Self-Driving Car Engineer Nanodegree

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 (https://github.com/udacity/CarND-LaneLines-P1/blob/master/writeup_template.md) 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 (https://review.udacity.com/#!/rubrics/322/view) for this project.

Let's have a look at our first image called 'test_images/solidWhiteRight.jpg'. Run the 2 cells below (hit Shift-Enter or the "play" button above) to display the image.

Note: If, at any point, you encounter frozen display windows or other confounding issues, you can always start again with a clean slate by going to the "Kernel" menu above and selecting "Restart & Clear Output".

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.

Import Packages

In [ ]:
```
# importing some useful packages
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import os
import imageio
imageio.plugins.ffmpeg.download()
from moviepy.editor import VideoFileClip
import cv2
import math
```

Read in an Image

In [ ]:
```
#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', for example, call as plt.imshow(gray, cmap='gray')
```

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!

Helper Functions

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

In [ ]:
```
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 o
n the input image
    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 col
or
    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=2):
    """
    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
    """
    if len(img.shape) == 2:  # grayscale image -> make a "color" image out of
 it
        img = np.dstack((img, img, img))

    for line in lines:
        for x1, y1, x2, y2 in line:
             if x1 > 0 and x1 < img.shape[1] and \
                y1 > 0 and y1 < img.shape[0] and \
                x2 > 0 and x2 < img.shape[1] and \
                y2 > 0 and y2 < img.shape[0]:
                print(x1, y1, x2, y2)
                cv2.line(img, (x1, y1), (x2, y2), color, thickness)
    # print(img)
    return img

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([]), minLineL
ength=min_line_len, maxLineGap=max_line_gap)
    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, α=0.8, β=1., λ=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 * α + img * β + λ
    NOTE: initial_img and img must be the same shape!
    """
    return cv2.addWeighted(initial_img, α, img, β, λ)
```

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

def extract_lane_lines(imshape, lines):

    slope_min = 0.5
    slope_max = 1.0

    # Left line
    m1 = np.array([])
    b1 = np.array([])

    # Right line
    m2 = np.array([])
    b2 = np.array([])

    yMin = imshape[0]
    yMax = imshape[0] - 1

    for line in lines:
        # print(line)
        for x_left_1, y_left_1, x_right_1, y_right_1 in line:
            m = (y_right_1 - y_left_1) / (x_right_1 - x_left_1)
            b = y_left_1 - m * x_left_1

            if abs(m) > slope_min and abs(m) < slope_max:
                if m > 0:
                    m1 = np.append(m1, m)
                    b1 = np.append(b1, b)
                else:
                    m2 = np.append(m2, m)
                    b2 = np.append(b2, b)

                yMin = min([yMin, y_left_1, y_right_1])

    m1 = np.mean(m1)
    b1 = np.mean(b1)

    m2 = np.mean(m2)
    b2 = np.mean(b2)

    y_left_1 = yMax
    x_left_1 = round((y_left_1 - b1) / m1)

    y_left_2 = yMin
    x_left_2 = round((y_left_2 - b1) / m1)

    y_right_1 = yMax
    x_right_1 = round((y_right_1 - b2) / m2)
```

```
    y_right_2 = yMin
    x_right_2 = round((y_right_2 - b2) / m2)

    linePoints = np.array(
        [[[x_left_1, y_left_1, x_left_2, y_left_2]], [[x_right_1, y_right_1, x
_right_2, y_right_2]]]).astype(int)
    return linePoints


def return_y_position(y_position):
    # It needs to be lover half of the picture
    return y_position * 0.6



def get_houg_lines(img, masked_edges):
    # Define the Hough transform parameters
    # Make a blank the same size as our image to draw on
    rho = 2  # distance resolution in pixels of the Hough grid
    theta = np.pi / 180  # angular resolution in radians of the Hough grid
    # 15 minimum number of votes (intersections in Hough grid cell)
    threshold = 75
    min_line_length = 10  # minimum number of pixels making up a line
    max_line_gap = 90    # maximum gap in pixels between connectable line segm
ents


    # Run Hough on edge detected image
    # Output "lines" is an array containing endpoints of detected line segment
s
    lines = hough_lines(masked_edges, rho, theta, threshold,
                        min_line_length, max_line_gap)
    return lines

def define_vertices(imshape):
    vertices = np.array([[(0, imshape[0]), (450, return_y_position(
        imshape[0])), (500, 330), (imshape[1], imshape[0])]], dtype=np.int32)
    return vertices

def apply_canny(blur_gray):
    low_threshold = 50
    high_threshold = 150
    edges = canny(blur_gray, low_threshold, high_threshold)
    return edges

def apply_gaussian_smoothing(gray_image):
    kernel_size = 5
    blur_gray = gaussian_blur(gray_image, kernel_size)
    return blur_gray


def pipeline(img):

    imshape = img.shape
    gray_image = grayscale(img)
```

```
    blur_gray = apply_gaussian_smoothing(gray_image)

    edges = apply_canny(blur_gray)

    vertices = define_vertices(imshape)

    masked_edges = region_of_interest(edges, vertices)

    line_image = np.copy(img) * 0  # creating a blank to draw lines on

    lines = get_houg_lines(img, masked_edges)

    lane_image = draw_lines(line_image, lines)

    return weighted_img(lane_image,img)
```

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 [ ]:
```
def process_image():
    test_images_directory = "test_images"
    result_dir= "results"

    if not os.path.isdir(result_dir):
        os.mkdir(result_dir)

    for filename in  os.listdir(test_images_directory):
        image_path= os.path.join(test_images_directory, filename)
        img = mpimg.imread(image_path)

        result_image=pipeline(img)
        plt.imshow(result_image, cmap='Greys_r')

        mpimg.imsave(os.path.join(result_dir, filename), result_image)
```

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 (https://discussions.udacity.com/t/project-error-of-test-on-videos/274082) for more troubleshooting tips across operating systems.

In [2]:
```
# Import everything needed to edit/save/watch video clips
from moviepy.editor import VideoFileClip
from IPython.display import HTML
```

In [3]:
```
def process_image(image):
    # NOTE: The output you return should be a color image (3 channel) for proc
essing video below
    # TODO: put your pipeline here,
    # you should return the final output (image where lines are drawn on lane
s)
    result=pipeline(image)
    return result
```

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

In [4]:
```
white_output = 'test_videos_output/solidWhiteRight.mp4'
## To speed up the testing process you may want to try your pipeline on a shor
ter subclip of the video
## 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 of the subclip
## You may also uncomment the following line for a subclip of the first 5 seco
nds
##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!!
white_clip.write_videofile(white_output, audio=False)
```

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 [5]:
```
HTML("""
<video width="960" height="540" controls>
  <source src="{0}">
</video>
""".format(white_output))
```

Out[5]:

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 [6]:
```
yellow_output = 'test_videos_output/solidYellowLeft.mp4'
## To speed up the testing process you may want to try your pipeline on a shor
ter subclip of the video
## 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 of the subclip
## You may also uncomment the following line for a subclip of the first 5 seco
nds
##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)
```

In [7]:
```
HTML("""
<video width="960" height="540" controls>
  <source src="{0}">
</video>
""".format(yellow_output))
```

Out[7]:

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 (https://github.com/udacity/CarND-LaneLines-P1/blob/master/writeup_template.md) to the writeup template file.

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

Reflection

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

My pipeline consisted of several steps. First, I converted the images to grayscale, then I apply Gaussian smoothing in order remove noise. After that image is ready for the applying canny edge detection. Canny will return all edges but region of interest is just lane lines so it is necessary to define four sided polygon mask and extract just edges in that region. After extracting edges in the masked region I calculated hough lines.

Hough lines are not optimal solution so it needs additional adjustmets. We are only interested for the lane lines but offten on the road we can find other markings which we can consider like a noise or errors so hough lines can give us lines that are almost horizontal in our region.

In order to remove lines that are not logical for our assigment I have used slope to identife just lines that relevant for us. So for every line it is necessary that we calculate line parameters (y=bx+m)

If the slope is positiv lines belong to the left lane otherwise to the right.

Our region of interest is lower half of the picture which means that top of lines will have minimum value of the y and it is the same for the both lines.

When we have all lines and sorted to the left and right line we need to do average the lines so we have just one line for left and right and we draw those lines to the empty picture.

After all this steps are done we are taking piture with lanes and original image and do blending in order to get final result

If you'd like to include images to show how the pipeline works, here is how to include an image: