Project: Finding Lane Lines on the Road

Develop a pipeline to identify lane lines on the road. You must apply it on a series of individual images, provided in the *test_images* folder.

Once you have a result that looks roughly like the image *line-segments-example* in the examples folder (also shown below), you'll need to try to average and/or extrapolate the line segments you've detected to map out the full extent of the lane lines.

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. 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).

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

Import Packages

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

Read in an Image

In [2]:
```
#reading in an image
image = cv2.imread('test_images/solidWhiteRight.jpg')
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

#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 'gr
```

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

Out[2]:
```
<matplotlib.image.AxesImage at 0x250db905b00>
```

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.

In [111]:

```
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.
    `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 th
    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=2):
    """
    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
    """
    for line in lines:
        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([]), minLineLengt
    line_img = np.zeros((img.shape[0], img.shape[1], 3), dtype=np.uint8)
    draw_lines(line_img, lines, thickness = 5)
    return line_img

def weighted_img(img, initial_img, alpha=0.8, beta=1., gamma=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 + γ
    NOTE: initial_img and img must be the same shape!
    """
    return cv2.addWeighted(initial_img, alpha, img, beta, gamma)
```

Test Images

Build your pipeline to work on the images in the directory "test_images"

In [112]:
```
import os

path = "test_images/"
files = os.listdir(path)
images = []

f, plots = plt.subplots((len(files)+3-1)//3, 3, figsize=(20,10))
plots = [plot for sublist in plots for plot in sublist]

for file, plot in zip(files, plots):
    image = cv2.cvtColor(cv2.imread(os.path.join(path, file)), cv2.COLOR_BGR2RGB)
    plot.set_title(file)
    plot.imshow(image)
    images.append((image, file))
```

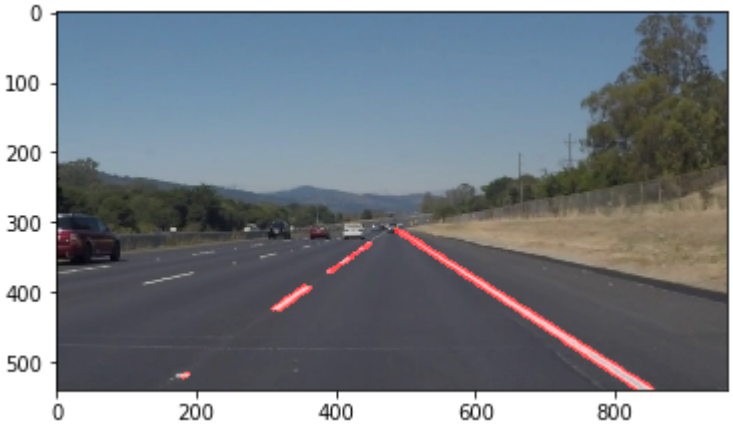
Build a Lane Finding Pipeline

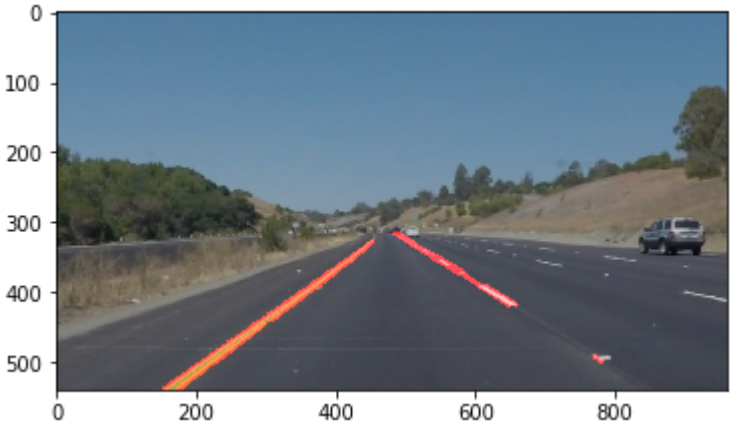
Build the pipeline and run your solution on all test_images.

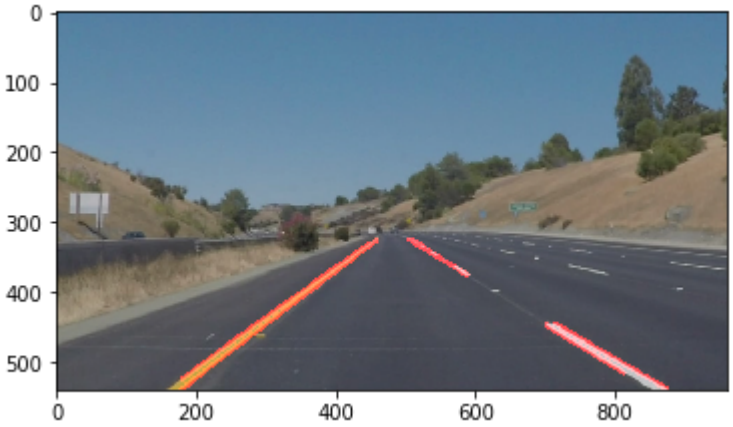
Try tuning the various parameters, especially the low and high Canny thresholds as well as the Hough lines parameters.

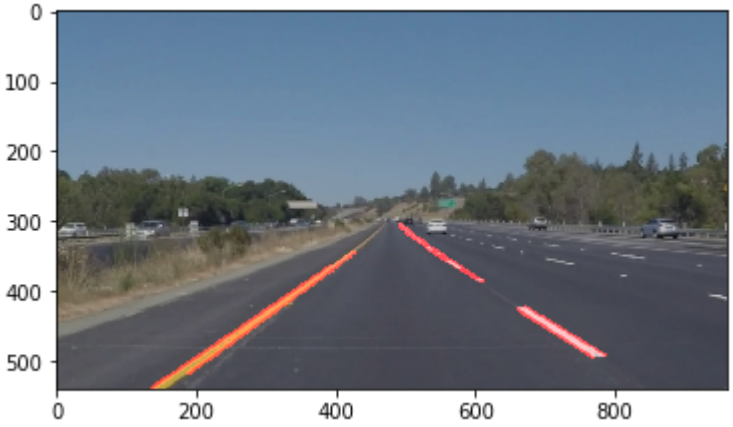
In [116]:
```
# TODO: Build your pipeline that will draw lane lines segments on the test_images
for i in range(0,5):
    gray_img = grayscale(images[i][0])
    canny_img = canny(gray_img, 180, 250)
    height = 600
    width = 1000
    vertices = [(0, height),(width / 2, height / 2),(width, height),]
    crop_img = region_of_interest(canny_img,(np.array([vertices],np.int32),))
    hough_img = hough_lines(crop_img, 8, np.pi / 180, 150, 5, 25)
    final_img = weighted_img(hough_img, images[i][0])

    plt.imshow(final_img)
    plt.show()
```





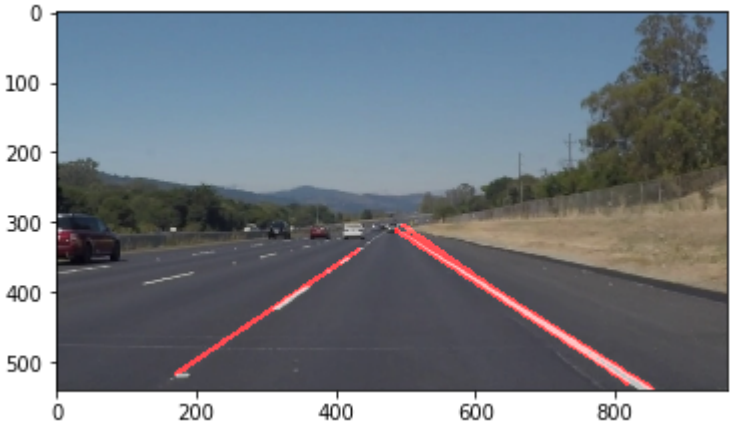


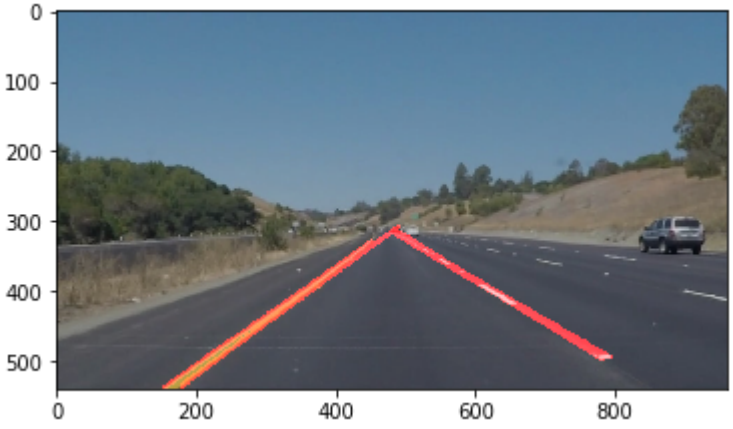


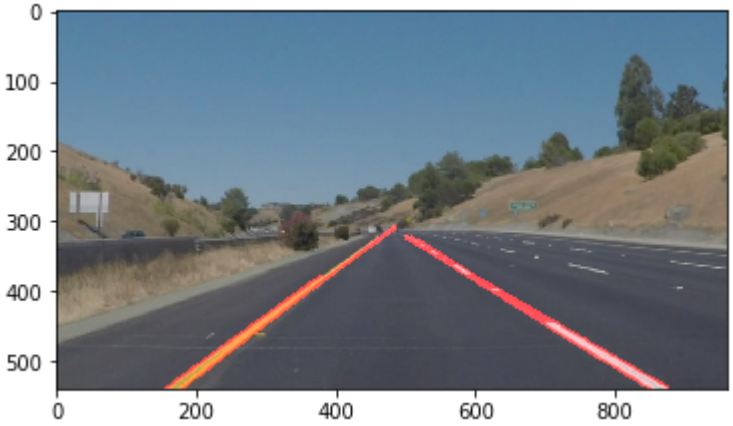
Improve the draw_lines() function

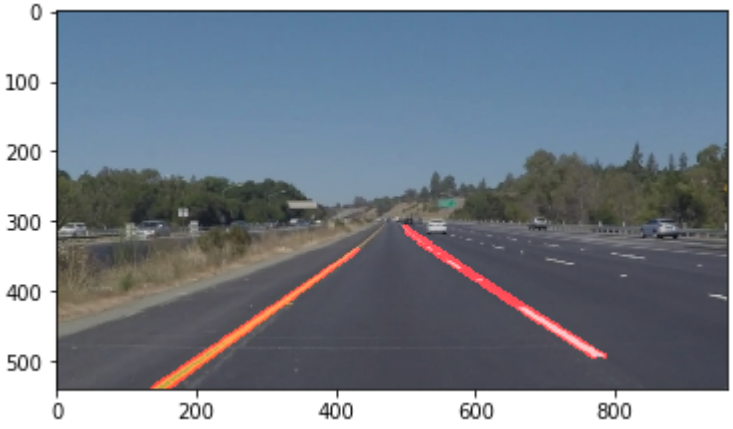
At this point, you should have the Hough line segments drawn onto the road. Extend your code to define a line to run the full length of the visible lane based on the line segments you identified with the Hough Transform. Try to average and/or extrapolate the line segments you've detected to map out the full extent of the lane lines. The 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.

In [114]:
```
# TODO: Build your pipeline that will draw complete lane lines on the test_images
for i in range(0,5):
    gray_img = grayscale(images[i][0])
    canny_img = canny(gray_img, 180, 250)
    height = 600
    width = 1000
    vertices = [(0, height),(width / 2, height / 2),(width, height),]
    crop_img = region_of_interest(canny_img,(np.array([vertices],np.int32),))
    hough_img = hough_lines(crop_img, 7, np.pi / 180, 170, 140, 180)
    final_img = weighted_img(hough_img, images[i][0])
    plt.imshow(final_img)
    plt.show()
```






0
100
200
300
400
500
0
200
400
600
800


0
100
200
300
400
500
0
200
400
600
800