Project 1: Finding Lane Lines using Open CV and Python

Getting Started:

To be able to detect lane lines accurately is a critical task for a self-driving car to drive autonomously. In this project, a simple pipeline using OpenCV to find lane lines in an image is developed and then applied to a full video feed, which is basically just a series of images.

On a sidenote, this is one of initial projects on Udacity's Self-Driving Car Nanodegree program, which is highly recommended for anyone interested in the field.
This lesson is with a series of helper methods that will all be combined into a single pipeline for processing.

Frameworks used:

- Python, Numpy, Matplotlib, MoviePy
- OpenCV 3

Input Image

Helper Methods/Functions:

Helper functions are used to describe methods like color selection, region of interest selection, grayscaling, Gaussian smoothing, Canny Edge Detection and Hough Transform line detection. The goal is to 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).

- Expected output after detecting line segments using the helper functions

- Goal is to connect/average/extrapolate line segments to get similar

```
import math
def grayscale(img):
    #Applies the Grayscale transform
    return cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

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.
     #defining a blank mask to start with
     mask = np.zeros_like(img)
     #defining a 3 channel or 1 channel color to fill the m
 ask with depending on the input image
     if len(img.shape) > 2:
         channel_count = img.shape[2]
         ignore_mask_color = (255,) * channel_count
     else:
         ignore_mask_color = 255

     #filling pixels inside the polygon defined
     cv2.fillPoly(mask, vertices, ignore_mask_color)

     #returning the image 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
starting point once you want to average 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 to decide which segments are part of the left
line vs.the right line.Then, you can average the position
of each ofthe 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).
"""
    for line in lines:
        for x1,y1,x2,y2 in line:
            cv2.line(img, (x1, y1), (x2, y2), color, thick
ness)

def hough_lines(img, rho, theta, threshold, min_line_len,
max_line_gap):

    #`img` should be the output of a Canny transform.
    lines = cv2.HoughLinesP(img, rho, theta, threshold, np
.array([]), minLineLength=min_line_len, maxLineGap=max_lin
e_gap)
    line_img = np.zeros((img.shape[0], img.shape[1], 3), d
type=np.uint8)

    draw_lines(line_img, lines)
    return line_img

# Python 3 has support for cool math symbols.

def weighted_img(img, initial_img, Î±=0.8, Î²=1., Î»=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 * Î± + img * Î² + Î»

        NOTE: initial_img and img must be the same shape!
        """
    return cv2.addWeighted(initial_img, Î±, img, Î², Î»)
```

Apply Canny Edge Detection and Masking a Region of Interest:

- Start by applying a Gaussian blur and converting the image to grayscale before isolating the region of interest.
- The second step is to run canny edge detection in OpenCV. The blur and grayscale step helps make the main lane lines stand out.
- Lastly, it's important to cut out as much of the noise as possible in the frame. As the general area in which the road will appear can be known form the input image, so that area is isolated with a trapezoid shape.

```
def pipeline(image):
    # Paramemters For Region Selection
    bot_left = [80, 540]
    bot_right = [980, 540]
    apex_right = [510, 315]
    apex_left = [450, 315]
     v = [np.array([bot_left, bot_right, apex_right, apex_lef
  t], dtype=np.int32)]
    #Apply Canny Edge Detection And Mask Region Of Interest
    gray = grayscale(image)
    blur = gaussian_blur(gray, 7)
    edge = canny(blur, 50, 125)
    mask = region_of_interest(edge, v)
```

Hough Lines Detection:

- Probabilistic Hough lines is used in OpenCV to identify the location of lane lines in the road.
- The HoughLinesP function in OpenCV returns an array of lines organized by endpoints (X1,Y1, X2, Y2), which can then be drawn onto the image.
- The problem here is that the two distinct lines needs to be detected, the right lane marker and the left lane marker.
- In the next section, these are organized separately by slope and reject outliers that throw off the intended slope of the line.

```
def pipeline(image):
    ## ...
    # Apply Hough Lines
    lines = cv2.HoughLinesP(mask, 0.8, np.pi/180, 25, np.arr
    ay([]), minLineLength=50, maxLineGap=200)
```

Separating Lines by Slope:

- Let's use slope equation, m = (Y2 - Y1) / (X2 - X1) to organize lines by their slope. It is important to point out that the y-axis is inverted in OpenCV when reading images.
- Thus, positive slopes will be the right lane and negative slopes will be the left lane.

```
def separate_lines(lines):
    #Separate Lines By +/- Slope
    right = []
    left = []
    for x1,y1,x2,y2 in lines[:, 0]:
        m = (float(y2) - y1) / (x2 - x1)
        if m >= 0:
            right.append([x1,y1,x2,y2,m])
        else:
            left.append([x1,y1,x2,y2,m])
    return right, left
```

Here's the updated pipeline to this point:

```
def pipeline(image):
    ## ...
    right_lines, left_lines = separate_lines(lines)
```

Extending Hough Lines into a Single Unified Lane Line:

- Now, lines need to be pieced together and unified into the best estimate for the lane line location.
- In this case, lines need to be extended outside the image frame and then mask them off with the region of interest helper described above.

Here's the step by step procedure:

- Line outliers are removed (e.g flat lines or lines that deviate significantly)
- Lines are merged by the mean of their endpoints.
- Endpoints are extended off the image canvas.

These helpers will be defined before running in the pipeline:

```
def extend_point(x1, y1, x2, y2, length):
    # Extends Line Endpoints By Specific Length
    line_len = np.sqrt((x1 - x2)**2 + (y1 - y2)**2)
    x = x2 + (x2 - x1) / line_len * length
    y = y2 + (y2 - y1) / line_len * length
    return x, y

def outliers(data, cutoff, thresh=0.08):
    # Reject Outliers
    data = np.array(data)
    data = data[(data[:, 4]>=cutoff[0])& (data[:, 4]<=
    cutoff[1])]
    m = np.mean(data[:, 4], axis=0)
    return data[(data[:, 4] <= m+thresh) & (data[:, 4] >= m-
thresh)]

def merge_lines(lines):
    # Merges and Extends Hough Detected Lines Across Image
    lines = np.array(lines)[:, :4]
    x1,y1,x2,y2 = np.mean(lines, axis=0)
    x1e, y1e = extend_point(x1,y1,x2,y2, -1000)
    x2e, y2e = extend_point(x1,y1,x2,y2, 1000)
    line = np.array([[x1e,y1e,x2e,y2e]])
    return np.array([line], dtype=np.int32)
```

Final Pipeline:

```
1 def pipeline(image):
2     # Parameмters For Region Selection
3     bot_left = [80, 540]
4     bot_right = [980, 540]
5     apex_right = [510, 315]
6     apex_left = [450, 315]
7     v = [np.array([bot_left, bot_right, apex_right, apex_le
      ft], dtype=np.int32)]
8
9     # Apply Canny Edge Detection And Mask Region Of Interes
      t
10    gray = grayscale(image)
11    blur = gaussian_blur(gray, 7)
12    edge = canny(blur, 50, 125)
13    mask = region_of_interest(edge, v)
14
15    # Apply Hough Lines
16    lines = cv2.HoughLinesP(mask, 0.8, np.pi/180, 25, np.ar
      ray([]), minLineLength=50, maxLineGap=200)
17    # Separate Lines By Slope
```

```
    right_lines, left_lines = separate_lines(lines)

    right = outliers(right_lines,  cutoff=(0.45, 0.75))
    right = merge_lines(right)

    left = outliers(left_lines, cutoff=(-0.85, -0.6))
    left = merge_lines(left)

    lines = np.concatenate((right, left))

    # Draw Lines
    line_img = np.copy((image)*0)
    draw_lines(line_img, lines, thickness=10)

    #Compute Final Image
    line_img = region_of_interest(line_img, v)
    final = weighted_img(line_img, image)
    return final
```

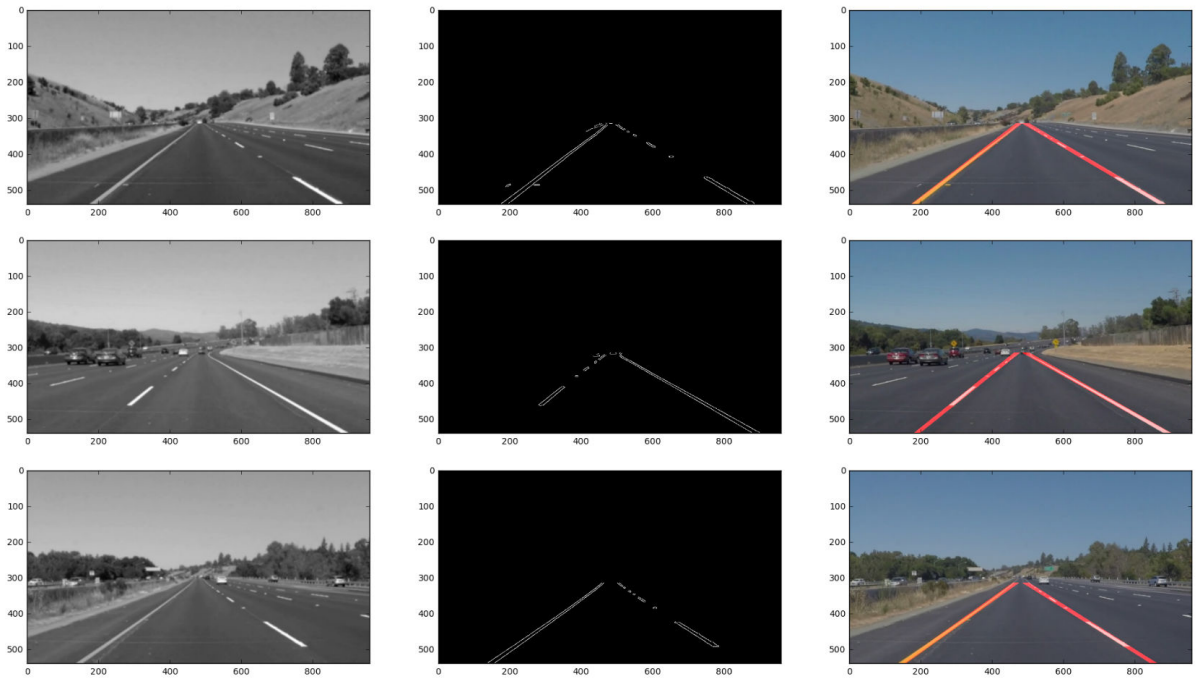
The pipeline plugged into MoviePy:

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

  def process_image(image):
      result = pipeline(image)
      return result
   white_output = 'final_white.mp4'
    clip1 = VideoFileClip("solidWhiteRight.mp4")
    white_clip = clip1.fl_image(process_image)
    #NOTE: this function expects color images!!
    %time white_clip.write_videofile(white_output,
audio=False)
```

Ssteps in the Pipeline Visual:

Here's a few visuals of the intermediate steps in the transformation process for three different input images:



Conclusion:

In this project lane lines are detected using the computer vision library OpenCV by applying concepts such as Gaussian Smoothing, Canny Edge Detection, Hough Transform.

The algorithm works fairly well in most cases but the algorithm can be made robust by creating a color mask that will highlights the whites and yellows in the frame. This ensures that Hough lines are more easily detected in shaded regions and low contrast regions.

An additional technique is to add a global variable for the line from the prior frame. This will be averaged with the current frame to prevent jittery line detection on the video footage

---

References:

Here’s the link to my github repository containing all the source code and output files: https://github.com/harish3110/Project-1---Lane-Line-Detection.git

---