P1_4

July 19, 2017

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, see this forum post for more troubleshooting tips.

1.2 Import Packages

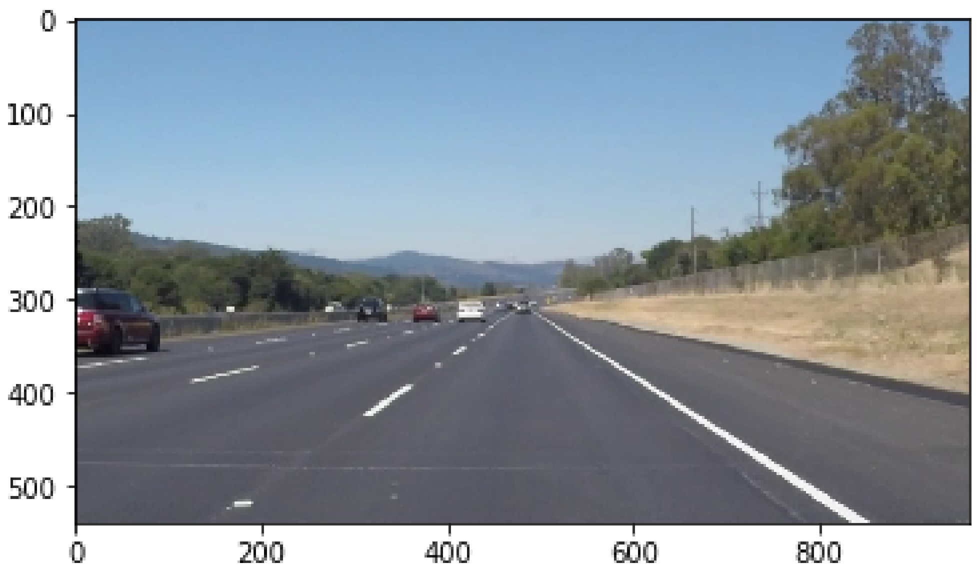
```
In [1]: #importing some useful packages
        import matplotlib.pyplot as plt
        import matplotlib.image as mpimg
        import numpy as np
        import cv2
        %matplotlib inline
```

1.3 Read in an Image

```
In [2]: #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)
        plt.imshow(image)  # if you wanted to show a single color channel image called 'gray', f
```

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

```
Out[2]: <matplotlib.image.AxesImage at 0x110f0ea20>
```



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 [4]: 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.
            """
            #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 inpu
            if len(img.shape) > 2:
                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 draw_lines(img, lines, color=[255, 0, 0], thickness=5):
    """
    NOTE: this is the function you might want to use as a starting point once you want t
    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
    """
    ysize, xsize, _ = img.shape
    threshold = 0.55 # ignore line segments near horizontal
    left_lines = []
    right_lines = []
    for line in lines:
        for x1,y1,x2,y2 in line:
            if x1!=x2:
                slop = float(y2-y1)/float(x2-x1+.0000001)
            else:
                slop = 999999.999 # set infinite
            # sort lines into the left and right lines
            if slop > threshold and x1 > xsize//2: # \
                right_lines += [[x1,y1],[x2,y2]]
            elif slop < -threshold and x1 < xsize//2:  # /
                left_lines += [[x1,y1],[x2,y2]]
    right = np.array(right_lines)
    left = np.array(left_lines)

    # fit lines: y = m*x + b; return slope m and constant b;
    right_m, right_b = np.polyfit(right[:,0], right[:,1], 1)
    left_m, left_b = np.polyfit(left[:,0], left[:,1], 1)
    # upper bound of the line: 60% of ysize; it's about 290;
    upper_bound = int(ysize*0.6)

    # find two points at both lines (x1,y1,x2,y2)
    left_line = (int((upper_bound - left_b)/left_m), upper_bound, int((ysize - left_b)/l
    right_line = (int((upper_bound - right_b)/right_m), upper_bound, int((ysize - right_

    # draw lines
    line = (left_line, right_line)
    for x1,y1,x2,y2 in line:
        cv2.line(img, (x1, y1), (x2, y2), color, thickness)

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_img = np.zeros((img.shape[0], img.shape[1], 3), dtype=np.uint8)
    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, , )

def vertices_shape(img):
    return np.array([[(-5,600),(500, 300), (550, 300), (960, 600)]], dtype=np.int32)
```

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 [5]: import os
        os.listdir("test_images/")

Out[5]: ['.DS_Store',
         'solidWhiteCurve.jpg',
         'solidWhiteRight.jpg',
         'solidYellowCurve.jpg',
         'solidYellowCurve2.jpg',
         'solidYellowLeft.jpg',
         'whiteCarLaneSwitch.jpg']
```

1.7 Build a Lane Finding Pipeline

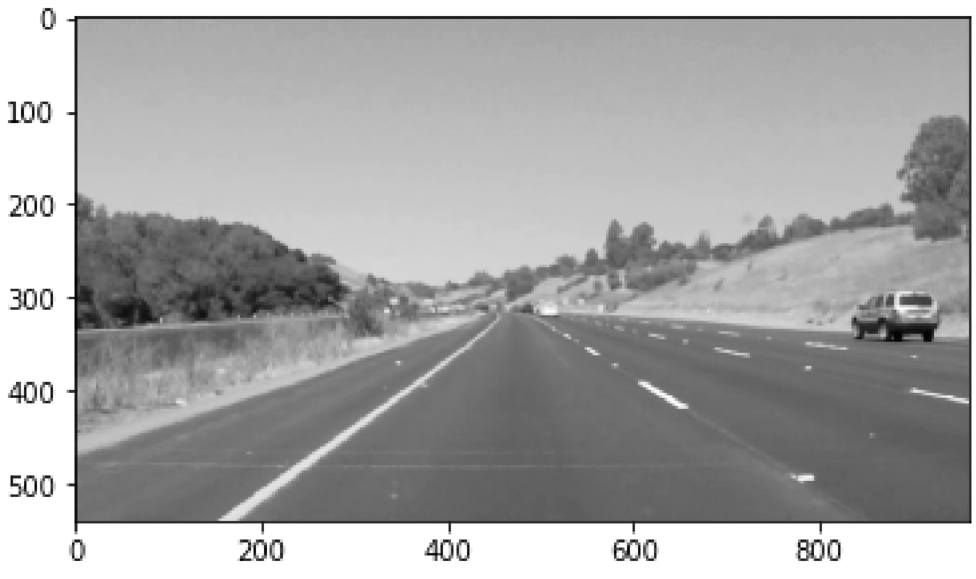
Build the pipeline and run your solution on all test_images. Make copies into the `test_images_output` directory, and you can use the images in your writeup report.

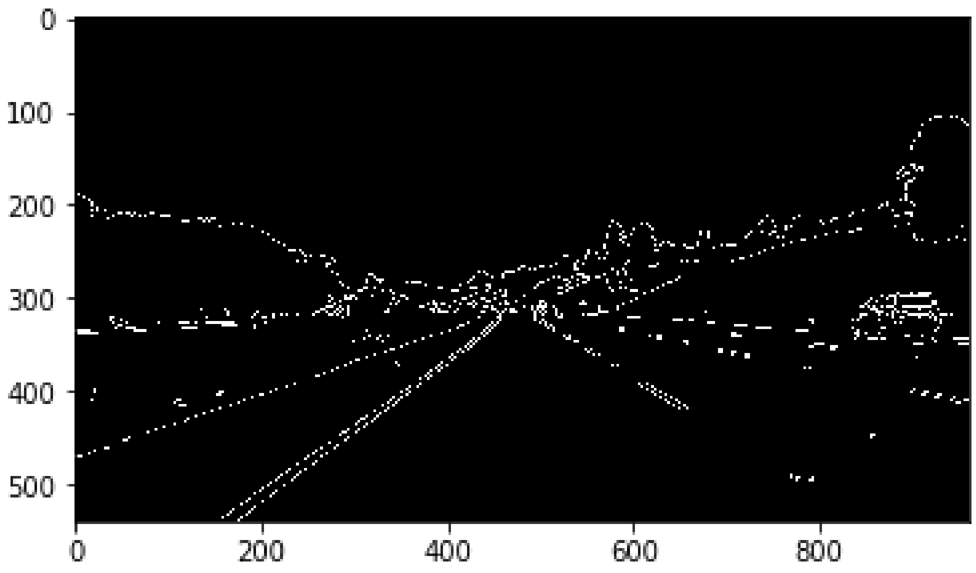
Try tuning the various parameters, especially the low and high Canny thresholds as well as the Hough lines parameters.

```
In [6]: # TODO: Build your pipeline that will draw lane lines on the test_images
        # then save them to the test_images directory.

        ## -- Parameter Setting --
        # define kernel_size
        kernel_size = 5 # (be an odd number)
        # define parameters for Canny
        low_threshold = 50
        high_threshold = 150
        # Hough transform
        rho = 1 # distance of the line from the origin
        theta = (2*np.pi/180) # the angle away from horizontal
        threshold = 20 # minimum number of votes (intersections in a given grid cell)
        min_line_length = 5 # lengths to be accepted in the output
        max_line_gap = 20 # max distance (in pixels) btw segments that are allowed to be connect

In [7]: def lane_lines_pipelines(img):
            # convert the image to grayscale
            gray = grayscale(img)
            # Gaussian smoothing
            blur_gray = gaussian_blur(gray, kernel_size=kernel_size)
            # Canny edges detection
            edges = canny(blur_gray, low_threshold=low_threshold, high_threshold=high_threshold)
            # Trapezoid shape adjustment
            vertices = vertices_shape(edges)
            # masked region of interest
            masked_edges = region_of_interest(edges, vertices)
            # Hough transform on the edges detection of the image
            lines_img = hough_lines(masked_edges, rho, theta, threshold, min_line_length, max_li
            # draw lane lines on the original image
            init_image = img.astype('uint8')
```

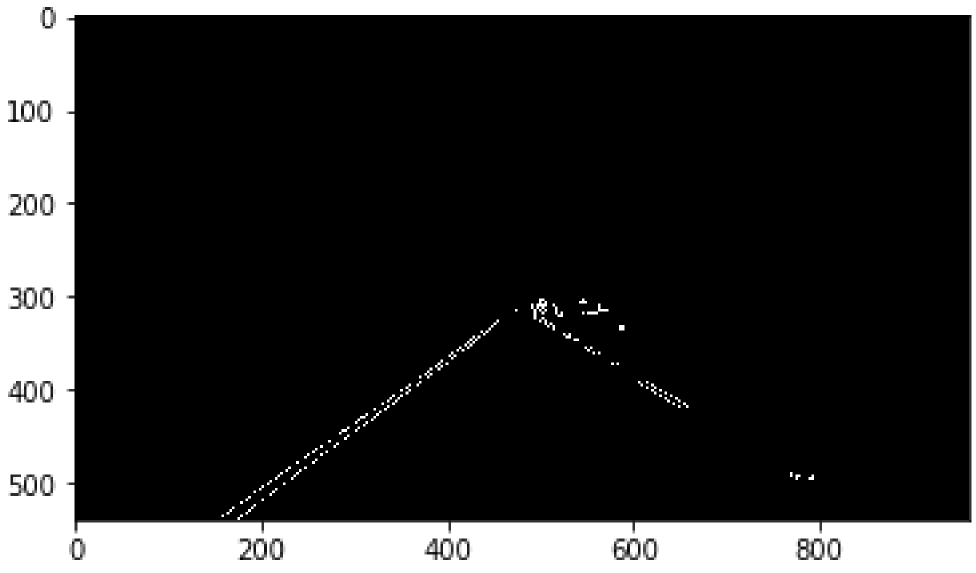
```
    return weighted_img(lines_img, init_image, =0.8, =1., =0.)
```

```
In [8]: # gray testing
        image1= mpimg.imread("test_images/solidYellowCurve.jpg")
        gray = grayscale(image1)

        # Gaussian smoothing
        blur_gray = gaussian_blur(gray, kernel_size=kernel_size)
        plt.imshow(blur_gray, cmap='Greys_r')
        plt.savefig('test_images_out/blur_gray_lines.jpg', dpi=300)
        plt.show()

        # Canny edges detection
        edges = canny(blur_gray, low_threshold=low_threshold, high_threshold=high_threshold)
        plt.imshow(edges, cmap='Greys_r')
        plt.savefig('test_images_out/canny_edges_lines.jpg', dpi=300)
        plt.show()

        # Trapezoid shape adjustment
        vertices = vertices_shape(edges)
        # masked region of interest
        masked_edges = region_of_interest(edges, vertices)
        plt.imshow(masked_edges, cmap='Greys_r')
        plt.savefig('test_images_out/masked_edges_region_interest.jpg', dpi=300)
        plt.show()

        # Hough transform on the edges detection of the image
        lines_img = hough_lines(masked_edges, rho, theta, threshold, min_line_length, max_line_g
        plt.imshow(lines_img, cmap='Greys_r')
        plt.savefig('test_images_out/hough_lines.jpg', dpi=300)
```

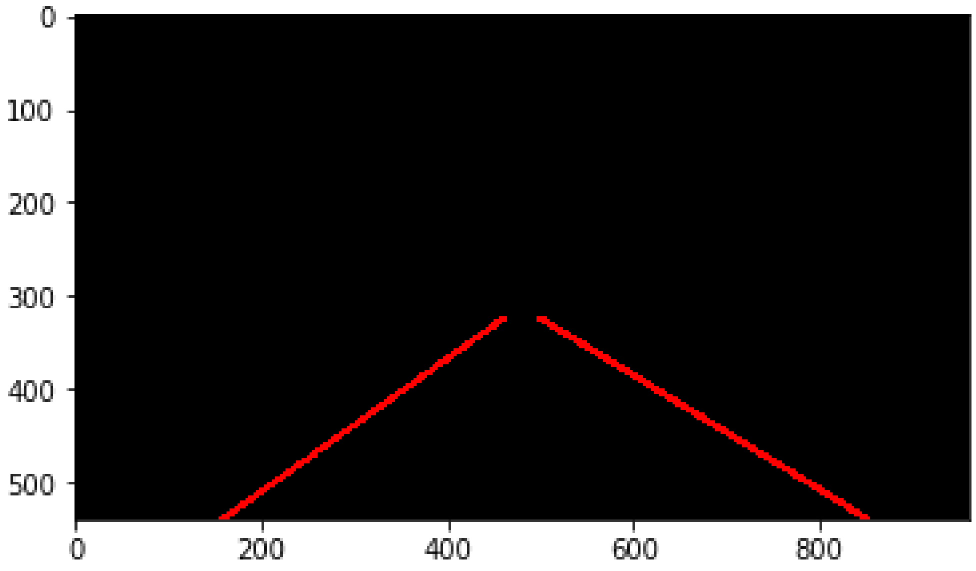




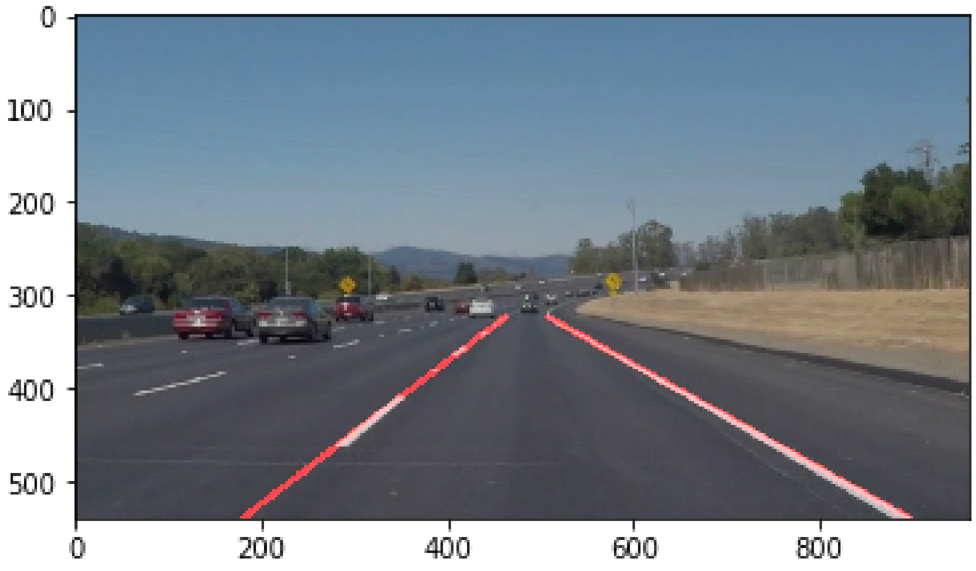
0
100
200
300
400
500
0
200
400
600
800

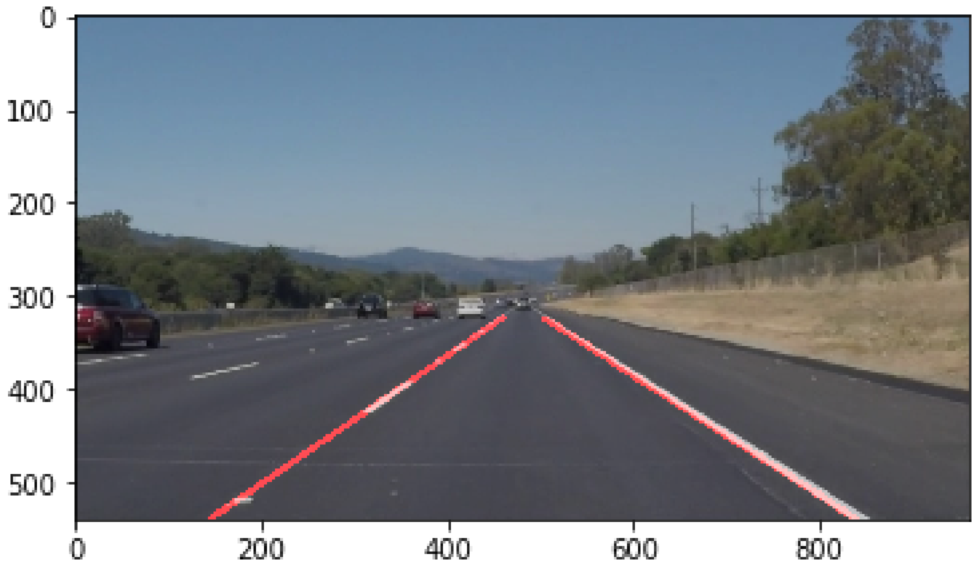


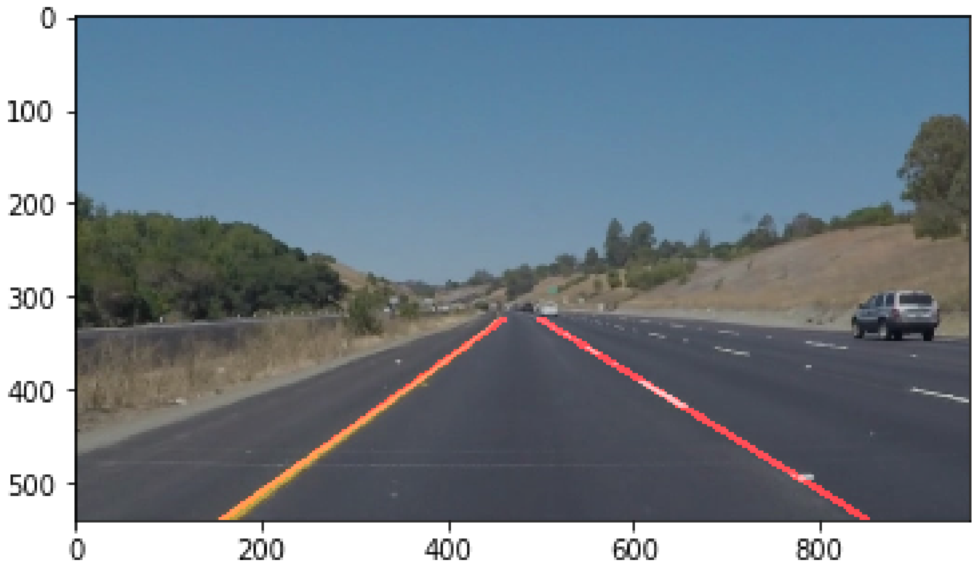
0
100
200
300
400
500
0
200
400
600
800

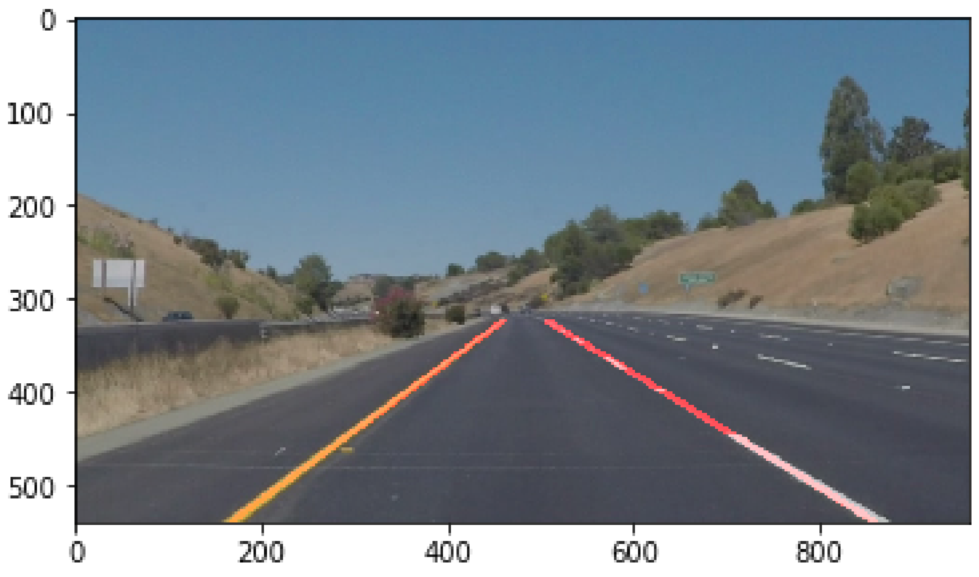


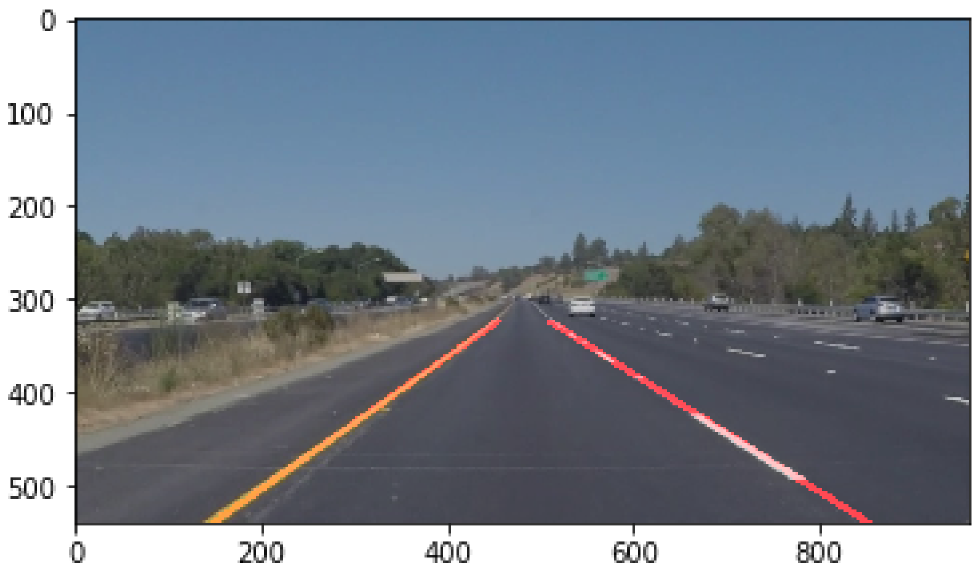
```
In [9]: # series of images
        test_images = os.listdir("test_images/")
        for image in test_images:
            if image!=".DS_Store":
                imgx = mpimg.imread("test_images/" + image)
                imageProcessed = lane_lines_pipelines(imgx)
                plt.imshow(imageProcessed)
                plt.show()
                mpimg.imsave('test_images_out/' + image[:-4] + '_modified.jpg', imageProcessed)
```

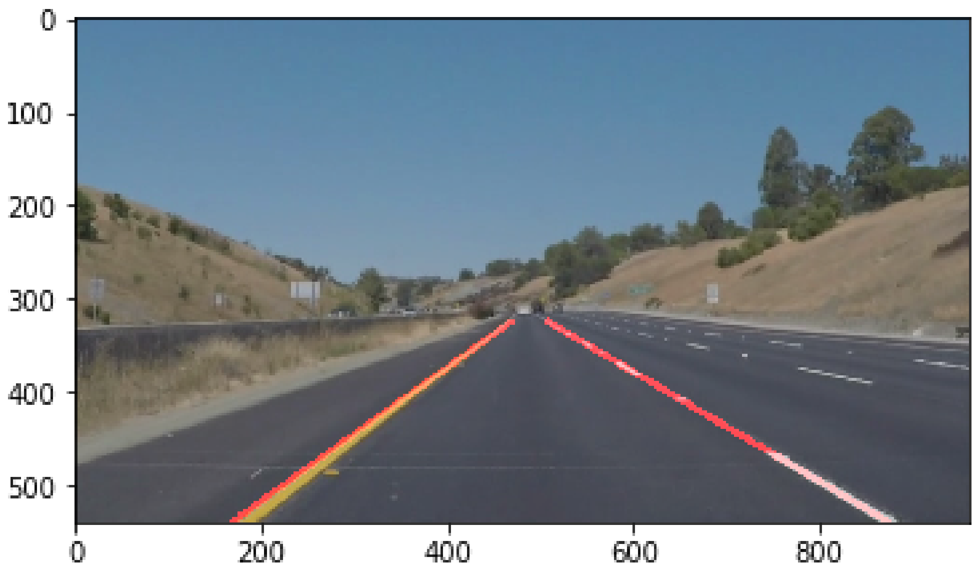



0
100
200
300
400
500
0
200
400
600
800


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



Looks nice!

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, check out this forum post 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 [10]: # Import everything needed to edit/save/watch video clips
         from moviepy.editor import VideoFileClip
         from IPython.display import HTML
```

```
In [11]: def process_image(image):
             # NOTE: The output you return should be a color image (3 channel) for processing vi
             # TODO: put your pipeline here,
             # you should return the final output (image where lines are drawn on lanes)
```

```
        result = lane_lines_pipelines(image)
        return result
```

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

```
In [12]: white_output = 'test_videos_output/solidWhiteRight.mp4'
         ## To speed up the testing process you may want to try your pipeline on a shorter subcl
         ## 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


100%|| 221/222 [00:02<00:00, 79.97it/s]


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

CPU times: user 2.21 s, sys: 629 ms, total: 2.84 s
Wall time: 3.18 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 [13]: HTML("""
         <video width="960" height="540" controls>
           <source src="{0}">
         </video>
         """.format(white_output))

Out[13]: <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 [14]: yellow_output = 'test_videos_output/solidYellowLeft.mp4'
         ## To speed up the testing process you may want to try your pipeline on a shorter subcl
         ## 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


100%|| 681/682 [00:08<00:00, 78.02it/s]


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

CPU times: user 7.04 s, sys: 1.72 s, total: 8.76 s
Wall time: 9.14 s
```

```
In [15]: HTML("""
         <video width="960" height="540" controls>
           <source src="{0}">
         </video>
         """.format(yellow_output))

Out[15]: <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 [16]: challenge_output = 'test_videos_output/challenge.mp4'
         ## To speed up the testing process you may want to try your pipeline on a shorter subcl
         ## 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


100%|| 251/251 [00:06<00:00, 36.09it/s]


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

CPU times: user 5.34 s, sys: 1.72 s, total: 7.06 s
Wall time: 7.75 s
```

```
In [17]: HTML("""
         <video width="960" height="540" controls>
           <source src="{0}">
         </video>
         """.format(challenge_output))

Out[17]: <IPython.core.display.HTML object>

In [ ]:
```