**Sobel Operator**

The Sobel operator is at the heart of the Canny edge detection algorithm you used in the Introductory Lesson. Applying the Sobel operator to an image is a way of taking the derivative of the image in the x*x* or y*y* direction. The operators for Sobel\_x*Sobelx*​ and Sobel\_y*Sobely*​, respectively, look like this:

A close up of a sign

Description automatically generated

These are examples of Sobel operators with a kernel size of 3 (implying a 3 x 3 operator in each case). This is the minimum size, but the kernel size can be any odd number. A larger kernel implies taking the gradient over a larger region of the image, or, in other words, a smoother gradient.

To understand how these operators take the derivative, you can think of overlaying either one on a 3 x 3 region of an image. If the image is flat across that region (i.e., there is little change in values across the given region), then the result (summing the element-wise product of the operator and corresponding image pixels) will be zero.

gradient = \sum(region \* S\_x)*gradient*=∑(*region*∗*Sx*​)

For example, given:

region = \begin{pmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \\ 2 & 2 & 2 \end{pmatrix}, S\_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}*region*=⎝⎛​222​222​222​⎠⎞​,*Sx*​=⎝⎛​−1−2−1​000​121​⎠⎞​

The element-wise product would be:

\begin{pmatrix} -2 & 0 & 2 \\ -4 & 0 & 4 \\ -2 & 0 & 2 \end{pmatrix}⎝⎛​−2−4−2​000​242​⎠⎞​

In which case, the sum of this matrix is 00, implying a flat gradient (in the x-direction in this calculation, although the y-direction is also zero in this example).

If, instead, for example, you apply the S\_x*Sx*​ operator to a region of the image where values are rising from left to right, then the result will be positive, implying a positive derivative.

Given:

region = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}, S\_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}*region*=⎝⎛​111​222​333​⎠⎞​,*Sx*​=⎝⎛​−1−2−1​000​121​⎠⎞​

The element-wise product would be:

\begin{pmatrix} -1 & 0 & 3 \\ -2 & 0 & 6 \\ -1 & 0 & 3 \end{pmatrix}⎝⎛​−1−2−1​000​363​⎠⎞​

This time, the sum of this matrix is 88, meaning a gradient exists in the x-direction. Note that in this example image region, if you applied the S\_y*Sy*​ operator, the result would be a gradient of 00 in the y-direction, as the values are not varying from top to bottom.

**Visual Example**

If we apply the Sobel x*x* and y*y* operators to this image:

A car driving on a highway

Description automatically generated

And then we take the absolute value, we get the result:

A close up of a logo

Description automatically generated

Absolute value of Sobel x (left) and Sobel y (right).

x*x***vs.**y*y*

In the above images, you can see that the gradients taken in both the x*x* and the y*y* directions detect the lane lines and pick up other edges. Taking the gradient in the x*x* direction emphasizes edges closer to vertical. Alternatively, taking the gradient in the y*y* direction emphasizes edges closer to horizontal.

In the upcoming exercises, you'll write functions to take various thresholds of the x*x* and y*y* gradients. Here's some code that might be useful:

**Examples of Useful Code**

You need to pass a single color channel to the cv2.Sobel() function, so first convert it to grayscale:

gray = cv2.cvtColor(im, cv2.COLOR\_RGB2GRAY)

**Note:**Make sure you use the correct grayscale conversion depending on how you've read in your images. Use cv2.COLOR\_RGB2GRAY if you've read in an image using mpimg.imread(). Use cv2.COLOR\_BGR2GRAY if you've read in an image using cv2.imread().

Calculate the derivative in the x*x* direction (the 1, 0 at the end denotes x*x* direction):

sobelx = cv2.Sobel(gray, cv2.CV\_64F, 1, 0)

Calculate the derivative in the y*y* direction (the 0, 1 at the end denotes y*y* direction):

sobely = cv2.Sobel(gray, cv2.CV\_64F, 0, 1)

Calculate the absolute value of the x*x* derivative:

abs\_sobelx = np.absolute(sobelx)

Convert the absolute value image to 8-bit:

scaled\_sobel = np.uint8(255\*abs\_sobelx/np.max(abs\_sobelx))

**Note:**It's not entirely necessary to convert to 8-bit (range from 0 to 255) but in practice, it can be useful in the event that you've written a function to apply a particular threshold, and you want it to work the same on input images of different scales, like jpg vs. png. You could just as well choose a different standard range of values, like 0 to 1 etc.

Create a binary threshold to select pixels based on gradient strength:

thresh\_min = 20

thresh\_max = 100

sxbinary = np.zeros\_like(scaled\_sobel)

sxbinary[(scaled\_sobel >= thresh\_min) & (scaled\_sobel <= thresh\_max)] = 1

plt.imshow(sxbinary, cmap='gray')

**Result**

A picture containing photo, sitting, monitor, table

Description automatically generated

Pixels have a value of 1 or 0 based on the strength of the x*x* gradient.

# Applying Sobel

Here's your chance to write a function that will be useful for the Advanced Lane-Finding Project at the end of this lesson! Your goal in this exercise is to identify pixels where the gradient of an image falls within a specified threshold range.

## Example

A screenshot of a cell phone

Description automatically generated

Here's the scaffolding for your function:

**def** **abs\_sobel\_thresh**(img, orient='x', thresh\_min=0, thresh\_max=255):

*# Grayscale*

*# Apply cv2.Sobel()*

*# Take the absolute value of the output from cv2.Sobel()*

*# Scale the result to an 8-bit range (0-255)*

*# Apply lower and upper thresholds*

*# Create binary\_output*

**return** binary\_output

Pass in img and set the parameter orient as 'x' or 'y' to take either the x*x* or y*y* gradient. Set thresh\_min, and thresh\_max to specify the range to select for binary output. You can use exclusive (<, >) or inclusive (<=, >=) thresholding.

**NOTE:** Your output should be an array of the same size as the input image. The output array elements should be 1 where gradients were in the threshold range, and 0 everywhere else.

As usual, if you run into any errors as you run your code, please refer to the **Examples of Useful Code** section in the previous video and make sure that your code syntax matches up! You can download the image used in the quiz [here](https://s3-us-west-1.amazonaws.com/udacity-selfdrivingcar/files/Advanced_Lane_Finding_Images/sobel_example/signs_vehicles_xygrad.png).

# Magnitude of the Gradient

A picture containing photo, sitting, table, player

Description automatically generated

With the result of the last quiz, you can now take the gradient in x or y and set thresholds to identify pixels within a certain gradient range. If you play around with the thresholds a bit, you'll find the x-gradient does a cleaner job of picking up the lane lines, but you can see the lines in the y-gradient as well.

In this next exercise, your goal is to apply a threshold to the overall magnitude of the gradient, in both x and y.

The magnitude, or absolute value, of the gradient is just the square root of the squares of the individual x and y gradients. For a gradient in both the x*x* **and** y*y* directions, the magnitude is the square root of the sum of the squares.

abs\_sobelx= \sqrt{(sobel\_x)^2}=(*sobelx*​)2​

abs\_sobely= \sqrt{(sobel\_y)^2}=(*sobely*​)2​

abs\_sobelxy= \sqrt{(sobel\_x)^2+(sobel\_y)^2}=(*sobelx*​)2+(*sobely*​)2​

It's also worth considering the size of the region in the image over which you'll be taking the gradient. You can modify the kernel size for the Sobel operator to change the size of this region. Taking the gradient over larger regions can smooth over noisy intensity fluctuations on small scales. The default Sobel kernel size is 3, but here you'll define a new function that takes kernel size as a parameter.

It's important to note here that the kernel size should be an **odd** number. Since we are searching for the gradient around a given pixel, we want to have an equal number of pixels in each direction of the region from this central pixel, leading to an odd-numbered filter size - a filter of size three has the central pixel with one additional pixel in each direction, while a filter of size five has an additional two pixels outward from the central pixel in each direction.

The function you'll define for the exercise below should take in an image and optional Sobel kernel size, as well as thresholds for gradient magnitude. Next, you'll compute the gradient magnitude, apply a threshold, and create a binary output image showing where thresholds were met.

**Steps to take in this exercise:**

1. Fill out the function in the editor below to return a thresholded gradient magnitude. Again, you can apply exclusive (<, >) or inclusive (<=, >=) thresholds.
2. Test that your function returns output similar to the example below for sobel\_kernel=9, mag\_thresh=(30, 100).

You can download the image used in the quiz [here](https://s3-us-west-1.amazonaws.com/udacity-selfdrivingcar/files/Advanced_Lane_Finding_Images/sobel_example/signs_vehicles_xygrad.png).

**Here's an example of the output you're going for:**

A screenshot of a car

Description automatically generated

# Direction of the Gradient

When you play around with the thresholding for the gradient magnitude in the previous exercise, you find what you might expect, namely, that it picks up the lane lines well, but with a lot of other stuff detected too. Gradient magnitude is at the heart of Canny edge detection, and is why Canny works well for picking up all edges.

In the case of lane lines, we're interested only in edges of a particular orientation. So now we will explore the direction, or orientation, of the gradient.

The direction of the gradient is simply the inverse tangent (arctangent) of the y*y* gradient divided by the x*x* gradient:

arctan{(sobel\_y/sobel\_x)}*arctan*(*sobely*​/*sobelx*​).

Each pixel of the resulting image contains a value for the angle of the gradient away from horizontal in units of radians, covering a range of -\pi/2−*π*/2 to \pi/2*π*/2. An orientation of 0 implies a vertical line and orientations of \footnotesize{+/-}\:\normalsize{\pi/2}+/−*π*/2 imply horizontal lines. (Note that in the quiz below, we actually utilize np.arctan2, which can return values between \footnotesize{+/-}\:\normalsize{\pi}+/−*π*; however, as we'll take the absolute value of sobel\_x*sobelx*​, this restricts the values to \footnotesize{+/-}\:\normalsize{\pi/2}+/−*π*/2, as shown [here](https://en.wikipedia.org/wiki/Atan2).)

In this next exercise, you'll write a function to compute the direction of the gradient and apply a threshold. The direction of the gradient is much noisier than the gradient magnitude, but you should find that you can pick out particular features by orientation.

### Steps to take in this exercise:

1. Fill out the function in the editor below to return a thresholded absolute value of the gradient direction. Use Boolean operators, again with exclusive (<, >) or inclusive (<=, >=) thresholds.
2. Test that your function returns output similar to the example below for sobel\_kernel=15, thresh=(0.7, 1.3).

You can download the image used in the quiz [here](https://s3-us-west-1.amazonaws.com/udacity-selfdrivingcar/files/Advanced_Lane_Finding_Images/sobel_example/signs_vehicles_xygrad.png).

A close up of a road

Description automatically generated

# Combining Thresholds

If you play around with the thresholds in the last exercise, you'll find that you can start to identify the lane lines by gradient direction alone by setting the threshold around thresh = (0.7, 1.3), but there's still a lot of noise in the resulting image.

Now consider how you can use various aspects of your gradient measurements (x, y, magnitude, direction) to isolate lane-line pixels. Specifically, think about how you can use thresholds of the x and y gradients, the overall gradient magnitude, and the gradient direction to focus on pixels that are likely to be part of the lane lines.

# Challenge:

In the project at the end of this section, you'll want to experiment with thresholding various aspects of the gradient, so now would be a great time to start coding it up on your local machine! Grab the image we've been working with for the last three quizzes [here](https://s3.amazonaws.com/udacity-sdc/advanced_lane_finding/signs_vehicles_xygrad.png) (or a smaller jpg file [here](https://s3.amazonaws.com/udacity-sdc/advanced_lane_finding/signs_vehicles_xygrad.jpg)).

Combine the selection thresholds from the last 3 quizzes to write a piece of code like the following, where you can play with various thresholds and see the output.

**def** **abs\_sobel\_thresh**(img, orient='x', sobel\_kernel=3, thresh=(0, 255)):

*# Calculate directional gradient*

*# Apply threshold*

**return** grad\_binary

**def** **mag\_thresh**(image, sobel\_kernel=3, mag\_thresh=(0, 255)):

*# Calculate gradient magnitude*

*# Apply threshold*

**return** mag\_binary

**def** **dir\_threshold**(image, sobel\_kernel=3, thresh=(0, np.pi/2)):

*# Calculate gradient direction*

*# Apply threshold*

**return** dir\_binary

*# Choose a Sobel kernel size*

ksize = 3 *# Choose a larger odd number to smooth gradient measurements*

*# Apply each of the thresholding functions*

gradx = abs\_sobel\_thresh(image, orient='x', sobel\_kernel=ksize, thresh=(0, 255))

grady = abs\_sobel\_thresh(image, orient='y', sobel\_kernel=ksize, thresh=(0, 255))

mag\_binary = mag\_thresh(image, sobel\_kernel=ksize, mag\_thresh=(0, 255))

dir\_binary = dir\_threshold(image, sobel\_kernel=ksize, thresh=(0, np.pi/2))

Try different combinations and see what you get.

For example, here is a selection for pixels where both the x*x* and y*y* gradients meet the threshold criteria, or the gradient magnitude and direction are both within their threshold values.

combined = np.zeros\_like(dir\_binary)

combined[((gradx == 1) & (grady == 1)) | ((mag\_binary == 1) & (dir\_binary == 1))] = 1

## Output

Here is an example of a binary result from multiple thresholds:

A picture containing photo

Description automatically generated

A **color space** is a specific organization of colors; color spaces provide a way to categorize colors and represent them in digital images.

**RGB** is red-green-blue color space. You can think of this as a 3D space, in this case a cube, where any color can be represented by a 3D coordinate of R, G, and B values. For example, white has the coordinate (255, 255, 255), which has the maximum value for red, green, and blue.

***Note***: If you read in an image using matplotlib.image.imread() you will get an RGB image, but if you read it in using OpenCV cv2.imread() this will give you a BGR image.

A close up of a logo

Description automatically generated

RGB color space

There are many other ways to represent the colors in an image besides just composed of red, green, and blue values.

There is also **HSV** color space (hue, saturation, and value), and **HLS** space (hue, lightness, and saturation). These are some of the most commonly used color spaces in image analysis.

To get some intuition about these color spaces, you can generally think of **Hue** as the value that represents color independent of any change in brightness. So if you imagine a basic red paint color, then add some white to it or some black to make that color lighter or darker -- the underlying color remains the same and the hue for all of these colors will be the same.

On the other hand, **Lightness** and **Value** represent different ways to measure the relative lightness or darkness of a color. For example, a dark red will have a similar hue but much lower value for lightness than a light red. **Saturation** also plays a part in this; saturation is a measurement of colorfulness. So, as colors get lighter and closer to white, they have a lower saturation value, whereas colors that are the most intense, like a bright primary color (imagine a bright red, blue, or yellow), have a high saturation value. You can get a better idea of these values by looking at the 3D color spaces pictured below.

Most of these different color spaces were either inspired by the human vision system, and/or developed for efficient use in television screen displays and computer graphics. You can read more about the history and the derivation of HLS and HSV color spaces [here](https://en.wikipedia.org/wiki/HSL_and_HSV).

A picture containing clock

Description automatically generated

(Left) HSV color space, (Right) HLS color space

In the code example, I used HLS space to help detect lane lines of different colors and under different lighting conditions.

OpenCV provides a function hls = cv2.cvtColor(im, cv2.COLOR\_RGB2HLS) that converts images from one color space to another. If you’re interested in the math behind this conversion, take a look at the equations below; note that all this math is for converting 8-bit images, which is the format for most road images in this course. These equations convert one color at a time from RGB to HLS.

**Constants**

V\_{max} \leftarrow max(R, G, B)*Vmax*​←*max*(*R*,*G*,*B*)

V\_{min} \leftarrow min(R, G, B)*Vmin*​←*min*(*R*,*G*,*B*)

These are the maximum and minimum values across all three RGB values for a given color (by pixel).

Note that in these equations, it would be necessary to divide the RGB values by 255 such that they fall in the range (0, 1)(0,1), but OpenCV will perform this for you in cv2.cvtColor(). **L** and **S** are scaled back up after the transformations take place, while **H** is halved for use in 8-bit images (see below).

**H channel conversion equations**

There are three different equations, which one is used depends on the the value of V\_{max}*Vmax*​ whether that's R, G, or B.

\large H \leftarrow \frac{30(G-B)}{V\_{max}-V\_{min}}*H*←*Vmax*​−*Vmin*​30(*G*−*B*)​ , if \: V\_{max}=R*Vmax*​=*R*

\large H \leftarrow 60 + \frac{30(B-R)}{V\_{max}-V\_{min}}*H*←60+*Vmax*​−*Vmin*​30(*B*−*R*)​ , if \: V\_{max}=G*Vmax*​=*G*

\large H \leftarrow 120 + \frac{30(R-G)}{V\_{max}-V\_{min}}*H*←120+*Vmax*​−*Vmin*​30(*R*−*G*)​ , if \: V\_{max}=B*Vmax*​=*B*

***Note***: In OpenCV, for 8-bit images, the range of H is from 0-179. It's typically from 0-359 for degrees around the cylindrical colorspace, but this number is divided in half so that the range can be represented in an 8-bit image whose color values range from 0-255.

**L channel conversion equation**

\large L \leftarrow \frac{V\_{max}+V\_{min}}{2}*L*←2*Vmax*​+*Vmin*​​

**S channel conversion equations**

There are two possible equations; one is used depending on the value of L.

\large S \leftarrow \frac{V\_{max}-V\_{min}}{V\_{max}+V\_{min}}*S*←*Vmax*​+*Vmin*​*Vmax*​−*Vmin*​​, if L < 0.5*L*<0.5

\large S \leftarrow \frac{V\_{max}-V\_{min}}{2 - (V\_{max}+V\_{min})}*S*←2−(*Vmax*​+*Vmin*​)*Vmax*​−*Vmin*​​, if L \geq 0.5*L*≥0.5

# HLS and Color Thresholds

A car parked on the side of a road

Description automatically generated

You've now seen that various color thresholds can be applied to find the lane lines in images. Here we'll explore this a bit further and look at a couple examples to see why a color space like HLS can be more robust. Let's first take another look at some of the images you saw in the last video.

Here I'll read in the same original image (the image above), convert to grayscale, and apply a threshold that identifies the lines:

**import** numpy **as** np

**import** cv2

**import** matplotlib.pyplot **as** plt

**import** matplotlib.image **as** mpimg

image = mpimg.imread('test6.jpg')

thresh = (180, 255)

gray = cv2.cvtColor(image, cv2.COLOR\_RGB2GRAY)

binary = np.zeros\_like(gray)

binary[(gray > thresh[0]) & (gray <= thresh[1])] = 1

And here's the result:

A screenshot of a computer

Description automatically generated

You might have also explored thresholding individual RGB color channels. You can take a look at them side by side to see which ones do a better job of picking up the lane lines:

R = image[:,:,0]

G = image[:,:,1]

B = image[:,:,2]

A screenshot of a social media post

Description automatically generated

The R channel does a reasonable job of highlighting the lines, and you can apply a similar threshold to find lane-line pixels:

thresh = (200, 255)

binary = np.zeros\_like(R)

binary[(R > thresh[0]) & (R <= thresh[1])] = 1

A screenshot of a computer

Description automatically generated

In this lesson, we're looking at different color spaces. While there are several that are worth exploring, here we'll look specifically at HLS. When we separate the H, L, and S channels we get the following result:

hls = cv2.cvtColor(image, cv2.COLOR\_RGB2HLS)

H = hls[:,:,0]

L = hls[:,:,1]

S = hls[:,:,2]

A screenshot of a social media post

Description automatically generated

The S channel picks up the lines well, so let's try applying a threshold there:

thresh = (90, 255)

binary = np.zeros\_like(S)

binary[(S > thresh[0]) & (S <= thresh[1])] = 1

A screenshot of a social media post

Description automatically generated

You can also see that in the H channel, the lane lines appear dark, so we could try a low threshold there and obtain the following result:

thresh = (15, 100)

binary = np.zeros\_like(H)

binary[(H > thresh[0]) & (H <= thresh[1])] = 1

A screenshot of a video game

Description automatically generated

From these examples, you can see that the S channel is probably your best bet. It's cleaner than the H channel result and a bit better than the R channel or simple grayscaling. But it's not clear that one method is far superior to the others.

In each case, I've tweaked the threshold parameters to do as good a job as possible of picking out the lines. Where we can really see a difference in results, however, is when we step to a new frame, where there are shadows and different colors in the pavement.

Look at the same thresholds applied to each of these four channels for this image:

A close up of a car going down the highway

Description automatically generated

Here's how the various channels and binaries (with the same threshold values as above) look:

A screenshot of a social media post

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a video game

Description automatically generated

A screenshot of a social media post

Description automatically generated

Now you can see that, the S channel is still doing a fairly robust job of picking up the lines under very different color and contrast conditions, while the other selections look messy. You could tweak the thresholds and get closer in the other channels, but the S channel is preferable because it is more robust to changing conditions.

It's worth noting, however, that the R channel still does rather well on the white lines, perhaps even better than the S channel. As with gradients, it's worth considering how you might combine various color thresholds to make the most robust identification of the lines.

# HLS Color Threshold

A close up of a street

Description automatically generated

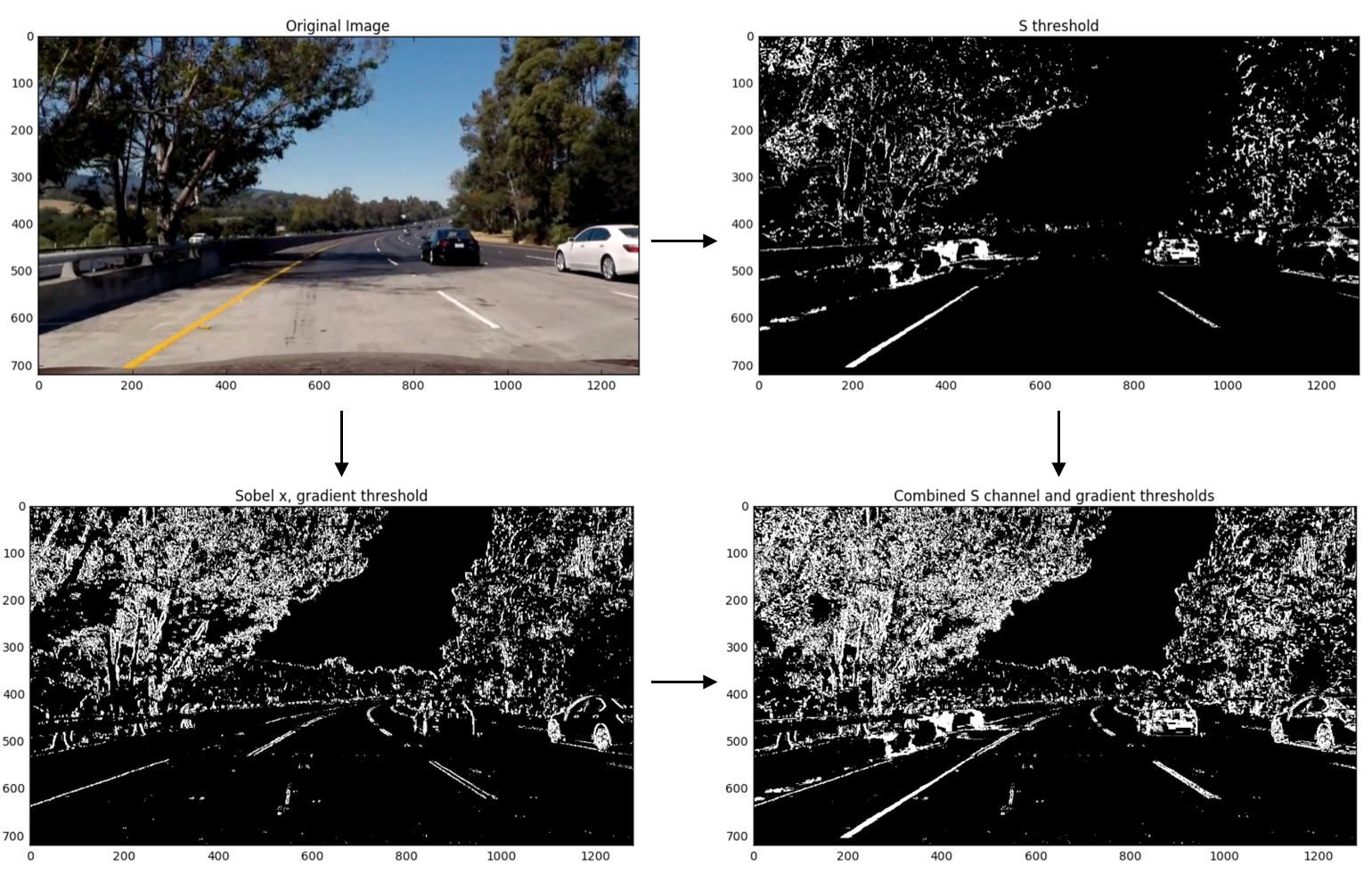
In this exercise, you can try applying a color threshold in the HLS color space. If you have any problems with the implementation, please refer to the previous video and instructor notes.

Your task here is to write a function that takes in an image and threshold values and returns a binary output from applying the threshold to the S-channel. Apply your thresholds as you did for gradients but this time use an exclusive (>) lower bound and an inclusive upper bound (<=).

You can download the test images in the quiz for local usage [here](https://s3-us-west-1.amazonaws.com/udacity-selfdrivingcar/files/Advanced_Lane_Finding_Images/color_spaces/colorspace_test_images.zip).

# Color and Gradient

Now it's time to combine what you know about color and gradient thresholding to get the best of both worlds. Here's an example of how that might look:



Combined color and gradient thresholds for lane detection.

At this point, it's okay to detect edges around trees or cars because these lines can be mostly filtered out by applying a mask to the image and essentially cropping out the area outside of the lane lines. It's most important that you reliably detect different colors of lane lines under varying degrees of daylight and shadow.

You can clearly see which parts of the lane lines were detected by the gradient threshold and which parts were detected by the color threshold by stacking the channels and seeing the individual components. You can create a binary combination of these two images to map out where either the color or gradient thresholds were met.

Here's what that looks like in code:

*# Convert to HLS color space and separate the S channel*

*# Note: img is the undistorted image*

hls = cv2.cvtColor(img, cv2.COLOR\_RGB2HLS)

s\_channel = hls[:,:,2]

*# Grayscale image*

*# NOTE: we already saw that standard grayscaling lost color information for the lane lines*

*# Explore gradients in other colors spaces / color channels to see what might work better*

gray = cv2.cvtColor(img, cv2.COLOR\_RGB2GRAY)

*# Sobel x*

sobelx = cv2.Sobel(gray, cv2.CV\_64F, 1, 0) *# Take the derivative in x*

abs\_sobelx = np.absolute(sobelx) *# Absolute x derivative to accentuate lines away from horizontal*

scaled\_sobel = np.uint8(255\*abs\_sobelx/np.max(abs\_sobelx))

*# Threshold x gradient*

thresh\_min = 20

thresh\_max = 100

sxbinary = np.zeros\_like(scaled\_sobel)

sxbinary[(scaled\_sobel >= thresh\_min) & (scaled\_sobel <= thresh\_max)] = 1

*# Threshold color channel*

s\_thresh\_min = 170

s\_thresh\_max = 255

s\_binary = np.zeros\_like(s\_channel)

s\_binary[(s\_channel >= s\_thresh\_min) & (s\_channel <= s\_thresh\_max)] = 1

*# Stack each channel to view their individual contributions in green and blue respectively*

*# This returns a stack of the two binary images, whose components you can see as different colors*

color\_binary = np.dstack(( np.zeros\_like(sxbinary), sxbinary, s\_binary)) \* 255

*# Combine the two binary thresholds*

combined\_binary = np.zeros\_like(sxbinary)

combined\_binary[(s\_binary == 1) | (sxbinary == 1)] = 1

*# Plotting thresholded images*

f, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))

ax1.set\_title('Stacked thresholds')

ax1.imshow(color\_binary)

ax2.set\_title('Combined S channel and gradient thresholds')

ax2.imshow(combined\_binary, cmap='gray')

The output is shown below. The final image color\_binary is a combination of binary thresholding the S channel (HLS) and binary thresholding the result of applying the Sobel operator in the x direction on the original image.

A screenshot of a computer

Description automatically generated

(Left) Stacked image; the green is the gradient threshold component and the blue is the color channel threshold component. (Right) black and white combined thresholded image - this one has combined both gradient and color thresholds into one image.

In this next exercise, you'll get the chance to play around with different combinations of color and gradient with the goal using the pipeline you come up with for your upcoming project. There's no correct submission, just explore!

Some interesting things to explore might include: the H channel, different threshold values for color and gradient binary images, and even a different color space, like HSV!

You can download the image used in the quiz [here](https://s3-us-west-1.amazonaws.com/udacity-selfdrivingcar/files/Advanced_Lane_Finding_Images/luv_example/bridge_shadow.jpg).