P1

June 20, 2018

1 Self-Driving Car Engineer Nanodegree

1.1 Project: Finding Lane Lines on the Road

In this project, you will use the tools you learned about in the lesson to identify lane lines on the road. You can develop your pipeline on a series of individual images, and later apply the result to a video stream (really just a series of images). Check out the video clip "raw-lines-example.mp4" (also contained in this repository) to see what the output should look like after using the helper functions below.

Once you have a result that looks roughly like "raw-lines-example.mp4", you'll need to get creative and try to average and/or extrapolate the line segments you've detected to map out the full extent of the lane lines. You can see an example of the result you're going for in the video "P1_example.mp4". Ultimately, you would like to draw just one line for the left side of the lane, and one for the right.

In addition to implementing code, there is a brief writeup to complete. The writeup should be completed in a separate file, which can be either a markdown file or a pdf document. There is a write up template that can be used to guide the writing process. Completing both the code in the Ipython notebook and the writeup template will cover all of the rubric points for this project.

The tools you have are color selection, region of interest selection, grayscaling, Gaussian smoothing, Canny Edge Detection and Hough Tranform line detection. You are also free to explore and try other techniques that were not presented in the lesson. Your goal is piece together a pipeline to detect the line segments in the image, then average/extrapolate them and draw them onto the image for display (as below). Once you have a working pipeline, try it out on the video stream below.

Your output should look something like this (above) after detecting line segments using the helper functions below

Your goal is to connect/average/extrapolate line segments to get output like this

Run the cell below to import some packages. If you get an `import error` for a package you've already installed, try changing your kernel (select the Kernel menu above –> Change Kernel). Still have problems? Try relaunching Jupyter Notebook from the terminal prompt. Also, consult the forums for more troubleshooting tips.

1.2 Import Packages

In [220]:
```
#importing some useful packages
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import cv2
%matplotlib inline
```

1.3 Read in an Image

In [221]:
```
#reading in an image
image = mpimg.imread('test_images/solidWhiteRight.jpg')

#printing out some stats and plotting
print('This image is:', type(image), 'with dimensions:', image.shape)

# if you wanted to show a single color channel image called 'gray',
# for example, call as plt.imshow(gray, cmap='gray')
plt.imshow(image)
```

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

Out[221]: `<matplotlib.image.AxesImage at 0x7efcc9d3a668>`

1.4 Ideas for Lane Detection Pipeline

Some OpenCV functions (beyond those introduced in the lesson) that might be useful for this project are:

`cv2.inRange()` for color selection
`cv2.fillPoly()` for regions selection
`cv2.line()` to draw lines on an image given endpoints
`cv2.addWeighted()` to coadd / overlay two images `cv2.cvtColor()` to grayscale or change color
`cv2.imwrite()` to output images to file
`cv2.bitwise_and()` to apply a mask to an image

Check out the OpenCV documentation to learn about these and discover even more awesome functionality!

1.5 Helper Functions

Below are some helper functions to help get you started. They should look familiar from the lesson!

In [222]:
```
import math

def grayscale(img):
    """Applies the Grayscale transform
    This will return an image with only one color channel
    but NOTE: to see the returned image as grayscale
    (assuming your grayscaled image is called 'gray')
    you should call plt.imshow(gray, cmap='gray')"""
    #return cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    # Or use BGR2GRAY if you read an image with cv2.imread()
    return cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

def canny(img, low_threshold, high_threshold):
    """Applies the Canny transform"""
    return cv2.Canny(img, low_threshold, high_threshold)

def gaussian_blur(img, kernel_size):
    """Applies a Gaussian Noise kernel"""
    return cv2.GaussianBlur(img, (kernel_size, kernel_size), 0)

def region_of_interest(img, vertices):
    """
    Applies an image mask.

    Only keeps the region of the image defined by the polygon
    formed from `vertices`. The rest of the image is set to black.
    `vertices` should be a numpy array of integer points.
    """
    #defining a blank mask to start with
    mask = np.zeros_like(img)

    #defining a 3 channel or 1 channel color to fill the
    #mask with depending on the input image
    if len(img.shape) > 2:
```

```python
        channel_count = img.shape[2]  # i.e. 3 or 4 depending on your image
        ignore_mask_color = (255,) * channel_count
    else:
        ignore_mask_color = 255

    #filling pixels inside the polygon defined by "vertices" with the fill color
    cv2.fillPoly(mask, vertices, ignore_mask_color)

    #returning the image only where mask pixels are nonzero
    masked_image = cv2.bitwise_and(img, mask)
    return masked_image


def drawLine(img, x, y, color=[255, 0, 0], thickness=20):
    if (len(x) == 0):
            return

    poly = np.polyfit(x, y, 1)

    slope = poly[0]
    intercept = poly[1]

    height = img.shape[0]
    width = img.shape[1]

    y1 = height
    x1 = int((y1 - intercept)/slope)
    y2 = int(height/2) + 60
    x2 = int((y2 - intercept)/slope)

    cv2.line(img, (x1, y1), (x2, y2), color, thickness)

def draw_lines(img, lines, color=[255, 0, 0], thickness=13):
    """
    NOTE: this is the function you might want to use as a starting point
    once you want to
    average/extrapolate the line segments you detect to map out the full
    extent of the lane (going from the result shown in raw-lines-example.mp4
    to that shown in P1_example.mp4).

    Think about things like separating line segments by their
    slope ((y2-y1)/(x2-x1)) to decide which segments are part of the left
    line vs. the right line.  Then, you can average the position of each of
    the lines and extrapolate to the top and bottom of the lane.

    This function draws `lines` with `color` and `thickness`.
    Lines are drawn on the image inplace (mutates the image).
    If you want to make the lines semi-transparent, think about combining
```

```
this function with the weighted_img() function below


"""
width = img.shape[1]
height = img.shape[0]
lines_slope_intercept = np.zeros(shape=(len(lines),2))
for index,line in enumerate(lines):
    for x1,y1,x2,y2 in line:
        slope = (y2-y1)/(x2-x1)
        intercept = y1 - x1 * slope
        lines_slope_intercept[index]=[slope,intercept]

# Calculating the intercepts for max and min slope lines
max_slope_line = lines_slope_intercept[lines_slope_intercept.argmax(axis=0)[0]]
min_slope_line = lines_slope_intercept[lines_slope_intercept.argmin(axis=0)[0]]

# Placeholders for left and right slope and intercepts.

left_slopes = []
left_intercepts = []
right_slopes = []
right_intercepts = []

# Calculating the slopes and intercepts and
# appending them to placeholders.

for line in lines_slope_intercept:
    if abs(line[0] - max_slope_line[0]) < 0.1 and \
        abs(line[1] - max_slope_line[1]) < (0.1 * width):
        left_slopes.append(line[0])
        left_intercepts.append(line[1])
    elif abs(line[0] - min_slope_line[0]) < 0.1 and \
        abs(line[1] - min_slope_line[1]) < (0.1 * width):
        right_slopes.append(line[0])
        right_intercepts.append(line[1])

# left and right lines are averages of these slopes and intercepts,
#extrapolate lines to edges and center*
# *roughly
new_lines = np.zeros(shape=(1,2,4), dtype=np.int32)

# Calculating the coordinates for the extended lines.

if len(left_slopes) > 0:
    left_line = [sum(left_slopes)/len(left_slopes),
                 sum(left_intercepts)/len(left_intercepts)]
    left_bottom_x = (height - left_line[1])/left_line[0]
```

```
        left_top_x = (height*.6 - left_line[1])/left_line[0]
        if (left_bottom_x >= 0):
            new_lines[0][0] =[left_bottom_x,height,left_top_x,height*.6]

    if len(right_slopes) > 0:
        right_line = [sum(right_slopes)/len(right_slopes),
                      sum(right_intercepts)/len(right_intercepts)]
        right_bottom_x = (height - right_line[1])/right_line[0]
        right_top_x = (height*.6 - right_line[1])/right_line[0]
        if (right_bottom_x <= width):
            new_lines[0][1]=[right_bottom_x,height,right_top_x,height*.6]

    # Drawing the lines.

    for line in new_lines:
        for x1,y1,x2,y2 in line:
            cv2.line(img, (x1, y1), (x2, y2), color, thickness)



def draw_lines2(img, lines, color=[255, 0, 0], thickness=13):
    """
    NOTE: this is the function you might want to use as a starting point once you want
    average/extrapolate the line segments you detect to map out the full
    extent of the lane (going from the result shown in raw-lines-example.mp4
    to that shown in P1_example.mp4).

    Think about things like separating line segments by their
    slope ((y2-y1)/(x2-x1)) to decide which segments are part of the left
    line vs. the right line.  Then, you can average the position of each of
    the lines and extrapolate to the top and bottom of the lane.

    This function draws `lines` with `color` and `thickness`.
    Lines are drawn on the image inplace (mutates the image).
    If you want to make the lines semi-transparent, think about combining
    this function with the weighted_img() function below
    """
    x_size = img.shape[1]
    y_size = img.shape[0]

    imshape = img.shape
    print(imshape)
    print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~")
    print("The length of the lines is: ", len(lines))
    print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~")
    positive_slopes = 0
    negative_slopes = 0
```

```
ns_line_x_min = 0
ns_line_x_max = 960
ns_line_y_min = 0
ns_line_y_max = 540

ps_line_x_min = 0
ps_line_x_max = 960
ps_line_y_min = 0
ps_line_y_max = 540

ns_line_x1 = []
ns_line_x2 = []
ns_line_y1 = []
ns_line_y2 = []

ps_line_x1 = []
ps_line_x2 = []
ps_line_y1 = []
ps_line_y2 = []


for line in lines:
    for x1,y1,x2,y2 in line:
        if ((y2-y1)/(x2-x1)) < 0:
            print("Points for negative slope; x1: ", x1, "y1: ", y1, "x2: ", x2, "
            negative_slopes += 1
            ns_line_x1.append(x1)
            ns_line_x2.append(x2)
            ns_line_y1.append(y1)
            ns_line_y2.append(y2)


        if ((y2-y1)/(x2-x1)) > 0:
            print("Points for positive slope; x1: ", x1, "y1: ", y1, "x2: ", x2, "
            positive_slopes += 1

            ps_line_x1.append(x1)
            ps_line_x2.append(x2)
            ps_line_y1.append(y1)
            ps_line_y2.append(y2)


        #cv2.line(line_image,(x1,y1),(x2,y2),(255,0,0),10)
ns_x1 = max(ns_line_x1)
ns_y1 = min(ns_line_y1)
ns_x2 = min(ns_line_x2)
ns_y2 = max(ns_line_y2)
```

```
if ns_x1 != ns_x2 and ns_y1 != ns_y2:
    ns_slope = (ns_y2 - ns_y1)/(ns_x2 - ns_x1)




    print ("ns_x1: ", ns_x1, "ns_x2: ", ns_x2, "ns_y1: ", ns_y1, "ns_y2: ", ns_y2)



    if ns_y1 > 0.6 * imshape[0]:
        ns_y1 = int(0.65 * imshape[0])
    ns_x1 = int(ns_x2 + (1/ns_slope) * (ns_y1 - ns_y2))
    print ("ns_x1: ", ns_x1, "ns_x2: ", ns_x2, "ns_y1: ", ns_y1, "ns_y2: ", ns_y2)


    if ns_y2 != imshape[0]:
        ns_y2 = imshape[0]
    ns_x2 = int(ns_x1 + (1/ns_slope) * (ns_y2 - ns_y1))
    print ("ns_x1: ", ns_x1, "ns_x2: ", ns_x2, "ns_y1: ", ns_y1, "ns_y2: ", ns_y2)

    #cv2.line(line_image,(max(ns_line_x1),min(ns_line_y1)),(min(ns_line_x2),max(ns

    cv2.line(img,(ns_x1-15,ns_y1),(ns_x2+15,ns_y2),(255,0,0),10)

ps_x1 = min(ps_line_x1)
ps_y1 = min(ps_line_y1)
ps_x2 = max(ps_line_x2)
ps_y2 = max(ps_line_y2)

if ps_x1 != ps_x2 and ps_y1 != ps_y2:
    ps_slope = (ps_y2 - ps_y1)/(ps_x2 - ps_x1)


    print ("ps_x1: ", ps_x1, "ps_x2: ", ps_x2, "ps_y1: ", ps_y1, "ps_y2: ", ps_y2)



    if ps_y1 > 0.6 * imshape[0]:
        ps_y1 = int(0.65 * imshape[0])
    ps_x1 = int(ps_x2 + (1/ps_slope) * (ps_y1 - ps_y2))
    print ("ps_x1: ", ps_x1, "ps_x2: ", ps_x2, "ps_y1: ", ps_y1, "ps_y2: ", ps_y2)

    if ps_y2 != imshape[0]:
        ps_y2 = imshape[0]
    ps_x2 = int(ps_x1 + (1/ps_slope) * (ps_y2 - ps_y1))
    print ("ps_x1: ", ps_x1, "ps_x2: ", ps_x2, "ps_y1: ", ps_y1, "ps_y2: ", ps_y2)
```

```
            cv2.line(img,(ps_x1,ps_y1),(ps_x2-15,ps_y2),(255,0,0),10)

    def hough_lines(img, rho, theta, threshold, min_line_len, max_line_gap):
        """
        `img` should be the output of a Canny transform.

        Returns an image with hough lines drawn.
        """
        lines = cv2.HoughLinesP(img, rho, theta, threshold, np.array([]),
                                minLineLength=min_line_len, maxLineGap=max_line_gap)
        line_img = np.zeros((img.shape[0], img.shape[1], 3), dtype=np.uint8)
        #print(line_img)
        draw_lines(line_img, lines)
        return line_img

    # Python 3 has support for cool math symbols.

    def weighted_img(img, initial_img, =0.8, =1., =0.):
        """
        `img` is the output of the hough_lines(), An image with lines drawn on it.
        Should be a blank image (all black) with lines drawn on it.

        `initial_img` should be the image before any processing.

        The result image is computed as follows:

        initial_img *  + img *  +
        NOTE: initial_img and img must be the same shape!
        """
        return cv2.addWeighted(initial_img, , img, , )
```

1.6 Test Images

Build your pipeline to work on the images in the directory "test_images"
You should make sure your pipeline works well on these images before you try the videos.

```
In [223]: import os
          test_images_folder = "test_images"
          test_images_names = os.listdir("test_images/")
          #print(test_images_folder)
          #print(test_images_names)

          test_images = [mpimg.imread(test_images_folder + '/' + image_name)
                         for image_name in test_images_names]

          print(test_images_folder)
          print(test_images_names)
```

```
test_images
['solidWhiteRight.jpg', 'solidYellowLeft.jpg', 'solidYellowCurve.jpg', 'solidWhiteCurve.jpg', 's
```

1.7 Build a Lane Finding Pipeline

Build the pipeline and run your solution on all test_images. Make copies into the `test_images_output` directory, and you can use the images in your writeup report.

Try tuning the various parameters, especially the low and high Canny thresholds as well as the Hough lines parameters.

The following is the pipeline for lane finding.

1. Convert images to grayscale
2. Apply Gaussian blur
3. Apply Canny transform
4. Apply region mask
5. Apply hough lines and draw

In the following cell, these actions are done on all test images.

In [224]:
```python
# TODO: Build your pipeline that will draw lane lines on the test_images
# then save them to the test_images_output directory.


#for img in test_images:
#    print(type(img))

# Original Images

#print("Original Images")
for img in test_images:
    plt.figure()
    plt.imshow(img)
    plt.show

# Convert to grayscale

#print("Grayscale Images")

test_images_gray = [grayscale(image) for image in test_images]

for img, index in zip(test_images_gray, range(len(test_images_gray))):
    mpimg.imsave(test_images_folder + '_output/' + 'gray_' + \
                 test_images_names[index], img)
    plt.figure()
    plt.imshow(img, cmap='Greys_r')
```

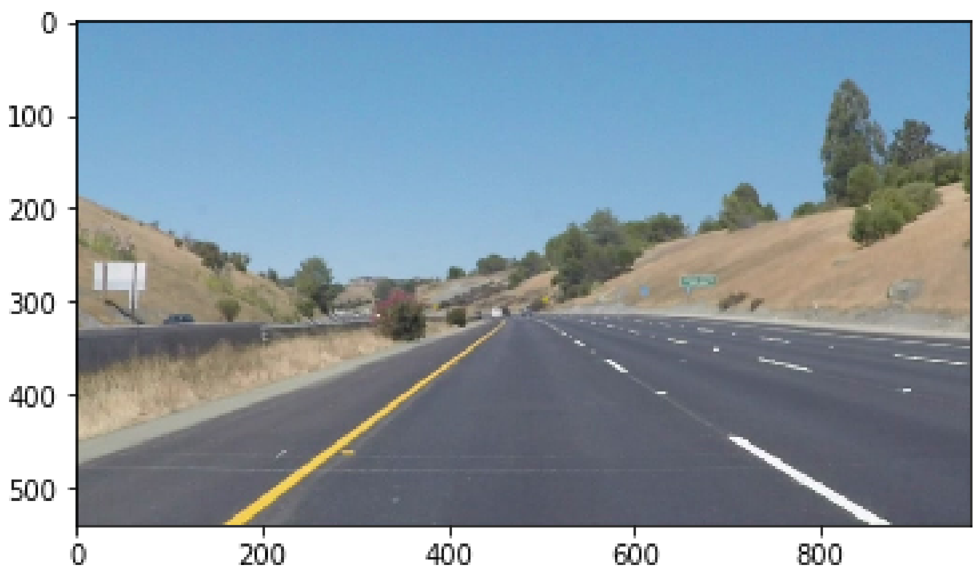
```
    plt.show


# Apply Gaussian blur

kernel_size = 15
test_images_gray_blur = [gaussian_blur(image, kernel_size)
                         for image in test_images_gray]
#print("Grayscale BLUR Images")

for img, index in zip(test_images_gray_blur, range(len(test_images_gray_blur))):
    mpimg.imsave(test_images_folder + '_output/' + 'blur_gray_' + \
                 test_images_names[index], img)
    plt.figure()
    plt.imshow(img, cmap='Greys_r')
    plt.show


#Apply Canny transform

canny_low_threshold = 50
canny_high_threshold = 120

test_images_gray_blur_canny = [canny(image, canny_low_threshold, \
                                     canny_high_threshold) \
                               for image in test_images_gray_blur]

for img, index in zip(test_images_gray_blur_canny, \
                      range(len(test_images_gray_blur_canny))):
    mpimg.imsave(test_images_folder + '_output/' + \
                 'canny_blur_gray_' + test_images_names[index], img)
    plt.figure()
    plt.imshow(img)
    plt.show


#Apply region mask

test_images_masked = []

for img in test_images_gray_blur_canny:
    imshape = img.shape
    vertices = np.array([[(0,imshape[0]),
                          (imshape[1]*0.45,
                           imshape[0]*0.6),
                          (imshape[1]*0.55,
                           imshape[0]*0.6),
                          (imshape[1],imshape[0])]], dtype=np.int32)
```

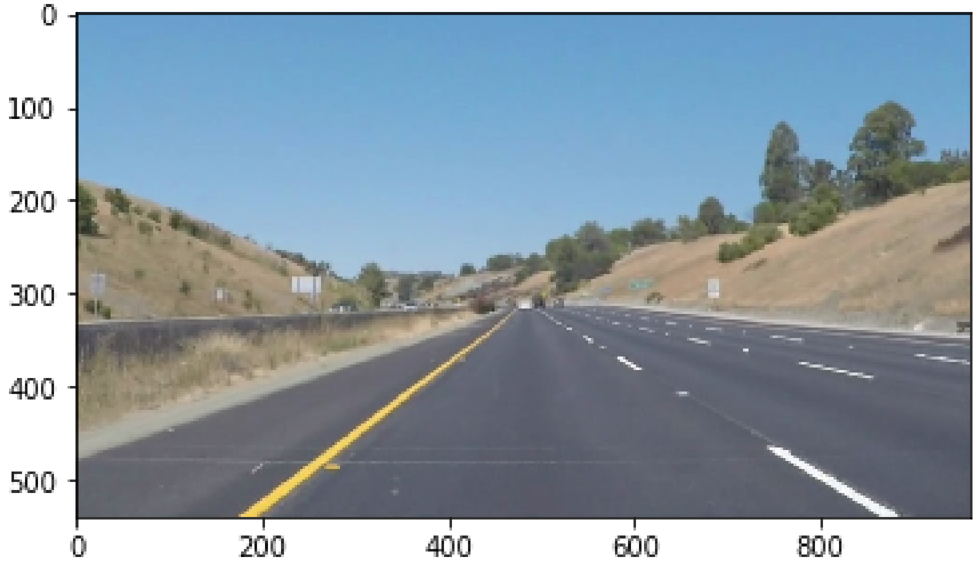
```python
    test_images_masked.append(region_of_interest(img, vertices))

for img, index in zip(test_images_masked, \
                      range(len(test_images_masked))):
    mpimg.imsave(test_images_folder + '_output/' + \
                 'masked_canny_blur_gray_' + test_images_names[index], img)
    plt.figure()
    plt.imshow(img)
    plt.show

# Apply hough lines and draw

rho  =  11  # distance resolution in pixels of the Hough grid# distanc
theta = np.pi/180 # angular resolution in radians of the Hough grid
threshold = 10     # minimum number of votes (intersections in Hough grid cell)
min_line_length = 20 #minimum number of pixels making up a line
max_line_gap = 1    # maximum gap in pixels between connectable line segments

test_images_hough = [hough_lines(img, rho, theta, \
                                 threshold, min_line_length, \
                                 max_line_gap) for img in test_images_masked]
for img, index in zip(test_images_hough, range(len(test_images_hough))):
    mpimg.imsave(test_images_folder + '_output/' + \
                 'hough_masked_canny_blur_gray_' + test_images_names[index], img)
    plt.figure()
    plt.imshow(img)
    plt.show
```

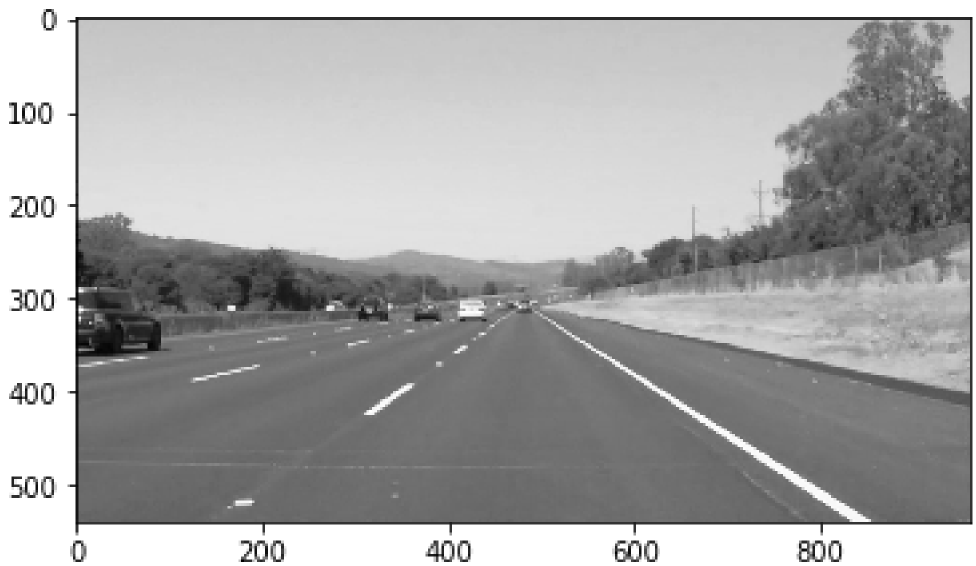

0
100
200
300
400
500
0
200
400
600
800

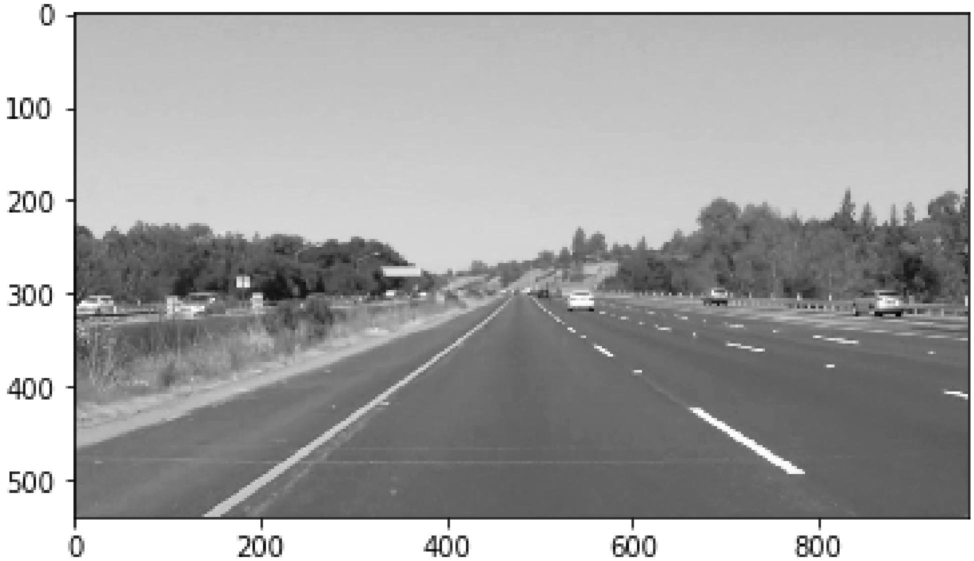

0
100
200
300
400
500
0
200
400
600
800

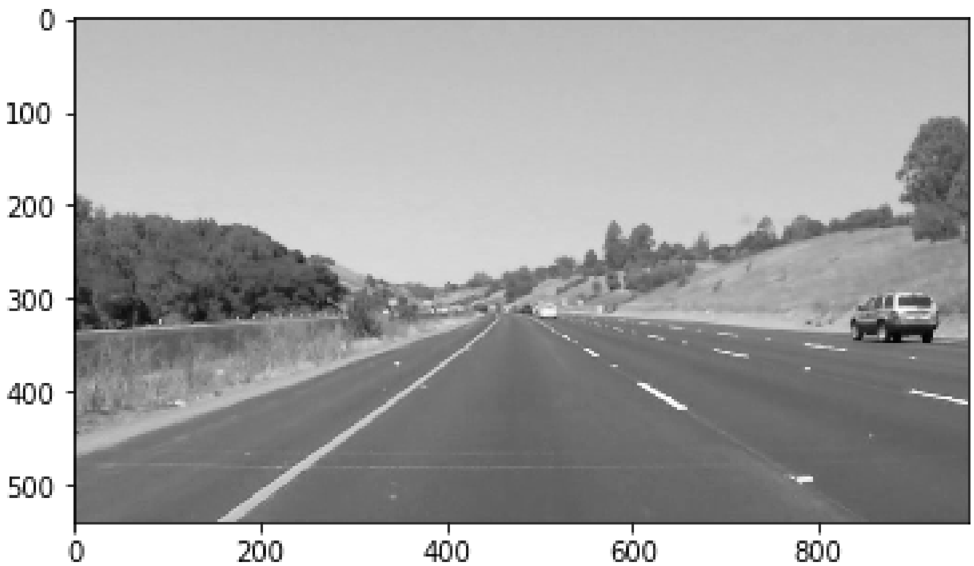

0
100
200
300
400
500
0
200
400
600
800

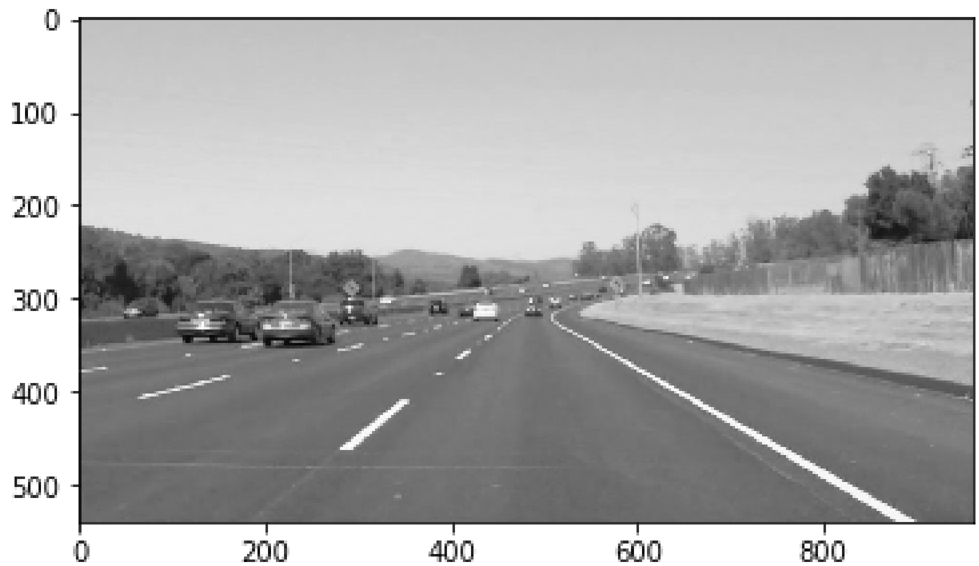

0
100
200
300
400
500
0
200
400
600
800

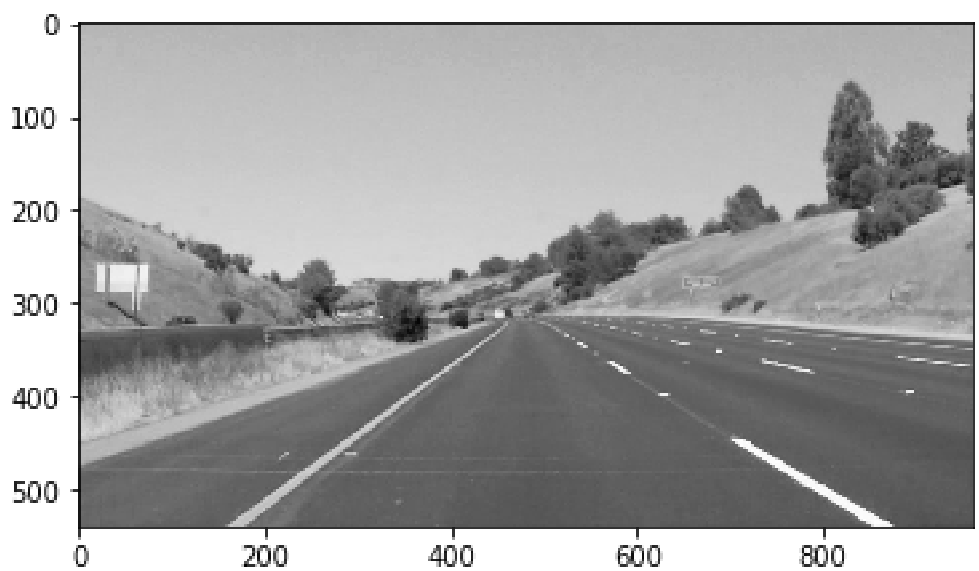

0
100
200
300
400
500
0
200
400
600
800

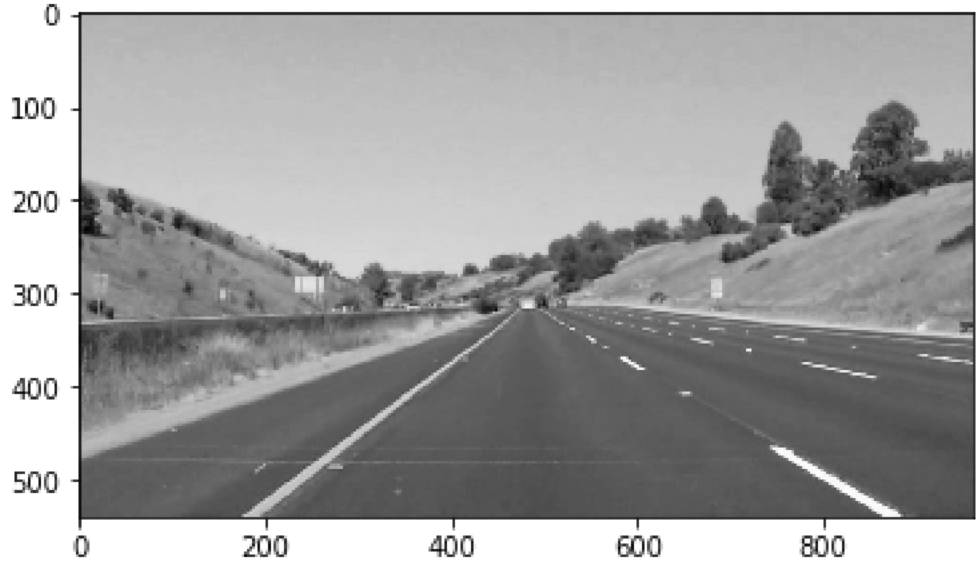

0
100
200
300
400
500
0
200
400
600
800

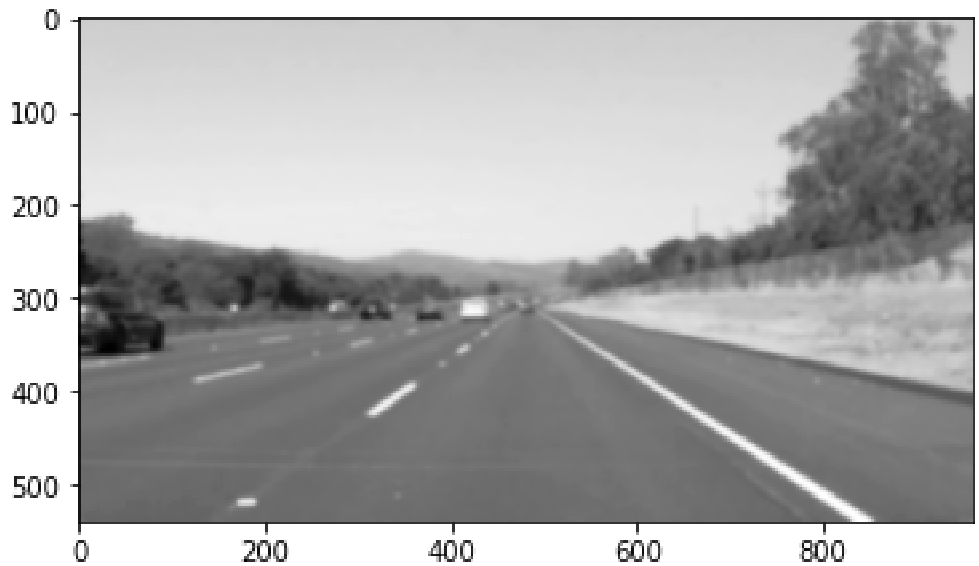

0
100
200
300
400
500
0
200
400
600
800

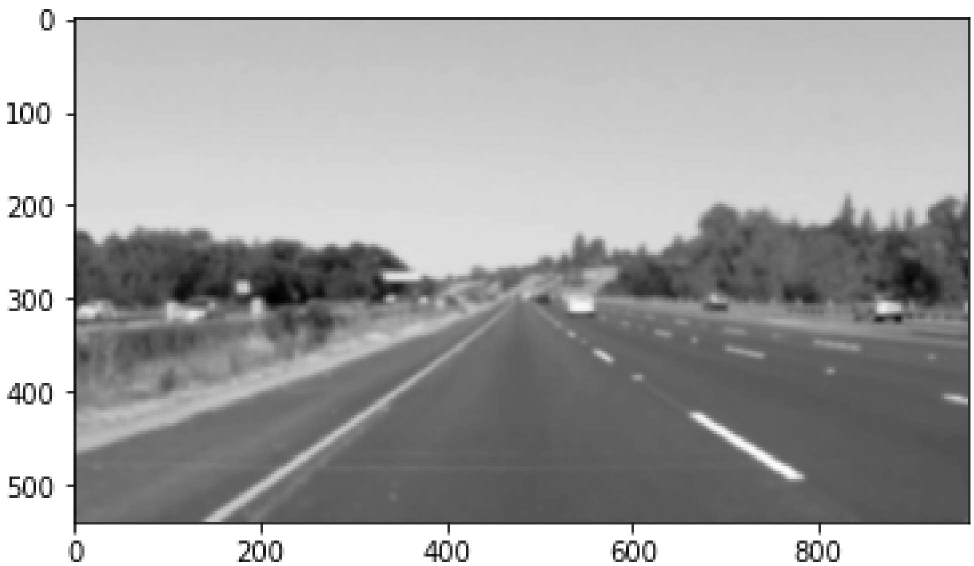

0
100
200
300
400
500
0
200
400
600
800

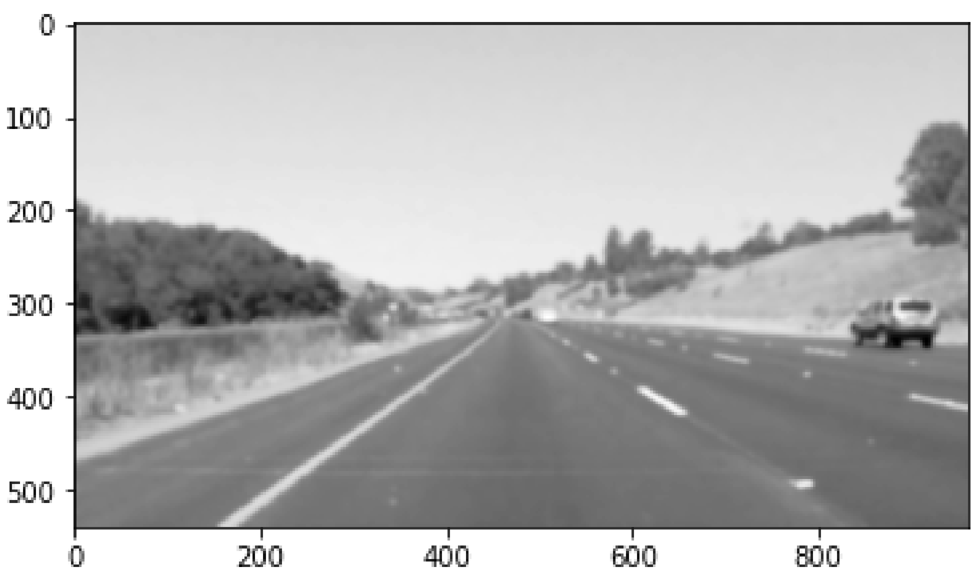

0
100
200
300
400
500
0
200
400
600
800

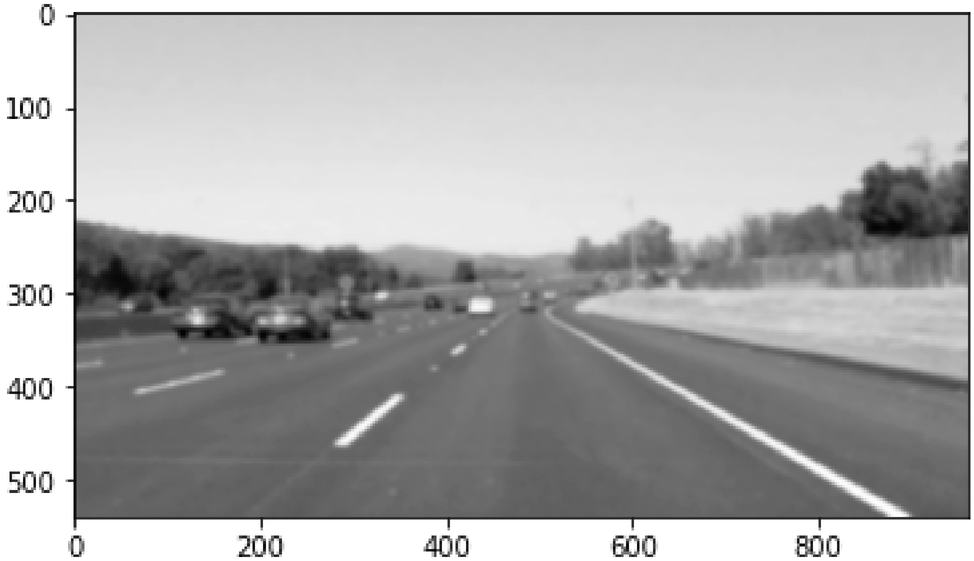

0
100
200
300
400
500
0
200
400
600
800

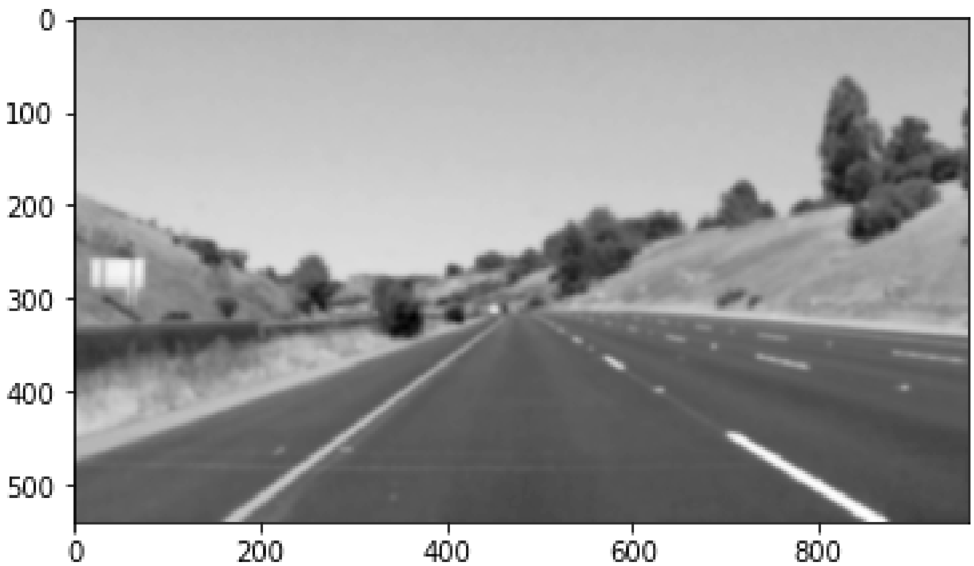

0
100
200
300
400
500
0
200
400
600
800

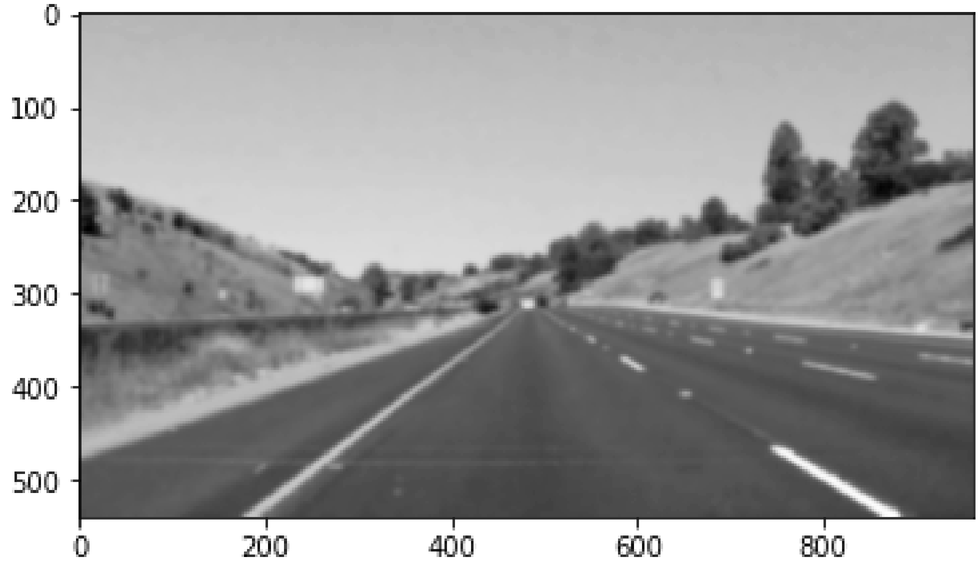

0
100
200
300
400
500
0
200
400
600
800

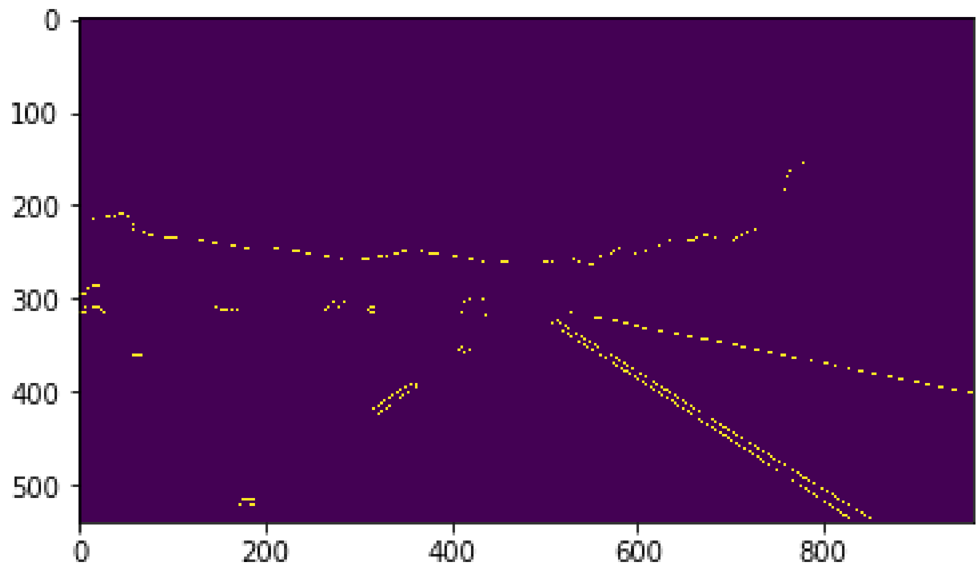

0
100
200
300
400
500
0
200
400
600
800

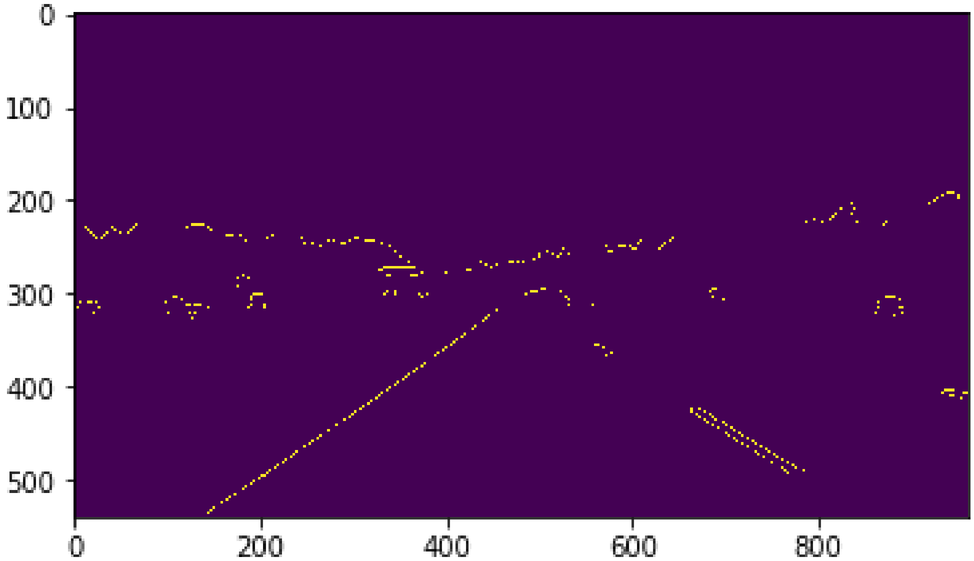

0
100
200
300
400
500
0
200
400
600
800


0
100
200
300
400
500
0
200
400
600
800

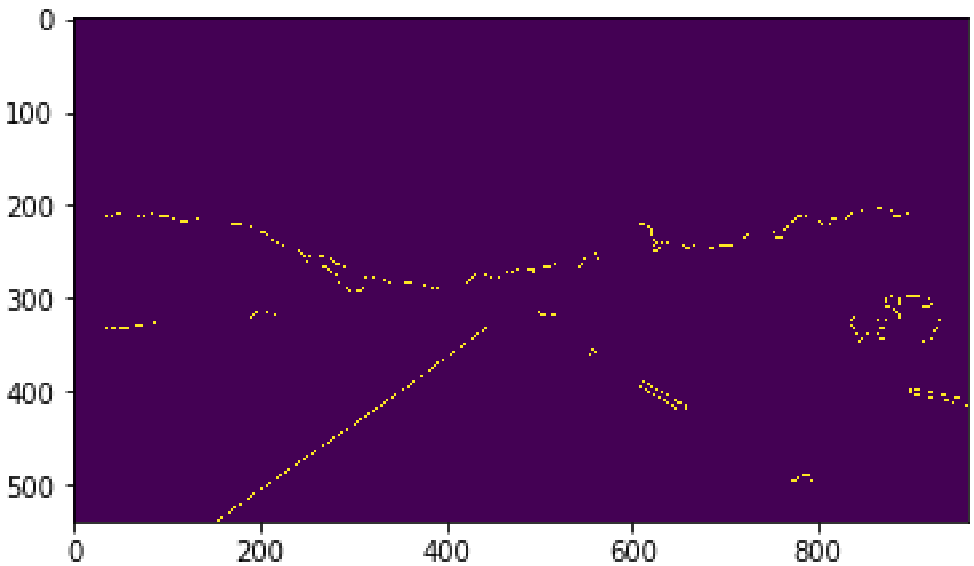

0
100
200
300
400
500
0
200
400
600
800


0
100
200
300
400
500
0
200
400
600
800

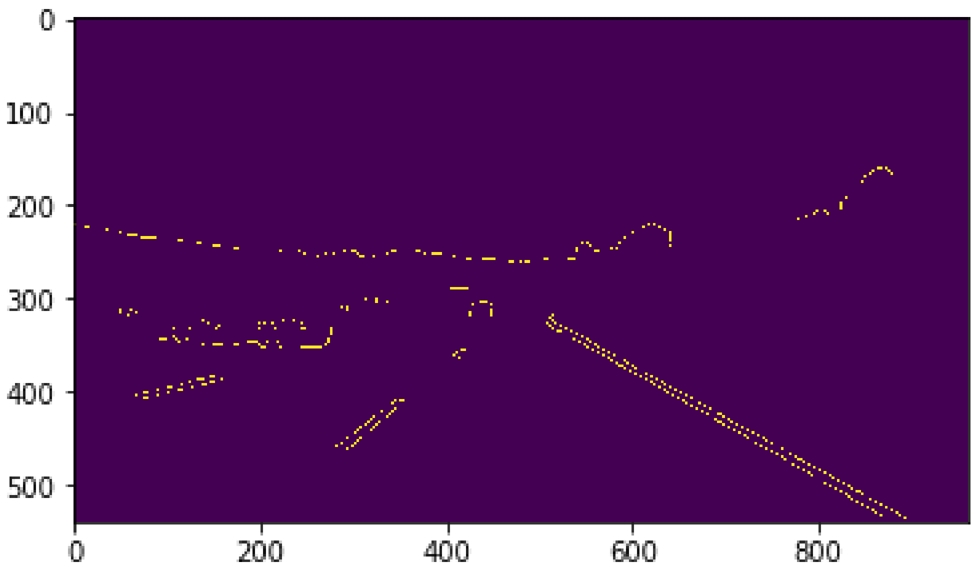

0
100
200
300
400
500
0
200
400
600
800

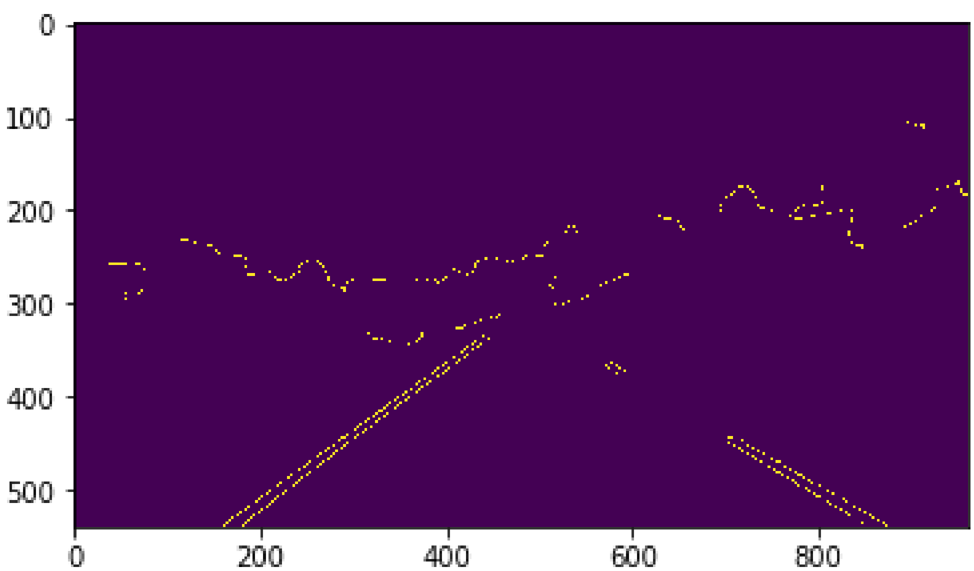


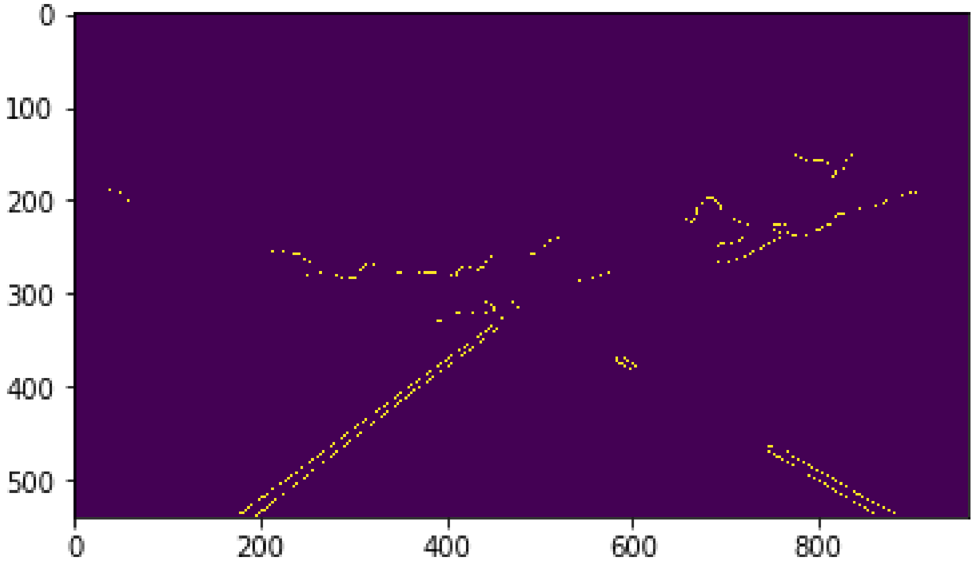
0
100
200
300
400
500
0
200
400
600
800

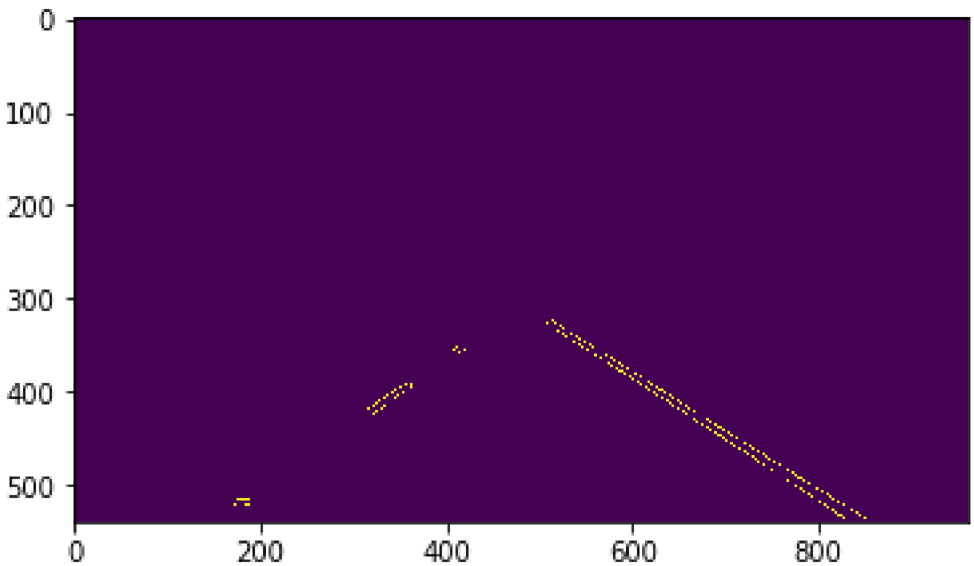


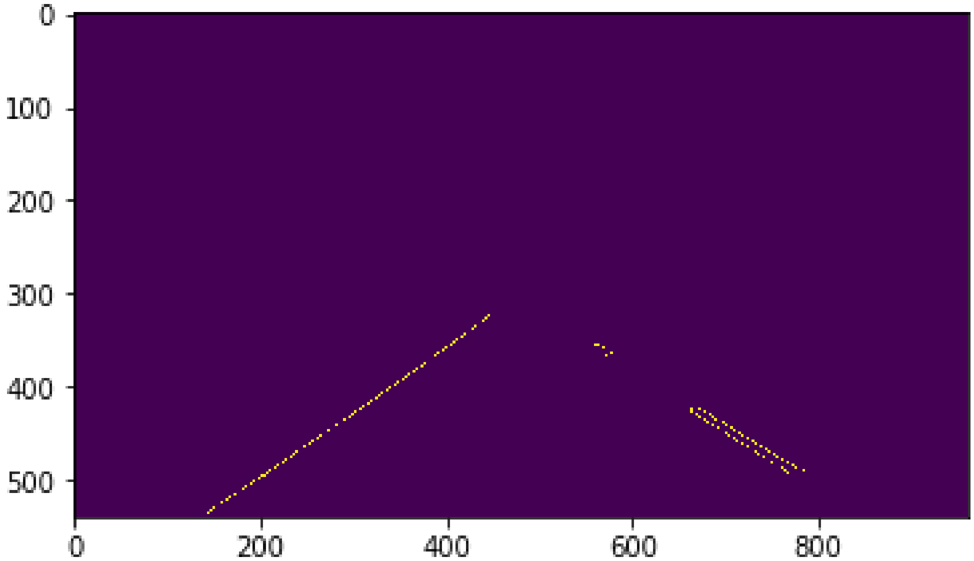
0
100
200
300
400
500
0
200
400
600
800

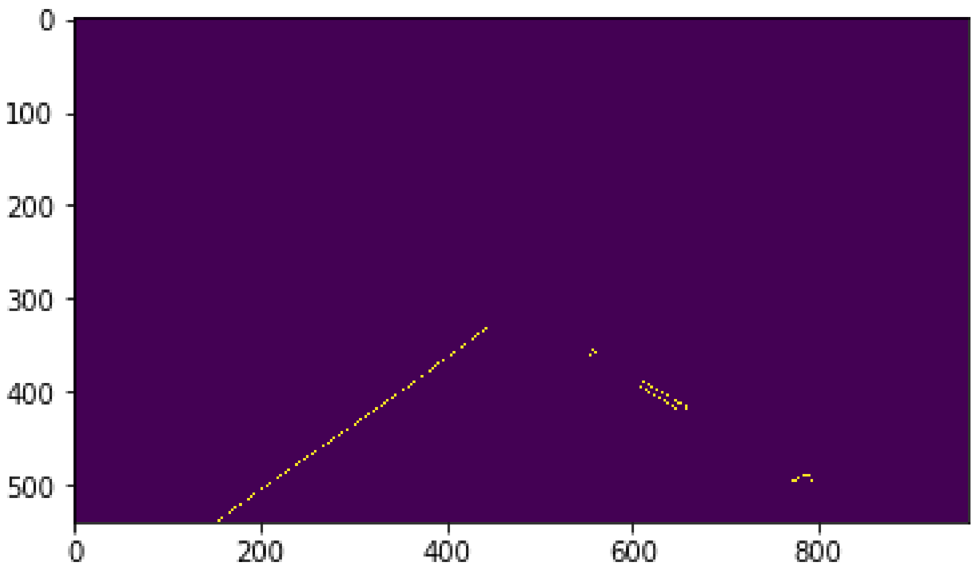

0
100
200
300
400
500
0
200
400
600
800

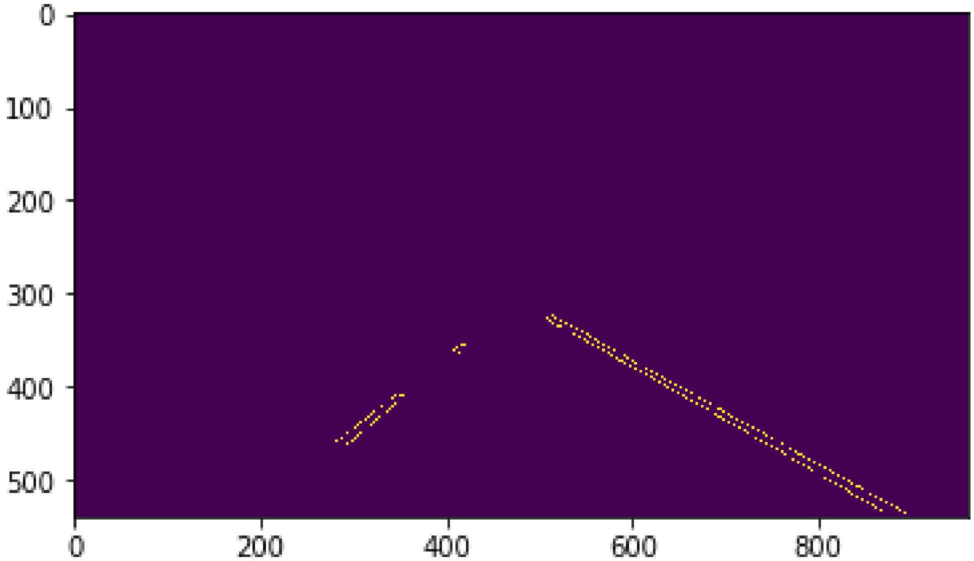

0
100
200
300
400
500
0
200
400
600
800

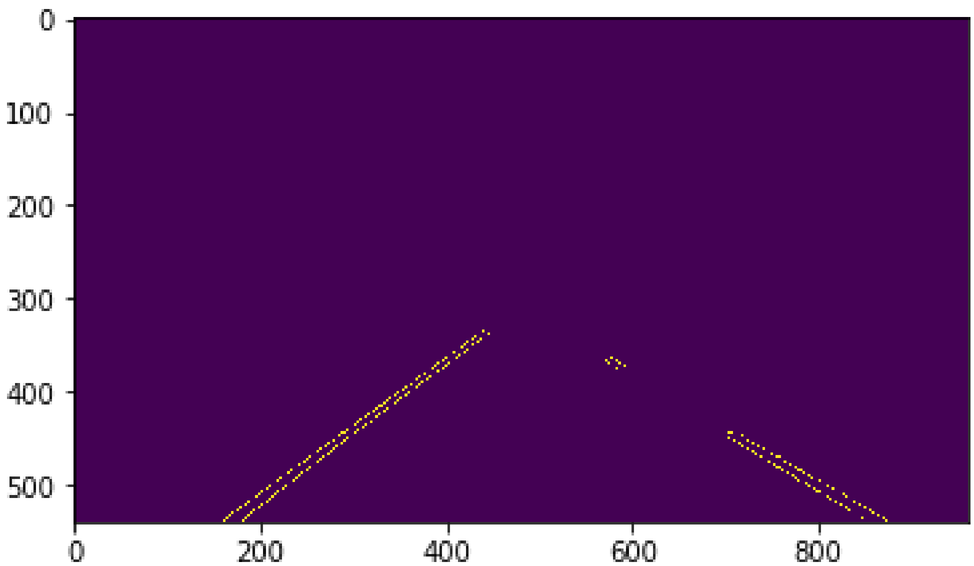

0
100
200
300
400
500
0
200
400
600
800

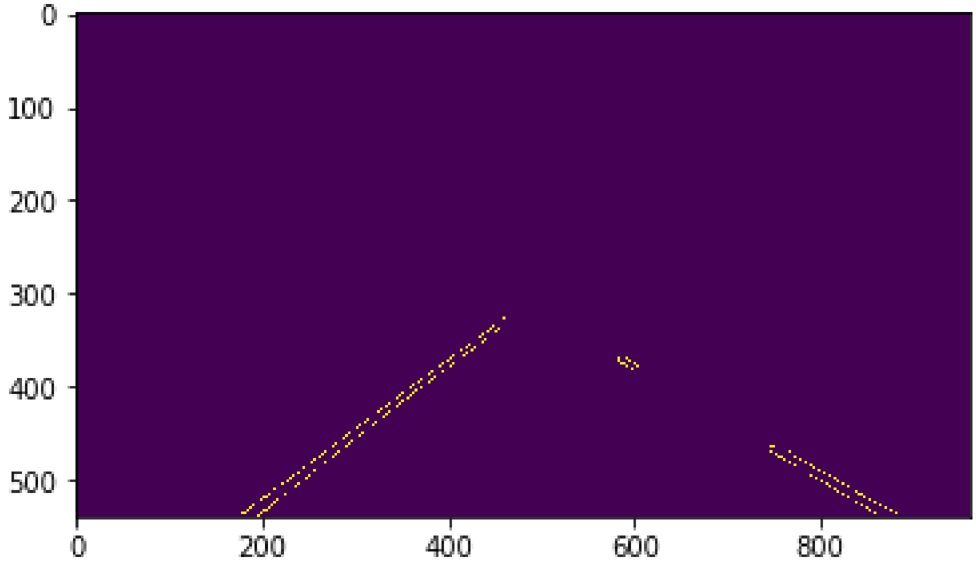

0
100
200
300
400
500
0
200
400
600
800


0
100
200
300
400
500
0
200
400
600
800

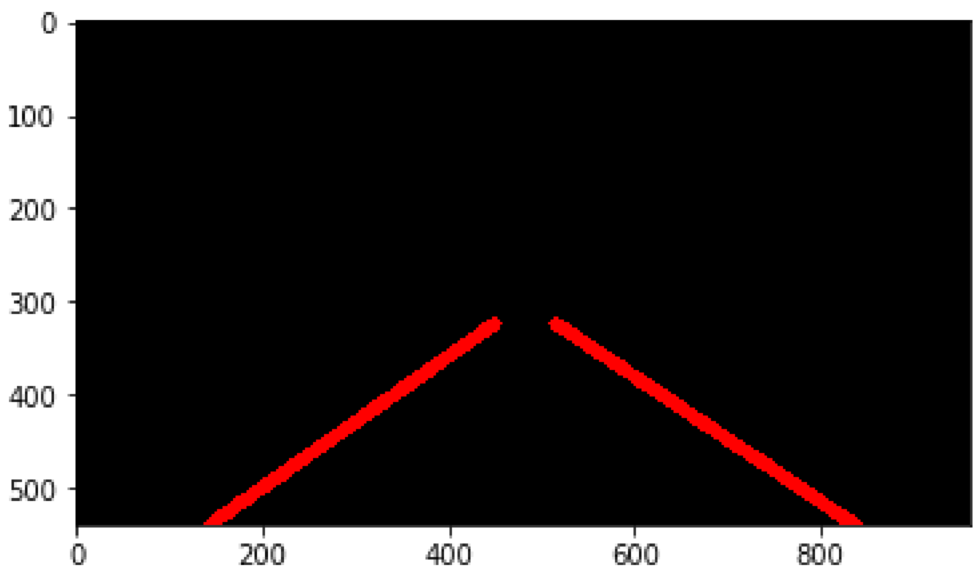

0
100
200
300
400
500
0
200
400
600
800


0
100
200
300
400
500
0
200
400
600
800

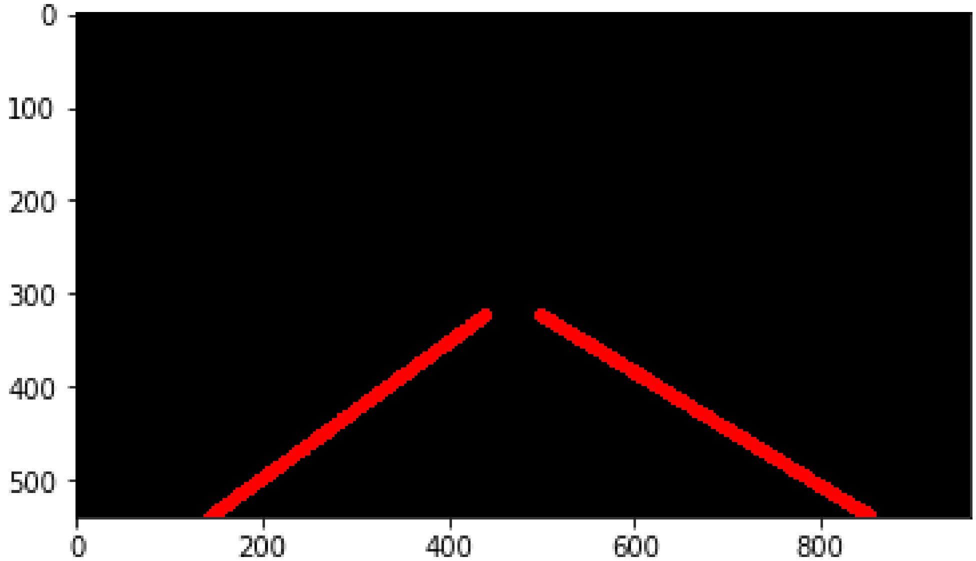

0
100
200
300
400
500
0
200
400
600
800

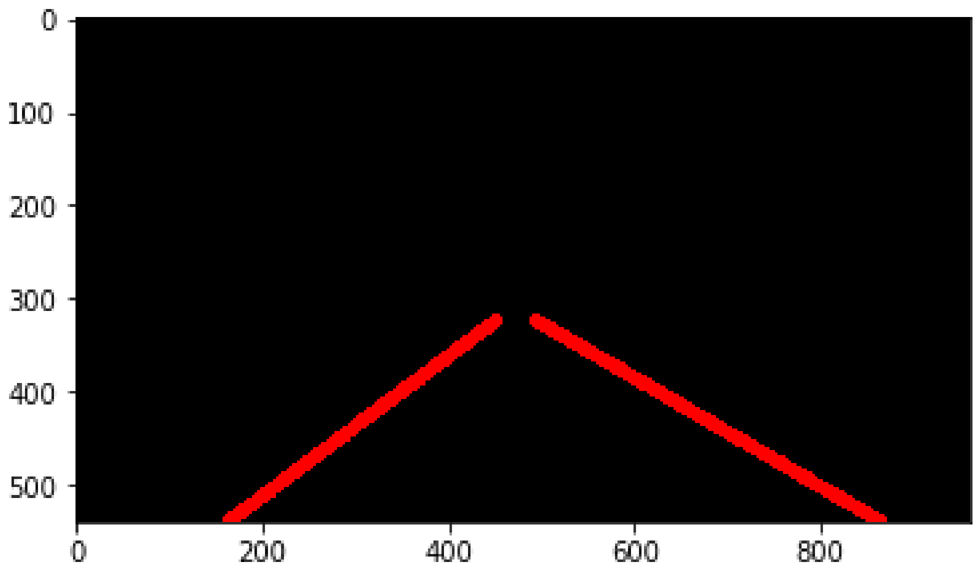


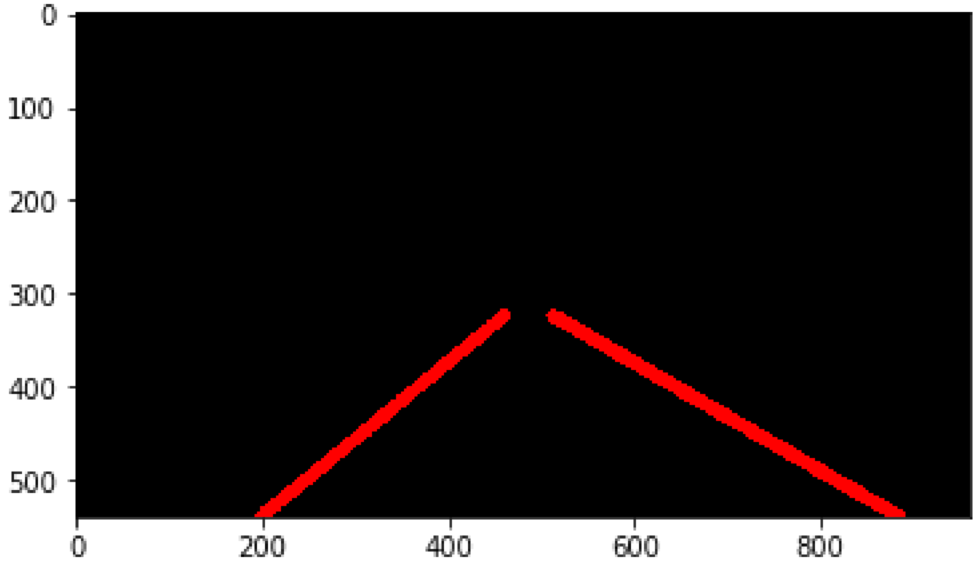
0
100
200
300
400
500
0
200
400
600
800



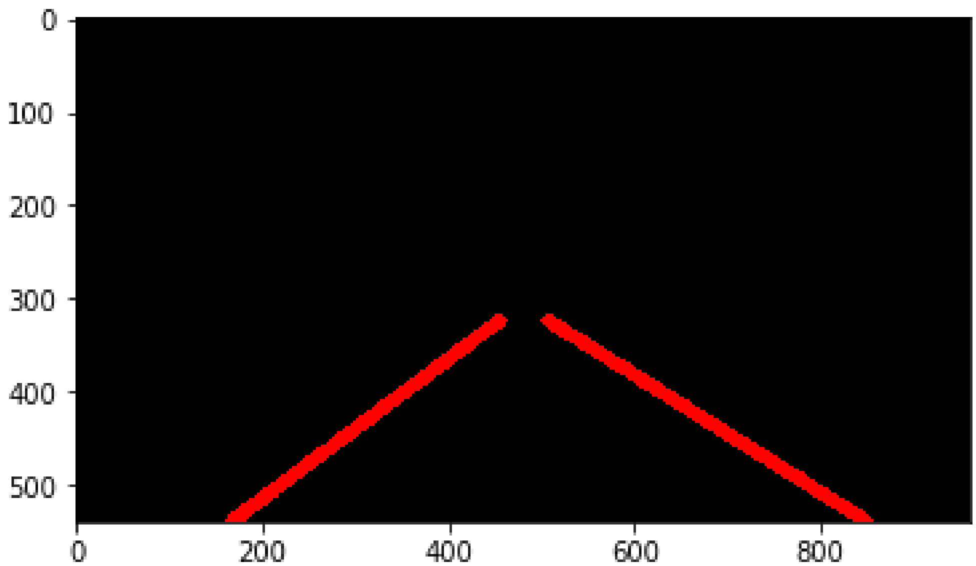
0
100
200
300
400
500
0
200
400
600
800


0
100
200
300
400
500
0
200
400
600
800

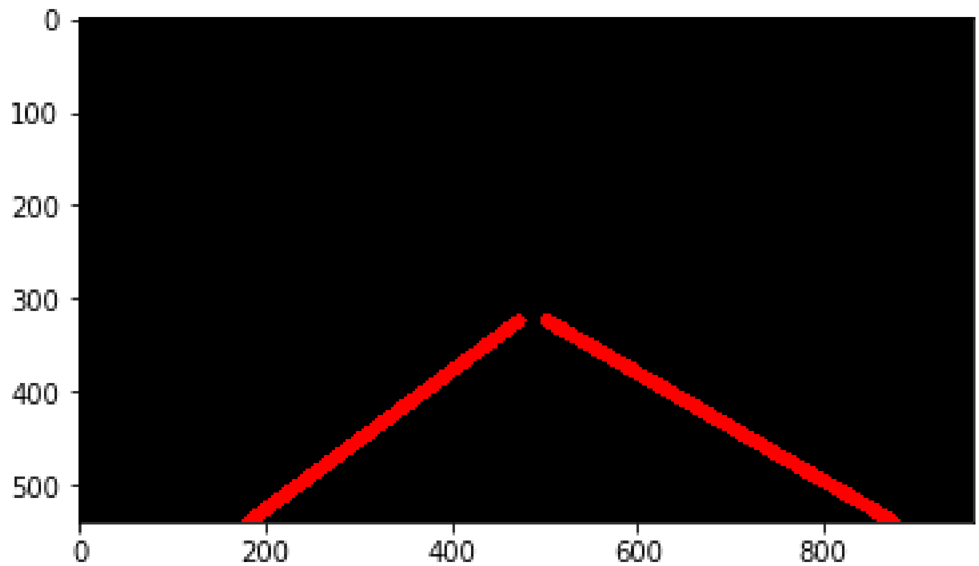

0
100
200
300
400
500
0
200
400
600
800



0
100
200
300
400
500
0
200
400
600
800



0
100
200
300
400
500
0
200
400
600
800



1.8 Test on Videos

You know what's cooler than drawing lanes over images? Drawing lanes over video!

We can test our solution on two provided videos:

`solidWhiteRight.mp4`

`solidYellowLeft.mp4`

Note: if you get an import error when you run the next cell, try changing your kernel (select the Kernel menu above –> Change Kernel). Still have problems? Try relaunching Jupyter Notebook from the terminal prompt. Also, consult the forums for more troubleshooting tips.

If you get an error that looks like this:

```
NeedDownloadError: Need ffmpeg exe.
You can download it by calling:
imageio.plugins.ffmpeg.download()
```

Follow the instructions in the error message and check out this forum post for more troubleshooting tips across operating systems.

```
In [225]: # Import everything needed to edit/save/watch video clips
          from moviepy.editor import VideoFileClip
          from IPython.display import HTML
```

```
In [226]: 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)
```

```
            # 1. Convert to grayscale
            gray_img = grayscale(image)

            # 2. Apply Gaussian blur
            kernel_size = 5
            blurred_img = gaussian_blur(gray_img,kernel_size)

            # 3. Canny edges
            canny_low_threshold = 50
            canny_high_threshold = 120
            edges_img = canny(blurred_img,canny_low_threshold,\
                              canny_high_threshold)

            # 4. Region mask
            # The masked region is a quadrangle from
            # bottom left corner,
            # 45% of left and 60% of height
            # 55% of right and 60% of height
            # bottom right corner
            # if we go beyond 60% the lines intersect and
            # in the corners we have lines in the middle of it.

            x = edges_img.shape[1]
            y = edges_img.shape[0]
            vertices = np.array([[(x*0.,y),(x*.45, y*.6), \
                                  (x*.55, y*.6), (x,y)]], dtype=np.int32)
            masked_img = region_of_interest(edges_img, vertices)

            # 5. Hough lines

            hough_rho = 3
            hough_theta = np.pi/180
            hough_threshold = 70
            hough_min_line_length = 70
            hough_max_line_gap = 250
            hough_img = hough_lines(masked_img,hough_rho,hough_theta,\
                                    hough_threshold,hough_min_line_length,\
                                    hough_max_line_gap)

            # 6. Combine original image and hough image and return it

            result = weighted_img(hough_img,image)

            return result
```

Let's try the one with the solid white lane on the right first ...

```
In [227]: white_output = 'test_videos_output/solidWhiteRight.mp4'
          ## To speed up the testing process you may want to try your pipeline on a shorter subc
```

```
## To do so add .subclip(start_second,end_second) to the end of the line below
## Where start_second and end_second are integer values representing the start and end
## You may also uncomment the following line for a subclip of the first 5 seconds
##clip1 = VideoFileClip("test_videos/solidWhiteRight.mp4").subclip(0,5)
clip1 = VideoFileClip("test_videos/solidWhiteRight.mp4")
white_clip = clip1.fl_image(process_image) #NOTE: this function expects color images!!
%time white_clip.write_videofile(white_output, audio=False)
```

```
[MoviePy] >>>> Building video test_videos_output/solidWhiteRight.mp4
[MoviePy] Writing video test_videos_output/solidWhiteRight.mp4
```

```
  0%|          | 0/222 [00:00<?, ?it/s]

  3%|          | 6/222 [00:00<00:04, 52.45it/s]

  6%|          | 13/222 [00:00<00:03, 57.72it/s]

  9%|          | 20/222 [00:00<00:03, 58.86it/s]

 12%|         | 26/222 [00:00<00:03, 58.96it/s]

 15%|         | 33/222 [00:00<00:03, 60.22it/s]

 18%|         | 40/222 [00:00<00:02, 60.99it/s]

 21%|         | 47/222 [00:00<00:03, 56.53it/s]

 24%|        | 53/222 [00:00<00:02, 56.84it/s]

 27%|        | 59/222 [00:01<00:02, 56.55it/s]

 30%|        | 66/222 [00:01<00:02, 56.97it/s]
```

```
32%|      | 72/222 [00:01<00:02, 57.11it/s]

36%|      | 79/222 [00:01<00:02, 57.50it/s]

38%|      | 85/222 [00:01<00:02, 57.28it/s]

41%|      | 91/222 [00:01<00:02, 57.29it/s]

44%|     | 97/222 [00:01<00:02, 57.45it/s]

47%|     | 104/222 [00:01<00:02, 57.82it/s]

50%|     | 110/222 [00:01<00:01, 57.91it/s]

52%|    | 116/222 [00:02<00:01, 57.83it/s]

55%|    | 122/222 [00:02<00:01, 57.45it/s]

58%|    | 129/222 [00:02<00:01, 57.63it/s]

61%|    | 135/222 [00:02<00:01, 57.62it/s]

64%|   | 141/222 [00:02<00:01, 57.69it/s]

67%|   | 148/222 [00:02<00:01, 57.81it/s]

69%|   | 154/222 [00:02<00:01, 57.70it/s]

72%|  | 160/222 [00:02<00:01, 57.71it/s]

75%|  | 166/222 [00:02<00:00, 57.78it/s]
```

```
 77%|  | 172/222 [00:02<00:00, 57.81it/s]


 81%|  | 179/222 [00:03<00:00, 57.97it/s]


 83%| | 185/222 [00:03<00:00, 57.91it/s]


 86%| | 191/222 [00:03<00:00, 57.94it/s]


 89%| | 197/222 [00:03<00:00, 57.88it/s]


 92%|| 204/222 [00:03<00:00, 57.99it/s]


 95%|| 211/222 [00:03<00:00, 58.17it/s]


 98%|| 218/222 [00:03<00:00, 58.22it/s]


100%|| 221/222 [00:03<00:00, 58.23it/s]

[MoviePy] Done.
[MoviePy] >>>> Video ready: test_videos_output/solidWhiteRight.mp4

CPU times: user 48.7 s, sys: 408 ms, total: 49.1 s
Wall time: 4.14 s
```

Play the video inline, or if you prefer find the video in your filesystem (should be in the same directory) and play it in your video player of choice.

```
In [228]: HTML("""
          <video width="960" height="540" controls>
            <source src="{0}">
          </video>
          """.format(white_output))
```

```
Out[228]: <IPython.core.display.HTML object>
```

1.9 Improve the draw_lines() function

At this point, if you were successful with making the pipeline and tuning parameters, you probably have the Hough line segments drawn onto the road, but what about identifying the full extent of the lane and marking it clearly as in the example video (P1_example.mp4)? Think

about defining a line to run the full length of the visible lane based on the line segments you identified with the Hough Transform. As mentioned previously, try to average and/or extrapolate the line segments you've detected to map out the full extent of the lane lines. You can see an example of the result you're going for in the video "P1_example.mp4".

Go back and modify your draw_lines function accordingly and try re-running your pipeline. The new output should draw a single, solid line over the left lane line and a single, solid line over the right lane line. The lines should start from the bottom of the image and extend out to the top of the region of interest.

Now for the one with the solid yellow lane on the left. This one's more tricky!

```
In [229]: yellow_output = 'test_videos_output/solidYellowLeft.mp4'
          ## To speed up the testing process you may want to try your pipeline on a shorter subc
          ## To do so add .subclip(start_second,end_second) to the end of the line below
          ## Where start_second and end_second are integer values representing the start and end
          ## You may also uncomment the following line for a subclip of the first 5 seconds
          ##clip2 = VideoFileClip('test_videos/solidYellowLeft.mp4').subclip(0,5)
          clip2 = VideoFileClip('test_videos/solidYellowLeft.mp4')
          yellow_clip = clip2.fl_image(process_image)
          %time yellow_clip.write_videofile(yellow_output, audio=False)
```

```
[MoviePy] >>>> Building video test_videos_output/solidYellowLeft.mp4
[MoviePy] Writing video test_videos_output/solidYellowLeft.mp4



  0%|          | 0/682 [00:00<?, ?it/s]


  1%|          | 6/682 [00:00<00:13, 51.43it/s]


  2%|         | 13/682 [00:00<00:11, 56.57it/s]


  3%|         | 20/682 [00:00<00:11, 59.08it/s]


  4%|         | 27/682 [00:00<00:10, 59.85it/s]


  5%|         | 34/682 [00:00<00:10, 60.24it/s]


  6%|         | 41/682 [00:00<00:10, 60.67it/s]
```

```
  7%|          | 47/682 [00:00<00:11, 56.02it/s]

  8%|          | 54/682 [00:00<00:11, 56.79it/s]

  9%|          | 61/682 [00:01<00:10, 57.35it/s]

 10%|          | 68/682 [00:01<00:10, 57.59it/s]

 11%|          | 74/682 [00:01<00:10, 57.44it/s]

 12%|         | 81/682 [00:01<00:10, 57.82it/s]

 13%|         | 88/682 [00:01<00:10, 58.14it/s]

 14%|         | 94/682 [00:01<00:10, 58.11it/s]

 15%|         | 100/682 [00:01<00:10, 58.19it/s]

 16%|         | 106/682 [00:01<00:09, 58.05it/s]

 17%|         | 113/682 [00:01<00:09, 58.37it/s]

 18%|         | 120/682 [00:02<00:09, 58.53it/s]

 19%|         | 127/682 [00:02<00:09, 58.35it/s]

 20%|         | 134/682 [00:02<00:09, 58.63it/s]

 21%|         | 141/682 [00:02<00:09, 58.54it/s]

 22%|        | 147/682 [00:02<00:09, 58.27it/s]
```

```
22%|          | 153/682 [00:02<00:09, 58.24it/s]

23%|          | 159/682 [00:02<00:08, 58.25it/s]

24%|          | 165/682 [00:02<00:08, 58.16it/s]

25%|          | 171/682 [00:02<00:08, 58.21it/s]

26%|          | 178/682 [00:03<00:08, 58.37it/s]

27%|          | 185/682 [00:03<00:08, 58.46it/s]

28%|          | 191/682 [00:03<00:08, 58.47it/s]

29%|          | 197/682 [00:03<00:08, 58.42it/s]

30%|          | 203/682 [00:03<00:08, 58.30it/s]

31%|          | 210/682 [00:03<00:08, 58.45it/s]

32%|         | 216/682 [00:03<00:07, 58.39it/s]

33%|         | 222/682 [00:03<00:07, 58.42it/s]

34%|         | 229/682 [00:03<00:07, 58.56it/s]

35%|         | 236/682 [00:04<00:07, 58.61it/s]

36%|         | 243/682 [00:04<00:07, 58.76it/s]

37%|         | 250/682 [00:04<00:07, 58.80it/s]
```

```
38%|      | 257/682 [00:04<00:07, 58.80it/s]

39%|      | 264/682 [00:04<00:07, 58.74it/s]

40%|      | 271/682 [00:04<00:06, 58.76it/s]

41%|      | 278/682 [00:04<00:06, 58.83it/s]

42%|     | 285/682 [00:04<00:06, 58.82it/s]

43%|     | 291/682 [00:04<00:06, 58.78it/s]

44%|     | 297/682 [00:05<00:06, 58.76it/s]

44%|     | 303/682 [00:05<00:06, 58.69it/s]

45%|     | 310/682 [00:05<00:06, 58.80it/s]

46%|     | 317/682 [00:05<00:06, 58.89it/s]

48%|     | 324/682 [00:05<00:06, 58.80it/s]

48%|     | 330/682 [00:05<00:05, 58.76it/s]

49%|     | 337/682 [00:05<00:05, 58.83it/s]

50%|     | 343/682 [00:05<00:05, 58.82it/s]

51%|     | 349/682 [00:05<00:05, 58.83it/s]

52%|    | 355/682 [00:06<00:05, 58.77it/s]
```

```
53%|     | 361/682 [00:06<00:05, 58.74it/s]

54%|     | 367/682 [00:06<00:05, 58.58it/s]

55%|     | 374/682 [00:06<00:05, 58.64it/s]

56%|     | 381/682 [00:06<00:05, 58.72it/s]

57%|     | 387/682 [00:06<00:05, 58.71it/s]

58%|     | 393/682 [00:06<00:04, 58.71it/s]

59%|     | 400/682 [00:06<00:04, 58.75it/s]

60%|     | 407/682 [00:06<00:04, 58.81it/s]

61%|     | 414/682 [00:07<00:04, 58.77it/s]

62%|    | 421/682 [00:07<00:04, 58.85it/s]

63%|    | 428/682 [00:07<00:04, 58.95it/s]

64%|    | 435/682 [00:07<00:04, 59.05it/s]

65%|    | 442/682 [00:07<00:04, 59.00it/s]

66%|    | 449/682 [00:07<00:03, 59.07it/s]

67%|    | 456/682 [00:07<00:03, 59.13it/s]

68%|    | 463/682 [00:07<00:03, 59.16it/s]
```

```
69%|   | 470/682 [00:07<00:03, 59.17it/s]

70%|   | 477/682 [00:08<00:03, 59.24it/s]

71%|   | 484/682 [00:08<00:03, 59.28it/s]

72%|  | 491/682 [00:08<00:03, 59.16it/s]

73%|  | 498/682 [00:08<00:03, 59.17it/s]

74%|  | 505/682 [00:08<00:02, 59.20it/s]

75%|  | 511/682 [00:08<00:02, 59.09it/s]

76%|  | 517/682 [00:08<00:02, 59.09it/s]

77%|  | 523/682 [00:08<00:02, 58.96it/s]

78%|  | 529/682 [00:08<00:02, 58.95it/s]

78%|  | 535/682 [00:09<00:02, 58.95it/s]

79%|  | 541/682 [00:09<00:02, 58.96it/s]

80%|  | 547/682 [00:09<00:02, 58.91it/s]

81%|  | 554/682 [00:09<00:02, 58.97it/s]

82%| | 560/682 [00:09<00:02, 58.95it/s]

83%| | 566/682 [00:09<00:01, 58.96it/s]
```

```
84%| | 573/682 [00:09<00:01, 58.98it/s]

85%| | 579/682 [00:09<00:01, 58.97it/s]

86%| | 585/682 [00:09<00:01, 58.98it/s]

87%| | 591/682 [00:10<00:01, 58.95it/s]

88%| | 598/682 [00:10<00:01, 58.97it/s]

89%| | 605/682 [00:10<00:01, 59.02it/s]

90%| | 612/682 [00:10<00:01, 59.01it/s]

91%| | 619/682 [00:10<00:01, 59.08it/s]

92%|| 626/682 [00:10<00:00, 59.09it/s]

93%|| 633/682 [00:10<00:00, 59.12it/s]

94%|| 640/682 [00:10<00:00, 59.15it/s]

95%|| 647/682 [00:10<00:00, 59.20it/s]

96%|| 654/682 [00:11<00:00, 59.15it/s]

97%|| 661/682 [00:11<00:00, 59.15it/s]

98%|| 668/682 [00:11<00:00, 59.16it/s]

99%|| 674/682 [00:11<00:00, 59.11it/s]
```

```
100%|| 681/682 [00:11<00:00, 59.14it/s]



[MoviePy] Done.
[MoviePy] >>>> Video ready: test_videos_output/solidYellowLeft.mp4

CPU times: user 2min 29s, sys: 1.48 s, total: 2min 30s
Wall time: 11.9 s
```

```
In [230]: HTML("""
          <video width="960" height="540" controls>
            <source src="{0}">
          </video>
          """.format(yellow_output))

Out[230]: <IPython.core.display.HTML object>
```

1.10 Writeup and Submission

If you're satisfied with your video outputs, it's time to make the report writeup in a pdf or markdown file. Once you have this Ipython notebook ready along with the writeup, it's time to submit for review! Here is a link to the writeup template file.

1.11 Optional Challenge

Try your lane finding pipeline on the video below. Does it still work? Can you figure out a way to make it more robust? If you're up for the challenge, modify your pipeline so it works with this video and submit it along with the rest of your project!

```
In [231]: challenge_output = 'test_videos_output/challenge.mp4'
          ## To speed up the testing process you may want to try your pipeline on a shorter subc
          ## To do so add .subclip(start_second,end_second) to the end of the line below
          ## Where start_second and end_second are integer values representing the start and end
          ## You may also uncomment the following line for a subclip of the first 5 seconds
          ##clip3 = VideoFileClip('test_videos/challenge.mp4').subclip(0,5)
          clip3 = VideoFileClip('test_videos/challenge.mp4')
          challenge_clip = clip3.fl_image(process_image)
          %time challenge_clip.write_videofile(challenge_output, audio=False)

[MoviePy] >>>> Building video test_videos_output/challenge.mp4
[MoviePy] Writing video test_videos_output/challenge.mp4
```

```
  0%|          | 0/251 [00:00<?, ?it/s]
  2%|          | 4/251 [00:00<00:07, 33.13it/s]
  3%|          | 8/251 [00:00<00:06, 35.34it/s]
  5%|          | 12/251 [00:00<00:06, 36.52it/s]
  6%|          | 16/251 [00:00<00:06, 36.36it/s]
  8%|          | 20/251 [00:00<00:06, 36.34it/s]
 10%|          | 24/251 [00:00<00:06, 36.47it/s]
 11%|          | 28/251 [00:00<00:06, 36.21it/s]
 13%|         | 32/251 [00:00<00:06, 36.27it/s]
 14%|         | 36/251 [00:00<00:05, 36.15it/s]
 16%|         | 40/251 [00:01<00:05, 35.81it/s]
 18%|         | 44/251 [00:01<00:05, 35.84it/s]
 19%|         | 48/251 [00:01<00:05, 34.69it/s]
 21%|         | 52/251 [00:01<00:05, 34.72it/s]
 22%|        | 56/251 [00:01<00:05, 34.82it/s]
 24%|        | 60/251 [00:01<00:05, 34.76it/s]
```

```
25%|        | 64/251 [00:01<00:05, 34.64it/s]

27%|        | 68/251 [00:01<00:05, 34.66it/s]

29%|        | 72/251 [00:02<00:05, 34.73it/s]

30%|        | 76/251 [00:02<00:05, 34.46it/s]

32%|       | 80/251 [00:02<00:04, 34.40it/s]

33%|       | 84/251 [00:02<00:04, 34.31it/s]

35%|       | 88/251 [00:02<00:04, 34.35it/s]

37%|       | 92/251 [00:02<00:04, 34.05it/s]

38%|       | 96/251 [00:02<00:04, 34.01it/s]

40%|       | 100/251 [00:02<00:04, 33.89it/s]

41%|      | 104/251 [00:03<00:04, 33.76it/s]

43%|      | 108/251 [00:03<00:04, 33.66it/s]

45%|      | 112/251 [00:03<00:04, 33.54it/s]

46%|      | 116/251 [00:03<00:04, 33.65it/s]

48%|      | 120/251 [00:03<00:03, 33.71it/s]

49%|      | 124/251 [00:03<00:03, 33.61it/s]
```

```
51%|     | 128/251 [00:03<00:03, 33.64it/s]

53%|    | 132/251 [00:03<00:03, 33.58it/s]

54%|    | 136/251 [00:04<00:03, 33.49it/s]

56%|    | 140/251 [00:04<00:03, 33.39it/s]

57%|    | 144/251 [00:04<00:03, 33.21it/s]

59%|    | 148/251 [00:04<00:03, 32.93it/s]

60%|    | 151/251 [00:04<00:03, 32.85it/s]

62%|   | 155/251 [00:04<00:02, 32.85it/s]

63%|   | 159/251 [00:04<00:02, 32.87it/s]

65%|   | 163/251 [00:04<00:02, 32.99it/s]

67%|   | 167/251 [00:05<00:02, 33.07it/s]

68%|   | 171/251 [00:05<00:02, 33.14it/s]

70%|   | 175/251 [00:05<00:02, 33.19it/s]

71%|  | 179/251 [00:05<00:02, 33.14it/s]

73%|  | 183/251 [00:05<00:02, 33.13it/s]

75%|  | 187/251 [00:05<00:01, 33.20it/s]
```

```
 76%|  | 191/251 [00:05<00:01, 33.25it/s]

 78%|  | 195/251 [00:05<00:01, 33.29it/s]

 79%|  | 199/251 [00:05<00:01, 33.33it/s]

 81%|  | 203/251 [00:06<00:01, 33.32it/s]

 82%| | 207/251 [00:06<00:01, 33.24it/s]

 84%| | 211/251 [00:06<00:01, 33.31it/s]

 86%| | 215/251 [00:06<00:01, 33.25it/s]

 87%| | 219/251 [00:06<00:00, 33.24it/s]

 89%| | 223/251 [00:06<00:00, 33.17it/s]

 90%| | 227/251 [00:06<00:00, 33.21it/s]

 92%|| 231/251 [00:06<00:00, 33.27it/s]

 94%|| 235/251 [00:07<00:00, 33.31it/s]

 95%|| 239/251 [00:07<00:00, 33.28it/s]

 97%|| 243/251 [00:07<00:00, 33.27it/s]

 98%|| 247/251 [00:07<00:00, 33.29it/s]

100%|| 251/251 [00:07<00:00, 33.35it/s]
```

```
[MoviePy] Done.
[MoviePy] >>>> Video ready: test_videos_output/challenge.mp4

CPU times: user 1min 4s, sys: 817 ms, total: 1min 4s
Wall time: 8.05 s
```

```
In [232]: HTML("""
          <video width="960" height="540" controls>
            <source src="{0}">
          </video>
          """.format(challenge_output))

Out[232]: <IPython.core.display.HTML object>
```

1.11.1 Reflection

In some places of the during the videos both project and challenge, we see lines crossing the actual lane, around the bonnet or around the road partition. These are happening when there is shade or the lanes are curved or the lane lines are lighter.

In addition, since we are predominently using the slope and intercept concepts, the lines will not be along the lane curvature. As i worked on this project and the advanced lanes project simultanesouly, i realized that there are far more sophisticated techniques we can use to improve this project.