Finding Lane Lines on the Road

Writeup Template

You can use this file as a template for your writeup if you want to submit it as a markdown file. But feel free to use some other method and submit a pdf if you prefer.

Finding Lane Lines on the Road

The goals / steps of this project are the following:

- Make a pipeline that finds lane lines on the road
- Reflect on your work in a written report

sample gray scale image

Reflection

1. Describe your pipeline. As part of the description, explain how you modified the draw_lines() function.

My pipeline consisted of 7 stages: 1) grayscale conversion 2) gaussian blur and smoothing to reduce nosie for shaper edges 3) canny transform to edge image 4) define region of interested to reduce unrelated areas 5) hough lines to generate lines around edges 6) filtering unusable lines and finding an average to form a single line 7) overlay the thickened line over the original image

Each stages are separately coded for easier debug process during development.

First, I defined a list of parameters that I needed at the beginnig of coding the pipeline for easier managing and updating parameters as I proceeded.

```
debug       = 0
edgeLoThres = 50  # canny edge detection low threshold
edgeHiThres = 250 # canny edge detection high threshold
kernelSize  = 3   # gaussian smoothing blurring kernel size
x_size      = 960
y_size      = 540
y_cap       = 300
vertices     = np.array([[50, 540], [910, 540], [480, 300]]) # vertices for region of interest [upper left, , lower left, u
rho          = 1         # distance resolution in pixels of the Hough grid
theta        = np.pi/180 # angular resolution in radians of the Hough grid
threshold    = 30        # minimum number of votes (intersections in Hough grid cell)
min_line_len = 20        # minimum number of pixels making up a line
max_line_gap = 250       # maximum gap in pixels between connectable line segments
alpha = 1     # orignial image color pixel density
beta  = 1     # line image color pixel density
lamda = -20   # additional color pixel to the combined original and line images
```

parameter defines

I converted the images to grayscale to help Canny transform later. Next, I applied gaussian blur and smoothing algorithm to reduce the noise so we could create better edges later on. Then we used Canny transform to turn an image to its gradient to show a wide range of edges in the image. By tuning the low and high thresholds of Canny transform algorithm between 50 and 250, I found it best to filter out the noise from small objects on the road.

Knowing that the road and its lanes were what we interested, I plotted out the coordinance focusing on the road as a shape of trapezoid. I chosed 50% of y axis to be the top of the trapezoid, and 100% of y axis to be the bottom, which the left and right most of trapezoid to be 10% from left and right sides of the image. Top top of the trapezoid extends 10 pixels from 50% of x axis to both sides.

After masking the edge images with the region of interest, I applied Hough transaform to generate all the lines associated with the edges of the lanes. From the collection of lines, there were few lines that ran across the image from left bottom to right middle. These were not lines that represented the lane lines. I needed to add conditions based on their slops to filter out any lines that were near to horizontal, i.e. absolute values of slopes that are less than 0.5. This allowed me to focus on the lines that matter. I separated the left and the right trending lines by using positve (right lane) vs negative (left lane) slopes. By resolving the simple slope equation, m=(y2-y1)/(x2-x1)=(y2-y')/(x2-x'), I could easily apply the y' as constains, (y_cap as top of trapezoid and y_size as bottom of trapezoid), to find out x' value where the left and right lines intercepts the y_cap and y_size. This allowed me to extrapolute the lines to cover the lanes.

```
for line in lines:
    for x1,y1,x2,y2 in line:
        if (debug==1): print("      line coordinants:x1,y1,x2,y2", x1,y1,x2,y2)
        if (x1!=x2 and y1!=y2): # skip striaght lines and horizontal lines and semi-flat lines, process only positive a
            m=slope(x1,y1,x2,y2)
            if (abs(m) >= 0.5):
                xbot=int(round(x2-(y2-y_size)/m))
                xtop=int(round(x1-(y1-y_cap)/m))
                if (debug==1): print("                                          final line coordinants:x1,y1
                cv2.line(img, (xbot, y_size), (xtop, y_cap), color, thickness)
```

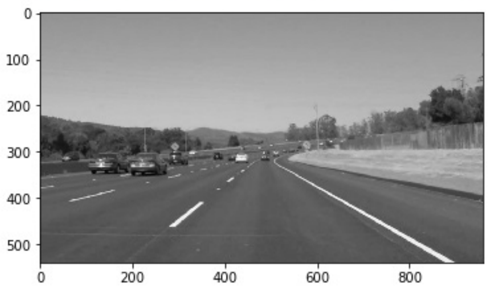
extrapolute lines function

The last pizzle was to overlay the newly drawed lines to the original image with a thickness defined.

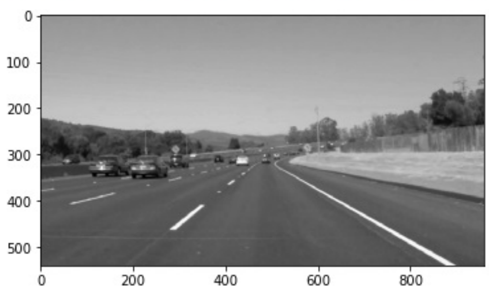
At the beginning, all provided images and videos worked fine with the pipeline above. Lines were following both left and right, yellow and white lanes. One problem was, there were multiple lines instead of a single line. Plus the pipeline only worked for the first two videos but not the challenged video. Something must be fixed to get that right. I will talk about these problems that I found in the next section.

```
print original image
This image is: <class 'numpy.ndarray'> with dimensions: (540, 960, 3)
```

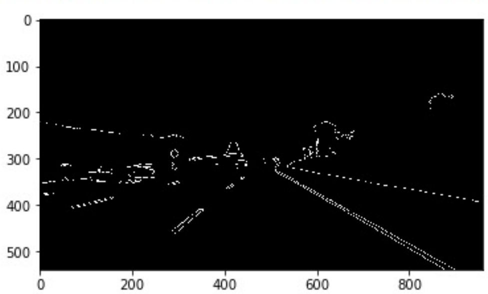
```
print gray image
This image is: <class 'numpy.ndarray'> with dimensions: (540, 960)
```



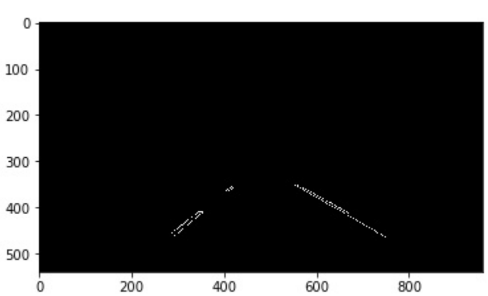
```
print gaussian smoothing image
This image is: <class 'numpy.ndarray'> with dimensions: (540, 960)
```



```
print canny edge image
This image is: <class 'numpy.ndarray'> with dimensions: (540, 960)
```

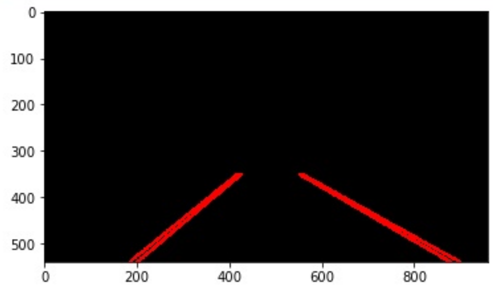


```
[[ 96 540]
 [864 540]
 [576 351]
 [384 351]]
print only region of interested edge image
This image is: <class 'numpy.ndarray'> with dimensions: (540, 960)
```



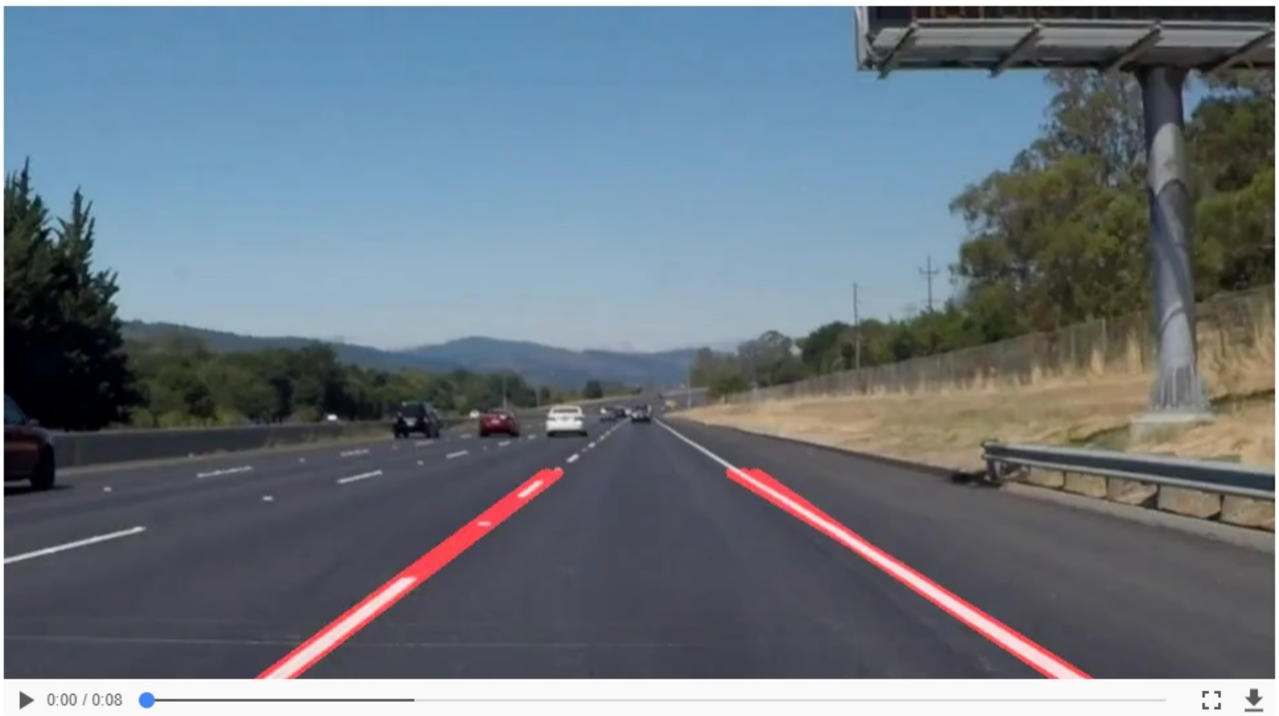
```
print only region of interested edge image
This image is: <class 'numpy.ndarray'> with dimensions: (540, 960, 3)
```

```
C:\Users\ckcheung\AppData\Local\Continuum\Miniconda3\envs\carnd\lib\s
g: elementwise == comparison failed; this will raise an error in the
```

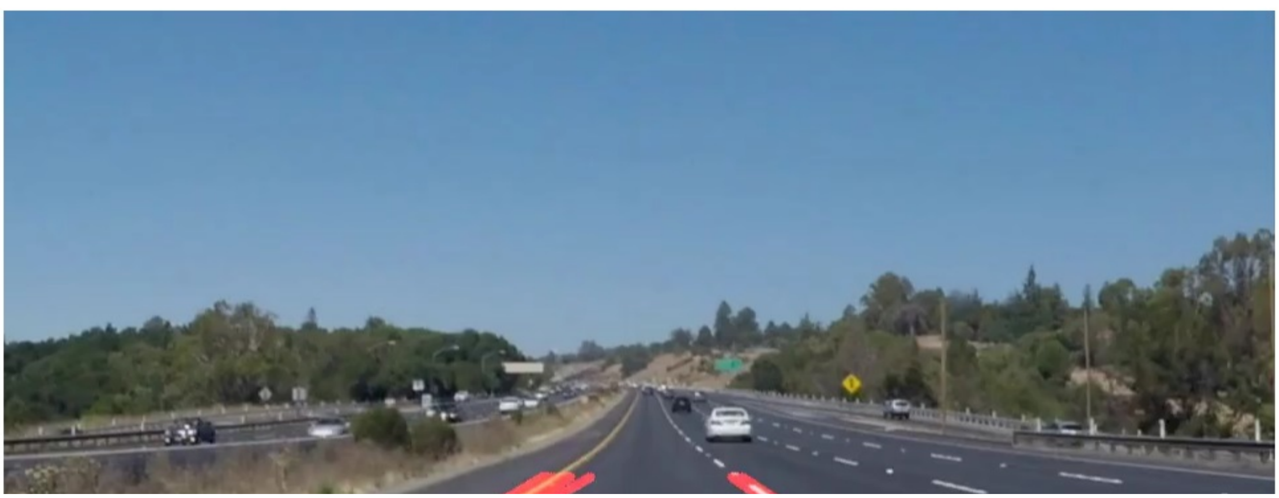


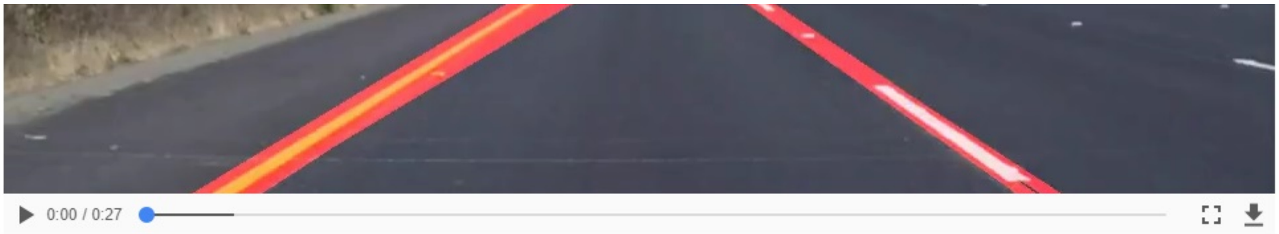

```
print only region of interested edge image
This image is: <class 'numpy.ndarray'> with dimensions: (540, 960, 3)
```

Out[18]:



Out[20]:






```
""".format(challenge_output))
```

Out[22]:

first round pipeline results

2. Identify potential shortcomings with your current pipeline

Here let me talk about the problem I had before talking about potential shortcommings.

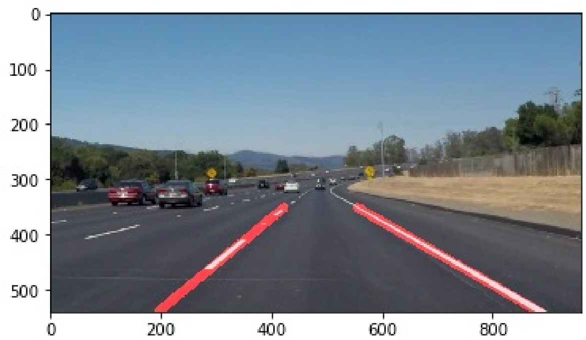
The current pipeline drawed multiples lines generated by Hough line transform instead of a single line. To do so, I needed to modify my filtering to include averaging the top and bottom coordinances of each lines on each lane. At the beginning, I thought averging the slope of the each line and their interception would do the trick, but turned out I was wrong on the math. I then realized each lines top and bottom coordinances were very close to each other, and simply averaging the each top and bottom coordinances would allow me to get a single line that was very close to represent each lane.

```
def averaging(lines):
    xbot=[]
    xtop=[]
    for line in lines:
        for x1,y1,x2,y2 in line:
            if (debug==1): print('averaging line:y2,y1,x2,x1', y2, y1, x2, x1)
            if (x1!=x2 and y1!=y2):
                m = slope(x1,y1,x2,y2)
                if (abs(m) >= 0.5):
                    xb=int(round(x2-(y2-y_size)/m))
                    xt=int(round(x1-(y1-y_cap)/m))
                    xbot.append(xb)
                    xtop.append(xt)
    if (debug==1): print('list of xbot line:', xbot)
    if (debug==1): print('list of xtop line:', xtop)
    if (xbot == []):
        print("==============================================================")
        print("xbot NaN: xbot|np.mean(xbot)", xbot, np.mean(xbot))
        print("xbot NaN: lines", lines)
        print("xtop NaN: xbot|np.mean(xtop)", xtop, np.mean(xtop))
        print("xtop NaN: lines", lines)
        print("==============================================================")
        return ([])
    else:
        avg_xbot=int(round(np.mean(xbot)))
        avg_xtop=int(round(np.mean(xtop)))
        if (debug==1): print('############   averaging xbot, y_size, xtop, y_cap: ############\n',
        return np.array([[avg_xbot, y_size, avg_xtop, y_cap]])
```

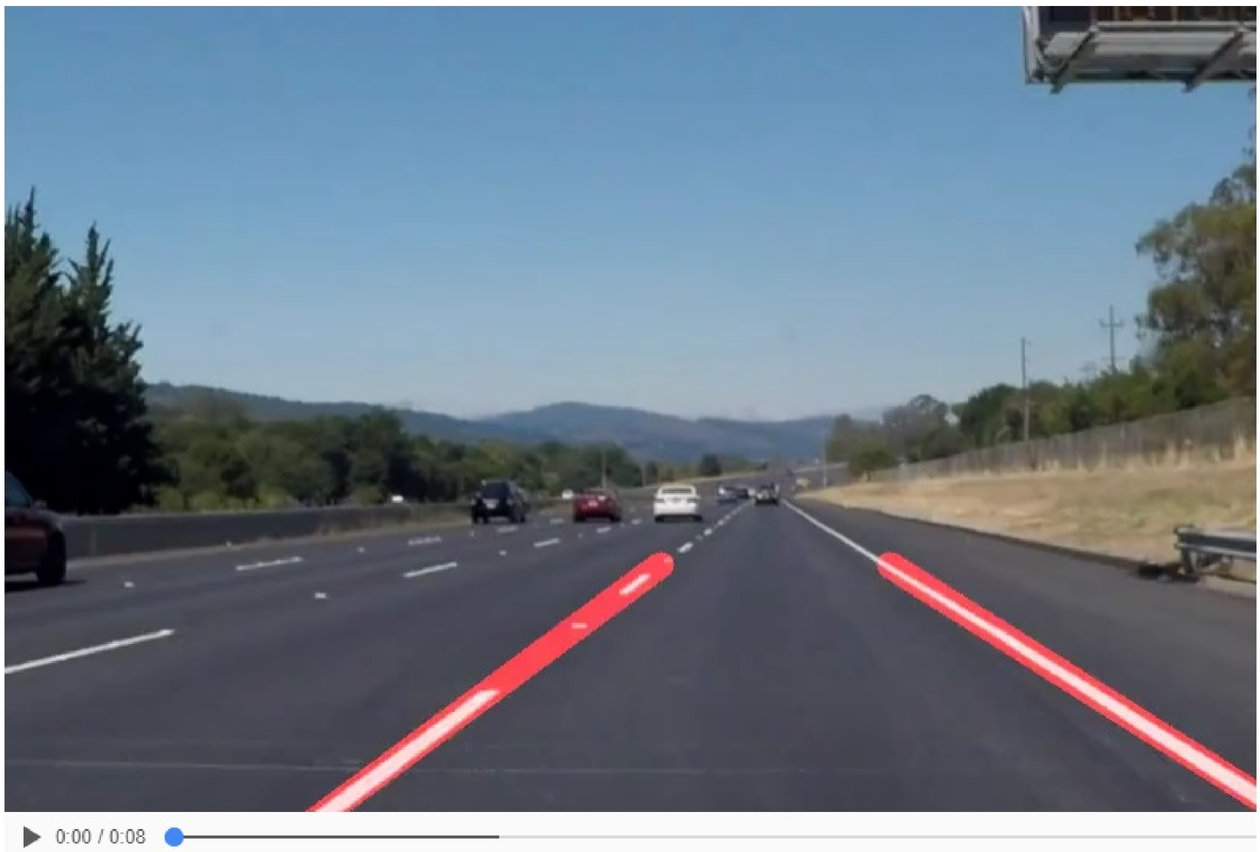
averaging function

With this minor fix, I was able to get an averaged line out of all lines generated for each lane.

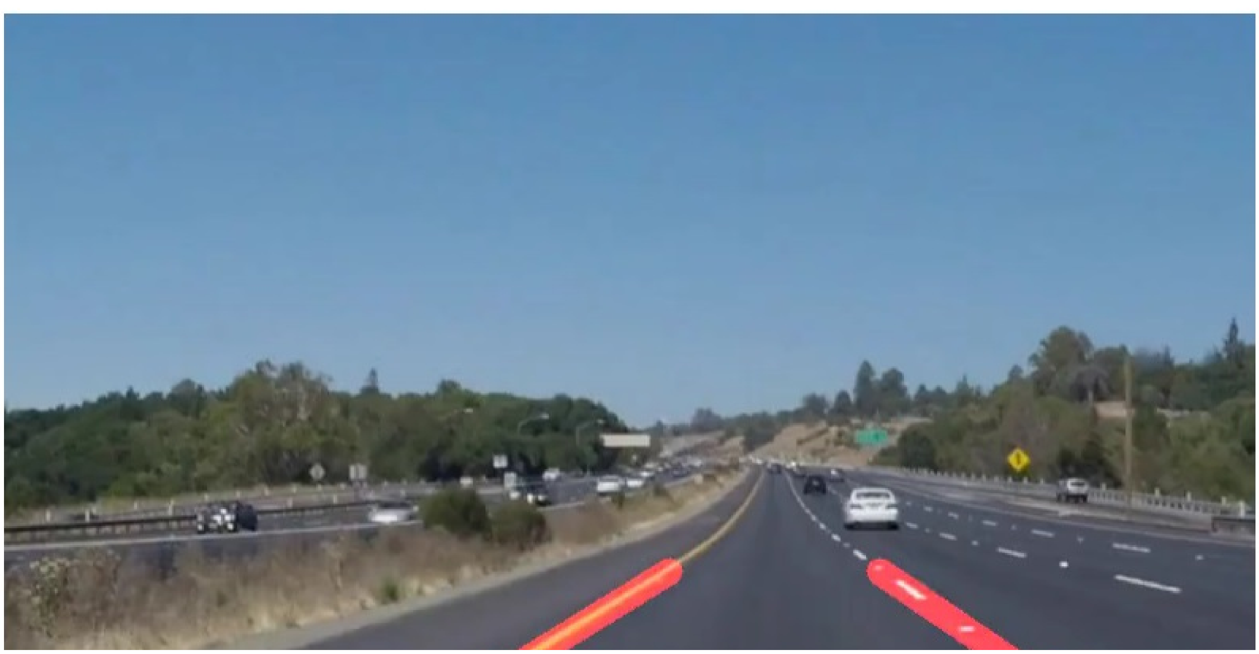
```
print only region of interested edge image
This image is: <class 'numpy.ndarray'> with dimensions: (540, 960, 3)
```

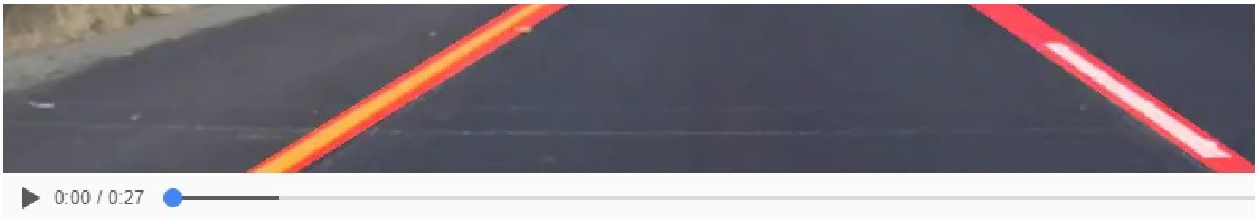


Out[18]:



Out[20]:





Out[22]:

single line results

However, the lines were still off from the bottom where the lane lines in the challenge video. As I ran the video with debug message on, I notice that the pixel of the image was larger than the images provided to test the pipeline. This is one potential shortcoming would happen or already happened when different sizes of video images are fed to the pipeline. This means my pipeline needs to be capability to support different pixel size. As a result, I updated my vertice, y_size, x_size, and corresponding parameters related to pixel size to variables based on image.shape[0] and image.shape[1]. To be able to udpate the image size in process_image(), I added globel variable declaration to make sure global varialbles were updated accordingly for functions using process_image and feeding their own images of different pixel size.

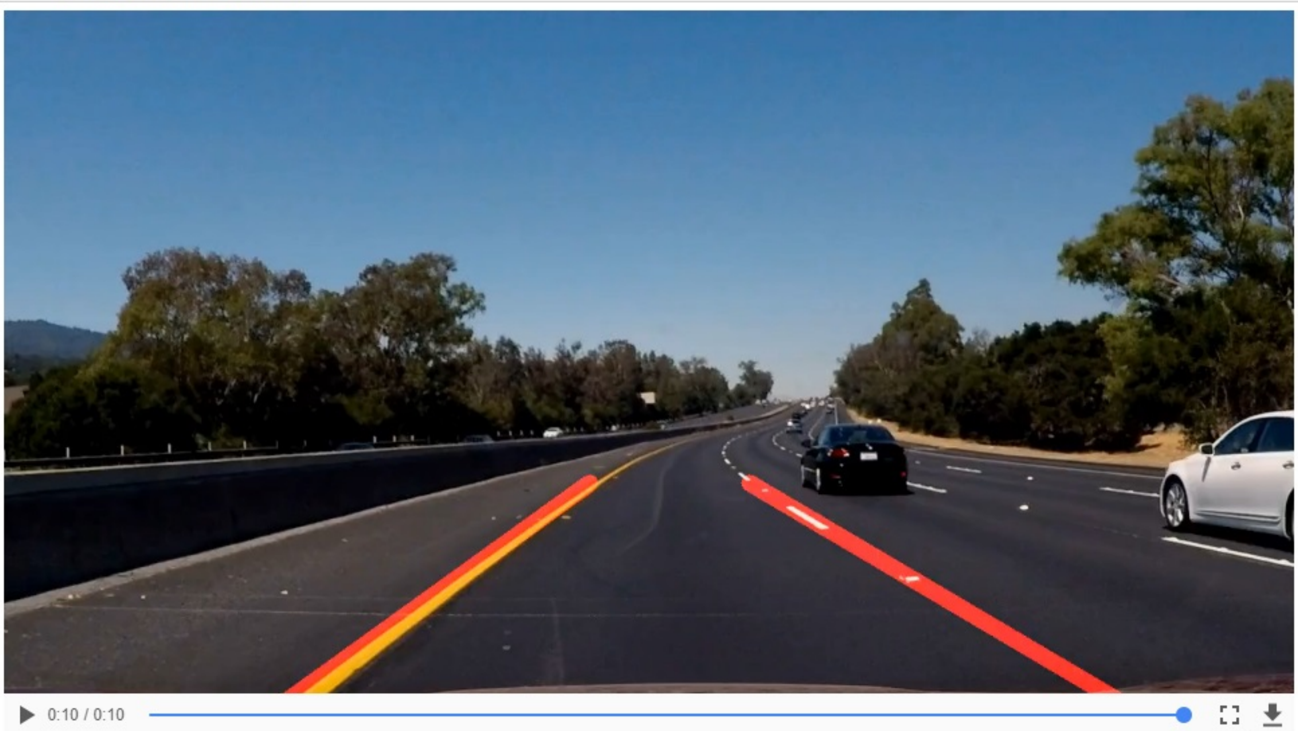
```
debug       = 0
edgeLoThres = 50  # canny edge detection low threshold
edgeHiThres = 250 # canny edge detection high threshold
kernelSize  = 5   # gaussian smoothing blurring kernel size
x_size      = image.shape[1]
y_size      = image.shape[0]
x_ctr       = int(x_size*0.5)  # taking 50% from right
x_lcap      = x_ctr-int(x_size*0.1)
x_rcap      = x_ctr+int(x_size*0.1)
y_cap       = int(y_size*0.65)  # taking 60% from bottom
vertices     = np.array([[int(x_size*0.1), y_size], [int(x_size*0.9), y_size], [x_rcap, y_cap], [x_lcap, y_cap]]) # vertices
rho          = 1          # distance resolution in pixels of the Hough grid
theta        = np.pi/180 # angular resolution in radians of the Hough grid
threshold    = 30         # minimum number of votes (intersections in Hough grid cell)
min_line_len = 25         # minimum number of pixels making up a line
max_line_gap = 250        # maximum gap in pixels between connectable line segments
alpha = 1    # orignial image color pixel density
beta  = 1    # line image color pixel density
lamda = -20  # additional color pixel to the combined original and line images
```

```
def process_image(image):
    # NOTE: The output you return should be a color image (3 channel) for processing video below
    # TODO: put your pipeline here,
    # you should return the final output (image where lines are drawn on lanes)

    # processing pipeline
    # setup parameters
    global debug
    global x_size
    global y_size
    global x_ctr
    global y_cap
    global vertices

    debug       = 0
    x_size      = image.shape[1]
    y_size      = image.shape[0]
    x_ctr       = int(x_size*0.5)  # taking 50% from right
    y_cap       = int(y_size*0.65)  # taking 60% from bottom
    vertices     = np.array([[int(x_size*0.1), y_size], [int(x_size*0.9), y_size], [x_ctr+int(x_size*0.1), y_cap], [x_ctr-int(x_s
```

update size parameters at process_image() function

```
""".format(challenge_output))
```

Out[22]:



result of challenge video

Another shortcoming was the fade left yellow lane on the white cement road in the challenge video. When the color of the lanes becoming too close to it background color Canny edge detection will not be able to identify a clear edge between lane and road.

Also, another shortcoming is the Hough lines transform may not be able to generate lanes since the the image fed to the pipeline is based on a moving frame of a video. If it happens to land on a wide gap between the next lane line could be detected, there may not be any lines gerneated and the pipeline will need to be able to skip that process instead of erroring out for null objects from the lines array.

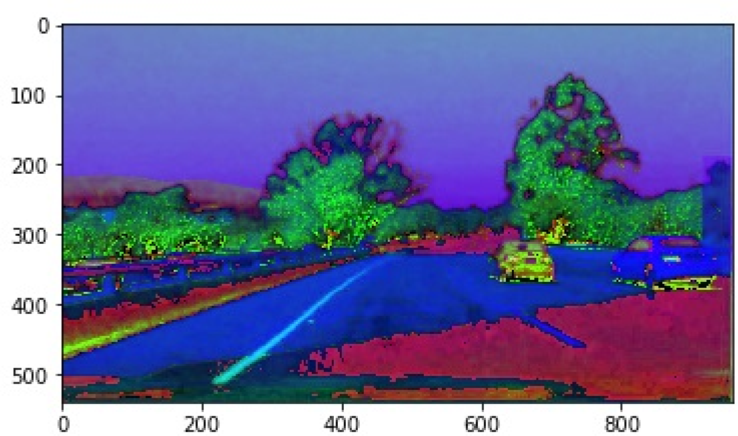
3. Suggest possible improvements to your pipeline

A possible improvement to overcome the shortcoming about the fade lane line is to used HSV scale instead of gray scale. HSV color space was able to extract the faded yellow lane from the similar cement road. With that, Canny edge detection algorithm will be able to generate the edges for the yellow lane.

```
print original image
This image is: <class 'numpy.ndarray'> with dimensions: (540, 960, 3)
```


```
# turn image to gray scale
#grayImage = grayscale(image)
hsvImage = hsvscale(image)
print('print HAV image')
plotImage(hsvImage,'gray')
```

```
print HAV image
This image is: <class 'numpy.ndarray'> with dimensions: (540, 960, 3)
```



hsv scale images

Another possible improvement to overcome the shortcoming about empyt line array is to add condition checking to make sure the pipeline will skip the empty line array and be able to continue processing the remaining video feed. (see averging image above for empty line array handling)

4. Conclusion

Overall the project is fun to work on and learning the trick to prefect the lane finding process is usefully to apply to future project. The product of the land finding pipeline still have room to improve, such as better filtering algorithm or fine turning the parameters for Canny transform and Hough Line detection.