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, see [this forum post (https://carnd-forums.udacity.com/cq/viewquestion.action?spaceKey=CAR&id=29496372&questionTitle=finding-lanes---import-cv2-fails-even-though-python-in-the-terminal-window-has-no-problem-with-import-cv2)](https://carnd-forums.udacity.com/cq/viewquestion.action?spaceKey=CAR&id=29496372&questionTitle=finding-lanes---import-cv2-fails-even-though-python-in-the-terminal-window-has-no-problem-with-import-cv2) for more troubleshooting tips.

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 [2]:

```
#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
```

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

Out[2]:

```
<matplotlib.image.AxesImage at 0x7b406a0>
```

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 [3]:

```
import math
# import numpy.polynomial.polynomial as poly

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 on the input i
    if len(img.shape) > 2:
        print("img.shape is larger than 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, mask


def draw_lines(img, lines, color=[0, 255, 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
    """
    for line in lines:
        for x1,y1,x2,y2 in line:
            cv2.line(img, (x1, y1), (x2, y2), color, thickness)

def draw_lane_lines(img, lines, color=[255, 0, 0], thickness=12):
    """
    draws the lane lines. lines must be an array of (2) lines.
    a line consists of slope and y-intercept

    returns true if succeeded, false else
    """
    if dev_mode:
        print("{} lines before reduction".format(lines.shape[0]))

    # minimum slope will help to filter out ourliers like contours on the street that are n
    # part of a lane marking but look similar
    min_slope = 0.2

    # get empty arrays to be filled up
    left_lane_lines_x = []   # x coordinates of dots that are part of lines that belong to
    left_lane_lines_y = []   # y coordinates of dots that are part of lines that belong to
    right_lane_lines_x = []  # same as above, just for the right lane
    right_lane_lines_y = []  # same as above, just for the right lane
    left_lane = []           # lane representations using [slope, y_intercept]
    right_lane = []          # same as above, just for the right lane

    i = 1
    for line in lines:
            for x1,y1,x2,y2 in line:
                slope = ((y2-y1)/(x2-x1))
                y_intercept = y1 - x1 * slope

                side = "default"
                if (slope < 0):
                    if slope < -1 * min_slope: # only if it is no outlier
                        left_lane_lines_x.extend([x1,x2])
                        left_lane_lines_y.extend([y1,y2])
                        left_lane.append([slope, y_intercept])
                        side = "left"
                    else:
                        side = "left (skipped)"
                else:
                    if slope > min_slope:
                        right_lane_lines_x.extend([x1,x2])
                        right_lane_lines_y.extend([y1,y2])
                        right_lane.append([slope, y_intercept])
                        side = "right"
                    else:
                        side = "right (skipped)"

                if dev_mode:
                    print("slope of line {} is {}, y_intercept={} ({}).".format(i, slope, y

    # if one of the two lanes is missing: return
    if len(left_lane_lines_x) == 0 or len(right_lane_lines_x) == 0:
        if dev mode:
```

```
            print("lane is missing, returning...")
        return False

    # convert the standard, python arays into np.Arrays to be able to apply stuff like mean
    left_lane = np.array(left_lane)
    right_lane = np.array(right_lane)

    # polyfit all dots that are part of a line in a lane specific manner. Even higher order
    # polygons are easily possible using this method without having to rewrite the whole
    # code again
    lin_l, const_l = np.polyfit(left_lane_lines_x, left_lane_lines_y, 1)
    lin_r, const_r = np.polyfit(right_lane_lines_x, right_lane_lines_y, 1)

    if dev_mode:
        print("right line y-intercept={}, slope={}".format(const_r, lin_r))
        print("left line y-intercept={}, slope={}".format(const_l, lin_l))

    # print left lane
    x_temp = int(round(im_dim_x/2 - im_x_width_distance/2))
    cv2.line(img, (0, int(round(const_l))), (x_temp, int(round(const_l + lin_l * x_temp))),

    # print right lane
    x_temp = int(round(im_dim_x/2 + im_x_width_distance/2))
    cv2.line(img, (x_temp, int(round( const_r + lin_r * x_temp))),
             (im_dim_x, int(round(const_r + lin_r * im_dim_x))), color, thickness)

    # return success
    return True

def hough_lines(img, rho, theta, threshold, min_line_len, max_line_gap, dev_mode = False):
    """
    `img` should be the output of a Canny transform.

    Returns an image with hough lines drawn.
    """
    success = False

    iteration = 1        # current iteration
    max_iterations = 10  # max number of iterations to avoid endless loop

    # loop until success or max_iterations reached
    while(success == False and iteration <= max_iterations):

        if dev_mode:
            print("iteration {} failed, trying again with lower restriction".format(iterati
        if iteration > 1:
            # from the second iteration ongoing, decrease the requirements for a line to be
            min_line_len = min_line_len - 3
            max_line_gap = max_line_gap + 3

        lines = cv2.HoughLinesP(img, rho, theta, threshold, np.array([]), minLineLength=mir

        if lines is not None:
            # draw both images in any case. This is an performance improvement #TODO.
            # If perfomance is a major concern, drawing the lines can be skipped (line_img
            line_img = np.zeros((img.shape[0], img.shape[1], 3), dtype=np.uint8)
            lane_img = np.zeros((img.shape[0], img.shape[1], 3), dtype=np.uint8)

            draw_lines(line_img, lines)

            success = draw_lane_lines(lane_img, lines)
```

```
        iteration = iteration + 1

    return lane_img, line_img, lines

# 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)
```

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 [4]:

```
import os
os.listdir("test_images/")
```

Out[4]:

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

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 [5]:

```
dev_mode = False
im = os.listdir("test_images/")[0]

# init all global vars
im_dim_x = 0
im_dim_y = 0
im_margin_x = 0
im_y_top = 0
im_x_width_distance = 0
```

In [6]:

```
def add_lane_markings_to_image(image, video_mode = False, draw_mode = 'lane'):

    # get global variables regarding images dimensions (needed by many functions)
    global im_dim_x
    global im_dim_y
    global im_margin_x
    global im_y_top
    global im_x_width_distance

    # only print text if not in video mode
    if not video_mode:
        print("original image")
        plt.imshow(image)
        plt.show()

    # set basic image size# set basic image size for every incoming image
    # necessary to master the challenge ;)
    imshape = image.shape
    im_dim_x = imshape[1]
    im_dim_y = imshape[0]
    im_margin_x = int(round(im_dim_x / 48)) # margin from the border of the image to x axis
    im_y_top = int(round(im_dim_y * 0.6111)) # y pixel from the top to the end of region of
    im_x_width_distance = int(round(im_dim_x * 0.1041)) # top width of polygon. 100 for 960

    # select pixels by color
    color_select = np.copy(image)
    safety = 10
    red_threshold = 224 - safety
    green_threshold = 137 - safety
    blue_threshold = 40 - safety
    rgb_threshold = [red_threshold, green_threshold, blue_threshold]
    # Do a boolean or with the "|" character to identify
    # pixels below the thresholds
    thresholds = (image[:,:,0] < rgb_threshold[0]) \
                | (image[:,:,1] < rgb_threshold[1]) \
                | (image[:,:,2] < rgb_threshold[2])
    color_select[thresholds] = [0,0,0]
    if dev_mode:
        print("color_select")
        plt.imshow(color_select)
        plt.show()

    # convert to grayscale image to save computational power (one channel instead of three)
    gray = cv2.cvtColor(color_select,cv2.COLOR_RGB2GRAY)
    if dev_mode:
        print("gray")
        plt.imshow(gray, cmap='gray')
        plt.show()

    # Define a kernel size and apply Gaussian smoothing
    kernel_size = 5
    blur_gray = gaussian_blur(gray,kernel_size)
    if dev_mode:
        print("blur_gray")
        plt.imshow(blur_gray, cmap='gray')
        plt.show()

    # Define our parameters for Canny and apply
    low_threshold = 70
```

```
    high_threshold = 150

    edges = canny(blur_gray, low_threshold, high_threshold)

    # mask image using a polygon that is defined dynamically based on the image size
    vertices = np.array([[(im_margin_x,im_dim_y),
                          (im_dim_x/2 - im_x_width_distance/2, im_y_top),
                          (im_dim_x/2 + im_x_width_distance/2, im_y_top),
                          (im_dim_x-im_margin_x,im_dim_y)]], dtype=np.int32)
    masked_edges, mask = region_of_interest(edges, vertices)
    if dev_mode:
        print("masked_edges")
        plt.imshow(masked_edges, cmap='gray')
        plt.show()

        # visualize masking
        color_mask = np.dstack((mask, mask, mask))
        orig_and_mask = weighted_img(img=image, initial_img=color_mask, α=0.5, β=0.5, λ=0.5
        print("orig_and_mask")
        plt.imshow(orig_and_mask, cmap='gray')
        plt.show()


    # Define the Hough transform parameters
    # Make a blank the same size as our image to draw on
    rho =  1             # distance resolution in pixels of the Hough grid
    theta = np.pi/180    # angular resolution in radians of the Hough grid
    threshold = 15       # minimum number of votes (intersections in Hough grid cell)
    min_line_length = 25  # minimum number of pixels making up a line
    max_line_gap = 20    # maximum gap in pixels between connectable line segments
    line_image = np.copy(image)*0 # creating a blank to draw lines on
    if dev_mode:
        print("rho={}, theta={}, threshold={}, min_line_length={}, max_line_gap={}".format(
                rho, theta, threshold, min_line_length, max_line_gap))
    # Run Hough on edge detected image
    # Output "lines" is an array containing endpoints of detected line segments
    lane_img, line_image, lines = hough_lines(masked_edges, rho, theta, threshold,
                                min_line_length, max_line_gap, dev_mode = dev_mode)
    if dev_mode:
        print("line_image")
        plt.imshow(line_image)
        plt.show()
    if dev_mode:
        print("lane_img")
        plt.imshow(lane_img)
        plt.show()

    # select image based on draw
    if draw_mode is 'lane':
        final_img = lane_img
    else:
        final_img = line_image

    # Draw the lines on the edge image
    lines_edges = cv2.addWeighted(image, 1, final_img, 1, 0)
    if not video_mode:
        print("lines_edges")
        plt.imshow(lines_edges)
        plt.show()

    return lines_edges
```

In [26]:

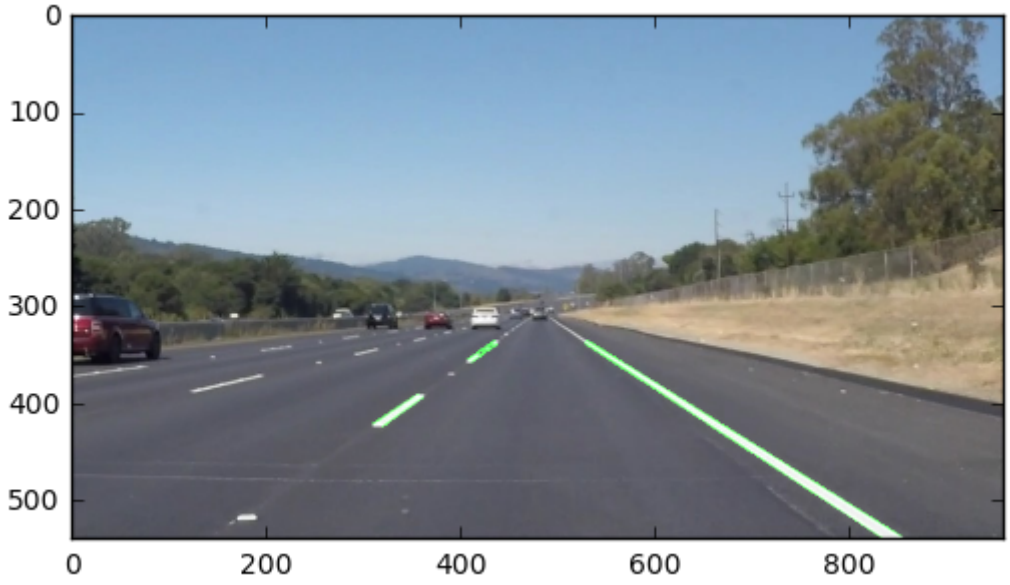
```
# TODO: Build your pipeline that will draw lane lines on the test_images
# then save them to the test_images directory.

dev_mode = False
all_images = os.listdir("test_images/")
for im in all_images:
    print("working image " + im)
    image = mpimg.imread('test_images/' + im)

    image_annotated = add_lane_markings_to_image(image, video_mode=False,  draw_mode = 'lir

    mpimg.imsave('test_images/line_' + im, image_annotated)

for im in all_images:
    print("working image " + im)
    image = mpimg.imread('test_images/' + im)

    image_annotated = add_lane_markings_to_image(image, video_mode=False,  draw_mode = 'lir

    mpimg.imsave('test_images/lane_'+ im, image_annotated)
```

```
working image solidWhiteCurve.jpg
original image
```

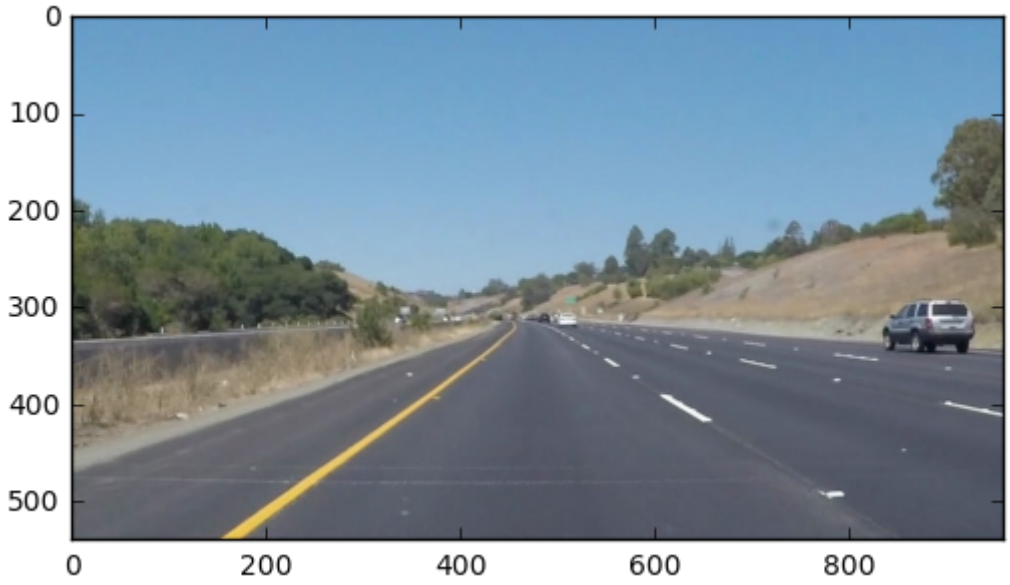

```
lines_edges
```


```
working image solidWhiteRight.jpg
original image
```

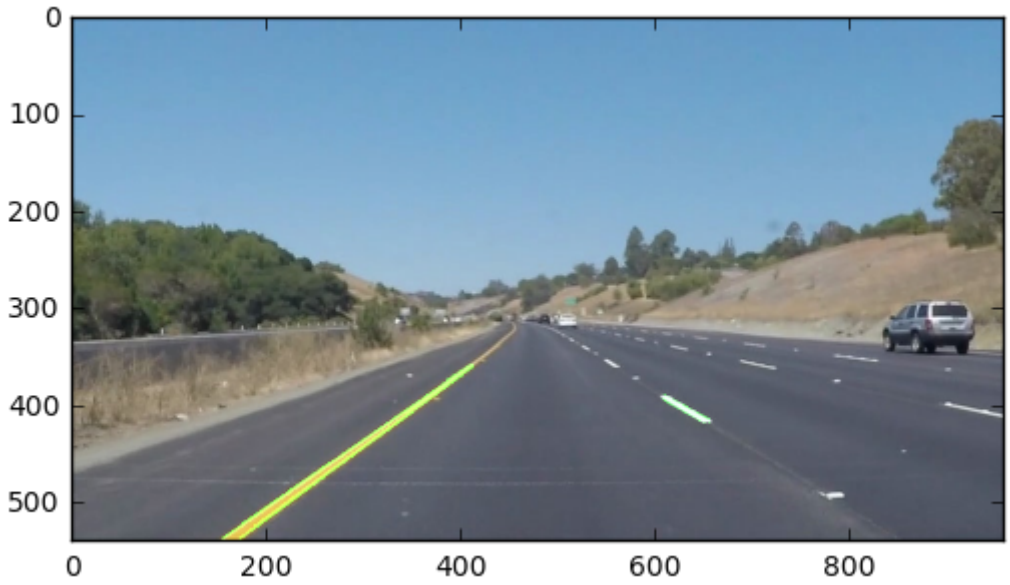

```
lines_edges
```



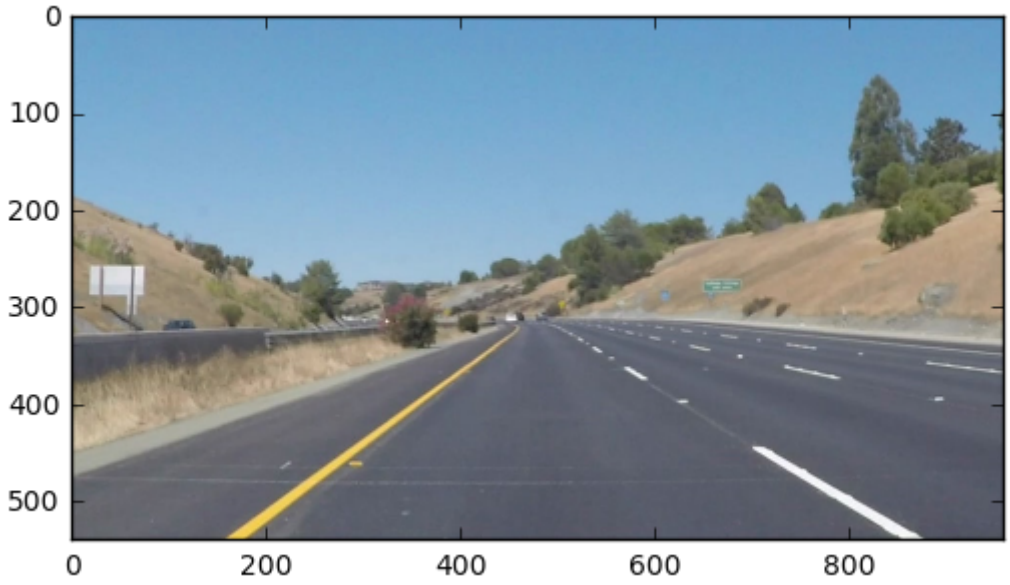
working image solidYellowCurve.jpg
original image



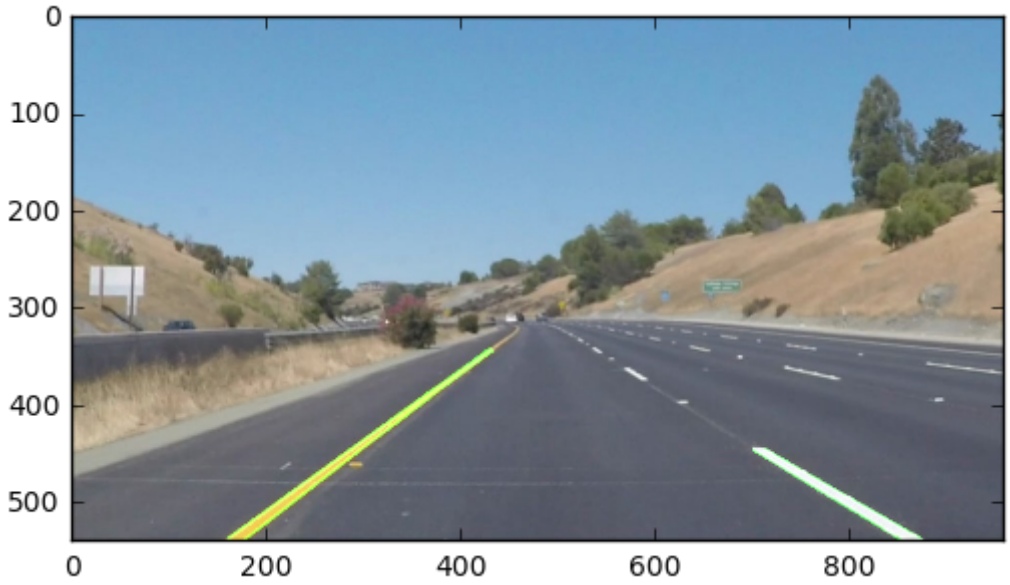
lines_edges



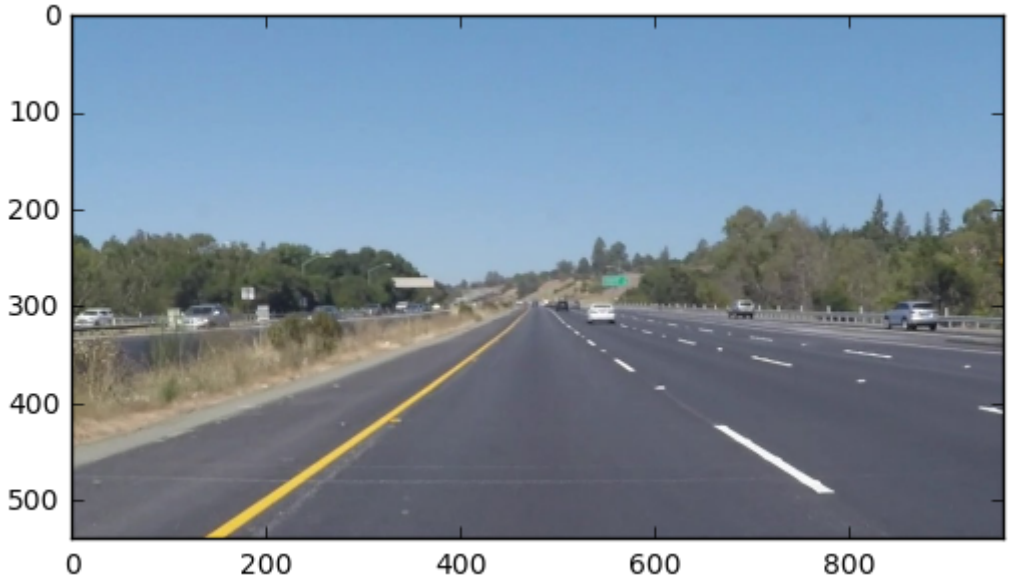
working image solidYellowCurve2.jpg
original image



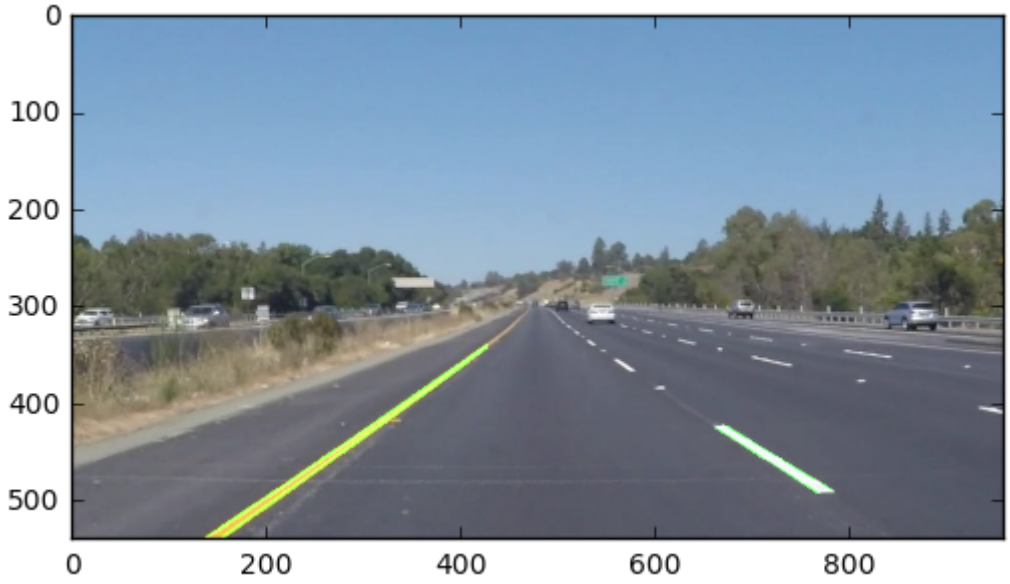
lines_edges



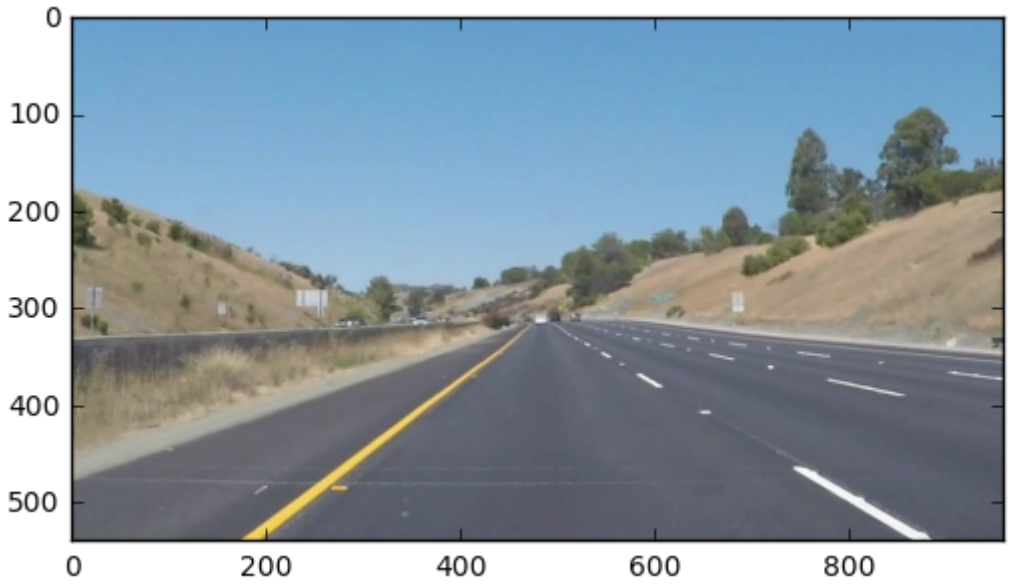
working image solidYellowLeft.jpg
original image



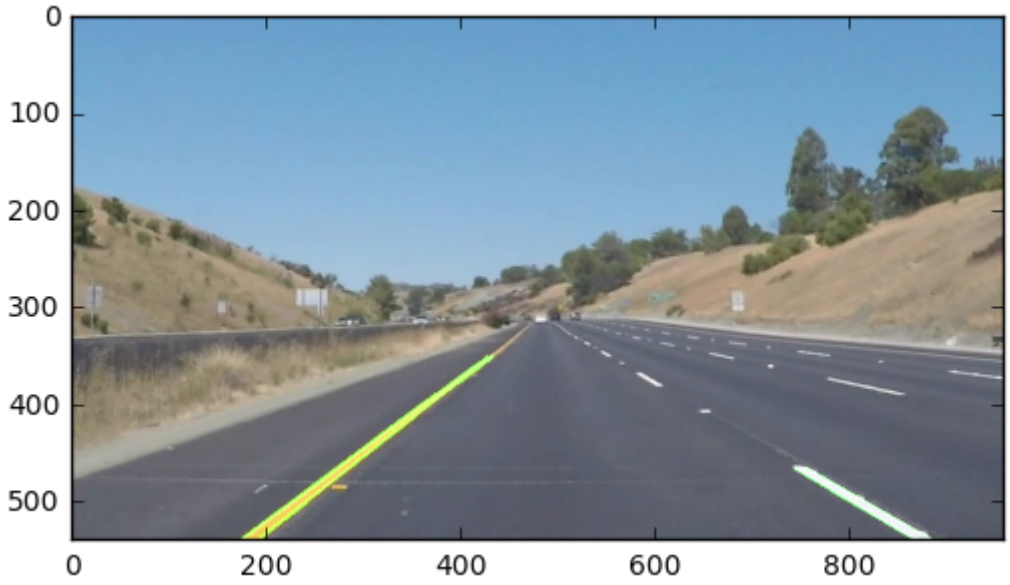
lines_edges



working image whiteCarLaneSwitch.jpg
original image



lines_edges

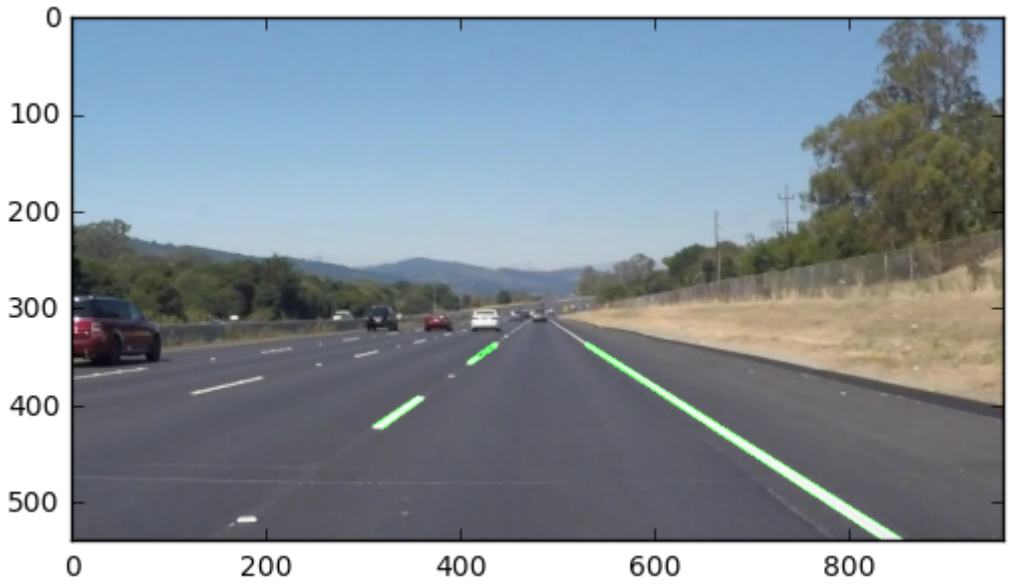


working image solidWhiteCurve.jpg
original image

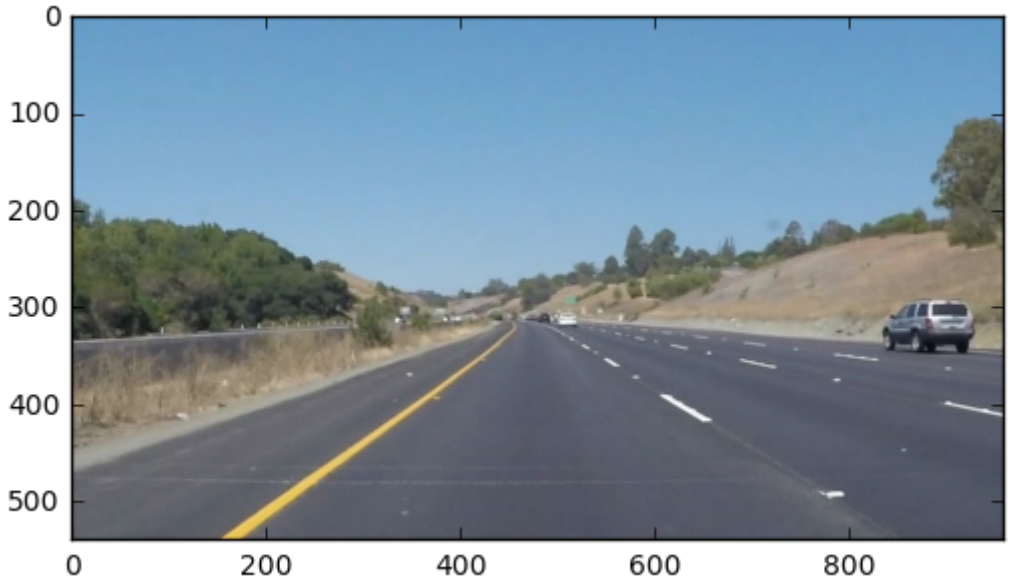
lines_edges

```
working image solidWhiteRight.jpg
original image
```

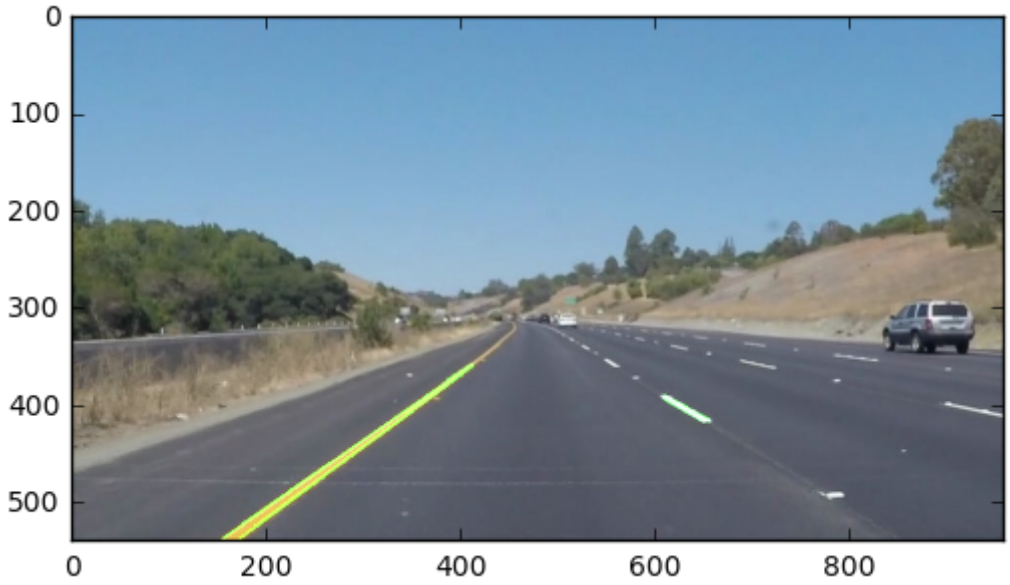

```
lines_edges
```



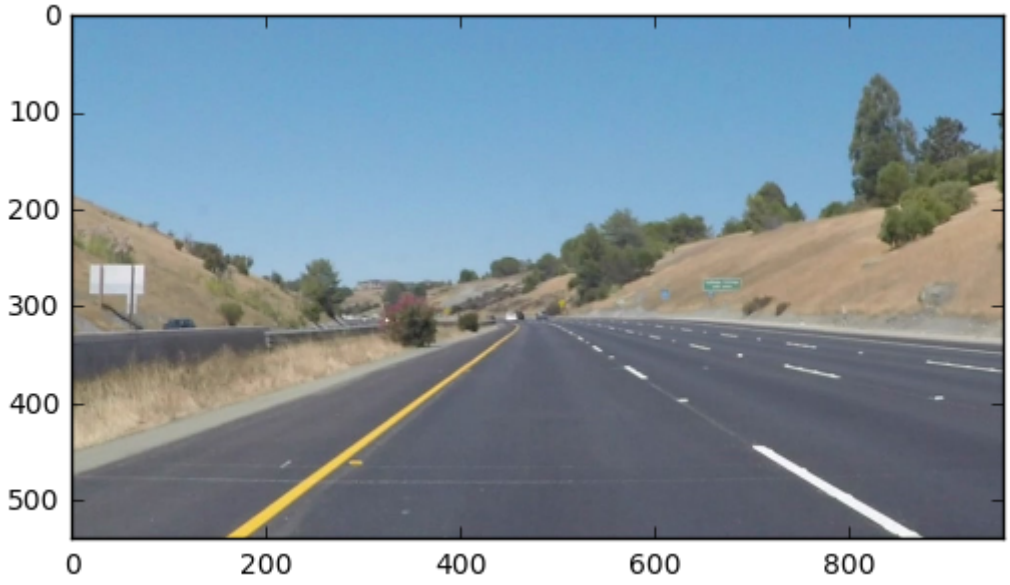

```
working image solidYellowCurve.jpg
original image
```



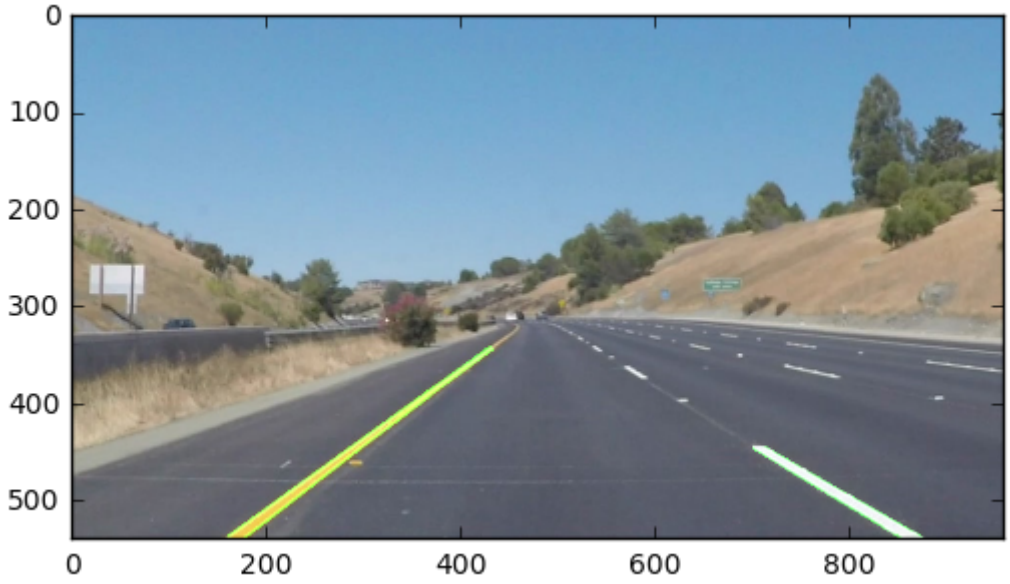
lines_edges



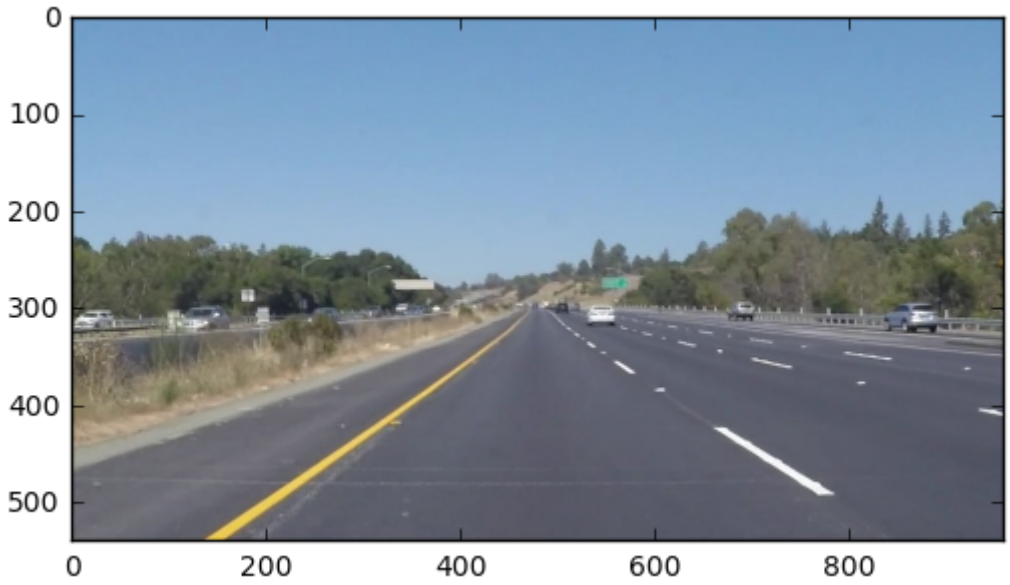
working image solidYellowCurve2.jpg
original image



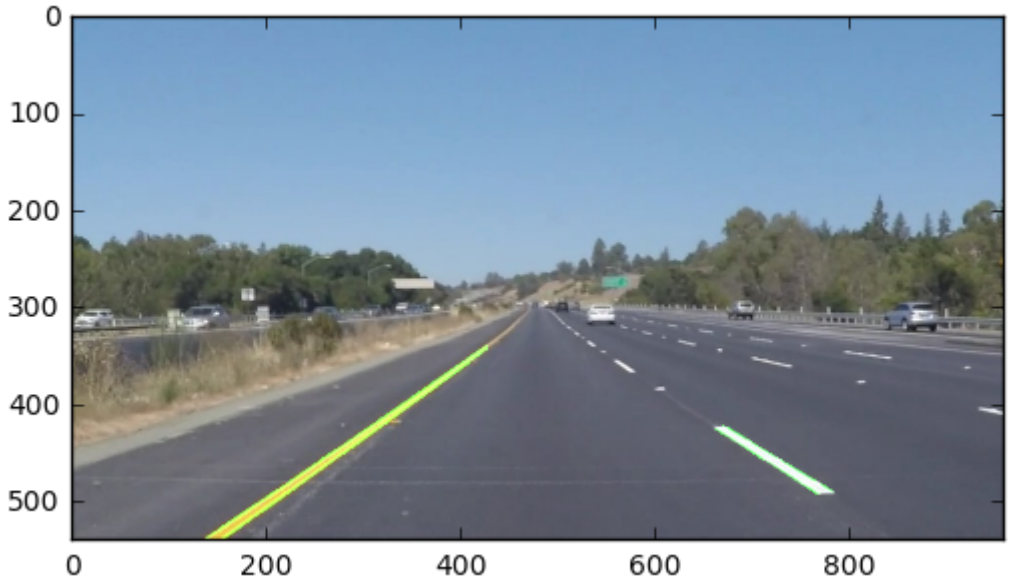
lines_edges



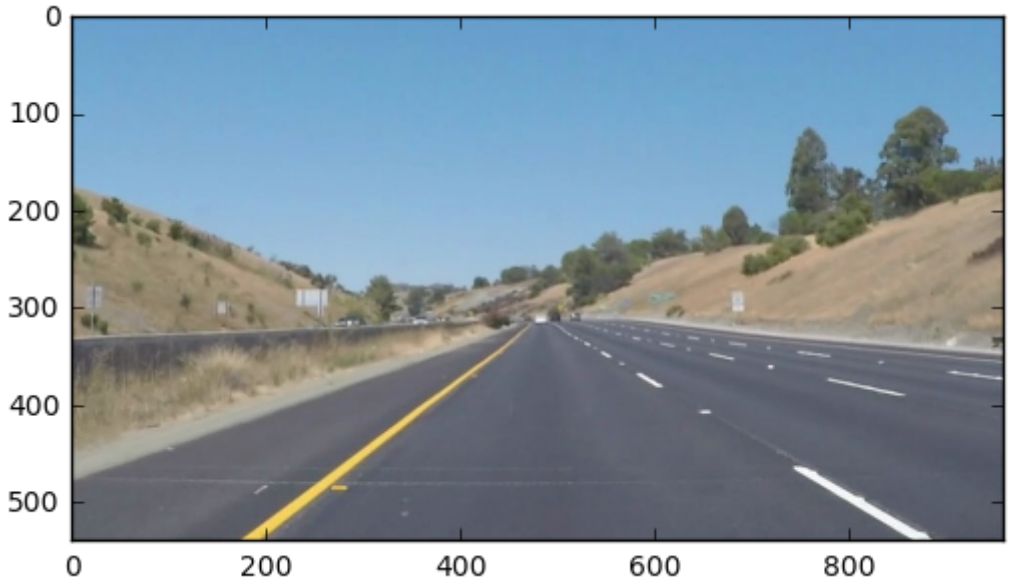
working image solidYellowLeft.jpg
original image



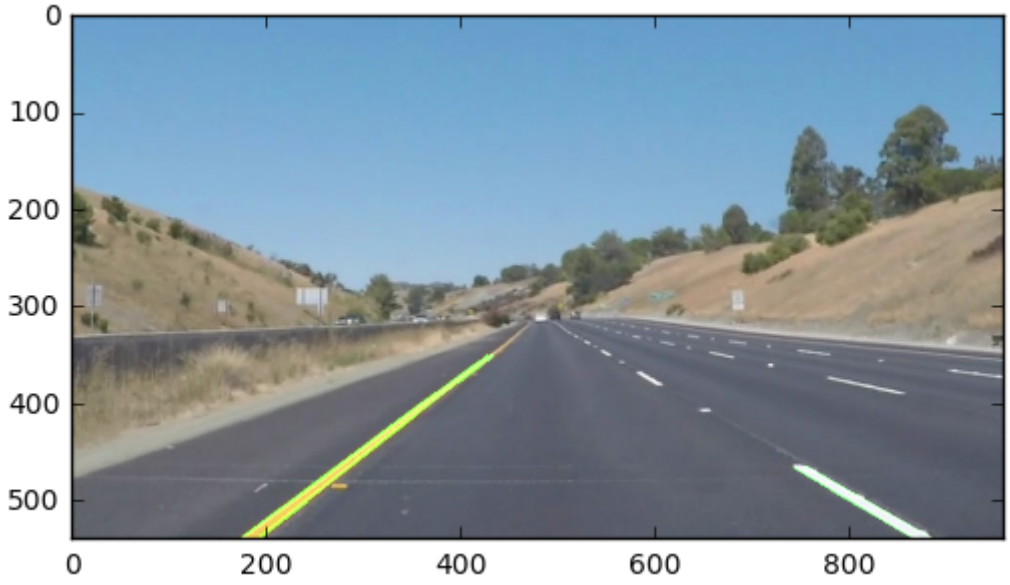
lines_edges




```
working image whiteCarLaneSwitch.jpg
original image
```




```
lines_edges
```



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, check out this forum post (https://carnd-forums.udacity.com/questions/22677062/answers/22677109) 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://carnd-forums.udacity.com/display/CAR/questions/26218840/import-videofileclip-error) for more troubleshooting tips across operating systems.

In [8]:

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

In [9]:

```
def process_image(image):
    return add_lane_markings_to_image(image, video_mode=True, draw_mode = 'line')

def process_image_lane(image):
    return add_lane_markings_to_image(image, video_mode=True, draw_mode = 'lane')
```

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

In [10]:

```
white_output = 'test_videos_output/solidWhiteRight.mp4'
## To speed up the testing process you may want to try your pipeline on a shorter subclip o
## 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 t
## 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)
```

```
[MoviePy] >>>> Building video test_videos_output/solidWhiteRight.mp4
[MoviePy] Writing video test_videos_output/solidWhiteRight.mp4

100%|████████████████████████████████████████████████████████████████████████████████
| 221/222 [00:09<00:00, 22.96it/s]

[MoviePy] Done.
[MoviePy] >>>> Video ready: test_videos_output/solidWhiteRight.mp4

Wall time: 10.7 s
```

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 [11]:

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

Out[11]:

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 [12]:

```
dev_mode = False
```

In [13]:

```
yellow_output = 'test_videos_output/solidYellowLeft.mp4'
## To speed up the testing process you may want to try your pipeline on a shorter subclip o
## 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 t
## 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_lane)
%time yellow_clip.write_videofile(yellow_output, audio=False)
```

```
[MoviePy] >>>> Building video test_videos_output/solidYellowLeft.mp4
[MoviePy] Writing video test_videos_output/solidYellowLeft.mp4

100%|████████████████████████████████████████████████████████████████
████████████████████████████▉| 681/682 [00:33<00:00, 23.39it/s]

[MoviePy] Done.
[MoviePy] >>>> Video ready: test_videos_output/solidYellowLeft.mp4

Wall time: 35.2 s
```

In [14]:

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

Out[14]:

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.

Optional Challenge

Try your lane finding pipeline on the video below. Does it still work? Can you figure out a way to make it more robust? If you're up for the challenge, modify your pipeline so it works with this video and submit it along with the rest of your project!

In [15]:

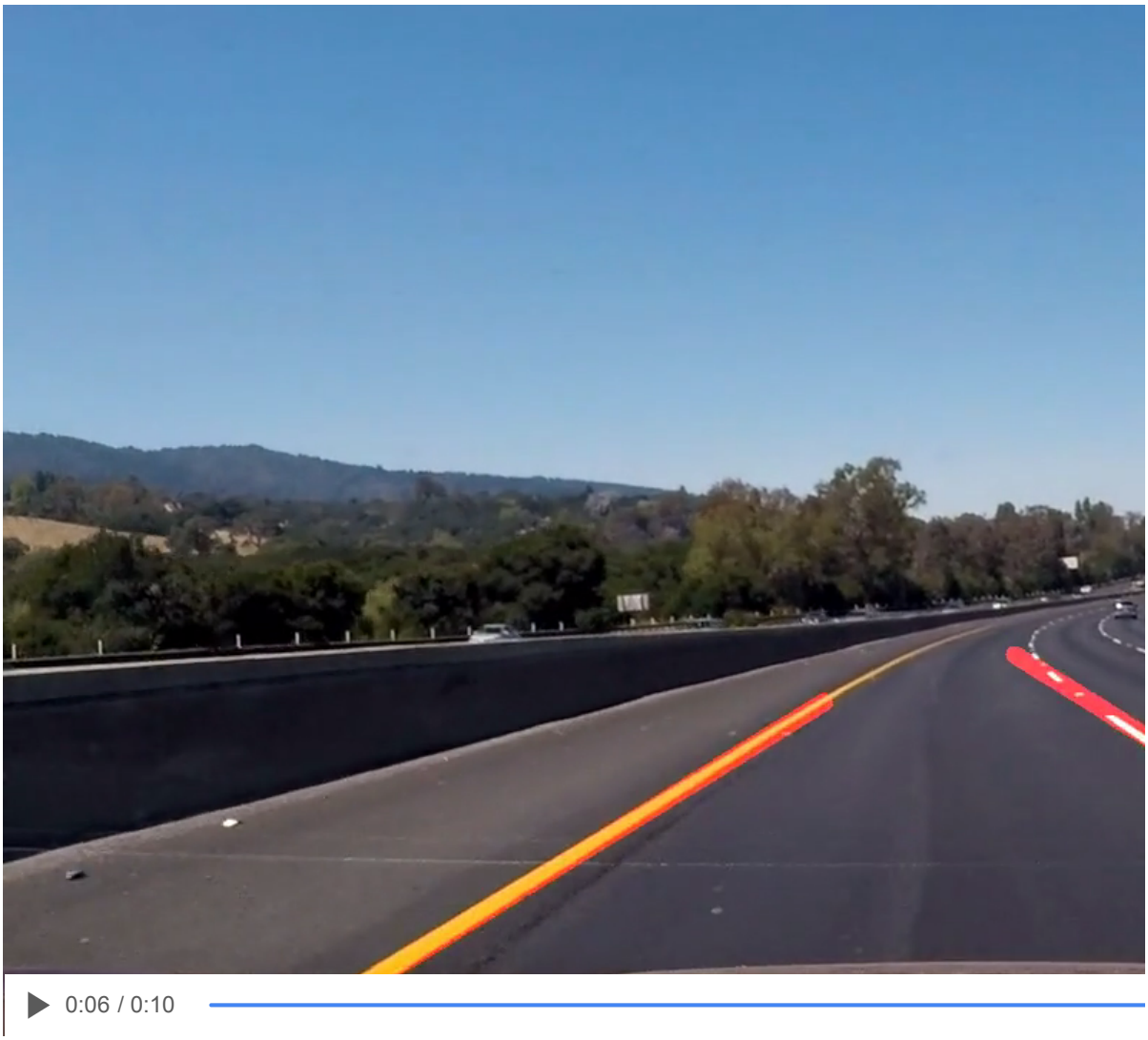
```
dev_mode = False
challenge_output = 'test_videos_output/challenge.mp4'
## To speed up the testing process you may want to try your pipeline on a shorter subclip o
## To do so add .subclip(start_second,end_second) to the end of the line below
## Where start_second and end_second are integer values representing the start and end of t
## You may also uncomment the following line for a subclip of the first 5 seconds
##clip3 = VideoFileClip('test_videos/challenge.mp4').subclip(0,5)
clip3 = VideoFileClip('test_videos/challenge.mp4')
challenge_clip = clip3.fl_image(process_image_lane)
%time challenge_clip.write_videofile(challenge_output, audio=False)
```

```
[MoviePy] >>>> Building video test_videos_output/challenge.mp4
[MoviePy] Writing video test_videos_output/challenge.mp4

100%|████████████████████████████████████████████████████████████████████████████████
████████████████████████████████| 251/251 [00:20<00:00, 13.49it/s]

[MoviePy] Done.
[MoviePy] >>>> Video ready: test_videos_output/challenge.mp4

Wall time: 22.5 s
```

In [16]:

```
HTML("""
<video width="960" height="540" controls>
  <source src="{0}">
</video>
""".format(challenge_output))
```

Out[16]:



Backup