P1

July 12, 2017

1 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.

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

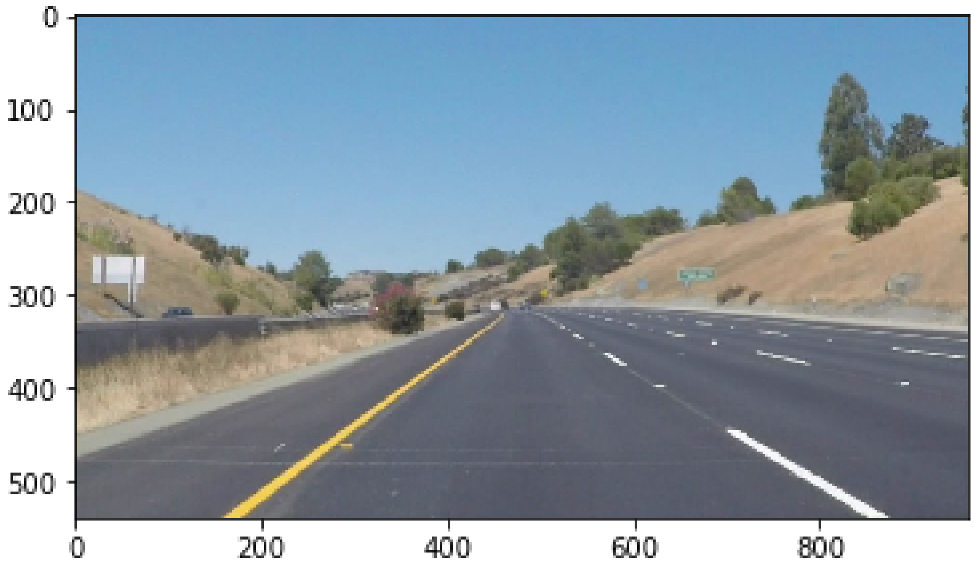
Your goal is to connect/average/extrapolate line segments to get output like this

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

```
In [2]: #reading in an image
        image = mpimg.imread('test_images/solidYellowCurve2.jpg')
        #printing out some stats and plotting
        print('This image is:', type(image), 'with dimesions:', image.shape)
        plt.imshow(image)  # if you wanted to show a single color channel image called 'gray',
```

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

```
Out[2]: <matplotlib.image.AxesImage at 0x2185d05ffd0>
```



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!

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

```
In [2]: 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)
```

```python
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 in
    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=10):
    """
    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).

    """
    global old_Lx1, old_Lx2, old_Rx1, old_Rx2
    y_max = img.shape[0]
    y_min = int(0.6*img.shape[0])
    left_slope = []
    right_slope = []
    left_y_cept = []
```

```
right_y_cept = []

for line in lines:
    for x1,y1,x2,y2 in line:
        m = (y2-y1)/(x2-x1)
        y_cept = y1 - m*x1
        if m < -0.3 and m > -4:                 #count negative slopes as left mar
            left_slope.append(m)
        elif m > 0.3 and m < 4:                 #count positive slopes as right ma
            right_slope.append(m)
        if y_cept > y_max:                      #left lane marker must have y-inte
            left_y_cept.append(y_cept)
        elif y_cept < 0.5*y_max:                #right lane marker must have y-int
            right_y_cept.append(y_cept)

#Averaging slopes and intercepts
l_slopeMean = np.mean(left_slope)
r_slopeMean = np.mean(right_slope)
l_yceptMean = np.mean(left_y_cept)
r_yceptMean = np.mean(right_y_cept)

if abs(l_slopeMean)>0.4:                            # only keep lines steeper than
    try:
        Lx1 = int(old_Lx1*0.8 + 0.2*((y_max - l_yceptMean)/l_slopeMean))  #reduce
    except:
        Lx1 = int((y_max - l_yceptMean)/l_slopeMean)
    try:
        Lx2 = int(old_Lx2*0.8 + 0.2*((y_min - l_yceptMean)/l_slopeMean))
    except:
        Lx2 = int((y_min - l_yceptMean)/l_slopeMean)
    #cv2.line(img, (Lx1, y_max), (Lx2, y_min), color, thickness)

if abs(r_slopeMean)>0.4:
    try:
        Rx1 = int(old_Rx1*0.8 + 0.2*((y_max - r_yceptMean)/r_slopeMean))
    except:
        Rx1 = int((y_max - r_yceptMean)/r_slopeMean)
    try:
        Rx2 = int(old_Rx2*0.8 + 0.2*((y_min - r_yceptMean)/r_slopeMean))
    except:
        Rx2 = int((y_min - r_yceptMean)/r_slopeMean)

cv2.line(img, (Rx1, y_max), (Rx2, y_min), color, thickness)
cv2.line(img, (Lx1, y_max), (Lx2, y_min), color, thickness)

old_Lx1 = Lx1                                       #for video smoothing
old_Lx2 = Lx2
old_Rx1 = Rx1
```

```
        old_Rx2 = Rx2

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=mi
    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, , )
```

1.1 Test on Images

Now you should 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 [3]:
```
import os
os.listdir("test_images/")
```

Out[3]:
```
['solidWhiteCurve.jpg',
 'solidWhiteRight.jpg',
 'solidYellowCurve.jpg',
 'solidYellowCurve2.jpg',
 'solidYellowLeft.jpg',
 'whiteCarLaneSwitch.jpg']
```

run your solution on all test_images and make copies into the test_images directory).

In [4]:
```
# TODO: Build your pipeline that will draw lane lines on the test_images
#reading in an image
old_Lx1 = None
```

```
old_Lx2 = None
old_Rx1 = None
old_Rx2 = None
image = mpimg.imread('test_images/whiteCarLaneSwitch.jpg')

gray = grayscale(image)

# Define a kernel size and apply Gaussian smoothing
kernel_size = 9
blur_gray = gaussian_blur(gray,kernel_size)

# Define parameters for Canny and apply
low_threshold = 50
high_threshold = 100
edges = canny(blur_gray, low_threshold, high_threshold)


# Next we'll create a masked edges image using cv2.fillPoly()

imshape = image.shape
vertices = np.array([[(75,imshape[0]),(0.45*imshape[1], 0.6*imshape[0]), (0.55*imshape

masked_edges = region_of_interest(edges, vertices)


# Define the Hough transform parameters

rho = 4 # distance resolution in pixels of the Hough grid
theta = np.pi/180 # angular resolution in radians of the Hough grid
threshold = 50     # minimum number of votes (intersections in Hough grid cell)
min_line_length = 40 #minimum number of pixels making up a line
max_line_gap = 200    # maximum gap in pixels between connectable line segments

# Run Hough on edge detected image
hough_out = hough_lines(masked_edges, rho, theta, threshold, min_line_length, max_line_

# Draw the lines on the image
lines_edges = weighted_img(hough_out, image)
plt.imshow(lines_edges)
#plt.savefig('Fig6out.jpg', dpi=150)
```

```
Out[4]: <matplotlib.image.AxesImage at 0x22e8542ac18>
```