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.

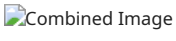
Once you have a result that looks roughly like "raw-lines-example.mp4", you'll need to get creative and try to average and/or extrapolate the line segments you've detected to map out the full extent of the lane lines. You can see an example of the result you're going for in the video "P1_example.mp4". 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 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 for this project.

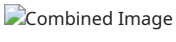
Let's have a look at our first image called 'test_images/solidWhiteRight.jpg'. Run the 2 cells below (hit Shift-Enter or the "play" button above) to display the image.

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, consult the forums for more troubleshooting tips.

Import Packages

In [1]:
```
import os
import math
import numpy as np
import scipy as sp
import scipy.optimize as spo
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import cv2


plt.rcParams["figure.figsize"] = [40, 40]
```

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 example, call as plt.imshow(gray, cmap='gray')
```

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

Out[2]:
```
<matplotlib.image.AxesImage at 0x7f8047f2d8e0>
```

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]:

```
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 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=2, vertices=None):  # Added vertices parameter, this will be used in the improved di
    """
    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 hough_lines(img, rho, theta, threshold, min_line_len, max_line_gap, vertices=None):  # Added vertices parameter, this will be used in th
    """
    `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, vertices=vertices)
    return line_img


def weighted_img(img, initial_img, α=0.8, β=1., γ=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, α, img, β, γ)
```

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]:
```
images = os.listdir("test_images/")
print(images)
```

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

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.

1. Pipleline and Parameters Tuning

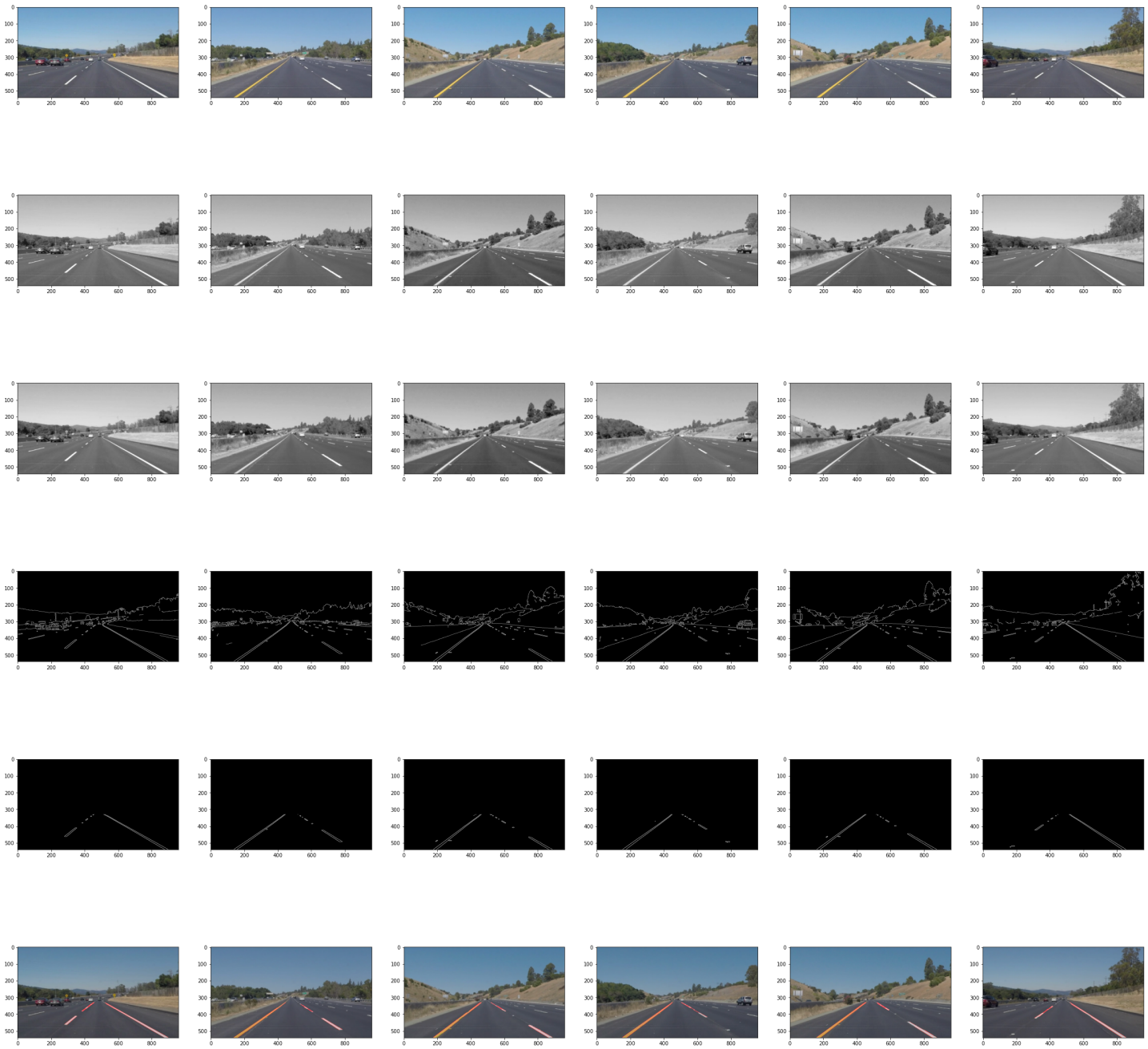
Here I write the code for the CV pipeline. I start with the parameters I determined in the lesson quizes. This will be first used to test the pipeline. The parameters will then be tuned here. Then I will improve the draw_lines function in part 2.

In [5]:
```
class Parameters:
    '''Struct to hold all model parameters.'''

    def __init__(self, imshape):
        ''' Definition of all paramters.'''
```

```
        self.gaussian_keral_size = 5             # Kernal size for the smoothing
        self.canny_low_threshold = 50            # Canny transform low threshhold
        self.canny_high_threshold = 150          # Canny transform high threshhold
        self.vertices = np.array([[(0, imshape[0]), (430, 330), (520, 330), (imshape[1], imshape[0])]], dtype=np.int32) # Vertices of 2D po
        self.hough_rho = 1                       # Distance resolution in pixels of the Hough grid
        self.hough_theta = np.pi / 180           # Agular resolution in radians of the Hough grid
        self.hough_threshold = 1                 # Minimum number of votes (intersections in Hough grid cell)
        self.hough_min_line_length = 50          # Minimum number of pixels making up a line
        self.hough_max_line_gap = 30             # Maximum gap in pixels between connectable line segments


def draw_lanes(image, parameters=None):
    '''Function to add lane lines to a given image.

    image : np.ndarray
        Pixel values of image. Shape of color image expected: (X,Y,3).
    paramters : Parameters
        Parameter struct holding all parameters of the model.
    returns : tuple
        Tuple of output images, showing each stage of the lane dwaring process, along with final image. Elements are:
            0 - original image
            1 - grayscale image
            2 - blurred grayscale image
            3 - edge detected image
            4 - masked edge detected image
            5 - fineal image showing lane lines superimposed on original image
    '''

    # Use default prameters if none given
    if parameters is None:
        parameters = Parameters(image.shape)

    # Convert to grayscale
    gray = grayscale(image)

    # Remove noise using Gaussian smoothing
    blur_gray = gaussian_blur(gray, parameters.gaussian_keral_size)

    # Apply Canny edge detection
    edges = canny(blur_gray, parameters.canny_low_threshold, parameters.canny_high_threshold)

    # Create a masked edges image
    masked_edges = region_of_interest(edges, parameters.vertices)

    # Run Hough on edge detected image
    line_image = hough_lines(masked_edges, parameters.hough_rho, parameters.hough_theta, parameters.hough_threshold, parameters.hough_min_li

    # Draw the lines on the edge image
    lines_edges = weighted_img(line_image, image, α=0.8, β=1., γ=0.)

    # Return tuple of result images
    return (image, gray, blur_gray, edges, masked_edges, lines_edges)
```

In [6]:

```
# Test parameters for the given images
lines_added = []
for image_file in images:
    image = mpimg.imread('test_images/{}'.format(image_file))
    lines_added.append(draw_lanes(image))


# Plot the results for the given images
fig, axs = plt.subplots(6, 6)
for i, ax in enumerate(axs):
    for j, a in enumerate(ax):
        if i in [1,2,3,4]:
            a.imshow(lines_added[j][i], cmap='Greys_r')
        else:
            a.imshow(lines_added[j][i])
plt.savefig('fig1.pdf')
```



2. Improving draw_lines

I will now redefine the draw_lines function to fix the issue with the dahsed lane lines not yeilding a clean and extrapolated lane line, and the instability in the solid lane lines.

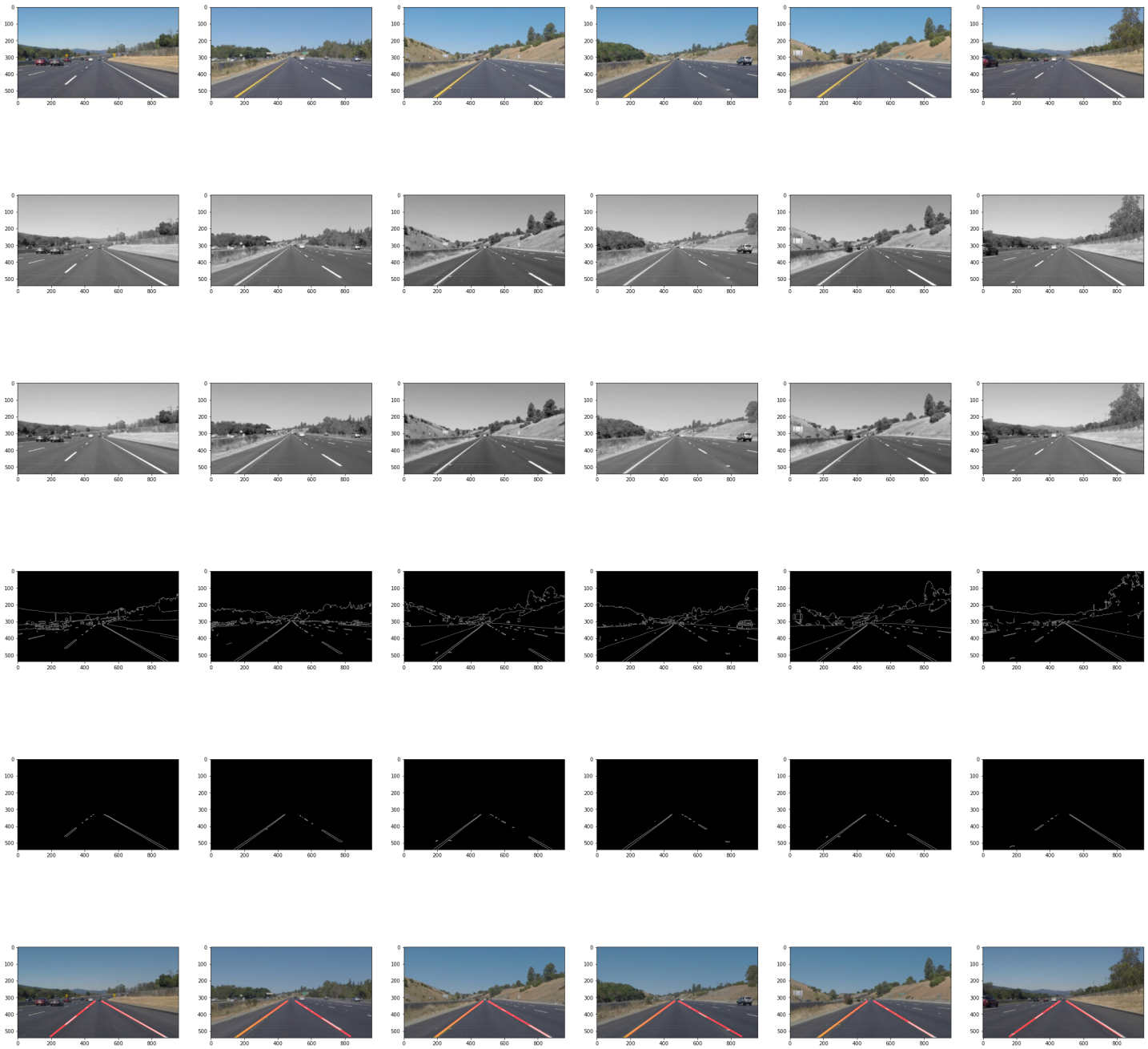
In [7]:

```
def draw_lines(img, lines, color=[255, 0, 0], thickness=10, vertices=None):
    '''New finction to draw lines smoothly using extrapolation.'''

    # Use the gradients of the line segemtns to group the points into two sets:
    # Those that are part of the left lane (positive gradient)
    # And those that are part of the right lane (negative gradient)
    # The resultant arrays are of shape (2,N), one col for the x and one for the y for all N points
    left_lane = []
    right_lane = []
    for line in lines:
        for x1,y1,x2,y2 in line:
            gradient = (y2 - y1) / (x2 - x1)
            if abs(gradient) > 0.5: # filter out spurious lines due to horizontal road markings
                if gradient >= 0:
                    left_lane.append([x1,y1])
                    left_lane.append([x2,y2])
                else:
                    right_lane.append([x1,y1])
                    right_lane.append([x2,y2])
    left_lane = np.array(left_lane).T
    right_lane = np.array(right_lane).T

    try:
        # Now fit a line to each set of points using scipy for the left lane
        f = lambda x, a, b: a*x + b       # y = ax + b
        g = lambda y, a, b: (y - b) / a   # x = (y - b) / a
```

```
        a, b = spo.curve_fit(f, left_lane[0,:], left_lane[1,:])[0]

        # Extrapolate in desired region determined by the y values in the vertices and add the line for the left lane
        x1 = g(vertices[0][0][1] - 8, a, b)
        x2 = g(vertices[0][1][1] - 8, a, b)
        y1 = f(x1, a, b)
        y2 = f(x2, a, b)
        cv2.line(img, (int(x1), int(y1)), (int(x2), int(y2)), color, thickness)

    except IndexError:
        pass # No lanes detected, do not draw a line

    try:
        # Now fit a line to each set of points using scipy for the right lane
        c, d = spo.curve_fit(f, right_lane[0,:], right_lane[1,:])[0]

        # Extrapolate in desired region determined by the y values in the vertices add the line for the right lane
        x1 = g(vertices[0][0][1] - 8, c, d)
        x2 = g(vertices[0][1][1] - 8, c, d)
        y1 = f(x1, c, d)
        y2 = f(x2, c, d)
        cv2.line(img, (int(x1), int(y1)), (int(x2), int(y2)), color, thickness)

    except IndexError:
        pass # No lanes detected, do not draw a line
```

In [8]:

```
# Test new method for the given images
lines_added = []
for image_file in images:
    image = mpimg.imread('test_images/{}'.format(image_file))
    lines_added.append(draw_lanes(image))


# Plot the results for the given images
fig, axs = plt.subplots(6, 6)
for i, ax in enumerate(axs):
    for j, a in enumerate(ax):
        if i in [1,2,3,4]:
            a.imshow(lines_added[j][i], cmap='Greys_r')
        else:
            a.imshow(lines_added[j][i])
plt.savefig('fig2.pdf')
```



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, consult the forums 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 for more troubleshooting tips across operating systems.

In [9]:
```
# Import everything needed to edit/save/watch video clips
from moviepy.editor import VideoFileClip
from IPython.display import HTML
```

In [10]:
```
def process_image(image):
    return draw_lanes(image)[5]
```

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

In [11]:
```
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)
```

```
t:  10%|         | 21/221 [00:00<00:00, 204.37it/s, now=None]
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: user 6.55 s, sys: 2.03 s, total: 8.58 s
Wall time: 2.03 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 [12]:
```
HTML("""
<video width="960" height="540" controls>
  <source src="{0}">
</video>
""".format(white_output))
```

Out[12]:

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 [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 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)
```

```
t:   0%|          | 0/681 [00:00<?, ?it/s, now=None]
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: user 22.8 s, sys: 7.26 s, total: 30 s
Wall time: 5.9 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 to the writeup template file.

Writeup found in writeup.pdf.