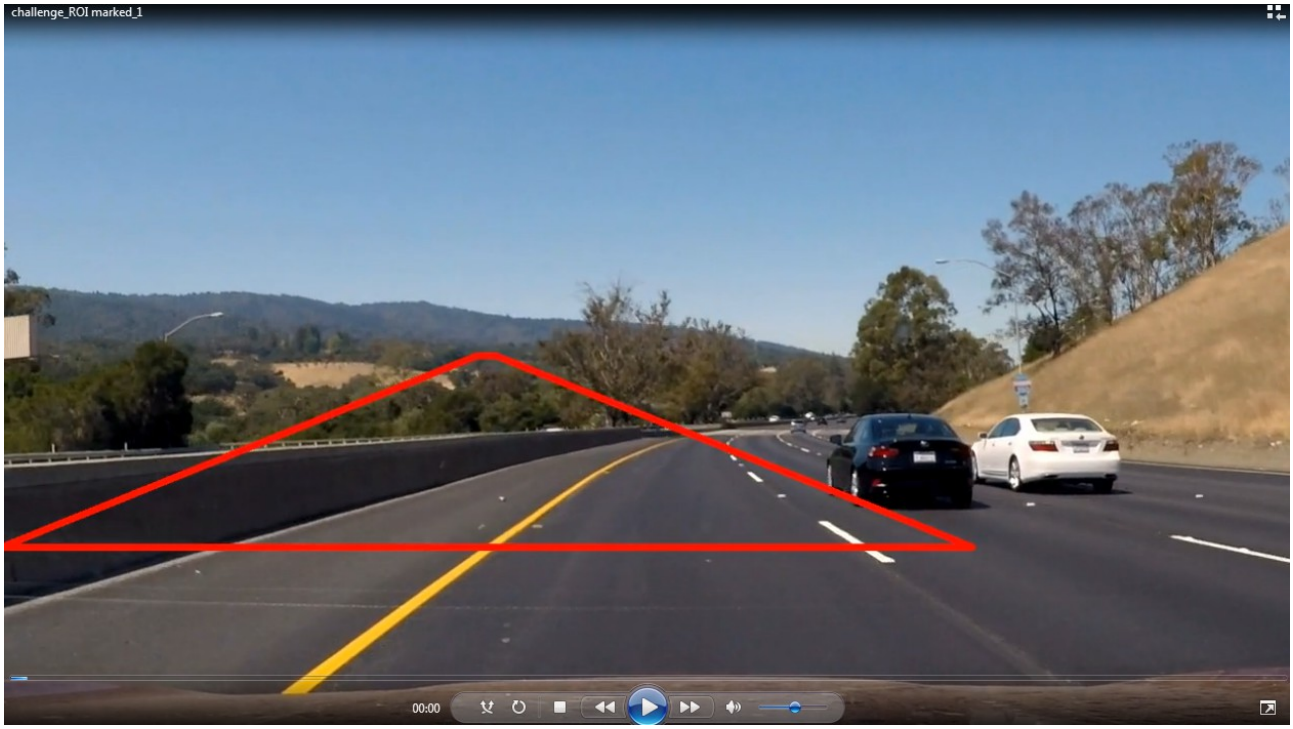
The initial result without any changes to the original region of interest and using the full pipeline was the video *challenge_with_original_ROI*. This ROI was actually incorrect because as mentioned earlier was using the ROI from the first set of still images in the project not the video clip. The errors was to use the word *"img"* (which was for the still image) when I should have used *"image"* which was for the video. See below. You can clearly see the ROI is up and to the left and the lines on the road wall (edge of road) are dragging the extrapolated line to the left by the averaging effect. From this it looks like if I can get the ROI correct I should have something that works reasonably. At his point I did not realise the error causing the ROI to be off so I focused on the ROI and the best way for me to do that was to tidy the code as best I could and draw the ROI onto the video so I could see it and work from there.



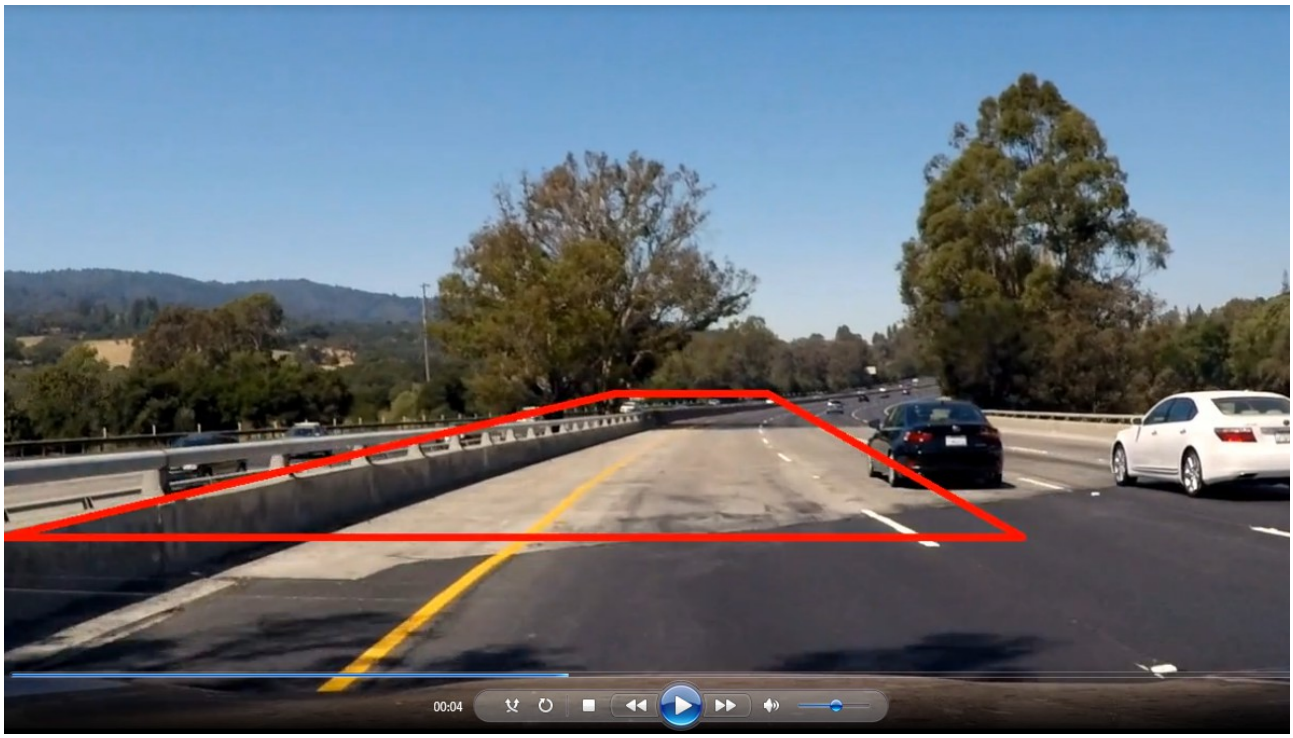
Video Screenshot *challenge_with_original_ROI*

Tweaking the ROI:

When I first drew in the ROI it looked like this.



When I adjusted it I got images like this:



Obviously this is not correct and it was proving difficult as I had not found the error yet but then I found the problem of *"img"* instead of *"image"* and I was able to line things up well

Finally I fixed the bug and got the correct ROI. Making it clear was very important here the code was clarified and the ROI draw to the video. This reduced the guess work and allowed me to solve the problem.

The following lines technically were not needed but they made everything clearer to read and that really helped.

From line 34 -41 in *process_image_challenge*

```
topLeftX     = int(imshape[1] * 0.45)
topLeftY     = int(imshape[0] * 0.60)
topRightX    = int(imshape[1] * 0.55)
topRightY    = int(imshape[0] * 0.60)
bottomLeftX  = int(imshape[1] * 0.10)
bottomLeftY  = int(imshape[0] * 0.90) #not sure this is what I am seeing .....img v's image!!
bottomRightX = int(imshape[1] * 0.90)
bottomRightY = int(imshape[0] * 0.90)
```

Also noteworthy is I have deliberately excluded from the ROI the bonnet (hood) of the car (automobile) …. I am writing this in Europe :)

I did at one point have the bonnet of the car included and got the following result from *"challenge_right_line_way_off"* also this video is only a couple of second long before the process crashed.

Video screenshot *"challenge_right_line_way_off"*

I expected the excluding of the bonnet of the car to eliminate the spurious errors that created the crashes. I commented out the code that created the ROI lines and with a good ROI reinstated the original pipeline. The result was many crashes (code not cars :)) . These cashes gave errors around values and types and lead to the many experiments to address these issues that eventually created the function *draft_draw_lines* explained above. *Draft_draw_lines* had lots of code including some that did not work so I edited it all down into something more manageable called *draw_lines* which is the one used in the pipeline in the submitted notebook for the project. The pipeline as it stands for the challenge video can produce clips of only split seconds the longest being about a second a screenshot below. I used the subclip function to make short clips hoping to find video sequences that produced less problems and indeed some clips are longer than others but the longest is still a only couple of seconds. Note the lines are extrapolated to the edge of the screen but the ROI stops just above the bonnet of the car as shown above.

The best screenshot is below and shows for a split second the lanes are marked well, the video is names "*Challenge_Best_Clip"*

Video Screenshot "*Challenge_Best_Clip*"

Identify short comings:

It is evident that the pipeline that worked well on the first two videos cannot handle the challenge video.

The fact that the pipeline for the challenge video can generate snippets of split second indicates that some of the video is fine but there are many sections that generate values that cannot be handled by the pipeline

Error messages after crashes indicate issues with *scalar* and *index* values suggesting that the values are perhaps null values or outside ranges or or incorrect data type.

Various attempt to resolve these issues such as range limiting and *"try"* statements may have been in principal the correct way to deal with these problems but they were not implemented sufficiently well to function correctly all the time.

Possible Improvements:

The idea of *"try"* statements seems likely to work. I expect more of the function *draw_lines* will be needed to be included in the *"try "* statement and better defaults or responses to failures of the try statement are needed.

Limiting ranges of values for slope and line point and line centres may prove useful and were attempted with limited success. For example limiting the slope to greater than zero work for separating left from right slopes but limiting to between 0.5 and 5 did not work causing more errors. How best to limit slopes is unknown at this stage but an obvious place to improve

Similarly limiting the ranges of line centres and points should be useful and did work to some extent but either caused errors elsewhere or did not limit the values as expected

In summary a robust errors handling, *exception handling* and range limiting using the above or other as yet unknown methods are needed and that will be my focus until the next project.

Question for Feedback:
Can you recommend a good source for learning about error handling, exception handling and range limiting for the Python language please? This is beyond the scope of the Python learning material I have seen so far. Thanks

Could you give an approximate figure for how many students get the challenge section working correctly? Thanks