CARND-LANELINES-P1

25/06/2018

UDACITY

PROJECT 1: FINDING LANE LINES ON THE ROAD

CREATED BY: SAMUEL ALLSOPP

SYSTEM ENGINEER
JAGUAR LAND ROVER

FINDING LANE LINES ON THE ROAD

PROJECT GOALS

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

PROJECT RUBIC

Required Files

Criteria	**Meets Specification**
Have all project files been included with the submission?	The project submission includes all required files: • Ipython notebook with code • A writeup report (either pdf or markdown)

Lane Finding Pipeline

CRITERIA	**MEETS SPECIFICATION**
Does the pipeline for line identification take road images from a video as input and return an annotated video stream as output?	The output video is an annotated version of the input video.
Has a pipeline been implemented that uses the helper functions and / or other code to roughly identify the left and right lane lines with either line segments or solid lines? (example solution included in the repository output: raw-lines-example.mp4)	In a rough sense, the left and right lane lines are accurately annotated throughout almost all of the video. Annotations can be segmented or solid lines
Have detected line segments been filtered / averaged / extrapolated to map out the full extent of the left and right lane boundaries? (example solution included in the repository: P1_example.mp4)	Visually, the left and right lane lines are accurately annotated by solid lines throughout most of the video.

Reflection

CRITERIA	**MEETS SPECIFICATION**
Has a thoughtful reflection on the project been provided in the notebook?	Reflection describes the current pipeline, identifies its potential shortcomings and suggests possible improvements. There is no minimum length. Writing in English is preferred but you may use any language.

PROJECT REFLECTIONS

LANE FINDING PIPELINE

Firstly I decided to initiate all my parameters in one place at the start of the pipeline, this allows easy access to make adjustments for optimization. The following table outlines these parameters and justifies their chosen value.

Parameters

Name	**Value**	**Justification**
kernal_size	3	I tried values up to 9 but saw a decrease in lines detected.
low_threshold	100	I started with the values suggested in the tutorial (50 and 150) and increased keeping the same ratio and settled for these values.
high_threshold	250	
rho	1	Maximum resolution of pixels for maximum coverage
theta	Pi/180	1 degree to give high resolution for maximum coverage
hough_threshold	35	I tried values between 1 and 50 in increments of 5 and chose what I thought worked best.
min_line_gap	5	A small gap b
max_lane_gap	2	Keep the gap between pixels at a minimum to maximize line detection

I will now explain the rest of the pipeline to detect lane lines. The for loop at the end of the pipeline allows all images in the folder "test_images" to be tested, from the outcome this gives a good indication to whether the code is working or not. There are 2 ways the outcome can be displayed and are chosen through inputs to the main function lane_find(); firstly the final outputs can be saved in the folder "test_images_output" by putting save_output to TRUE. Secondly 8 key stages of the process can be displayed in a subplot by putting display_images to TRUE, and inputting chosen images into display_list array.

def lane_find()

This is the main part of the code, that allows the lane lines to be found from an image, this consist of various stages, performed in sub-functions, which each perform a process on the image. To show an example I will show an image (whiteCarLaneSwitch.jpeg) at each stage, starting with the original.

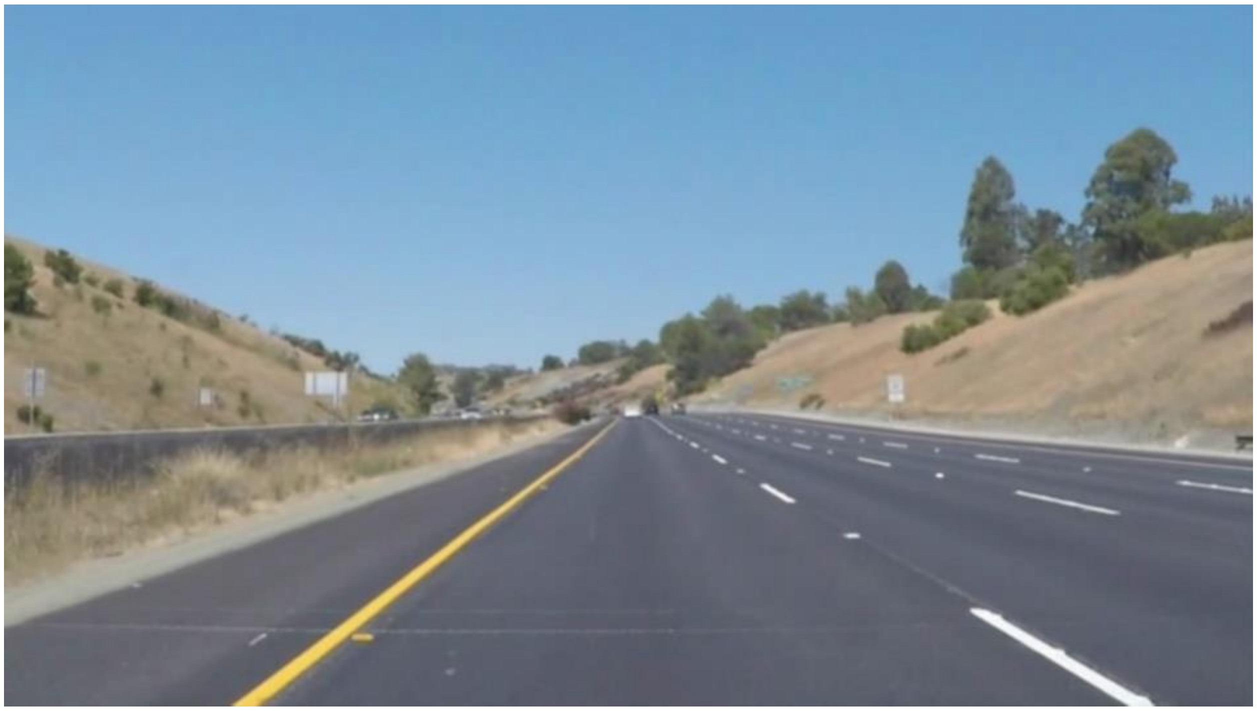

FIGURE 1 - ORIGINAL WHITECARLANESWITCH TEST IMAGE

STAGE 1: First step is to apply Gaussian smoothing, this takes the original image passed to the function. This gives the following output as an example:

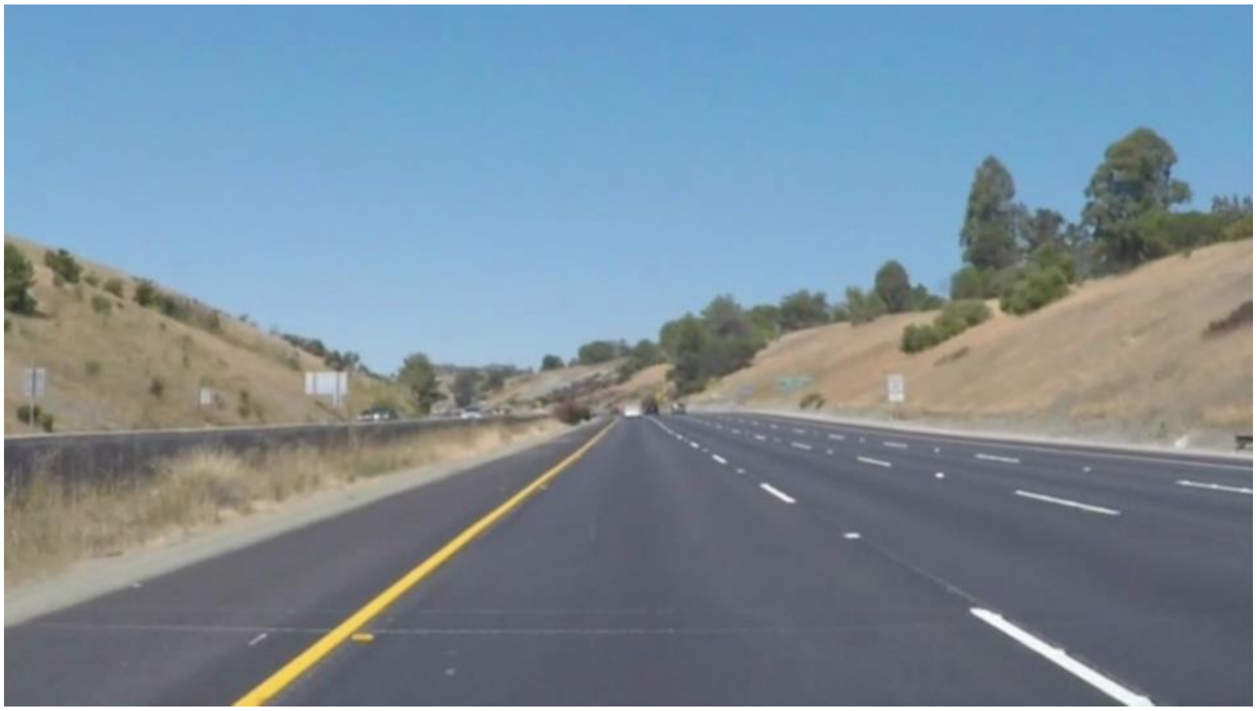

FIGURE 2 - GAUSSIAN BLUR WHITECARLANESWITCH TEST IMAGE

STAGE 2: Second step is to change the image to grayscale

NOTE: The detect_yellow() step is not required but has been added to improve line detection in difficult situations, I will talk about this is the “Improvements for Challenge” section.

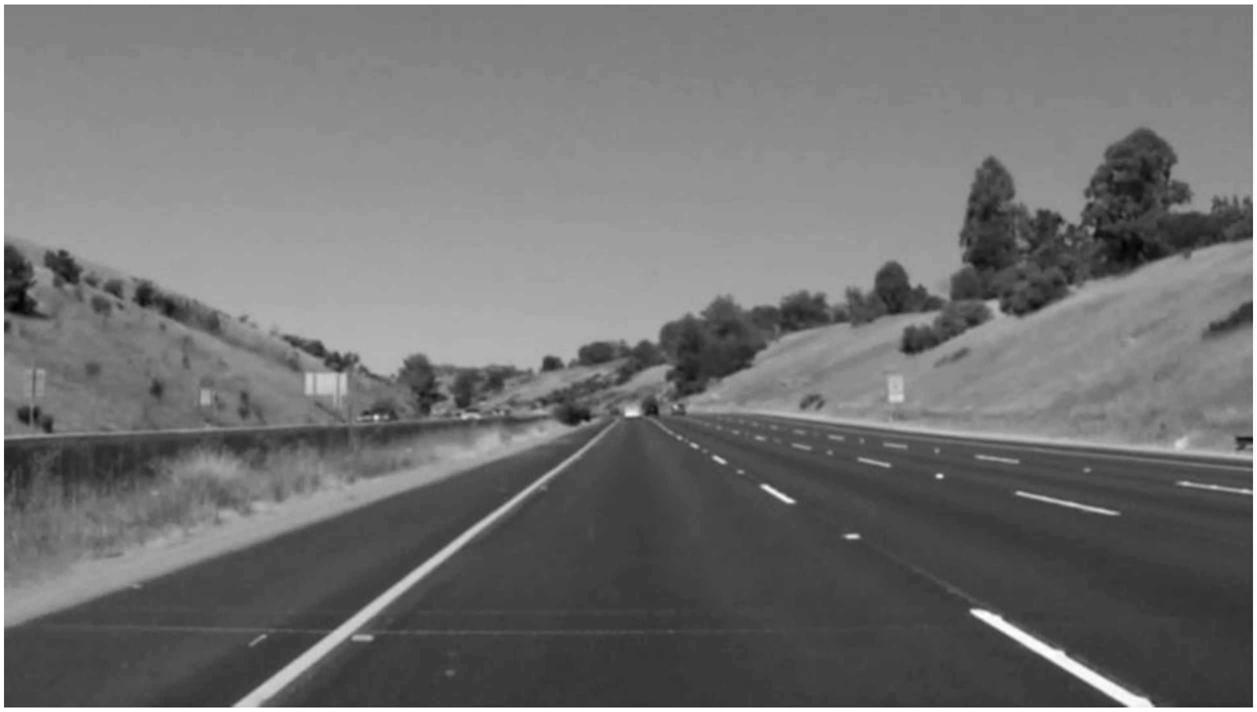

FIGURE 3 – GRAYSCALE WHITECARLANESWITCH TEST IMAGE

STAGE 3: Third step is to apply a Canny edge detection

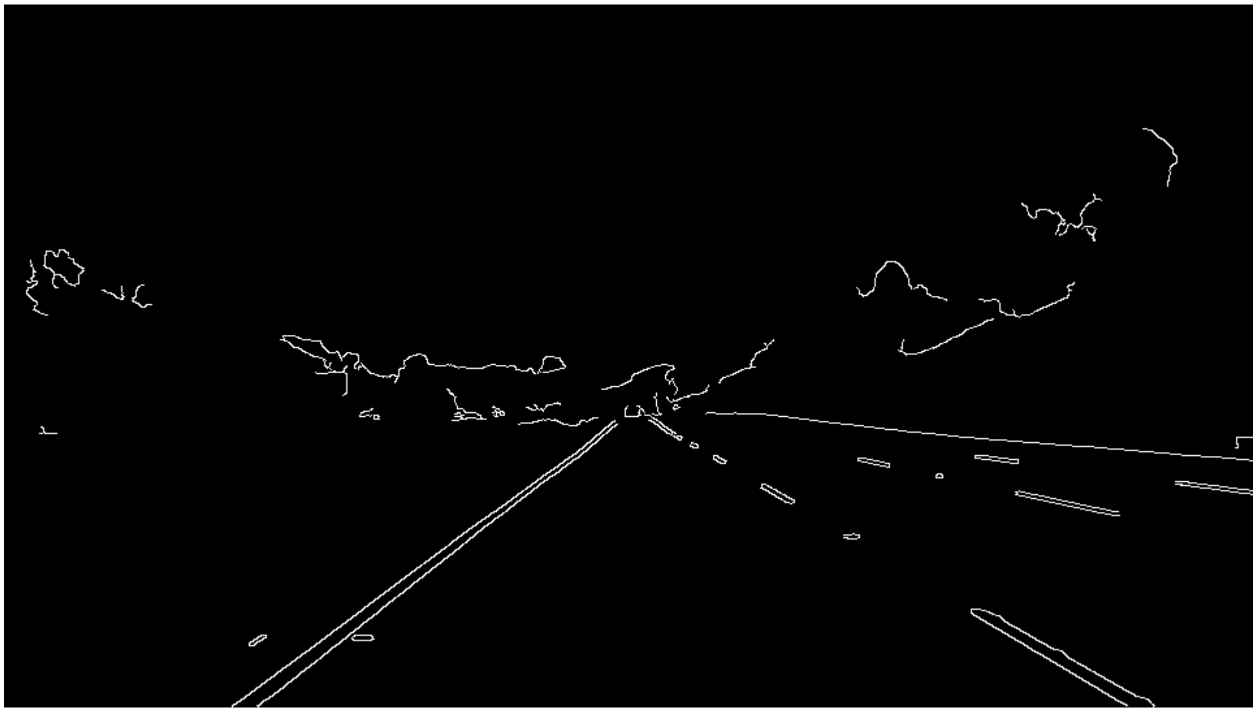

FIGURE 4 - CANNY EDGE WHITECARLANESWITCH TEST IMAGE

STAGE 4: Forth is to apply a mask to allow only the area of importance to be displayed

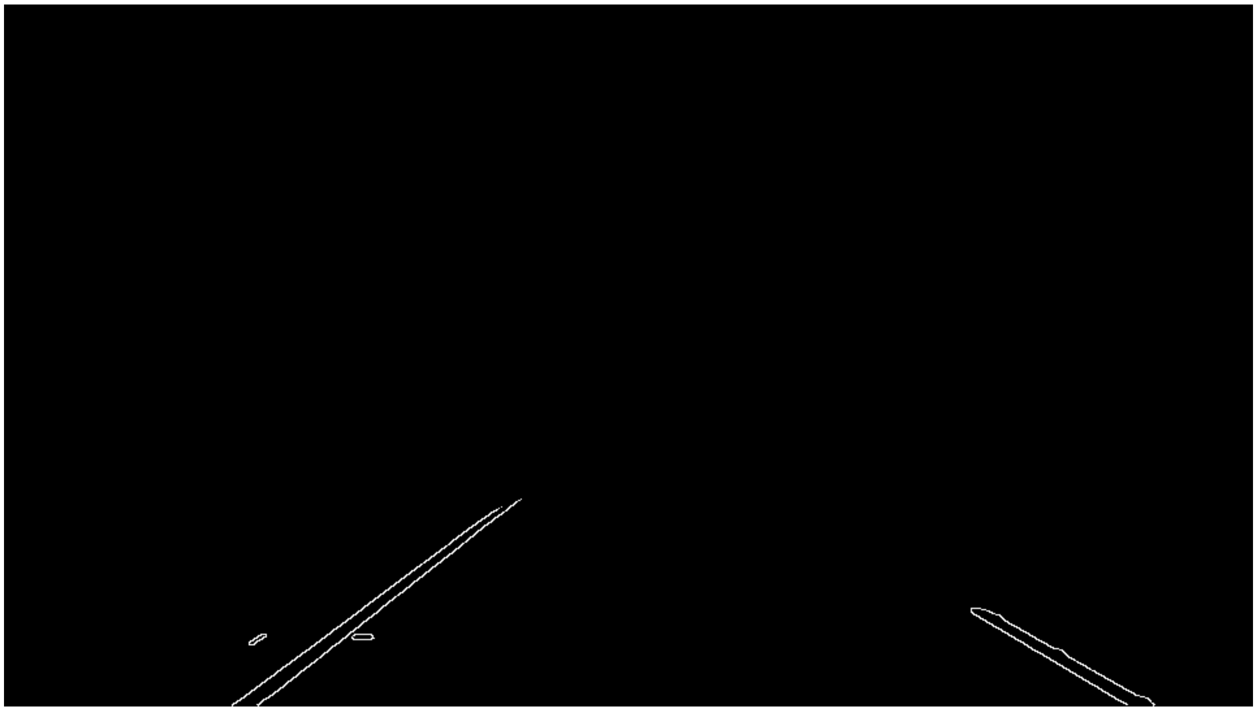

FIGURE 5 – MASK WHITECARLANESWITCH TEST IMAGE

STAGE 5: Fifth is to find the Hough lines, and draw them on the image

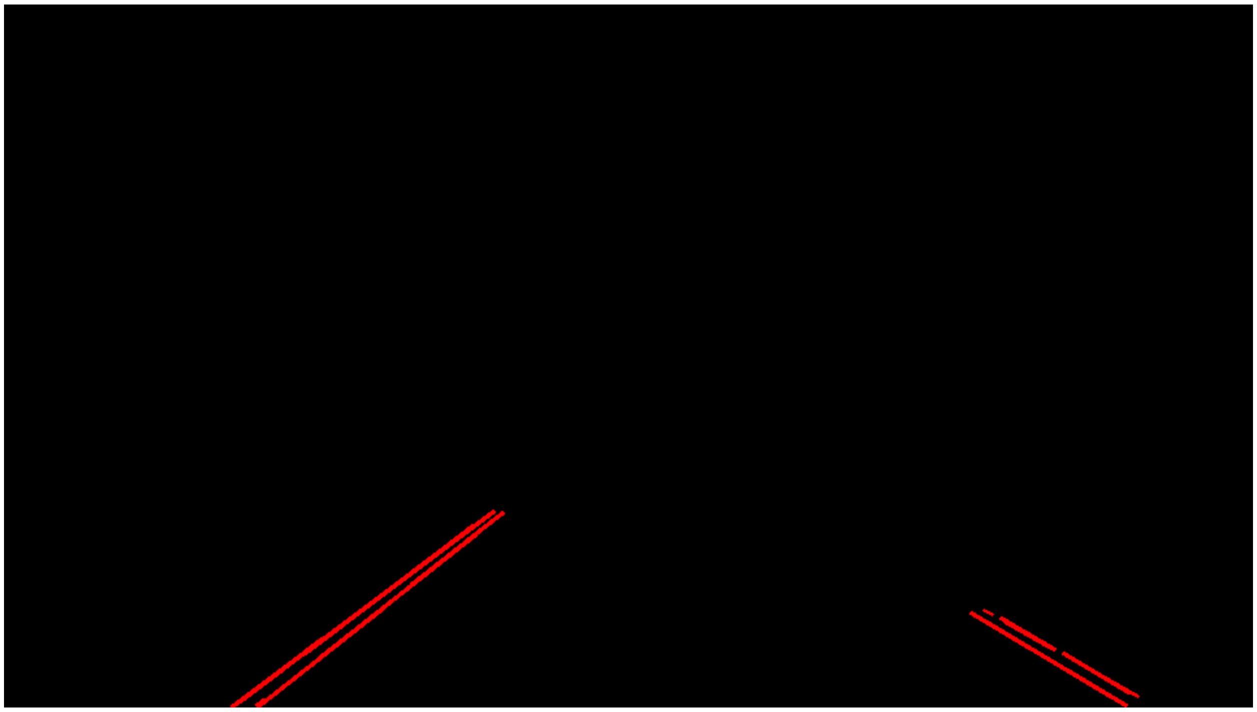

FIGURE 6 - HOUGH LINES WHITECARLANESWITCH TEST IMAGE

Once these steps are complete the final thing to do is fuse the outcome image on top of the original image this is done by the function weighted_img().

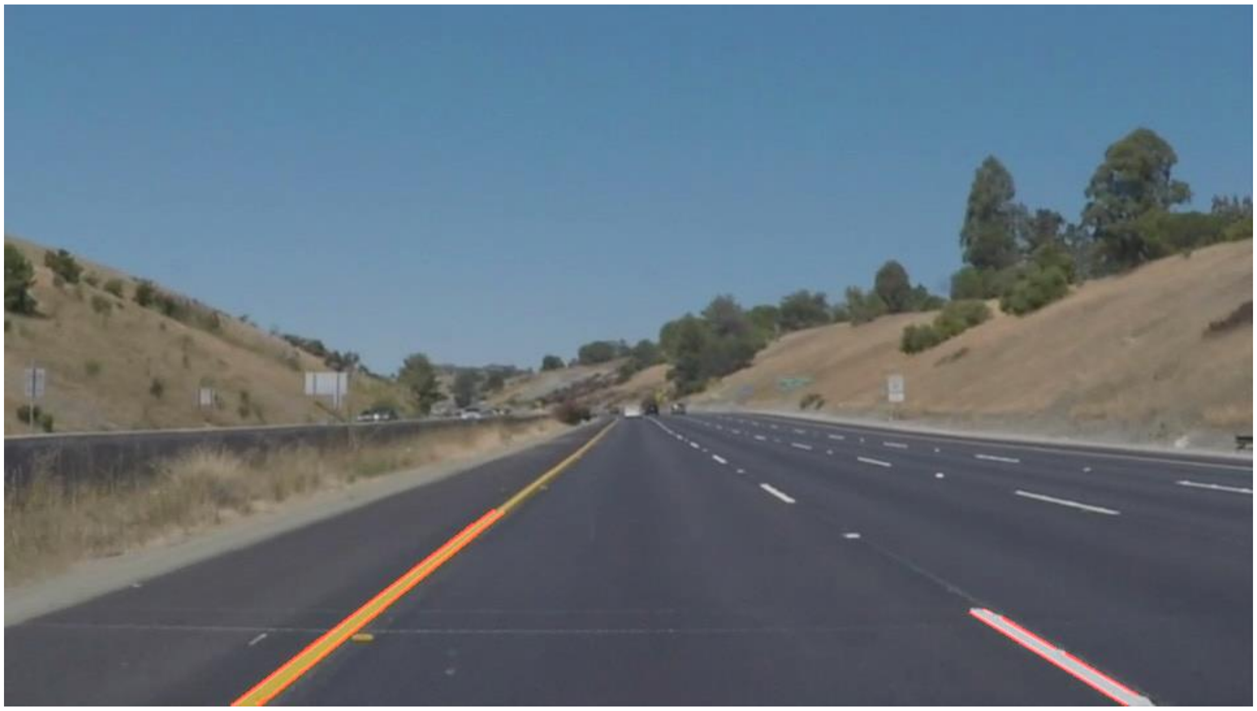

FIGURE 7 - FINAL WEIGHTED WHITECARLANESWITCH TEST IMAGE

def calc_verticles()

This code calculated the verticals for a specific image, outputted as an array of [x,y] coordinates. This is calculated by using the size of the image in y and x direction to calculate the centers. Using these center points the values are then shifted away in equal direction to a value parameterized by a percentage of the total size of x and y respectively.

def display()

This code allows a list of up to 8 images to be displayed as a set of small images, this allows quick traceability of each stage of processing to evaluate the performance of the algorithm.

def grayscale()

This code applies the Grayscale transform which will return an image with only one color channel.

def canny()

This code applies the Canny transform to an image between 2 threshold values specified.

def gaussian_blur()

This code applies a Gaussian Noise kernel to an image, with the kernel size specified.

def region_of_interest()

This code applies an image mask. It only keeps the region of the image defined by the polygon formed from `vertices` with the rest of the image being set to black. The `vertices` are an array of integer points on the corners of the required mask.

def draw_lines()

This code takes all x and y coordinates for all the end points of lines created by the Hough lines function and then draws a line from the start to the end of each independent line with specified color and thickness.

def hough_lines()

This code returns an image with the Hough lines drawn using a number of parameters specified to scan the edge points in an image and transform them into lines.

def weighted_img()

This code takes 2 images of the same shape and fuses them together, with different visibility levels being defined in α and β. 0.8 used for α to make the lines drawn slightly see through.

IMPROVEMENTS TO DRAW_LINES

In order to draw a single line on the left and right lanes, I created a modified version of the draw_lines() function which I called draw_lines2(); this allowed me to flick between the original and new code easier when developing.

def draw_lines2()

This function still takes the same inputs as the original, with the default thickness being changed to 8 as this is the value that suited the line thickness represented in the image.

The first part of this function is to calculate the size of the image, this is the same method as described in calc_vertices(), this is used later to calculate the start and end points for the final lines.

This premise of the first part of the function is to calculate the slope of each Hough line and separate them for the right and left lane. This is done by initiating an array for both right and left points, the points arrive in groups of 4; coordinates for the start of line and coordinates for the end of line. The slope is then calculated for each group with the formula; slope = (y2-y1)/(x2-x1).

Simply if the slope is positive it is “travelling” right, negative it is left, these then get added to the correct array following that rule. I have added limits to avoid lines that are too vertical and too horizontal, these can be changed in the parameters slope_limit_high and slope_limit_low respectively.

NOTE: The bias will be explained in “Improvements for Challenge” section.

Once the points are collected for both right and left lines they are average using np.average() discounting the initial zeros and keeping the x1,x2,y1,y2 values separate using axis=0.

The average values for x1, x2, y1 and y2 are now used to calculate the linear equation for the final left and right line, in the format y=mx+c. m is the slope and is calculated using the formula earlier, c is the intercept on the y axis and is calculated by rearranging the above formula to c = -mx+y

This gives us the formula but to display the lines on the final image we need the start and end coordinates of each line. As we know the y values are at ymax and ycenter ±yshift, we can calculate the x required by putting these y values back into the equation.

When running this code initially it became problematic when no points were detected on either side, as the equation code could not be calculated from an empty array. Firstly I tried to mirror the values collected from right to left, left to right when nothing was available from either side, however I thought this was not the best solution as the lines will not always be symmetrical distance from center.

The solution I came to was to store the previous correct values for right and left lines in a global variable when the average point arrays contain no value, the values would be drawn from the global variables. The last part of this function is to draw the line of specified color and thickness.

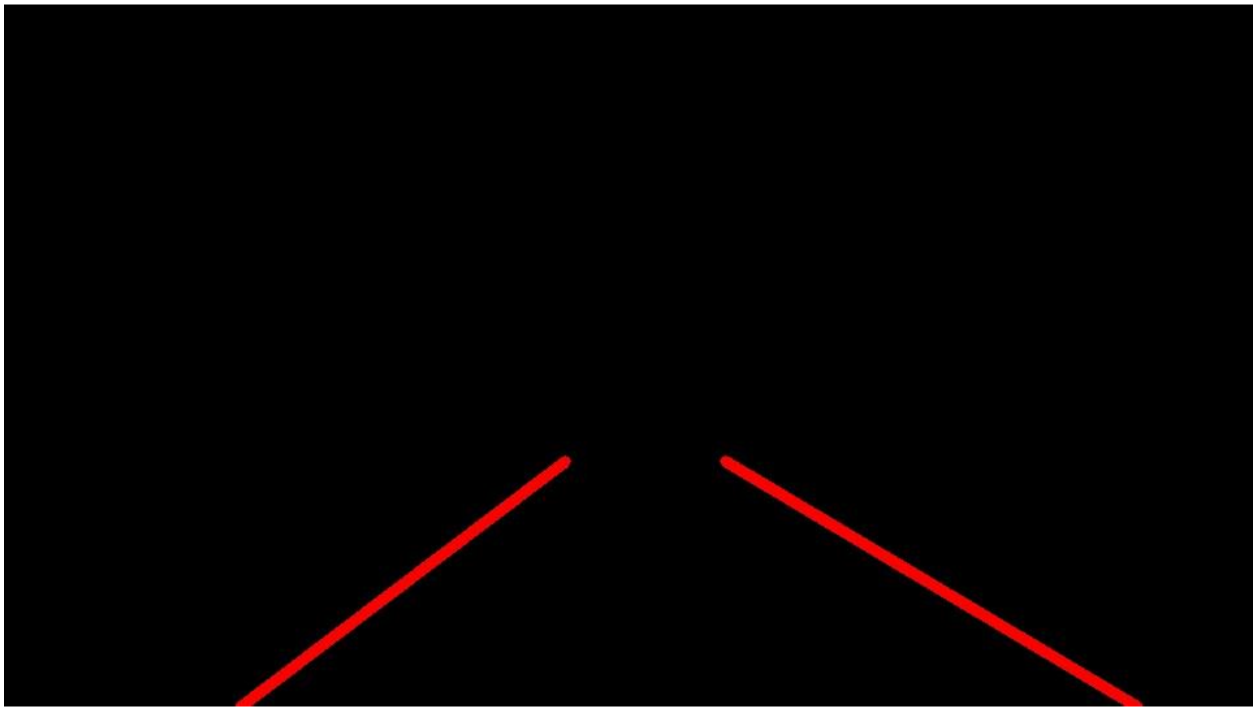

Figure 8 – Singular Lane Lines whiteCarLaneSwitch Test Image

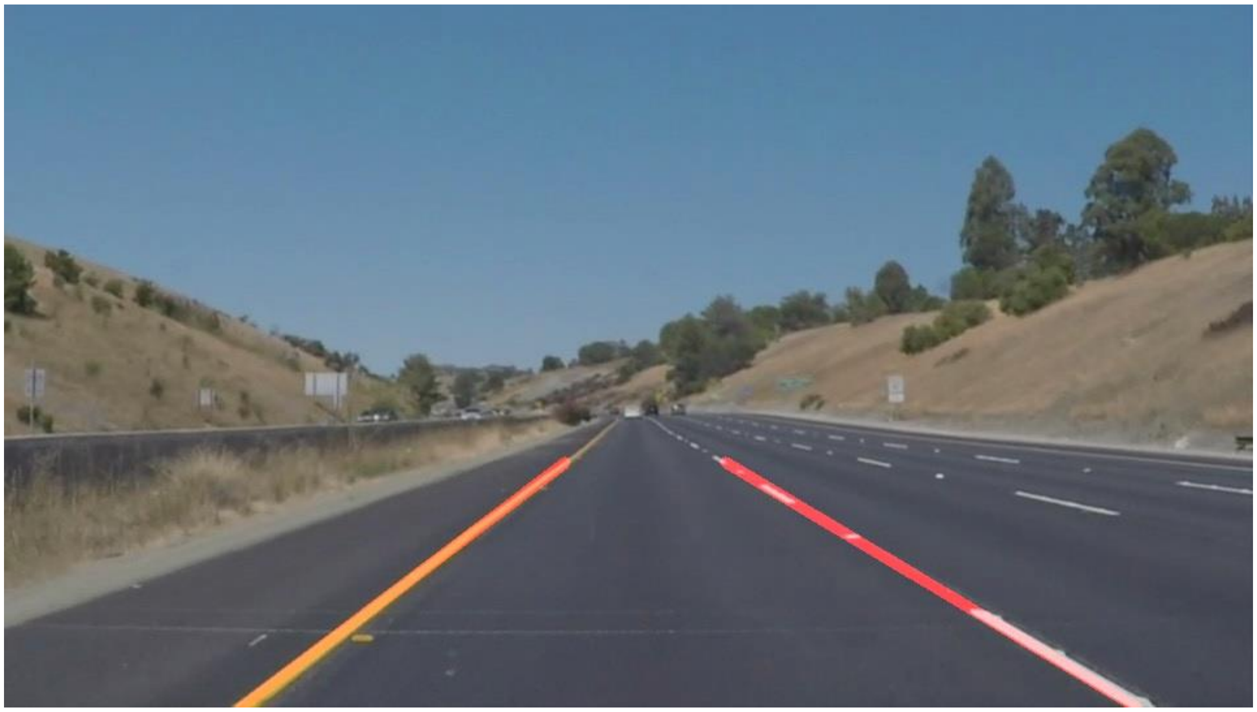

Figure 9 - Final Image WHITECARLANESWITCH Test Image

IMPROVEMENTS FOR CHALLENGE

def draw_lines2() – Bias

To manage the average points deviating from the correct values; usually caused by shadows or barriers. I added a way to bias the input of the Hough lines. I achieved this by weighting the input in three ways and then adding multiples of the points depending on the weighting. Higher score more points, more influence on average values.

The 3 ways are as follows; length, distance from y max and distance from x center. These were calculated in values between 0 and 1 and then multiplied together. The amount the bias influenced the result was given by the number this result was multiplied by, I chose 1000 as it gave influence but did not take too long to compute.

def detect_yellow()

To manage the change in sun exposure and road texture I decided that the yellow line was being affected the worst when being transposed into grayscale. I created a function that would work alongside the grayscale step but focus only on detecting the color yellow.

To do this I drew on inspiration from a project looking at detecting different colors in Rubik cube by Vikas Gupta (https://www.learnopencv.com/color-spaces-in-opencv-cpp-python/).

It consisted of looking at the image in 4 color spaces; RGB, HSV, YCB and LAB. Using the function cv2.inRange() I firstly converted the image into each color separately (functions hsv(), ycb() and lab() perform this action using cv2.cvtColor()) and specified a range of values to pick out the yellow in each color space.

This allowed me to play with the image components in the following ways;

- **RGB** - Red, Green and Blue,
- **HSV** - Hue, Saturation and Intensity
- **YCB** – Luma of RGB, Delta of luma from red and Delta of luma from blue
- **LAB** - Lightness, Component of Green and Magenta and Component of Blue and Yellow

However I mainly used the values in the Rubik cube example. After experimenting I changed only the RGB limits, as this was the most influential. The final step is fusing all these detection methods together and outputting the result.

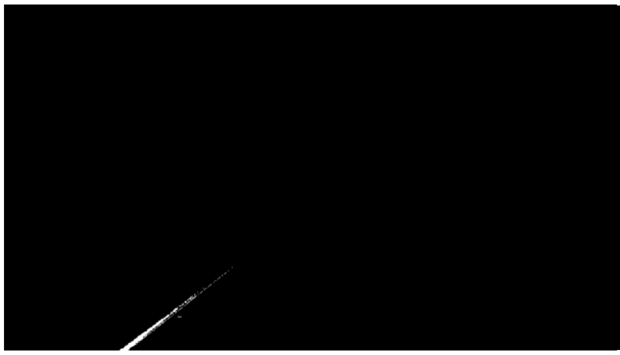
FIGURE 10 - YELLOW DETECTION WITH RGB ONLY

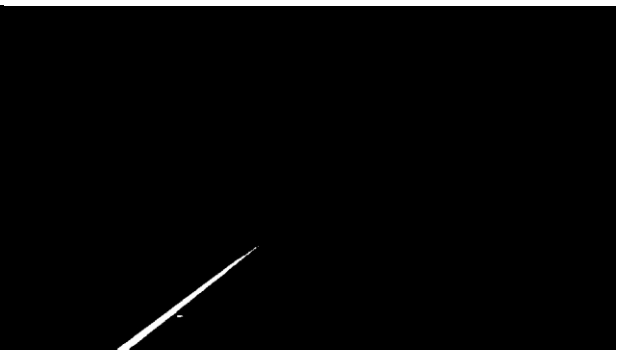
FIGURE 11 - YELLOW DETECTION WITH ALL COLOR SPACES

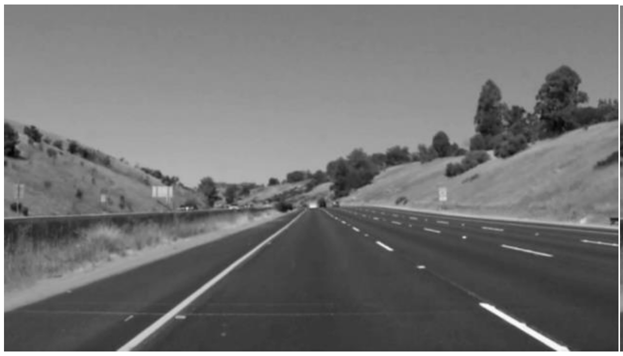
FIGURE 12 - GRAYSCALE ONLY

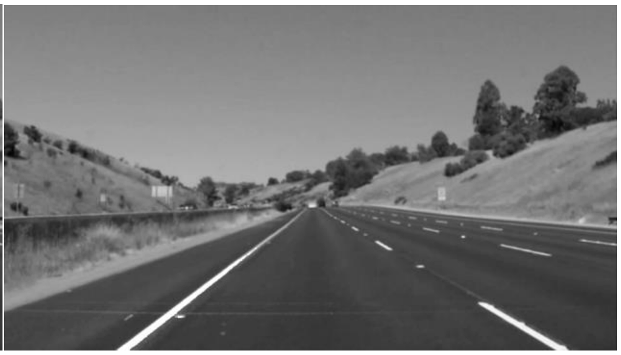
FIGURE 13 - GRAYSCALE WITH YELLOW DETECTION

POTENTIAL SHORTCOMINGS

One potential shortcoming would be what would happen when there are no lines appearing in the masked area either initially; as no previous values could be drawn upon to calculate the formula for the line.

A problem would also potentially occur if the Hough lines have not been detected for an extended period of time as the previous value may start to differ dramatically from what should be represented.

POSSIBLE IMPROVEMENTS

A possible improvement would be to improve calculation of where the image mask is placed and the final lines start and finish, currently it is just using a percentage of the x and y and their max from image.shape(). This percentage could be optimized better or a method of detecting the line closest to center and following it linearly.

Another potential improvement could be initially mask the image left and right to neglect possible lines curving from negative to positive or vice versa. You could also then use this to to draw a clothoid curve rather than a linear line, dependent on the number of reverse Hough lines detected on either side.

It would be great to color code lines depending on the confidence given for the equation. I think this could be done by using the number of points you average and the range of those points. This could help combat the no line detected problem as an initial estimate could be used with low confidence. Another method could be to explore the mirroring method again but give them a low bias.