Road Lane Line Detection

A report submitted in partial fulfillment of the requirements for the award of the certificate of

B. TECH

IN

COMPUTER SCIENCE ENGINEERING

By

Kaustubh sanadhya (201599013)
Ajay Yadav (201599002)
Shweta Singh(191500808)
Aman Thapak(191500099)
Akhil Dubey(191500079)

Department of Computer Science Engineering

DECLARATION

We hereby declare that the work which is presented in B. Tech Sixth semester. "**Road Lane Line Detection"** in partial fulfillment of the requirements for submitted to the department of Computer Science engineering of GLA University, Mathura is an authentic record of my own work carried under the supervision of **Mr. Vinay Agarwal (Assistant Professor).**

Name of the candidates: **Signature of the candidates**:

Kaustubh sanadhya (201599013)
Ajay Yadav (201599002)
Aman Thapak(191500099)
Akhil Dubey(191500079)
Shweta Singh(191500808)

<u>CERTIFICATE</u>

This is Certified that this project report **“Road Lane Line Detection”** is the Bonafede work of “**Kaustubh Sanadhya, Shweta Singh, Ajay Yadav, Aman Thapak, Akhil Dubey”** who carried out the project work under my supervision.

Signature of Project Guide

Mr. Vinay Agarwal
Assistant Professor
GLA University

ACKNOWLEDGEMENT

It gives me a great sense of pleasure to present the progress report work, undertaken during B. Tech Sixth Semester. I owe special debt of gratitude to **Mr. Vinay Agarwal**, Department of Computer Science Engineering, GLA University, Mathura for their constant support and guidance throughout the course of my work. His sincerity, thoroughness and perseverance have been a constant source of inspiration for me. It is only their cognizant efforts that my endeavors have seen light of the day.

I also do not like to miss the opportunity to acknowledge the contribution of**,** all faculty members of the department for their kind assistance and co-operation during the development of my report. Last but not the least, I acknowledge my friends for their contribution in the completion of the project.

CONTENTS

1. Introduction……………………………………………………………06
2. About Tools……………………………………………………………07
3. Open-CV………………………………………………………………08
    - Open-CV Python…………………………………………….………08
4. NumPy Library……………………………………………………..…09
    - Why is NumPy fast………………………………………………..10
5. Matplotlib Library……………………………………………………11
6. Working Of Project………………………………………………..12-14
7. Data Flow Diagram…………………………………………………15
8. The Proposed Algorithm…………………………………………...16
    - Canny Edge Detection………………………………………………17
9. Snapshots…………………………………………………………...18-25
10. Future Scope…...……………………………………………………...25
11. Conclusion…...…………………………………………………………...26

INTRODUCTION

Road lane-line detection is a primary function for an autonomous car and has been applied in various smart vehicle systems, but it is challenging for computer vision. It helps build a decision-making and path planning algorithm and more like steering control, accelerating action, breaking, relative vehicle trajectory motions, and much more. This algorithm provides an accurate and reliable fit to road lane-lines, and with it, we can easily estimate image sequence alignment such as drift, shift, rotation, etc. where we can adjust or align the frame according to our requirement. We have created a chart explaining the steps involved in the real-time road lane-line detection and image sequence alignment. The annual increase in car ownerships has caused traffic safety to become an important factor affecting the development of a city. To a large extent, the frequent occurrence of traffic accidents is caused by subjective reasons related to the driver, such as drunk, fatigue and incorrect driving operations. Smart cars can eliminate these human factors to a certain extent . In recent years, the development of smart cars has gradually attracted the attention of researchers in related fields worldwide. Smart cars can intelligently help humans perform driving tasks based on real-time traffic information, thereby indicating their significance in improving the safety of automobile driving and liberating human beings from tedious driving environments . Lane detection is an important foundation in the course of intelligent vehicle development that directly affects the implementation of driving behaviours. Based on the driving lane, determining an effective driving direction for the smart car and providing the accurate position of the vehicle in the lane are possible; Therefore, conducting an in-depth study on this is necessary.

ABOUT TOOLS

Software Used :

1. Jupiter Notebook
2. Python Programming
3. Open – Computer Vision

Hardware Used:

1. Intel i3 processor
2. RAM: Minimum 2 GB

OPEN-CV

OpenCV supports a wide variety of programming languages such as C++, Python, Java, etc., and is available on different platforms including Windows, Linux, OS X, Android, and iOS. Interfaces for high-speed GPU operations based on CUDA and OpenCL are also under active development.

OpenCV-Python is the Python API for OpenCV, combining the best qualities of the OpenCV C++ API and the Python language.

OpenCV-Python

OpenCV-Python is a library of Python bindings designed to solve computer vision problems.

Python is a genera purpose programming language started by **Guido van Rossum** that became very popular very quickly, mainly because of its simplicity and code readability. It enables the programmer to express ideas in fewer lines of code without reducing readability.

Compared to languages like C/C++, Python is slower. This gives us two advantages: first, the code is as fast as the original C/C++ code (since it is the actual C++ code working in background) and second, it easier to code in Python than C/C++. OpenCV-Python is a Python wrapper for the original OpenCV C++ implementation.

OpenCV-Python makes use of NumPy, which is a highly optimized library for numerical operations with a MATLAB-style syntax. All the OpenCV array structures are converted to and from NumPy arrays. This also makes it easier to integrate with other libraries that use NumPy such as SciPy and Matplotlib.

NumPy

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

At the core of the NumPy package, is the ndarray object. This encapsulates n-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance. There are several important differences between NumPy arrays and the standard Python sequences:

NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original.

The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.

NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.

A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays. In other words, in order to efficiently use much (perhaps even most) of today's scientific/mathematical Python-based software, just knowing how to use Python's built-in sequence types is insufficient - one also needs to know how to use NumPy arrays.

Why is NumPy Fast?

Vectorization describes the absence of any explicit looping, indexing, etc., in the code - these things are taking place, of course, just "behind the scenes" in optimized, pre-compiled C code. Vectorized code has many advantages. Without vectorization, our code would be littered with inefficient and difficult to read for loops.

Broadcasting is the term used to describe the implicit element-by-element behavior of operations; generally speaking, in NumPy all operations, not just arithmetic operations, but logical, bit-wise, functional, etc., behave in this implicit element-by-element fashion, i.e., they broadcast. Moreover, in the example above, a and b could be multidimensional arrays of the same shape, or a scalar and an array, or even two arrays of with different shapes, provided that the smaller array is "expandable" to the shape of the larger in such a way that the resulting broadcast is unambiguous.

JUPYTER NOTEBOOK

The Jupyter Notebook is an open source web application that you can use to create and share documents that contain live code, equations, visualizations, and text. Jupyter Notebook is maintained by the people at Jupyter Notebook.

Jupyter Notebooks are a spin-off project from the IPython project, which used to have an IPython Notebook project itself. The name, Jupyter, comes from the core supported programming languages that it supports: Julia, Python, and R. Jupyter ships with the IPython kernel, which allows you to write your programs in Python, but there are currently over 100 other kernels that you can also use.

The Jupyter Notebook is not included with Python, so if you want to try it out, you will need to install Jupyter.

There are many distributions of the Python language. This article will focus on just two of them for the purposes of installing Jupyter Notebook. The most popular is CPython, which is the reference version of Python that you can get from their website. It is also assumed that you are using **python 3.**

WORKING OF PROJECT

The detailed steps involved in our Road Lane Line Detection are given as follows:

Load the target image: We can easily capture and load the test images one by one in which we have to detect lane-lines using OpenCV inbuilt features.

Selection of suitable Color: This is the fundamental part that helps us locate the road dashed lane-line so that we can extract it in our image views and aim for it. Pictures are basically in RGB format. To underline our lane line, we have to choose different colour spaces like HSL, HSV. We specifically chose the HSL color space to opt-out of the dashed lane lines. Once this colour selection is applied to HSL images, it will suppress everything except for the white (in our case) lane lines.

Smoothening of the images: Before proceeding to locate the edges in the image, we have to do some smoothing of our target images. As we know, edge detection methods include measurement of intensity gradients, so we need to convert our images to grayscale to detect edges in our image because it is faster than handling an RGB coloured image. After that, we have to apply some filters to smooth the edges and remove noises as noise can create false edges. We have used Gaussian filters to achieve our objectives.

Canny Edge Detection: For edge detection in the image, the Canny edge detection technique is beneficial for detecting road-dashed lane-lines successfully. It measures gradient in several directions and tracks the edges of our blurred picture with a large intensity shift canny.

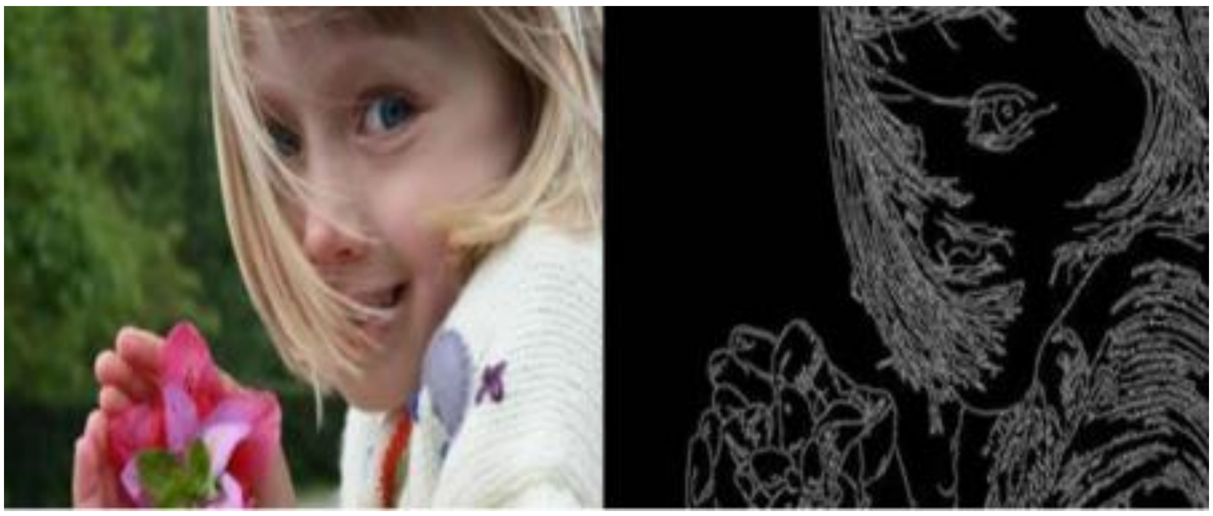


Fig – 1.1
Edge Filtering

Selecting a suitable region of interest: The field facing the camera is of concern, where there are lane lines. Therefore, we apply a mask to this area, and everything else will be suppressed.

Fig – 1.2
Original Image

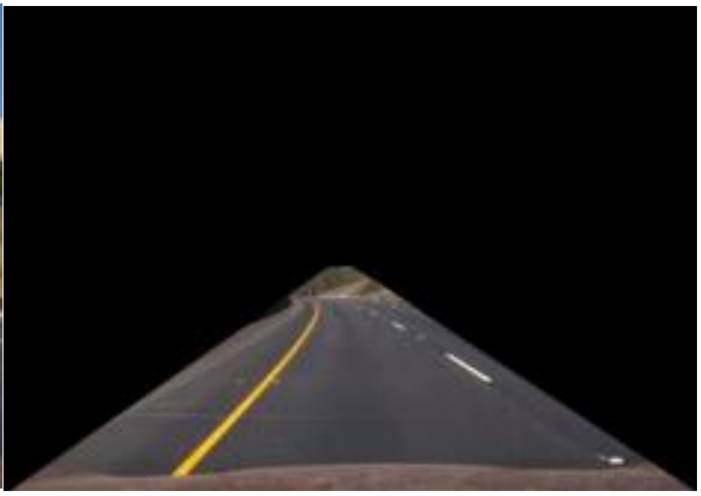


Fig – 1.3
Region of Interest

Hough Transform: In image processing, line detection is an algorithm that takes a collection of edge points and finds all the lines on which these edge points lie. The Hough transform is a procedure used to separate components of a particular shape in a picture. The most popular line detector is the Hough transform techniques. The notion of Hough transformation is for each edge point in the edge map to be converted into the line conceivable across that point.

Lane lines evaluation, averaging, and extrapolating: For each side, we have several lines identified. Both these lines must be combined, and a single line drawn for each line must be drawn. Lane lines can either be extended or extrapolated to the longest path.

Alignment of Image sequence: Once the lane-line is detected and extrapolated, we can easily find the intercept and slope of the line of the first frame. We can further apply a linear regression technique to find out the average slope and intercept. We are going to relate this slope and intercept with the subsequent frames. Upon successfully estimating the difference in slope and intercept, we can easily align and warp the image sequences.

DATA FLOW DIAGRAM

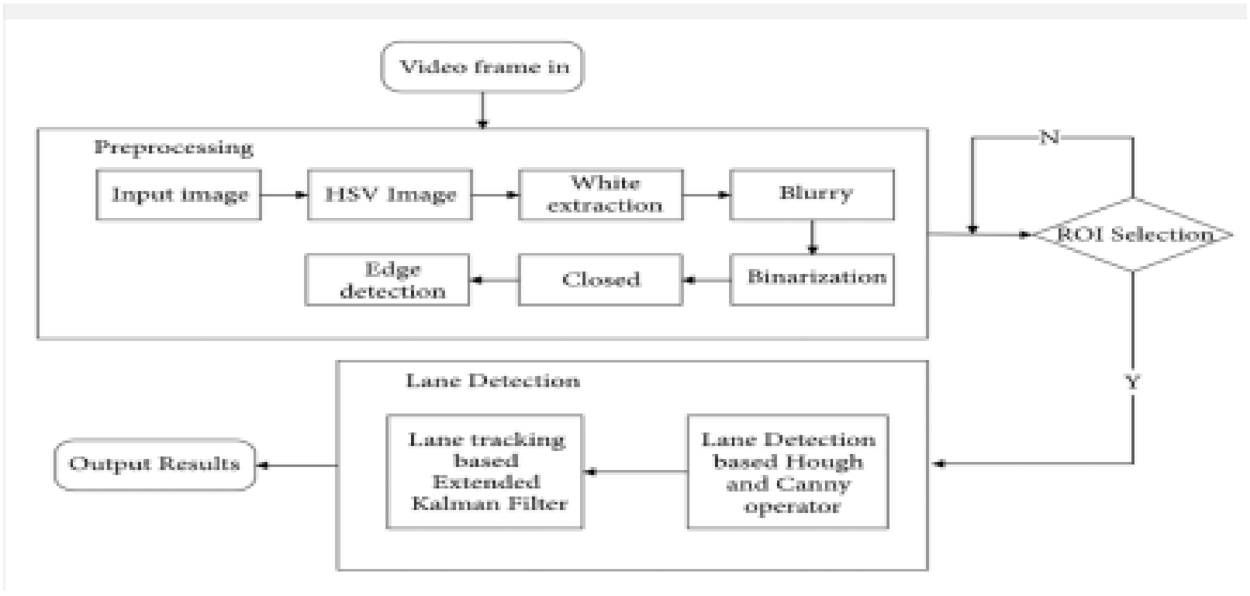
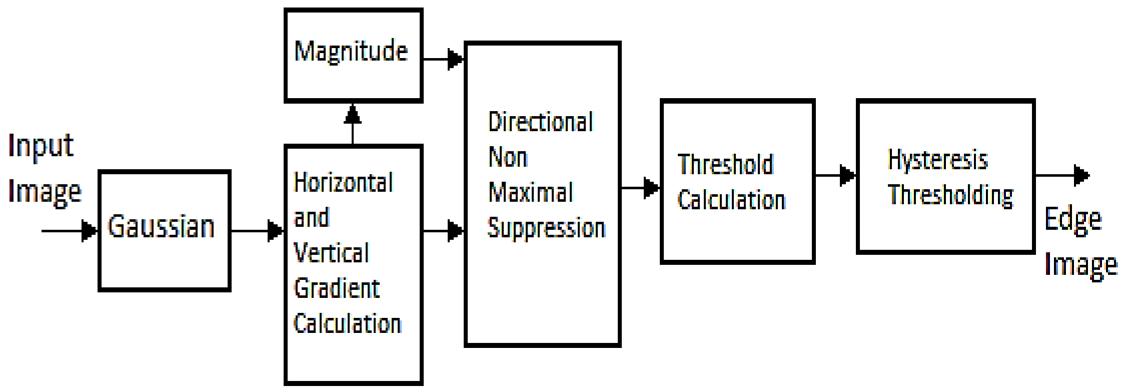


Fig – 1.4

The Proposed Algorithm: Canny-Edge Detection



Block Diagram

Fig – 1.5

Steps Of Canny Edge Detection:

Step 1: Gaussian Smoothing In order to implement the canny edge detector algorithm, a series of steps must be followed. The first step is to filter out any noise in the original image before trying to locate and detect any edges. And because the Gaussian filter can be computed using a simple mask, it is used exclusively in the Canny algorithm. Once a suitable mask has been calculated, the Gaussian smoothing can be performed using standard convolution methods.. Image smoothing is the first stage of the canny edge detection. The pixel values of the input image are convolved with predefined operators, to create an intermediate image. This process is used to reduce the noise within an image or to produce a less pixilated image. Image smoothing is performed by convolving the input image with a Gaussian filter.

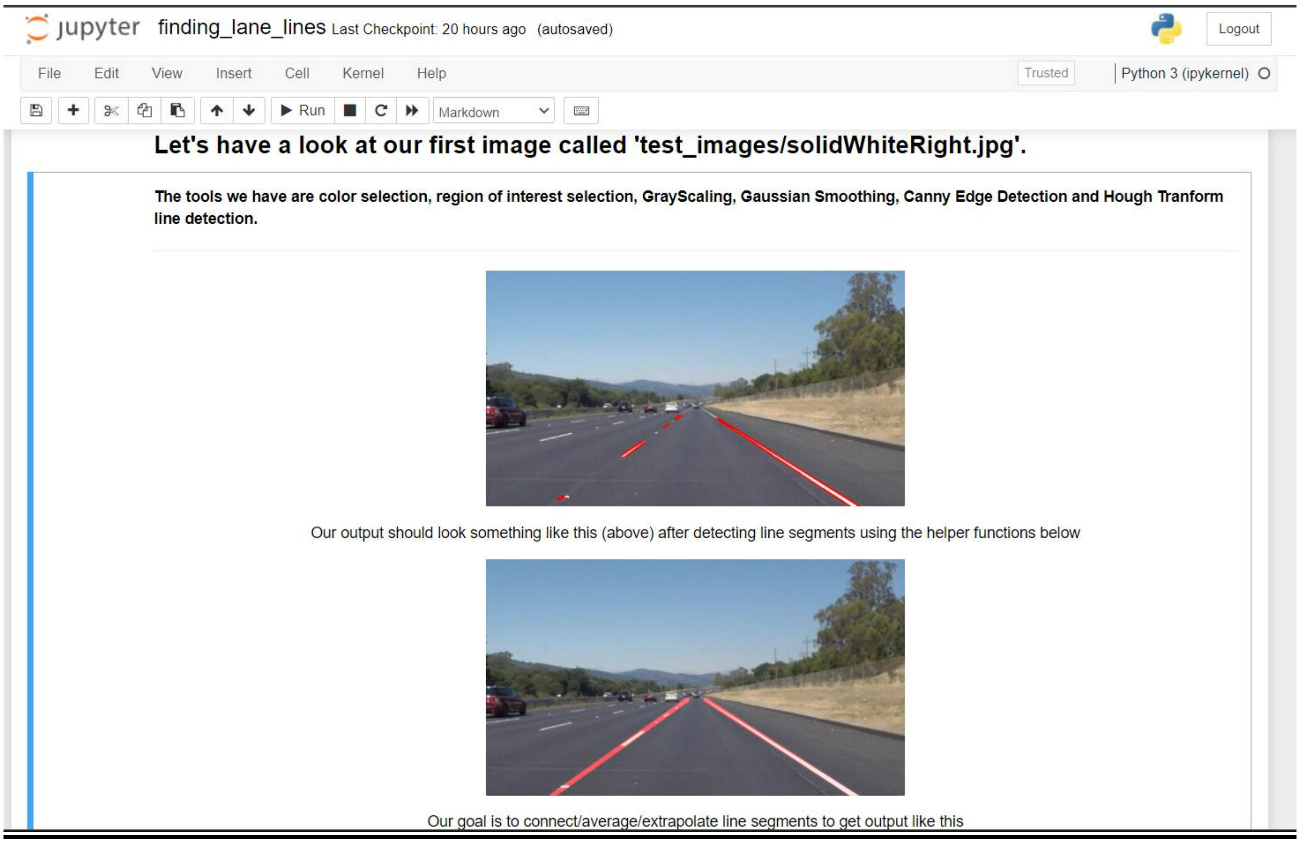
Step 2: Gradient Calculation After smoothing the image and eliminating the noise, the next step is to find the edge strength by taking the gradient of the image. The operator performs a 2-D spatial gradient measurement on an image. In this stage, the blurred image obtained from the image smoothing stage is convolved with a 3x3 operator. The operator is a discrete differential operator that generates a gradient image .The operators used to calculate the horizontal and vertical gradients. To obtain the gradient image, a smoothened image from the first stage is convolved with the horizontal and vertical operators.

Step3: Gradient Magnitude and Direction Gx and Gy are images with pixel values that are the magnitude of the horizontal and vertical gradient, respectively. The magnitude, or edge strength, of the gradient is then approximated using the formula, The direction of the edge is computed using the gradient in the x and y directions. Edge direction is defined as the direction of the tangent to the contour that the edge defines in 2-dimensions. The edge direction of each pixel in an edge direction image is determined using the arctangent.

Step4: Non Maximum Suppression The edge strength for each pixel in an image obtained from equation is used in non maximum suppression stage. The edge directions obtained from equation are rounded off to one of four angles 0 degree, 45 degree, 90 degree or 135 degree before using it in non-maximum suppression. After the edge directions are known, non maximum suppression now has to be applied. Non maximum suppression is used to trace along the edge in the edge direction and suppress any pixel value (sets it equal to 0) that is not considered to be an edge. This will give a thin line in the output image. Non-maximum suppression (NMS) is used normally in edge detection algorithms. It is a process in which all pixels whose edge strength is not maximal are marked as zero within a certain local neighborhood. This local neighborhood can be a linear window at different directions of length 5 pixels.

Step5: Thresholding and Hysterisis Thresholding with hysteresis is the last stage in canny edge detection, which is used to eliminate spurious points and non-edge pixels from the results of non-maximum suppression. The input image for thresholding with hysteresis has gone through Image smoothing, calculating edge strength and edge pixel, and the Non-maximum suppression stage to obtain thin edges in the image. Results of this stage should give us only the valid edges in the given image, which is performed by using the two threshold values,

SNAPSHOTS


jupyter finding_lane_lines Last Checkpoint: 20 hours ago (autosaved)
Logout
File Edit View Insert Cell Kernel Help
Trusted
Python 3 (ipykernel)
Run
Markdown

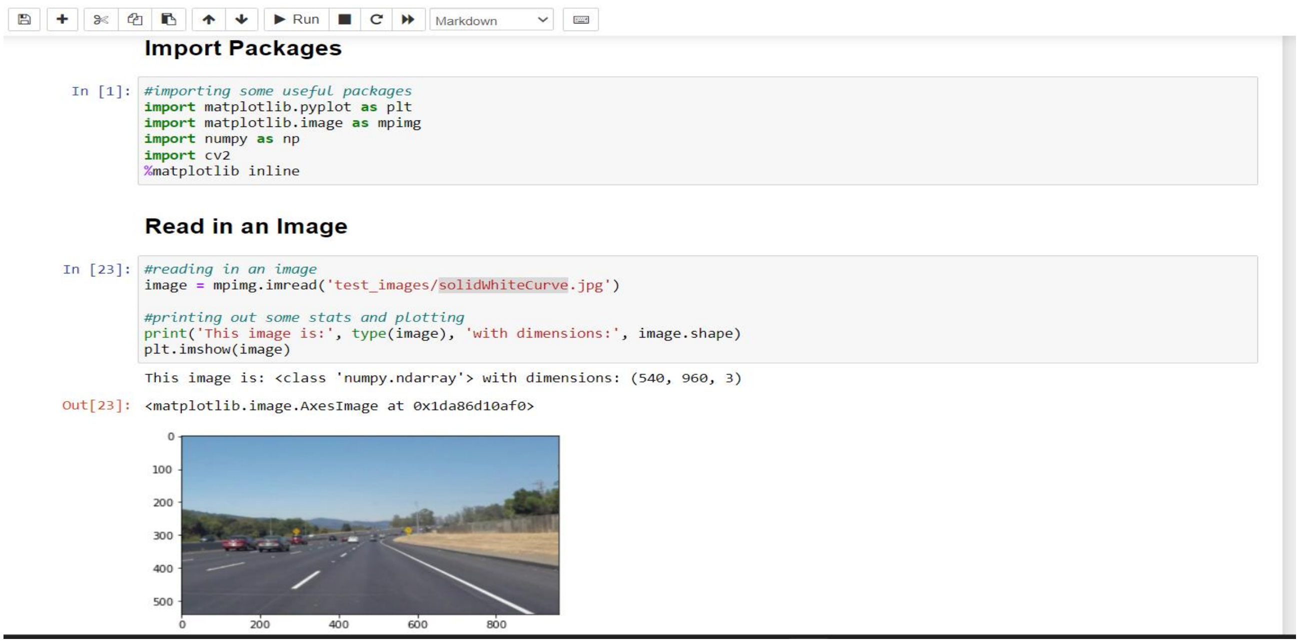
Let's have a look at our first image called 'test_images/solidWhiteRight.jpg'.

The tools we have are color selection, region of interest selection, GrayScaling, Gaussian Smoothing, Canny Edge Detection and Hough Tranform line detection.

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

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

Fig – 1.6



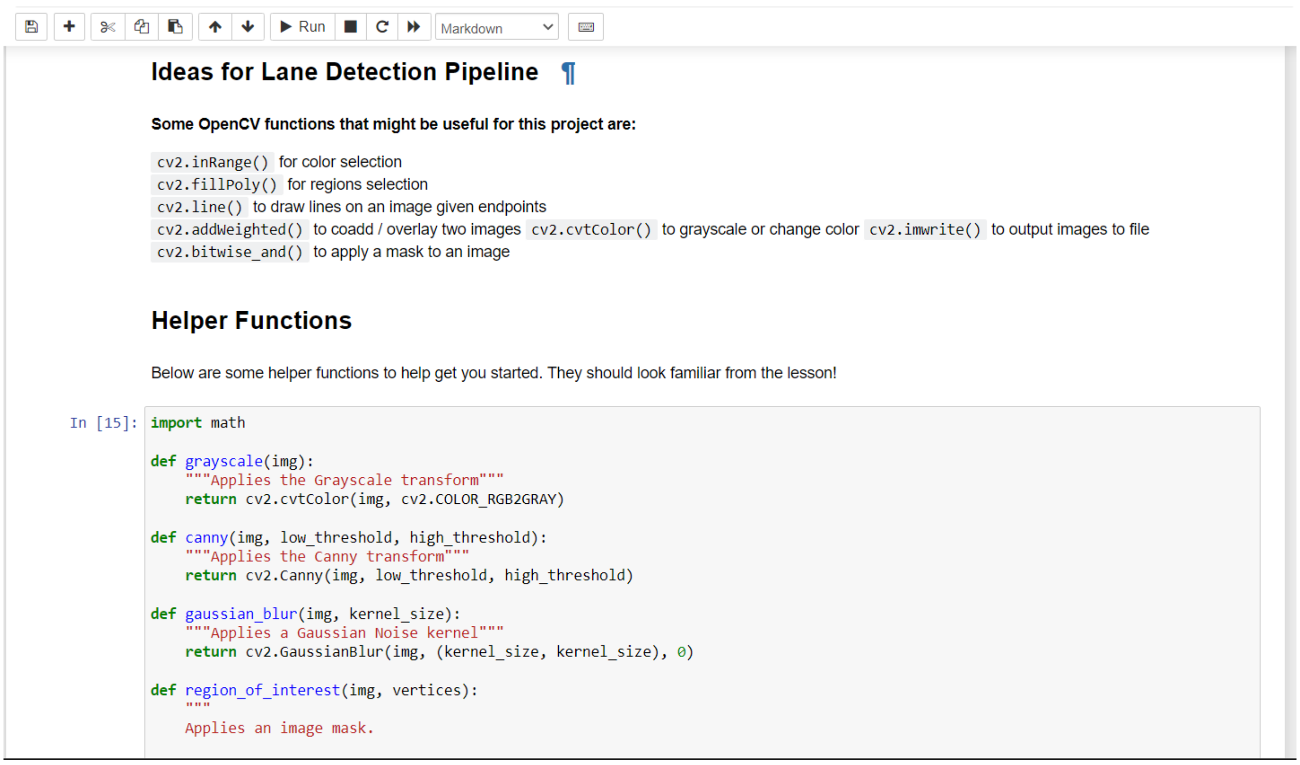
Import Packages

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

Read in an Image

```
In [23]: #reading in an image
         image = mpimg.imread('test_images/solidWhiteCurve.jpg')

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

         This image is: <class 'numpy.ndarray'> with dimensions: (540, 960, 3)
Out[23]: <matplotlib.image.AxesImage at 0x1da86d10af0>
```

Fig – 1.7



Ideas for Lane Detection Pipeline ¶

Some OpenCV functions that might be useful for this project are:

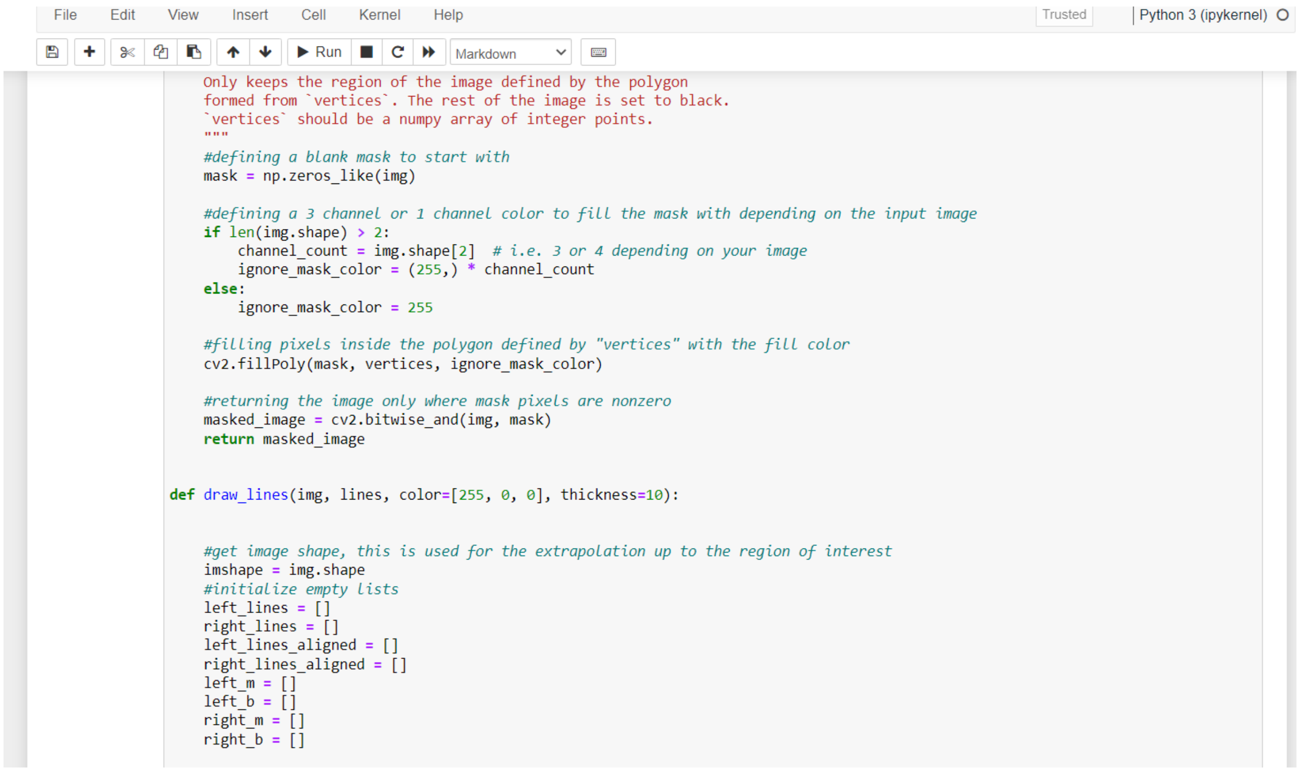
`cv2.inRange()` for color selection
`cv2.fillPoly()` for regions selection
`cv2.line()` to draw lines on an image given endpoints
`cv2.addWeighted()` to coadd / overlay two images `cv2.cvtColor()` to grayscale or change color `cv2.imwrite()` to output images to file
`cv2.bitwise_and()` to apply a mask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

Helper Functions

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

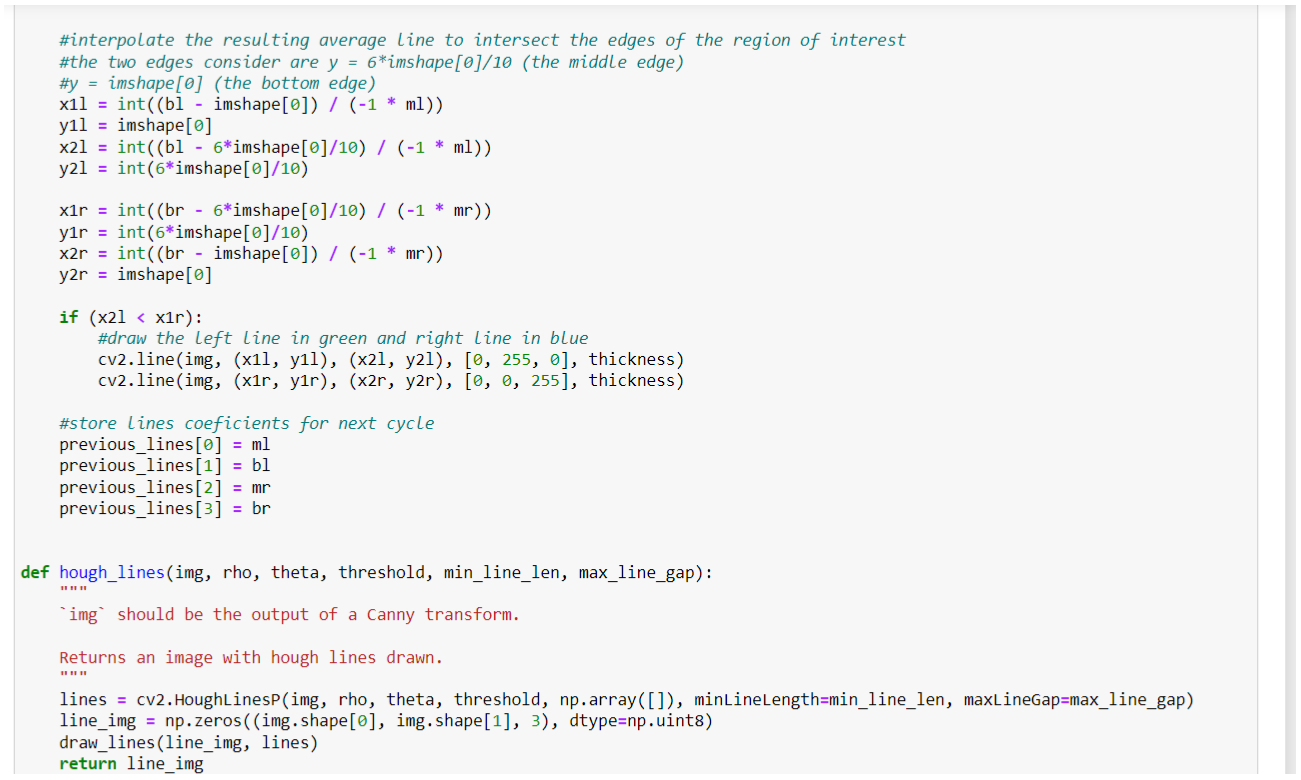
```
In [15]: import math

         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.
```

Fig – 1.8



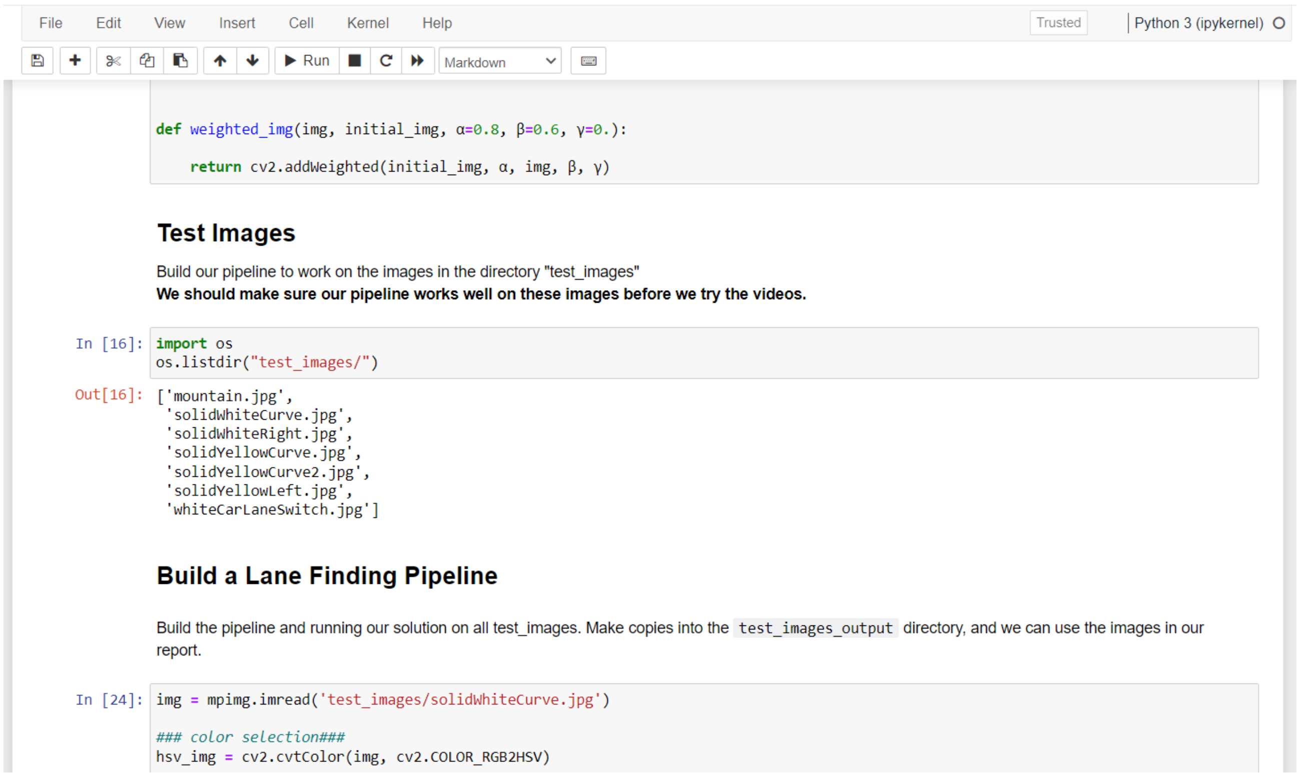
```python
    Only keeps the region of the image defined by the polygon
    formed from `vertices`. The rest of the image is set to black.
    `vertices` should be a numpy array of integer points.
    """
    #defining a blank mask to start with
    mask = np.zeros_like(img)

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


    #get image shape, this is used for the extrapolation up to the region of interest
    imshape = img.shape
    #initialize empty lists
    left_lines = []
    right_lines = []
    left_lines_aligned = []
    right_lines_aligned = []
    left_m = []
    left_b = []
    right_m = []
    right_b = []
```

Fig – 1.9



```python
    #interpolate the resulting average line to intersect the edges of the region of interest
    #the two edges consider are y = 6*imshape[0]/10 (the middle edge)
    #y = imshape[0] (the bottom edge)
    x1l = int((bl - imshape[0]) / (-1 * ml))
    y1l = imshape[0]
    x2l = int((bl - 6*imshape[0]/10) / (-1 * ml))
    y2l = int(6*imshape[0]/10)

    x1r = int((br - 6*imshape[0]/10) / (-1 * mr))
    y1r = int(6*imshape[0]/10)
    x2r = int((br - imshape[0]) / (-1 * mr))
    y2r = imshape[0]

    if (x2l < x1r):
        #draw the left line in green and right line in blue
        cv2.line(img, (x1l, y1l), (x2l, y2l), [0, 255, 0], thickness)
        cv2.line(img, (x1r, y1r), (x2r, y2r), [0, 0, 255], thickness)

    #store lines coeficients for next cycle
    previous_lines[0] = ml
    previous_lines[1] = bl
    previous_lines[2] = mr
    previous_lines[3] = br


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)
    draw_lines(line_img, lines)
    return line_img
```

Fig – 1.10



```
def weighted_img(img, initial_img, α=0.8, β=0.6, γ=0.):

    return cv2.addWeighted(initial_img, α, img, β, γ)
```

Test Images

Build our pipeline to work on the images in the directory "test_images"
We should make sure our pipeline works well on these images before we try the videos.

In [16]:
```
import os
os.listdir("test_images/")
```

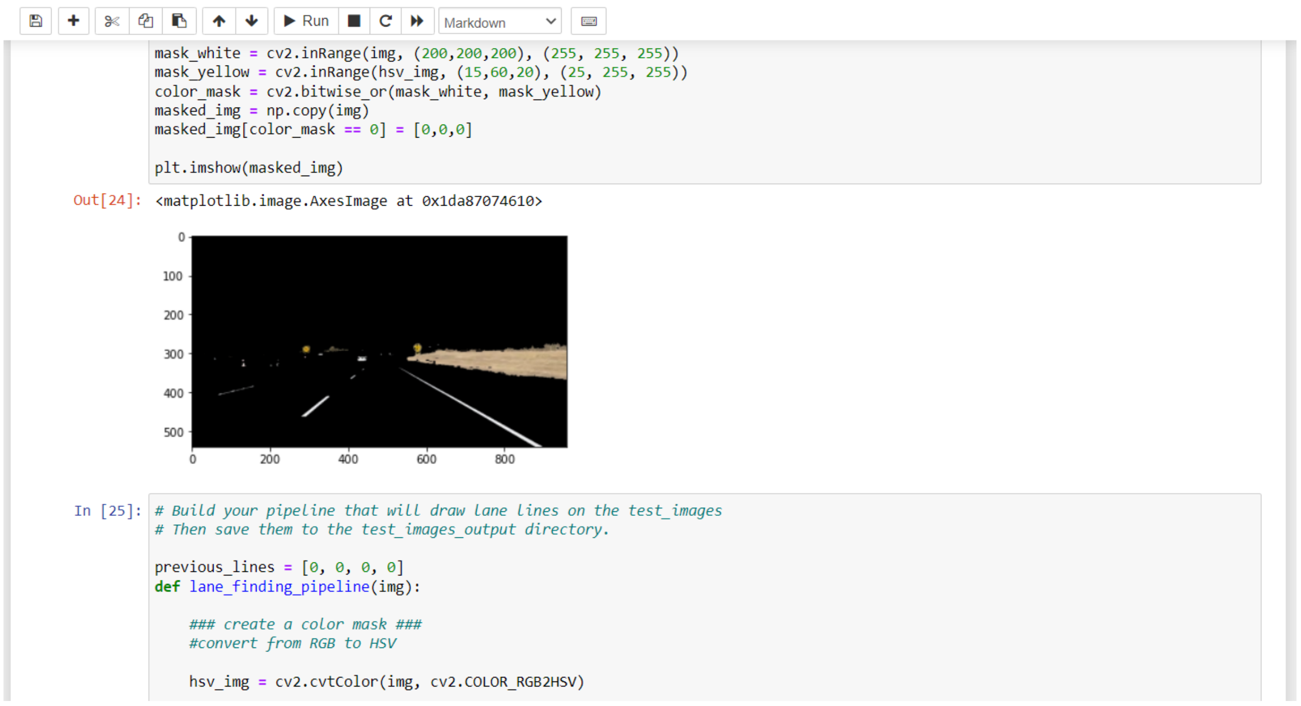
Out[16]:
```
['mountain.jpg',
 'solidWhiteCurve.jpg',
 'solidWhiteRight.jpg',
 'solidYellowCurve.jpg',
 'solidYellowCurve2.jpg',
 'solidYellowLeft.jpg',
 'whiteCarLaneSwitch.jpg']
```

Build a Lane Finding Pipeline

Build the pipeline and running our solution on all test_images. Make copies into the `test_images_output` directory, and we can use the images in our report.

In [24]:
```
img = mpimg.imread('test_images/solidWhiteCurve.jpg')

### color selection###
hsv_img = cv2.cvtColor(img, cv2.COLOR_RGB2HSV)
```

Fig – 1.11

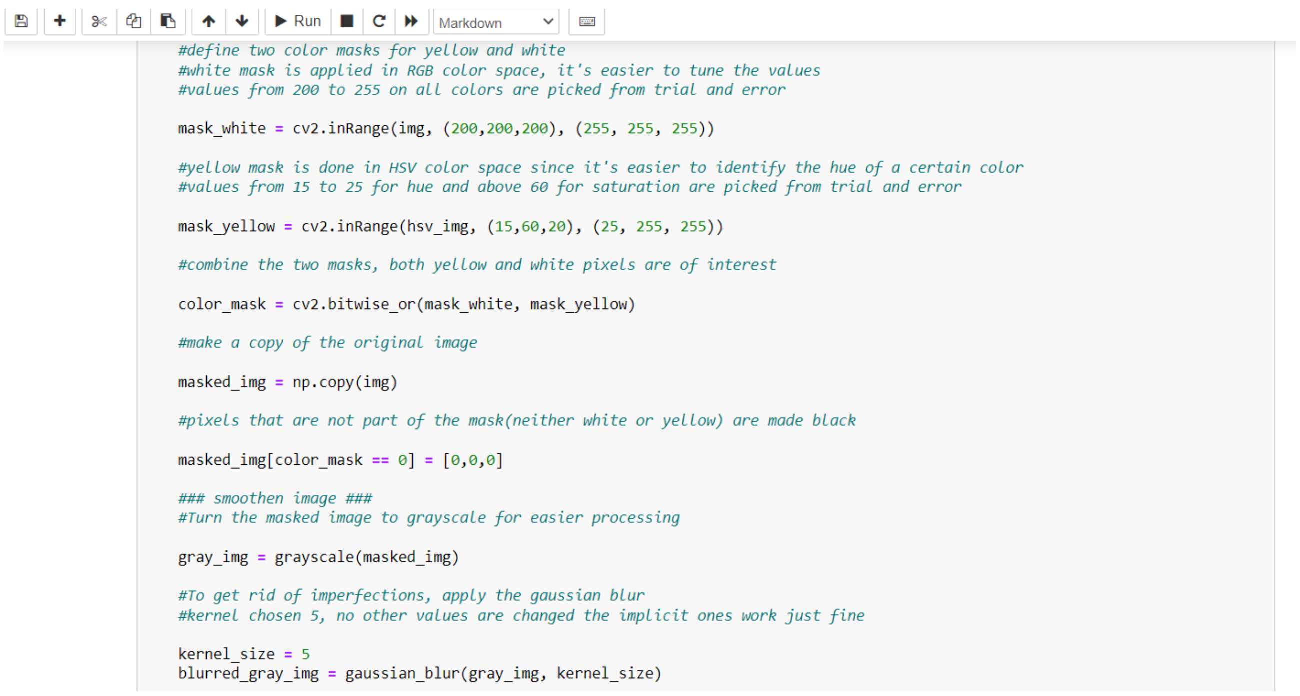


```
mask_white = cv2.inRange(img, (200,200,200), (255, 255, 255))
mask_yellow = cv2.inRange(hsv_img, (15,60,20), (25, 255, 255))
color_mask = cv2.bitwise_or(mask_white, mask_yellow)
masked_img = np.copy(img)
masked_img[color_mask == 0] = [0,0,0]

plt.imshow(masked_img)
```

Out[24]:
```
<matplotlib.image.AxesImage at 0x1da87074610>
```

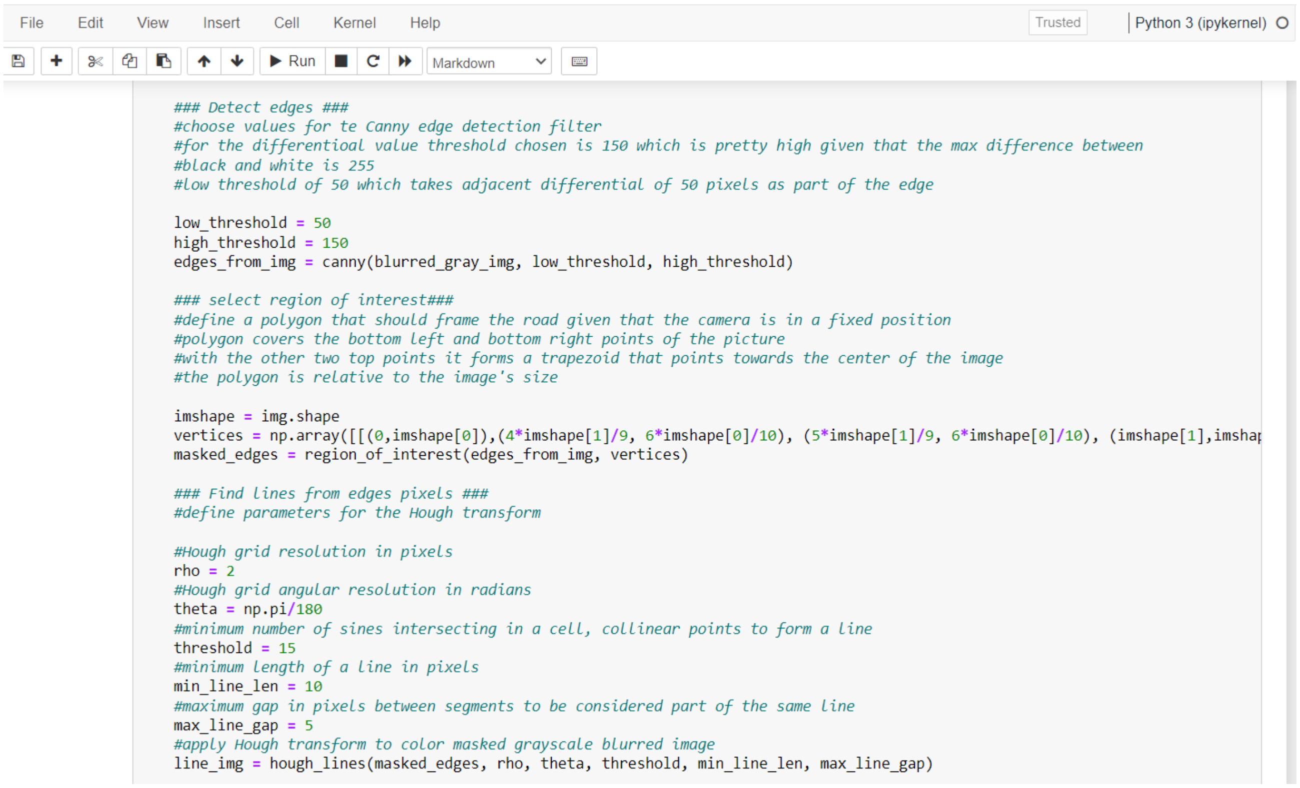
In [25]:
```
# Build your pipeline that will draw lane lines on the test_images
# Then save them to the test_images_output directory.

previous_lines = [0, 0, 0, 0]
def lane_finding_pipeline(img):

    ### create a color mask ###
    #convert from RGB to HSV

    hsv_img = cv2.cvtColor(img, cv2.COLOR_RGB2HSV)
```

Fig – 1.12



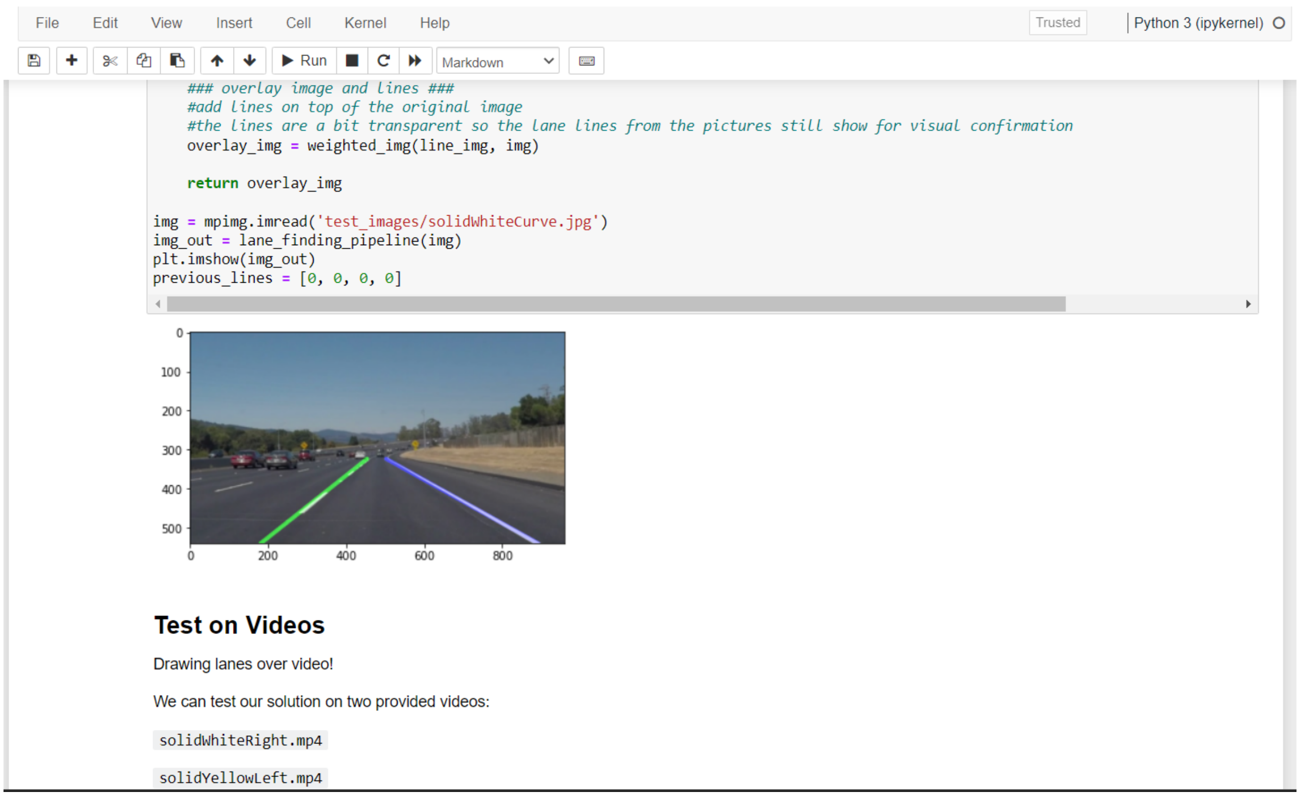
```python
#define two color masks for yellow and white
#white mask is applied in RGB color space, it's easier to tune the values
#values from 200 to 255 on all colors are picked from trial and error

mask_white = cv2.inRange(img, (200,200,200), (255, 255, 255))

#yellow mask is done in HSV color space since it's easier to identify the hue of a certain color
#values from 15 to 25 for hue and above 60 for saturation are picked from trial and error

mask_yellow = cv2.inRange(hsv_img, (15,60,20), (25, 255, 255))

#combine the two masks, both yellow and white pixels are of interest

color_mask = cv2.bitwise_or(mask_white, mask_yellow)

#make a copy of the original image

masked_img = np.copy(img)

#pixels that are not part of the mask(neither white or yellow) are made black

masked_img[color_mask == 0] = [0,0,0]

### smoothen image ###
#Turn the masked image to grayscale for easier processing

gray_img = grayscale(masked_img)

#To get rid of imperfections, apply the gaussian blur
#kernel chosen 5, no other values are changed the implicit ones work just fine

kernel_size = 5
blurred_gray_img = gaussian_blur(gray_img, kernel_size)
```

Fig – 1.13



```python
### Detect edges ###
#choose values for te Canny edge detection filter
#for the differentioal value threshold chosen is 150 which is pretty high given that the max difference between
#black and white is 255
#low threshold of 50 which takes adjacent differential of 50 pixels as part of the edge

low_threshold = 50
high_threshold = 150
edges_from_img = canny(blurred_gray_img, low_threshold, high_threshold)

### select region of interest###
#define a polygon that should frame the road given that the camera is in a fixed position
#polygon covers the bottom left and bottom right points of the picture
#with the other two top points it forms a trapezoid that points towards the center of the image
#the polygon is relative to the image's size

imshape = img.shape
vertices = np.array([[(0,imshape[0]),(4*imshape[1]/9, 6*imshape[0]/10), (5*imshape[1]/9, 6*imshape[0]/10), (imshape[1],imshap
masked_edges = region_of_interest(edges_from_img, vertices)

### Find lines from edges pixels ###
#define parameters for the Hough transform

#Hough grid resolution in pixels
rho = 2
#Hough grid angular resolution in radians
theta = np.pi/180
#minimum number of sines intersecting in a cell, collinear points to form a line
threshold = 15
#minimum length of a line in pixels
min_line_len = 10
#maximum gap in pixels between segments to be considered part of the same line
max_line_gap = 5
#apply Hough transform to color masked grayscale blurred image
line_img = hough_lines(masked_edges, rho, theta, threshold, min_line_len, max_line_gap)
```

Fig – 1.14



```
    ### overlay image and lines ###
    #add lines on top of the original image
    #the lines are a bit transparent so the lane lines from the pictures still show for visual confirmation
    overlay_img = weighted_img(line_img, img)

    return overlay_img

img = mpimg.imread('test_images/solidWhiteCurve.jpg')
img_out = lane_finding_pipeline(img)
plt.imshow(img_out)
previous_lines = [0, 0, 0, 0]
```

Test on Videos

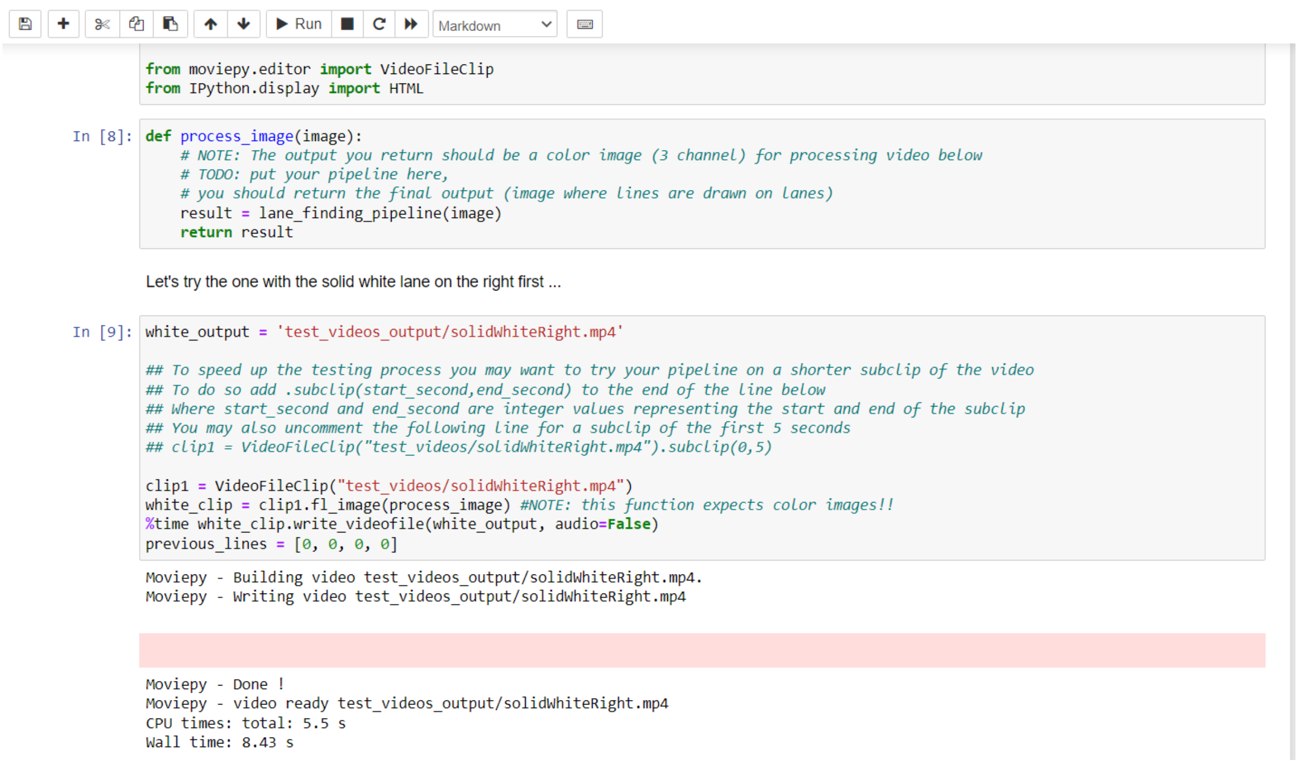
Drawing lanes over video!

We can test our solution on two provided videos:

`solidWhiteRight.mp4`

`solidYellowLeft.mp4`

Fig – 1.15



```
from moviepy.editor import VideoFileClip
from IPython.display import HTML
```

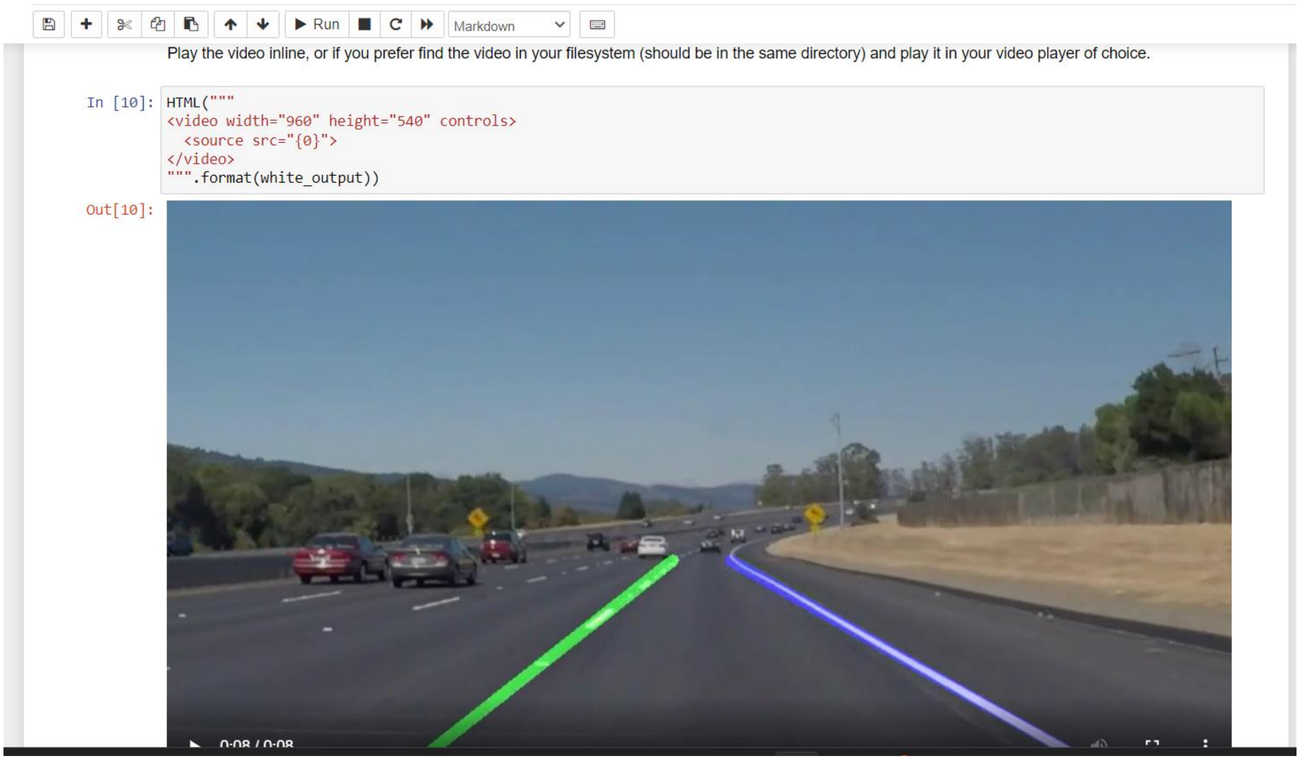
```
In [8]: def process_image(image):
            # NOTE: The output you return should be a color image (3 channel) for processing video below
            # TODO: put your pipeline here,
            # you should return the final output (image where lines are drawn on lanes)
            result = lane_finding_pipeline(image)
            return result
```

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

```
In [9]: white_output = 'test_videos_output/solidWhiteRight.mp4'

        ## To speed up the testing process you may want to try your pipeline on a shorter 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 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)
        previous_lines = [0, 0, 0, 0]
```

```
Moviepy - Building video test_videos_output/solidWhiteRight.mp4.
Moviepy - Writing video test_videos_output/solidWhiteRight.mp4

Moviepy - Done !
Moviepy - video ready test_videos_output/solidWhiteRight.mp4
CPU times: total: 5.5 s
Wall time: 8.43 s
```

Fig – 1.16

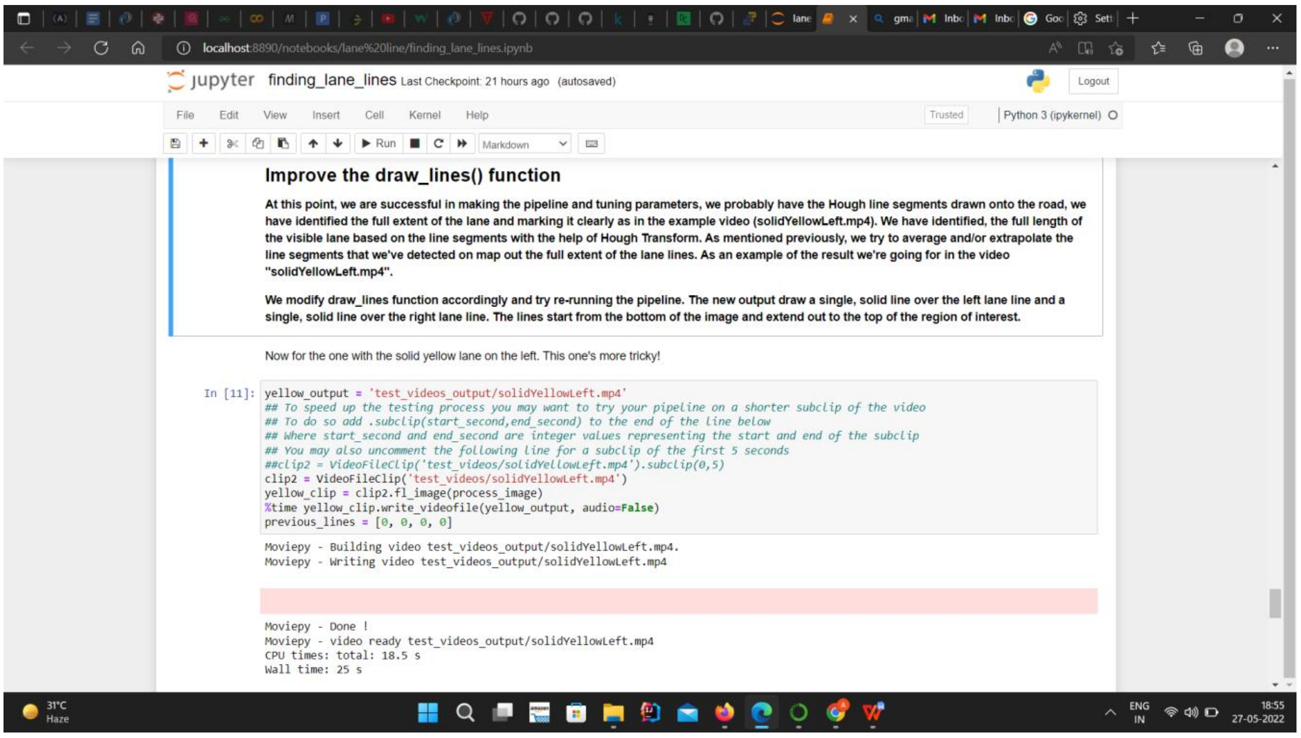


Play the video inline, or if you prefer find the video in your filesystem (should be in the same directory) and play it in your video player of choice.

```
In [10]: HTML("""
<video width="960" height="540" controls>
  <source src="{0}">
</video>
""".format(white_output))
```

```
Out[10]:
```

Fig – 1.17



localhost:8890/notebooks/lane%20line/finding_lane_lines.ipynb
jupyter
finding_lane_lines Last Checkpoint: 21 hours ago (autosaved)
Logout
File
Edit
View
Insert
Cell
Kernel
Help
Trusted
Python 3 (ipykernel)
Run
Markdown

Improve the draw_lines() function

At this point, we are successful in making the pipeline and tuning parameters, we probably have the Hough line segments drawn onto the road, we have identified the full extent of the lane and marking it clearly as in the example video (solidYellowLeft.mp4). We have identified, the full length of the visible lane based on the line segments with the help of Hough Transform. As mentioned previously, we try to average and/or extrapolate the line segments that we've detected on map out the full extent of the lane lines. As an example of the result we're going for in the video "solidYellowLeft.mp4".

We modify draw_lines function accordingly and try re-running the pipeline. The new output draw a single, solid line over the left lane line and a single, solid line over the right lane line. The lines 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 [11]: yellow_output = 'test_videos_output/solidYellowLeft.mp4'
## To speed up the testing process you may want to try your pipeline on a shorter 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 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)
previous_lines = [0, 0, 0, 0]
```

```
Moviepy - Building video test_videos_output/solidYellowLeft.mp4.
Moviepy - Writing video test_videos_output/solidYellowLeft.mp4

Moviepy - Done !
Moviepy - video ready test_videos_output/solidYellowLeft.mp4
CPU times: total: 18.5 s
Wall time: 25 s
```

Fig -1.18

```
In [12]: HTML("""
         <video width="960" height="540" controls>
           <source src="{0}">
         </video>
         """.format(yellow_output))
```

Out[12]:

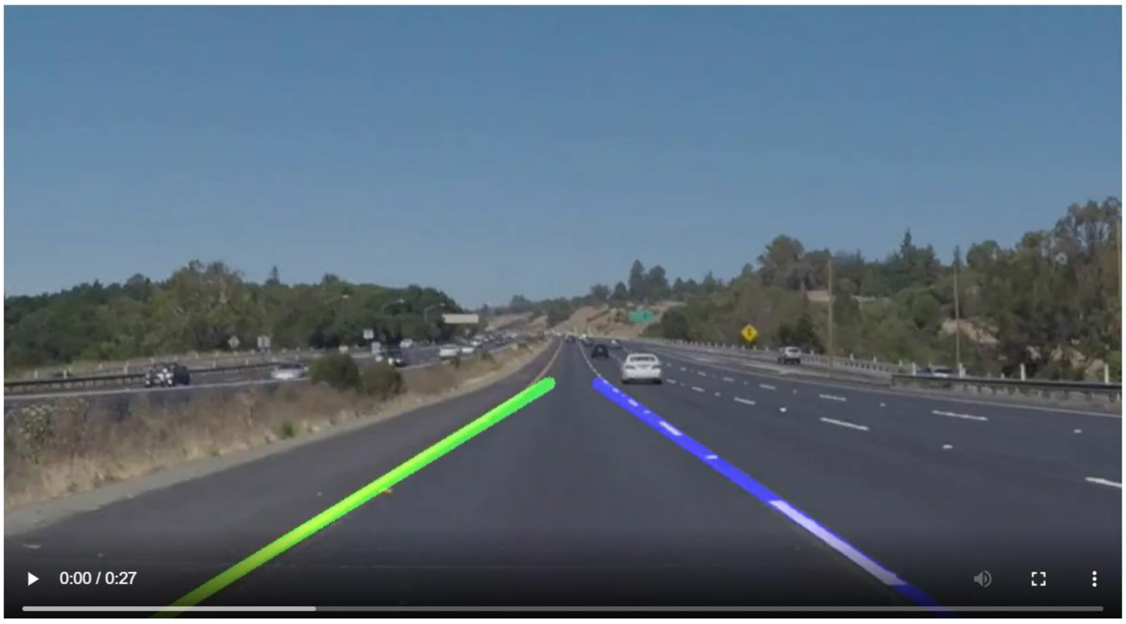


Fig -1.19

FUTURE SCOPE

The future scope of the lane detection consists of complicated surroundings thinking of the extraordinary environments inclusive of the all-weather conditions: sunny, rainy, fog, cloudy, mist vivid day light, darker, shadow or whilst there happens barriers and Speed Breakers, humps with inside the road. We can become aware of actual time lane detection. Further we are able to expand this version to the roads which do now no longer comprise lanes.

CONCLUSION

The Lane Detection is needed for today’s everyday life. Accidents on street are the primary hassle for authorities of any country. The foremost cause of injuries is surprising alternate in lane on speedy using roads. Most of this hassle happens whilst in terrible environmental situation whilst it unsuccessful to discover or withinside the road curves in which detection is just too tedious. Actual time imaginative and prescient[1]primarily based totally lane detection technique become developed. This system is efficient in detecting the road lanes.