**《Computer Vision Foundations and Applications》Homework 1**

1. **Requirements**

**Attention: Please hand in hw1\_release’s hw1.ipynb file with all the outputs and your answers to the written questions.**

Homework 1 focuses on edge detection, applying it to lane detection to aid self-driving cars. This assignment covers **Canny edge detector** and **Hough transform**.



**Part 1: Canny Edge Detector**

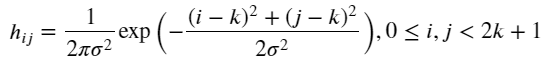
In this part, you are going to implement a Canny edge detector. The Canny edge detection algorithm can be broken down in to five steps:

1. Smoothing
2. Finding gradients
3. Non-maximum suppression
4. Double thresholding
5. Edge tracking by hysteresis

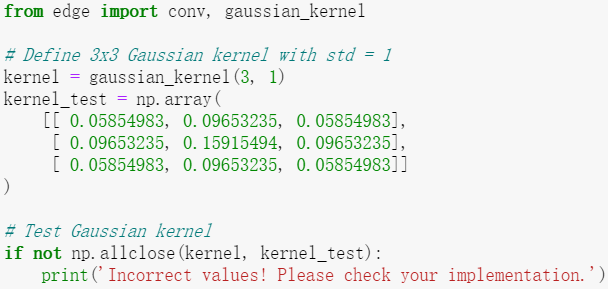
**1.1 Smoothing**

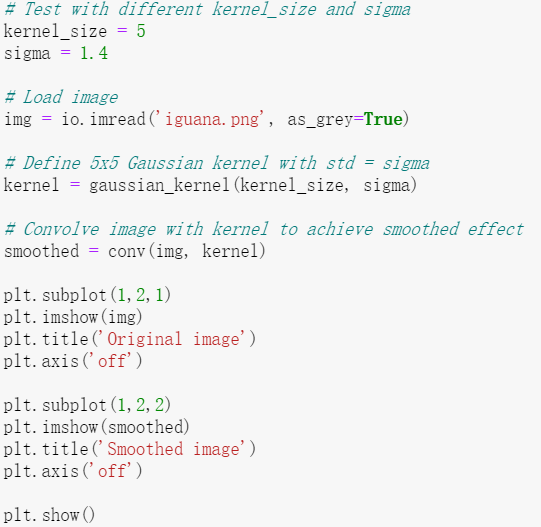
Implementation

We first smooth the input image by convolving it with a Gaussian kernel. The equation for a Gaussian kernel of size is given by:



**Implement1** *gaussian\_kernel* in **edge.py** and run the code below.



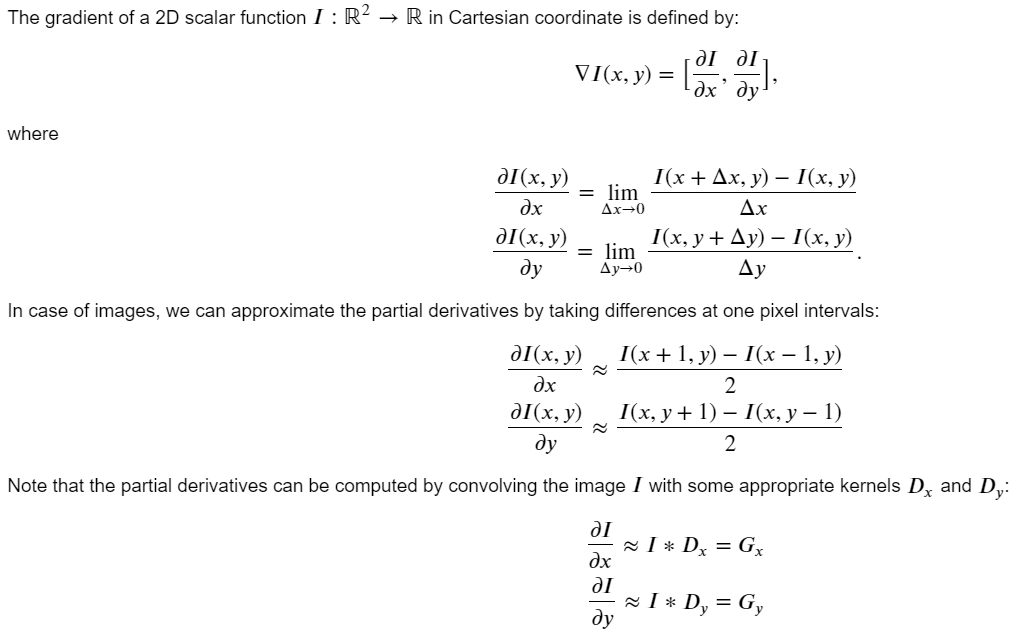


**Question1:**

What is the effect of changing kernel\_size and sigma?

**Your Answer:**

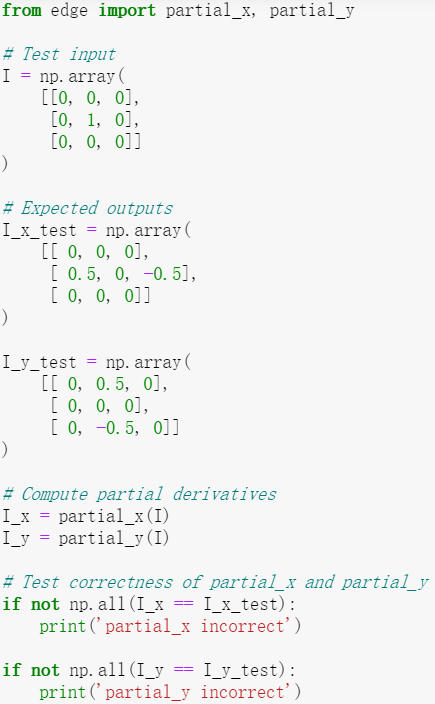
**1.2 Finding gradients**

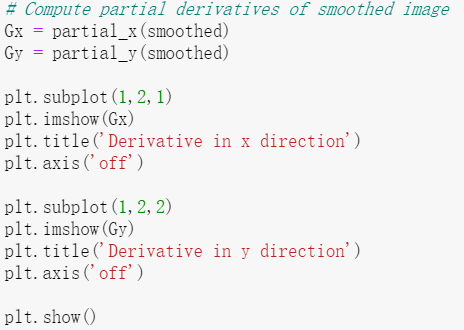


**Implementation**

Find the kernels and  and **implement2** *partial\_x* and *partial\_y* using *conv* defined in **edge.py**.

**-Hint:** Remeber that convolution flips the kernel.





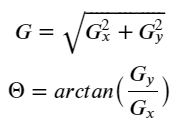
**Question2：**

What is the reason for performing smoothing prior to computing the gradients?

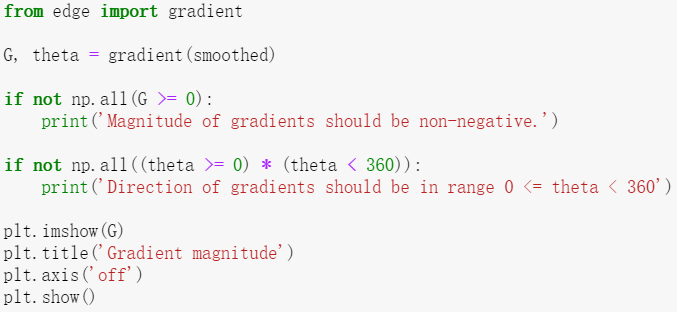
**Your Answer:**

**Implementation**

Now, we can compute the magnitude and direction of gradient with the two partial derivatives:



**Implement3** *gradient* in **edge.py** which takes in an image and outputs 𝐺 and Θ .



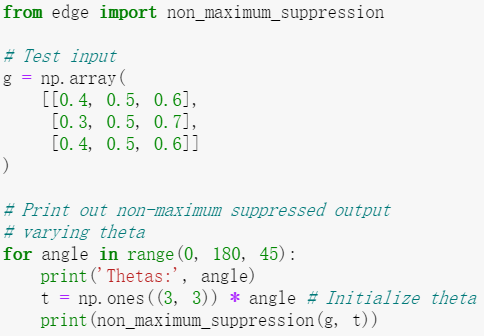
**1.3 Non-maximum suppression**

You should be able to see that the edges extracted from the gradient of the smoothed image are quite thick and blurry. The purpose of this step is to convert the "blurred" edges into "sharp" edges. Basically, this is done by preserving all local maxima in the gradient image and discarding everything else. The algorithm is for each pixel (x,y) in the gradient image:

1. Round the gradient direction Θ[𝑦,𝑥] to the nearest 45 degrees, corresponding to the use of an 8-connected neighbourhood.
2. Compare the edge strength of the current pixel with the edge strength of the pixel in the positive and negative gradient directions. For example, if the gradient direction is south (theta=90), compare with the pixels to the north and south.
3. If the edge strength of the current pixel is the largest; preserve the value of the edge strength. If not, suppress (i.e. remove) the value.

**Implement4** *non\_maximum\_suppression* in **edge.py**.

We provide the correct output and the difference between it and your result for debugging purposes. If you see white spots in the Difference image, you should check your implementation.

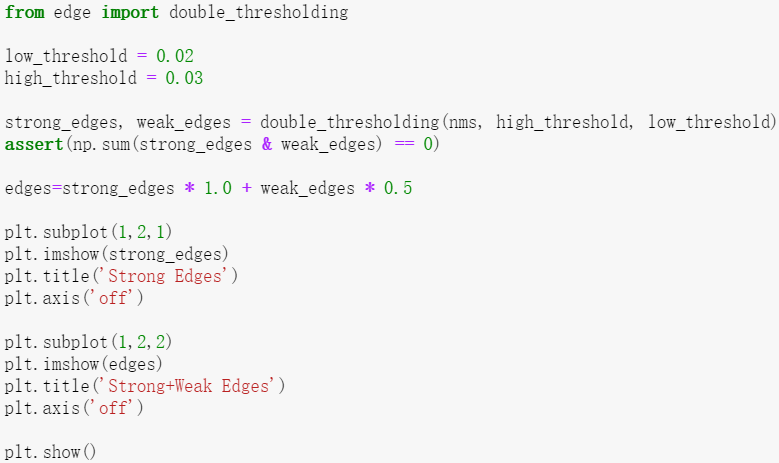




**1.4 Double Thresholding**

The edge-pixels remaining after the non-maximum suppression step are (still) marked with their strength pixel-by-pixel. Many of these will probably be true edges in the image, but some may be caused by noise or color variations, for instance, due to rough surfaces. The simplest way to discern between these would be to use a threshold, so that only edges stronger that a certain value would be preserved. The Canny edge detection algorithm uses double thresholding. Edge pixels stronger than the high threshold are marked as strong; edge pixels weaker than the low threshold are suppressed and edge pixels between the two thresholds are marked as weak.

**Implement5** *double\_thresholding* in **edge.py**

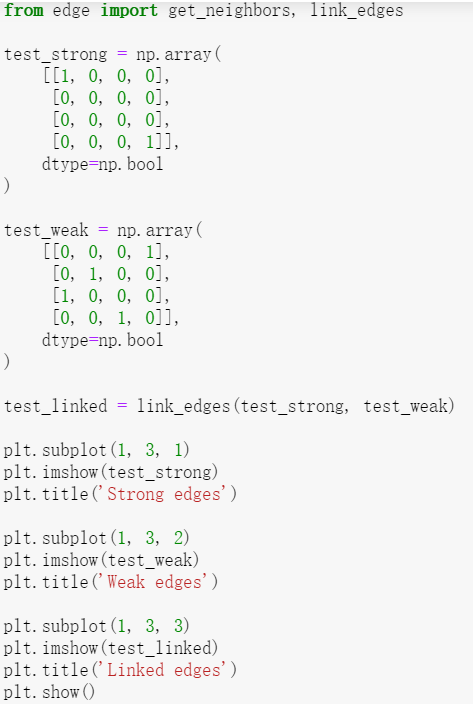


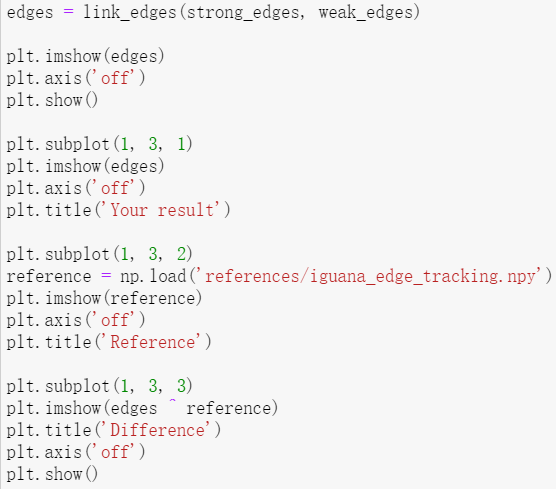
**1.5 Edge tracking**

Strong edges are interpreted as “certain edges”, and can immediately be included in the final edge image. Weak edges are included if and only if they are connected to strong edges. The logic is of course that noise and other small variations are unlikely to result in a strong edge (with proper adjustment of the threshold levels). Thus strong edges will (almost) only be due to true edges in the original image. The weak edges can either be due to true edges or noise/color variations. The latter type will probably be distributed independently of edges on the entire image, and thus only a small amount will be located adjacent to strong edges. Weak edges due to true edges are much more likely to be connected directly to strong edges.

**Implement6** *link\_edges* in **edge.py**.

We provide the correct output and the difference between it and your result for debugging purposes. If you see white spots in the Difference image, you should check your implementation.





**1.6 Canny edge detector**

**Implement7** *canny* in **edge.py** using the functions you have implemented so far. Test edge detector with different parameters.

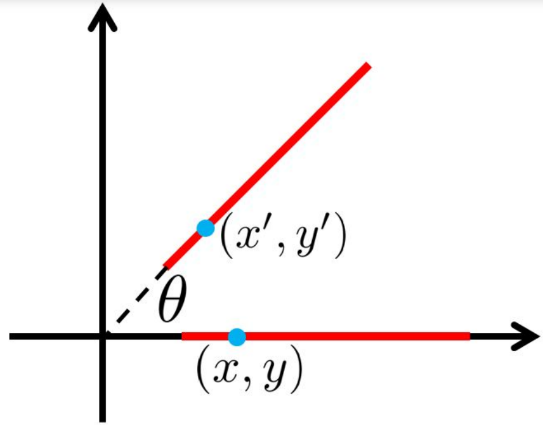
Here is an example of the output:



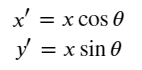
We provide the correct output and the difference between it and your result for debugging purposes. If you see white spots in the Difference image, you should check your implementation.



* 1. **Question3:**



(a) Suppose that the Canny edge detector successfully detects an edge in an image. The edge (see the figure above) is then rotated by θ, where the relationship between a point on the original edge (𝑥,𝑦) and a point on the rotated edge (𝑥′,𝑦′) is defined as



Will the rotated edge be detected using the same Canny edge detector? Provide either a mathematical proof or a counter example.

**-Hint:** The detection of an edge by the Canny edge detector depends only on the magnitude of its derivative. The derivative at point (x, y) is determined by its components along the x and y directions. Think about how these magnitudes have changed because of the rotation.

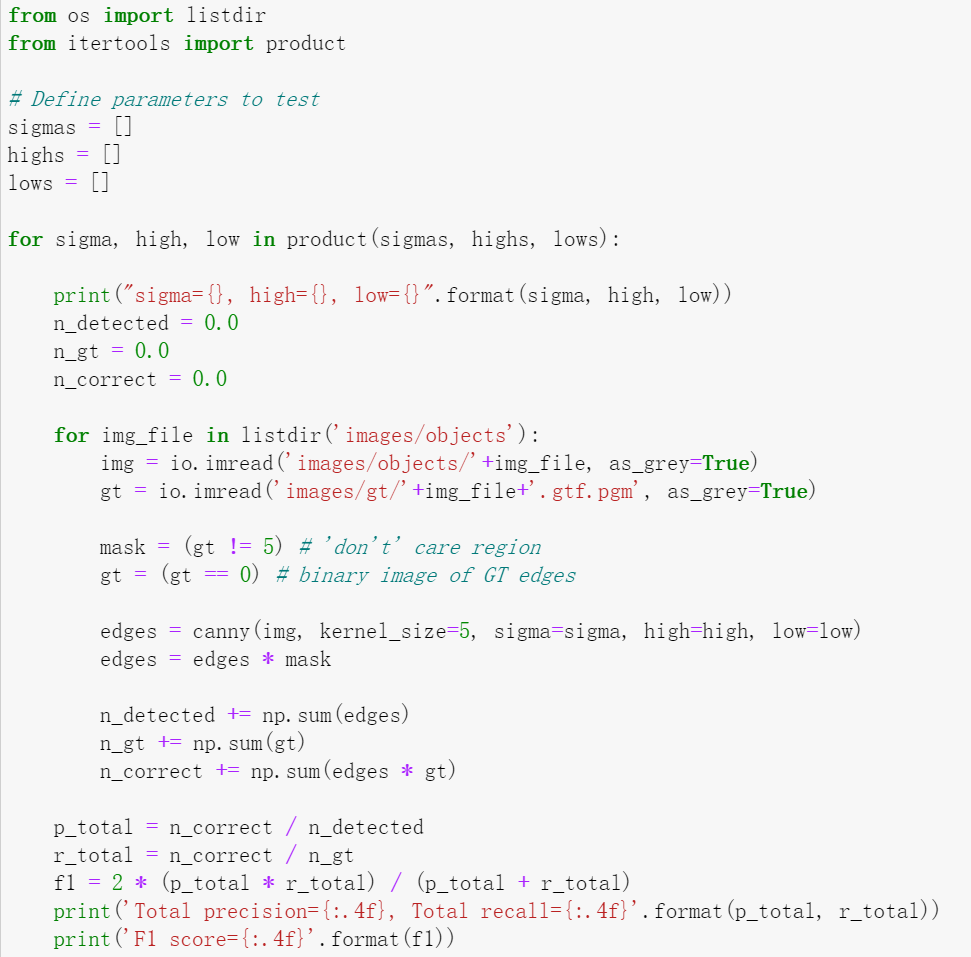
**Your Answer:**

(b) After running the Canny edge detector on an image, you notice that long edges are broken into short segments separated by gaps. In addition, some spurious edges appear. For each of the two thresholds (low and high) used in hysteresis thresholding, explain how you would adjust the threshold (up or down) to address both problems. Assume that a setting exists for the two thresholds that produces the desired result. Briefly explain your answer.

**Your Answer:**

**Extra Credit: Optimizing Edge Detector**

One way of evaluating an edge detector is to compare detected edges with manually specified ground truth edges. Here, we use precision, recall and F1 score as evaluation metrics. We provide you 40 images of objects with ground truth edge annotations. Run the code below to compute precision, recall and F1 score over the entire set of images. Then, tweak the parameters of the Canny edge detector to get as high F1 score as possible. You should be able to achieve F1 score higher than 0.31 by carefully setting the parameters.



**Part2: Lane Detection**

In this section we will implement a simple lane detection application using Canny edge detector and Hough transform. Here are some example images of how your final lane detector will look like.



The algorithm can broken down into the following steps:

1. Detect edges using the Canny edge detector.
2. Extract the edges in the region of interest (a triangle covering the bottom corners and the center of the image).
3. Run Hough transform to detect lanes.

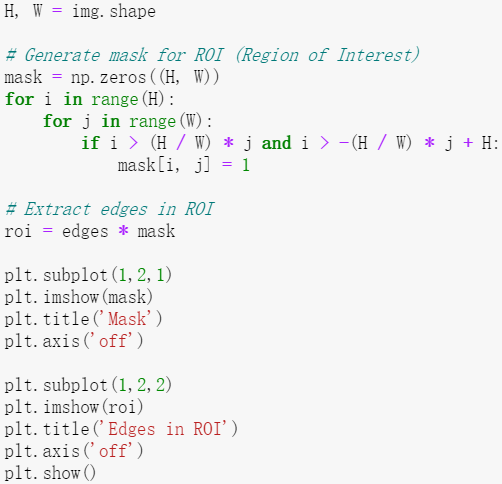
**2.1 Edge detection**

Lanes on the roads are usually thin and long lines with bright colors. Our edge detection algorithm by itself should be able to find the lanes pretty well. Run the code cell below to load the example image and detect edges from the image.



**2.2 Extracting region of interest (ROI)**

We can see that the Canny edge detector could find the edges of the lanes. However, we can also see that there are edges of other objects that we are not interested in. Given the position and orientation of the camera, we know that the lanes will be located in the lower half of the image. The code below defines a binary mask for the ROI and extract the edges within the region.



**2.3 Fitting lines using Hough transform**

The output from the edge detector is still a collection of connected points. However, it would be more natural to represent a lane as a line parameterized as 𝑦=𝑎𝑥+𝑏 , with a slope 𝑎 and y-intercept 𝑏 . We will use Hough transform to find parameterized lines that represent the detected edges.

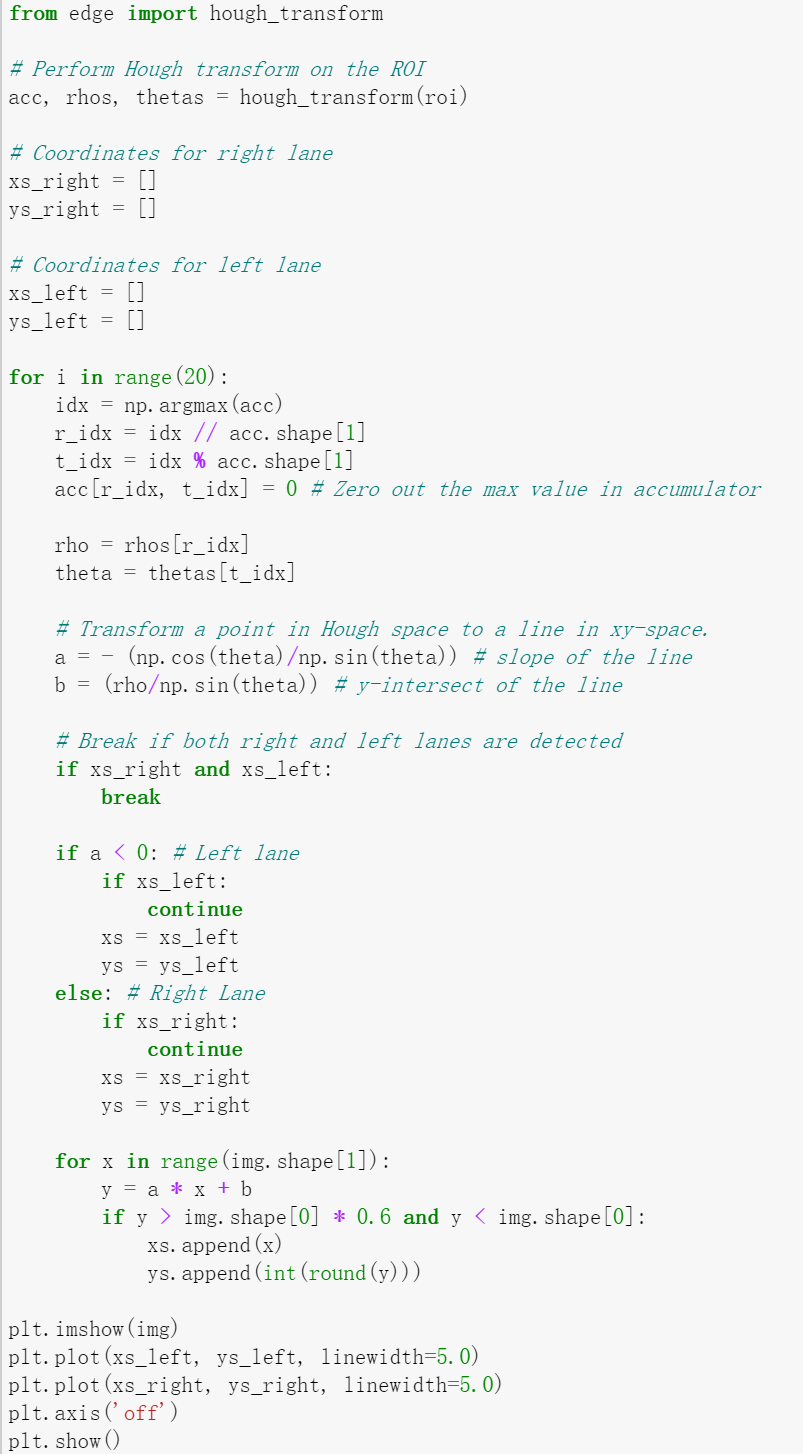
In general, a straight line can be represented as a point in the parameter space. However, this cannot represent vertical lines as the slope parameter will be unbounded. Alternatively, we parameterize a line using 𝜃∈[−𝜋,𝜋] and 𝜌∈ℝ as follows:

C:\Users\夏诗禹\AppData\Roaming\Tencent\Users\1030717635\TIM\WinTemp\RichOle\V3AWX6E{ALL})_9VKC)(U@8.png

Using this parameterization, we can map every point in 𝑥𝑦 -space to a sine-like line in 𝜃𝜌 -space (or Hough space). We then accumulate the parameterized points in the Hough space and choose points (in Hough space) with highest accumulated values. A point in Hough space then can be transformed back into a line in 𝑥𝑦 -space.

See notes on Hough transform.

**Implement8** *hough\_transform* in **edge.py**.



*-Hint: There will be 8* ***Implements*** *and 3* ***Questions*** *overall.*