**Coding up a Color Selection**

Let’s code up a simple color selection in Python.

No need to download or install anything, you can just follow along in the browser for now.

We'll be working with the same image you saw previously.

An empty road

Description automatically generated

Check out the code below. First, I import pyplot and image from matplotlib. I also import numpy for operating on the image.

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

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

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

I then read in an image and print out some stats. I’ll grab the x and y sizes and make a copy of the image to work with. NOTE: Always make a copy of arrays or other variables in Python. If instead, you say "a = b" then all changes you make to "a" will be reflected in "b" as well!

*# Read in the image and print out some stats*

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

print('This image is: ',type(image),

'with dimensions:', image.shape)

*# Grab the x and y size and make a copy of the image*

ysize = image.shape[0]

xsize = image.shape[1]

*# Note: always make a copy rather than simply using "="*

color\_select = np.copy(image)

Next I define a color threshold in the variables red\_threshold, green\_threshold, and blue\_threshold and populate rgb\_threshold with these values. This vector contains the minimum values for red, green, and blue (R,G,B) that I will allow in my selection.

*# Define our color selection criteria*

*# Note: if you run this code, you'll find these are not sensible values!!*

*# But you'll get a chance to play with them soon in a quiz*

red\_threshold = 0

green\_threshold = 0

blue\_threshold = 0

rgb\_threshold = [red\_threshold, green\_threshold, blue\_threshold]

Next, I'll select any pixels below the threshold and set them to zero.

After that, all pixels that meet my color criterion (those above the threshold) will be retained, and those that do not (below the threshold) will be blacked out.

*# Identify pixels below the threshold*

thresholds = (image[:,:,0] < rgb\_threshold[0]) \

| (image[:,:,1] < rgb\_threshold[1]) \

| (image[:,:,2] < rgb\_threshold[2])

color\_select[thresholds] = [0,0,0]

*# Display the image*

plt.imshow(color\_select)

plt.show()

The result, color\_select, is an image in which pixels that were above the threshold have been retained, and pixels below the threshold have been blacked out.

In the code snippet above, red\_threshold, green\_threshold and blue\_threshold are all set to 0, which implies all pixels will be included in the selection.

In the next quiz, you will modify the values of red\_threshold, green\_threshold and blue\_threshold until you retain as much of the lane lines as possible while dropping everything else. Your output image should look like the one below.

A picture containing indoor, table, computer, photo

Description automatically generated

Image after color selection

Here's how I did it… I started by just trying some guesses.

Eventually, I found that with red\_threshold = green\_threshold = blue\_threshold = 200, I get a pretty good result, where I can clearly see the lane lines, but most everything else is blacked out.

At this point, however, it would still be tricky to extract the exact lines automatically, because we still have many other pixels detected around the periphery.

An empty road

Description automatically generated A picture containing indoor, computer, computer, photo

Description automatically generated

The original image (top), and color selection applied (bottom).

**Coding up a Region of Interest Mask**

Awesome! Now you've seen that with a simple color selection we have managed to eliminate almost everything in the image except the lane lines.

At this point, however, it would still be tricky to extract the exact lines automatically, because we still have some other objects detected around the periphery that aren't lane lines.

A picture containing indoor, table, computer, photo

Description automatically generated

In this case, I'll assume that the front facing camera that took the image is mounted in a fixed position on the car, such that the lane lines will always appear in the same general region of the image. Next, I'll take advantage of this by adding a criterion to only consider pixels for color selection in the region where we expect to find the lane lines.

Check out the code below. The variables left\_bottom, right\_bottom, and apex represent the vertices of a triangular region that I would like to retain for my color selection, while masking everything else out. Here I'm using a triangular mask to illustrate the simplest case, but later you'll use a quadrilateral, and in principle, you could use any polygon.

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

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

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

*# Read in the image and print some stats*

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

print('This image is: ', type(image),

'with dimensions:', image.shape)

*# Pull out the x and y sizes and make a copy of the image*

ysize = image.shape[0]

xsize = image.shape[1]

region\_select = np.copy(image)

*# Define a triangle region of interest*

*# Keep in mind the origin (x=0, y=0) is in the upper left in image processing*

*# Note: if you run this code, you'll find these are not sensible values!!*

*# But you'll get a chance to play with them soon in a quiz*

left\_bottom = [0, 539]

right\_bottom = [900, 300]

apex = [400, 0]

*# Fit lines (y=Ax+B) to identify the 3 sided region of interest*

*# np.polyfit() returns the coefficients [A, B] of the fit*

fit\_left = np.polyfit((left\_bottom[0], apex[0]), (left\_bottom[1], apex[1]), 1)

fit\_right = np.polyfit((right\_bottom[0], apex[0]), (right\_bottom[1], apex[1]), 1)

fit\_bottom = np.polyfit((left\_bottom[0], right\_bottom[0]), (left\_bottom[1], right\_bottom[1]), 1)

*# Find the region inside the lines*

XX, YY = np.meshgrid(np.arange(0, xsize), np.arange(0, ysize))

region\_thresholds = (YY > (XX\*fit\_left[0] + fit\_left[1])) & \

(YY > (XX\*fit\_right[0] + fit\_right[1])) & \

(YY < (XX\*fit\_bottom[0] + fit\_bottom[1]))

*# Color pixels red which are inside the region of interest*

region\_select[region\_thresholds] = [255, 0, 0]

*# Display the image*

plt.imshow(region\_select)

*# uncomment if plot does not display*

*# plt.show()*

# Combining Color and Region Selections

Now you've seen how to mask out a region of interest in an image. Next, let's combine the mask and color selection to pull only the lane lines out of the image.

Check out the code below. Here we’re doing both the color and region selection steps, requiring that a pixel meet both the mask and color selection requirements to be retained.

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

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

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

*# Read in the image*

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

*# Grab the x and y sizes and make two copies of the image*

*# With one copy we'll extract only the pixels that meet our selection,*

*# then we'll paint those pixels red in the original image to see our selection*

*# overlaid on the original.*

ysize = image.shape[0]

xsize = image.shape[1]

color\_select= np.copy(image)

line\_image = np.copy(image)

*# Define our color criteria*

red\_threshold = 0

green\_threshold = 0

blue\_threshold = 0

rgb\_threshold = [red\_threshold, green\_threshold, blue\_threshold]

*# Define a triangle region of interest (Note: if you run this code,*

*# Keep in mind the origin (x=0, y=0) is in the upper left in image processing*

*# you'll find these are not sensible values!!*

*# But you'll get a chance to play with them soon in a quiz ;)*

left\_bottom = [0, 539]

right\_bottom = [900, 300]

apex = [400, 0]

fit\_left = np.polyfit((left\_bottom[0], apex[0]), (left\_bottom[1], apex[1]), 1)

fit\_right = np.polyfit((right\_bottom[0], apex[0]), (right\_bottom[1], apex[1]), 1)

fit\_bottom = np.polyfit((left\_bottom[0], right\_bottom[0]), (left\_bottom[1], right\_bottom[1]), 1)

*# Mask pixels below the threshold*

color\_thresholds = (image[:,:,0] < rgb\_threshold[0]) | \

(image[:,:,1] < rgb\_threshold[1]) | \

(image[:,:,2] < rgb\_threshold[2])

*# Find the region inside the lines*

XX, YY = np.meshgrid(np.arange(0, xsize), np.arange(0, ysize))

region\_thresholds = (YY > (XX\*fit\_left[0] + fit\_left[1])) & \

(YY > (XX\*fit\_right[0] + fit\_right[1])) & \

(YY < (XX\*fit\_bottom[0] + fit\_bottom[1]))

*# Mask color selection*

color\_select[color\_thresholds] = [0,0,0]

*# Find where image is both colored right and in the region*

line\_image[~color\_thresholds & region\_thresholds] = [255,0,0]

*# Display our two output images*

plt.imshow(color\_select)

plt.imshow(line\_image)

*# uncomment if plot does not display*

*# plt.show()*

In the next quiz, you can vary your color selection and the shape of your region mask (vertices of a triangle left\_bottom, right\_bottom, and apex), such that you pick out the lane lines and nothing else.

In this next quiz, I've given you the values of red\_threshold, green\_threshold, and blue\_threshold but now you need to modify left\_bottom, right\_bottom, and apex to represent the vertices of a triangle identifying the region of interest in the image. When you run the code in the quiz, your output result will be several images. Tweak the vertices until your output looks like the examples shown below.

An empty road

Description automatically generated A picture containing sitting, dark, table, computer

Description automatically generated A close up of a road

Description automatically generated

The original image (Top left), region and color selection applied (Top right) and lines identified (bottom left).

Here’s how I did it: After selecting red\_threshold = green\_threshold = blue\_threshold = 200, I chose the vertices of my triangle to be left\_bottom = [0, 539], right\_bottom [900, 539], and apex = [475, 320], which produces the result shown below.

An empty road

Description automatically generated A picture containing sitting, dark, table, computer

Description automatically generated A close up of a road

Description automatically generatedThe original image (left), region and color selection applied (center) and lines identified (right).

**Finding Lines of any Color**

A close up of a road

Description automatically generated

So you found the lane lines... simple right? Now you’re ready to upload the algorithm to the car and drive autonomously right?? Well, not quite yet ;)

As it happens, lane lines are not always the same color, and even lines of the same color under different lighting conditions (day, night, etc) may fail to be detected by our simple color selection.

What we need is to take our algorithm to the next level to detect lines of any color using sophisticated computer vision methods.

So, what is computer vision?

In rest of this lesson, we’ll introduce some computer vision techniques with enough detail for you to get an intuitive feel for how they work.

You'll learn much more about these topics during the Computer Vision module later in the program.

We also recommend the free Udacity course, [Introduction to Computer Vision](https://www.udacity.com/course/introduction-to-computer-vision--ud810).

A picture containing drawing

Description automatically generated

Throughout this Nanodegree Program, we will be using Python with OpenCV for computer vision work. OpenCV stands for Open-Source Computer Vision. For now, you don't need to download or install anything, but later in the program we'll help you get these tools installed on your own computer.

OpenCV contains extensive libraries of functions that you can use. The OpenCV libraries are well documented, so if you’re ever feeling confused about what the parameters in a particular function are doing, or anything else, you can find a wealth of information at [opencv.org](http://opencv.org/).

**Note! The standard location of the origin (x=0, y=0) for images is in the top left corner with y values increasing downward and x increasing to the right. This might seem weird at first, but if you think about an image as a matrix, it makes sense that the "00" element is in the upper left.**

Now let's try a quiz. Below, I’m plotting a cross section through this image. Where are the areas in the image that are most likely to be identified as edges?

A sign on the side of a road

Description automatically generated

A screenshot of a cell phone

Description automatically generated

**Canny Edge Detection in Action**

Now that you have a conceptual grasp on how the Canny algorithm works, it's time to use it to find the edges of the lane lines in an image of the road. So let's give that a try.

First, we need to read in an image:

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

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

image = mpimg.imread('exit-ramp.jpg')

plt.imshow(image)

A sign on the side of a road

Description automatically generated

Here we have an image of the road, and it's fairly obvious by eye where the lane lines are, but what about using computer vision?

Let's go ahead and convert to grayscale.

**import** cv2 *#bringing in OpenCV libraries*

gray = cv2.cvtColor(image, cv2.COLOR\_RGB2GRAY) *#grayscale conversion*

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

A car parked on the side of a road

Description automatically generated

Let’s try our Canny edge detector on this image. This is where OpenCV gets useful. First, we'll have a look at the parameters for the OpenCV Canny function. You will call it like this:

edges = cv2.Canny(gray, low\_threshold, high\_threshold)

In this case, you are applying Canny to the image gray and your output will be another image called edges. low\_threshold and high\_threshold are your thresholds for edge detection.

The algorithm will first detect strong edge (strong gradient) pixels above the high\_threshold, and reject pixels below the low\_threshold. Next, pixels with values between the low\_threshold and high\_threshold will be included as long as they are connected to strong edges. The output edges is a binary image with white pixels tracing out the detected edges and black everywhere else. See the [OpenCV Canny Docs](http://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/canny_detector/canny_detector.html) for more details.

What would make sense as a reasonable range for these parameters? In our case, converting to grayscale has left us with an [8-bit](https://en.wikipedia.org/wiki/8-bit) image, so each pixel can take 2^8 = 256 possible values. Hence, the pixel values range from 0 to 255.

This range implies that derivatives (essentially, the value differences from pixel to pixel) will be on the scale of tens or hundreds. So, **a reasonable range for your threshold parameters would also be in the tens to hundreds**.

As far as a ratio of low\_threshold to high\_threshold, [John Canny himself recommended](http://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/canny_detector/canny_detector.html#steps) a low to high ratio of 1:2 or 1:3.

We'll also include Gaussian smoothing, before running Canny, which is essentially a way of suppressing noise and spurious gradients by averaging (check out the [OpenCV docs for GaussianBlur](http://docs.opencv.org/2.4/modules/imgproc/doc/filtering.html?highlight=gaussianblur#gaussianblur)). cv2.Canny() actually applies Gaussian smoothing internally, but we include it here because you can get a different result by applying further smoothing (and it's not a changeable parameter within cv2.Canny()!).

You can choose the kernel\_size for Gaussian smoothing to be any odd number. A larger kernel\_size implies averaging, or smoothing, over a larger area. The example in the previous lesson was kernel\_size = 3.

*Note:* If this is all sounding complicated and new to you, don't worry! We're moving pretty fast through the material here, because for now we just want you to be able to use these tools. If you would like to dive into the math underpinning these functions, please check out the free Udacity course, [Intro to Computer Vision](https://www.udacity.com/course/introduction-to-computer-vision--ud810), where the third lesson covers Gaussian filters and the sixth and seventh lessons cover edge detection.

*#doing all the relevant imports*

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

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

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

**import** cv2

*# Read in the image and convert to grayscale*

image = mpimg.imread('exit-ramp.jpg')

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

*# Define a kernel size for Gaussian smoothing / blurring*

*# Note: this step is optional as cv2.Canny() applies a 5x5 Gaussian internally*

kernel\_size = 3

blur\_gray = cv2.GaussianBlur(gray,(kernel\_size, kernel\_size), 0)

*# Define parameters for Canny and run it*

*# NOTE: if you try running this code you might want to change these!*

low\_threshold = 1

high\_threshold = 10

edges = cv2.Canny(blur\_gray, low\_threshold, high\_threshold)

*# Display the image*

plt.imshow(edges, cmap='Greys\_r')

A close up of a logo

Description automatically generated

Here I've called the OpenCV function Canny on a Gaussian-smoothed grayscaled image called blur\_gray and detected edges with thresholds on the gradient of high\_threshold, and low\_threshold.

In the next quiz you'll get to try this on your own and mess around with the parameters for the Gaussian smoothing and Canny Edge Detection to optimize for detecting the lane lines and not a lot of other stuff.

Now it’s your turn! Try using Canny on your own and fiddle with the parameters for the Gaussian smoothing and Edge Detection to optimize for detecting the lane lines well without detecting a lot of other stuff. Your result should look like the example shown below.

A sign on the side of a road

Description automatically generated A picture containing photo, dark, large, bridge

Description automatically generated

The original image (up), and edge detection applied (down ).

Here’s how I did it…

I chose a kernelSize of 5 for Gaussian smoothing, a lowThreshold of 50 and a highThreshold of 150. These selections nicely extract the lane lines, while minimizing the edges detected in the rest of the image, producing the result shown below.

A sign on the side of a road

Description automatically generated A picture containing photo, dark, large, bridge

Description automatically generated

The original image (top), and edge detection applied (bottom).

**Using the Hough Transform to Find Lines from Canny Edges**

In image space, a line is plotted as x vs. y, but in 1962, Paul Hough devised a method for representing lines in parameter space, which we will call “Hough space” in his honor.

In Hough space, I can represent my "x vs. y" line as a point in "m vs. b" instead. The Hough Transform is just the conversion from image space to Hough space. So, the characterization of a line in image space will be a single point at the position (m, b) in Hough space.

So now I’d like to check your intuition… if a **line** in image space corresponds to a **point** in Hough space, what would **two parallel lines** in image space correspond to in Hough space?

A picture containing dark, screen, computer, night

Description automatically generated

In image space, a line is plotted as x vs. y, but in 1962, Paul Hough devised a method for representing lines in parameter space, which we will call “Hough space” in his honor.

In Hough space, I can represent my "x vs. y" line as a point in "m vs. b" instead. The Hough Transform is just the conversion from image space to Hough space. So, the characterization of a line in image space will be a single point at the position (m, b) in Hough space.

So now I’d like to check your intuition… if a **line** in image space corresponds to a **point** in Hough space, what would **two parallel lines** in image space correspond to in Hough space?

A picture containing dark, screen, computer, night

Description automatically generated

A screenshot of a social media post

Description automatically generated

Alright, so a line in image space corresponds to a point in Hough space. What does a point in image space correspond to in Hough space?

A single point in image space has many possible lines that pass through it, but not just any lines, only those with particular combinations of the m and b parameters. Rearranging the equation of a line, we find that a single point (x,y) corresponds to the line b = y - xm.

So what is the representation of a **point** in image space in Hough space?

A screenshot of a computer

Description automatically generated

A screenshot of a cell phone

Description automatically generated

What if you have **2 points** in image space. What would that look like in Hough space?

A picture containing object, clock

Description automatically generated

A screenshot of a cell phone

Description automatically generated

Alright, now we have two intersecting lines in Hough Space. How would you represent their **intersection** at the point (m0, b0) in image space?

A screenshot of a cell phone

Description automatically generated

A screenshot of a social media post

Description automatically generated

So, what happens if we run a Hough Transform on an image of a square? What will the corresponding plot in Hough space look like?

A picture containing clock

Description automatically generated

A screenshot of a cell phone

Description automatically generated

**Implementing a Hough Transform on Edge Detected Image**

Now you know how the Hough Transform works, but to accomplish the task of finding lane lines, we need to specify some parameters to say what kind of lines we want to detect (i.e., long lines, short lines, bendy lines, dashed lines, etc.).

To do this, we'll be using an OpenCV function called HoughLinesP that takes several parameters. Let's code it up and find the lane lines in the image we detected edges in with the Canny function (for a look at coding up a Hough Transform from scratch, check [this](https://alyssaq.github.io/2014/understanding-hough-transform/) out.) .

Here's the image we're working with:

A close up of a logo

Description automatically generated

Let's look at the input parameters for the OpenCV function HoughLinesP that we will use to find lines in the image. You will call it like this:

lines = cv2.HoughLinesP(masked\_edges, rho, theta, threshold, np.array([]),

min\_line\_length, max\_line\_gap)

In this case, we are operating on the image masked\_edges (the output from Canny) and the output from HoughLinesP will be lines, which will simply be an array containing the endpoints (x1, y1, x2, y2) of all line segments detected by the transform operation. The other parameters define just what kind of line segments we're looking for.

First off, rho and theta are the distance and angular resolution of our grid in Hough space. Remember that, in Hough space, we have a grid laid out along the (Θ, ρ) axis. You need to specify rho in units of pixels and theta in units of radians.

So, what are reasonable values? Well, rho takes a minimum value of 1, and a reasonable starting place for theta is 1 degree (pi/180 in radians). Scale these values up to be more flexible in your definition of what constitutes a line.

The threshold parameter specifies the minimum number of votes (intersections in a given grid cell) a candidate line needs to have to make it into the output. The empty np.array([]) is just a placeholder, no need to change it. min\_line\_length is the minimum length of a line (in pixels) that you will accept in the output, and max\_line\_gap is the maximum distance (again, in pixels) between segments that you will allow to be connected into a single line. You can then iterate through your output lines and draw them onto the image to see what you got!

So, here's what its going to look like:

*# Do relevant imports*

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

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

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

**import** cv2

*# Read in and grayscale the image*

image = mpimg.imread('exit-ramp.jpg')

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

*# Define a kernel size and apply Gaussian smoothing*

kernel\_size = 5

blur\_gray = cv2.GaussianBlur(gray,(kernel\_size, kernel\_size),0)

*# Define our parameters for Canny and apply*

low\_threshold = 50

high\_threshold = 150

masked\_edges = cv2.Canny(blur\_gray, low\_threshold, high\_threshold)

*# Define the Hough transform parameters*

*# Make a blank the same size as our image to draw on*

rho = 1

theta = np.pi/180

threshold = 1

min\_line\_length = 10

max\_line\_gap = 1

line\_image = np.copy(image)\*0 *#creating a blank to draw lines on*

*# Run Hough on edge detected image*

lines = cv2.HoughLinesP(masked\_edges, rho, theta, threshold, np.array([]),

min\_line\_length, max\_line\_gap)

*# Iterate over the output "lines" and draw lines on the blank*

**for** line **in** lines:

**for** x1,y1,x2,y2 **in** line:

cv2.line(line\_image,(x1,y1),(x2,y2),(255,0,0),10)

*# Create a "color" binary image to combine with line image*

color\_edges = np.dstack((masked\_edges, masked\_edges, masked\_edges))

*# Draw the lines on the edge image*

combo = cv2.addWeighted(color\_edges, 0.8, line\_image, 1, 0)

plt.imshow(combo)

A picture containing scene, road

Description automatically generated

As you can see I've detected lots of line segments! Your job, in the next exercise, is to figure out which parameters do the best job of optimizing the detection of the lane lines. Then, you'll want to apply a region of interest mask to filter out detected line segments in other areas of the image. Earlier in this lesson you used a triangular region mask, but this time you'll get a chance to use a quadrilateral region mask using the cv2.fillPoly() function (keep in mind though, you could use this same method to mask an arbitrarily complex polygon region). When you're finished you'll be ready to apply the skills you've learned to do the project at the end of this lesson.

Now it's your turn to play with the Hough Transform on an edge-detected image. You'll start with the image on the left below. If you "Test Run" the quiz, you'll get output that looks like the center image. Your job is to modify the parameters for the Hough Transform and impose a region of interest mask to get output that looks like the image on the right. In the code, I've given you a framework for defining a quadrilateral region of interest mask.

A sign on the side of a road

Description automatically generated A picture containing scene, road

Description automatically generated A picture containing dark, skiing, pole, snow

Description automatically generatedThe original image (left), edge detection and Hough transform (center), parameters optimized and region masked on the right.

Here's how I did it: I went with a low\_threshold of 50 and high\_threshold of 150 for Canny edge detection.

For region selection, I defined vertices = np.array([[(0,imshape[0]),(450, 290), (490, 290), (imshape[1],imshape[0])]], dtype=np.int32)

I chose parameters for my Hough space grid to be a rho of 2 pixels and theta of 1 degree (pi/180 radians). I chose a threshold of 15, meaning at least 15 points in image space need to be associated with each line segment. I imposed a min\_line\_length of 40 pixels, and max\_line\_gap of 20 pixels.

With these parameters, I'm picking up the lanes lines and nothing else, so looks like a decent solution!

A sign on the side of a road

Description automatically generated A picture containing dark, skiing, pole, snow

Description automatically generated

The original image (left), with edge detection, region masking, and Hough transform applied (right).

# Tuning Parameters

Parameter tuning is one of the biggest challenges in computer vision - what works well for one image may not work at all with different lighting and/or backgrounds.

Computer Vision Engineers gain an intuition over time for ranges of parameters and different techniques that might work best for a set of situations. When getting started, this can be a big hill to climb. Oftentimes, building a tool to help speed up your iteration between different techniques and thresholds can help you in parameter tuning.

While it's not required for the project, you might consider reading [this blog post](https://medium.com/@maunesh/finding-the-right-parameters-for-your-computer-vision-algorithm-d55643b6f954) from a fellow Self-Driving Car student on their approach to a parameter tuning tool, and consider building one of your own in the future!