Question 1

First, we are going to interduce and compare 2 edge detection, Canny Edge Detection and Marr-Hildreth.

Canny Edge Detection:

The Canny edge detector is an edge detection operator that uses a multi-stage algorithm to detect a wide range of edges in images. This algorithm consists of 5 stages like below:

1) Apply Gaussian filter to smooth the image in order to remove the noise
2) Find the intensity gradients of the image
3) Apply non-maximum suppression to get rid of spurious response to edge detection
4) Apply double threshold to determine potential edges
5) Track edge by hysteresis: Finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges

Marr-Hildreth:

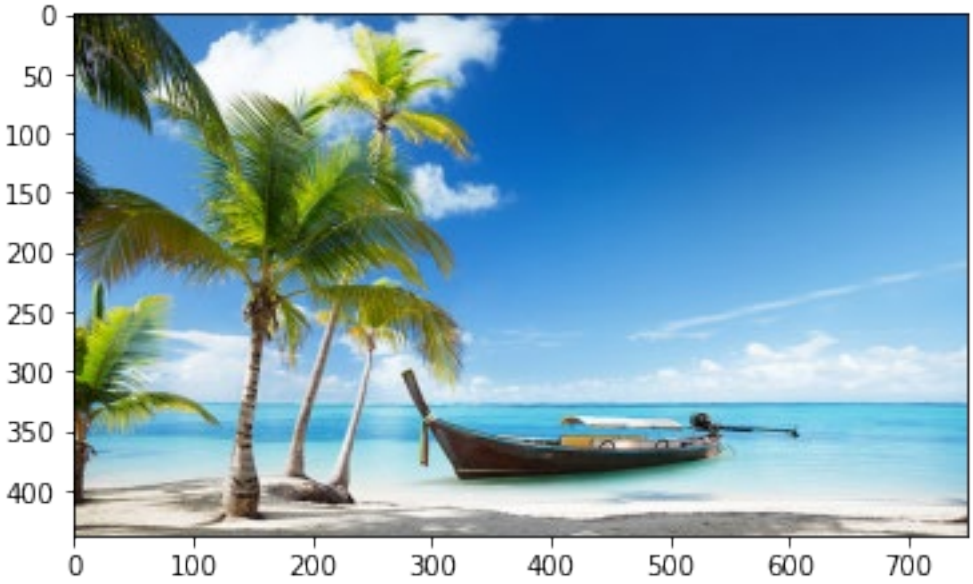
Marr–Hildreth algorithm is a method of detecting edges in digital images, that is, continuous curves where there are strong and rapid variations in image brightness. The Marr–Hildreth edge detection method is simple and operates by convolving the image with the Laplacian of the Gaussian function, or, as a fast approximation by difference of Gaussians. Then, zero crossings are detected in the filtered result to obtain the edges. two main limitations are It generates responses that do not correspond to edges, so-called "false edges", and the localization error may be severe at curved edges

As we can see canny edge detection with some improvement can reach higher accuracy with more robustness than Marr-Hildreth.

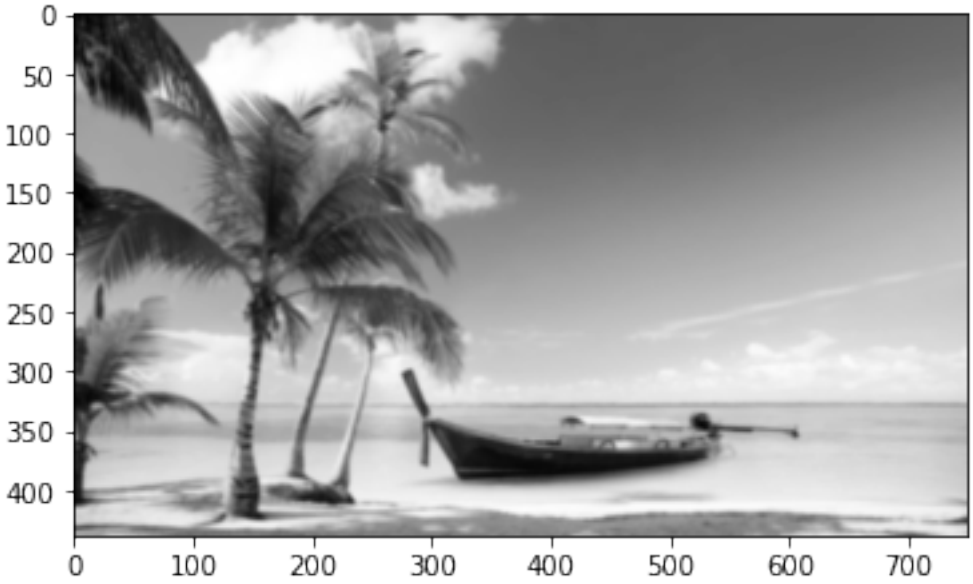
In this Part we are going to implement Canny edge detection from scratch and compare it to the in-built open-cv package.

Steps and its relative codes are described in the attached notebook. So we are going to visualize the outputs here:

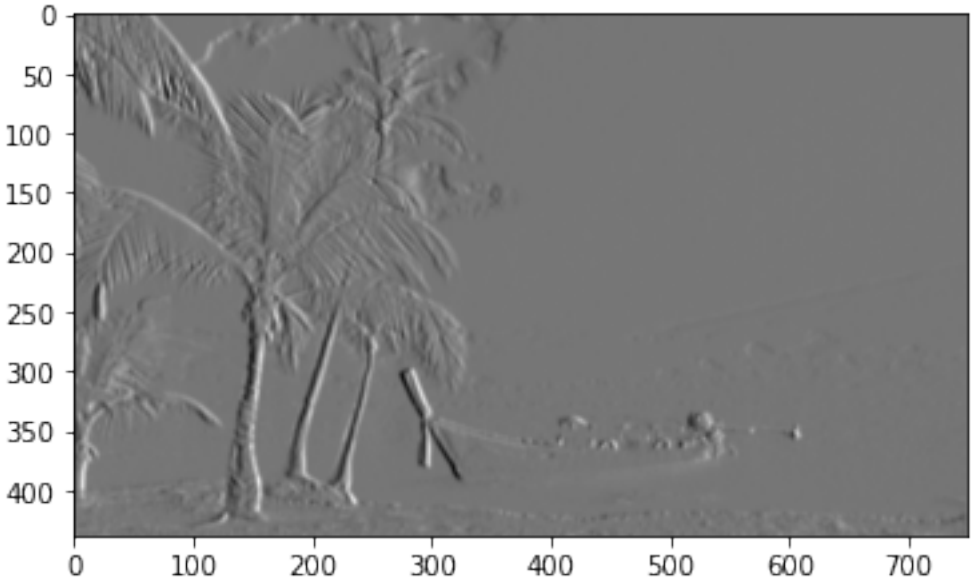
The input image is:

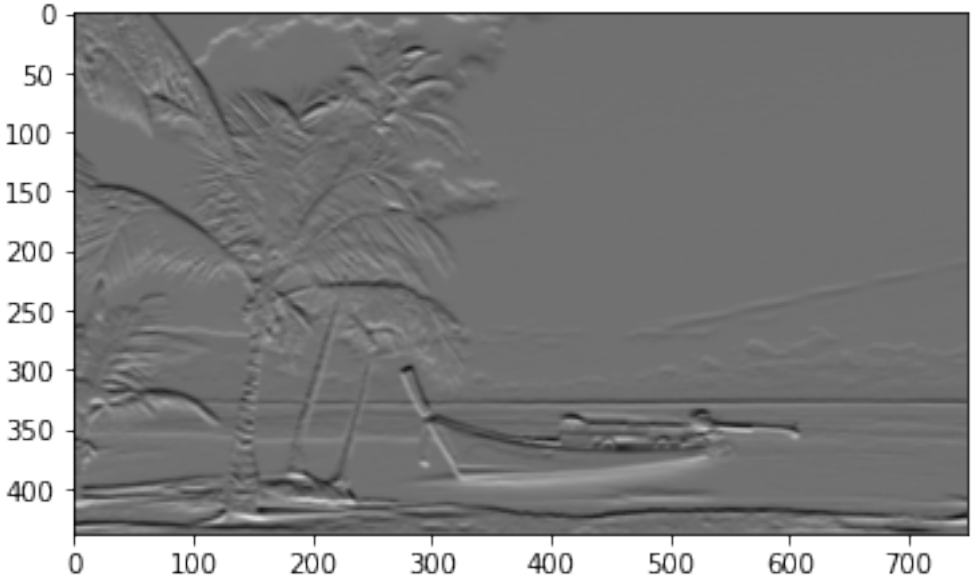


We the convert it to gray scale and smooth it using a gaussian blur filter with kernel size of 11 and sigma of 1.4:

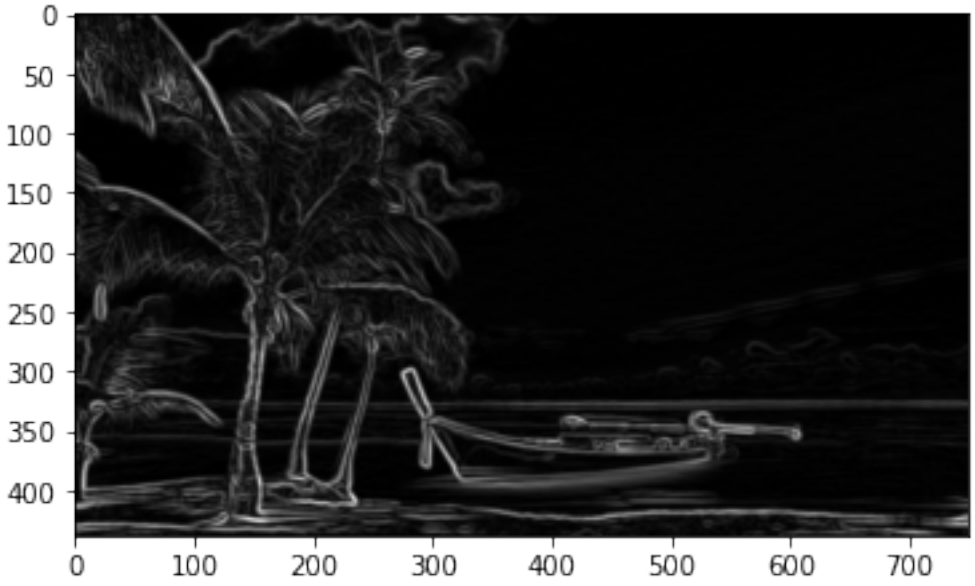


We then calculate the gradient of x and y directions:

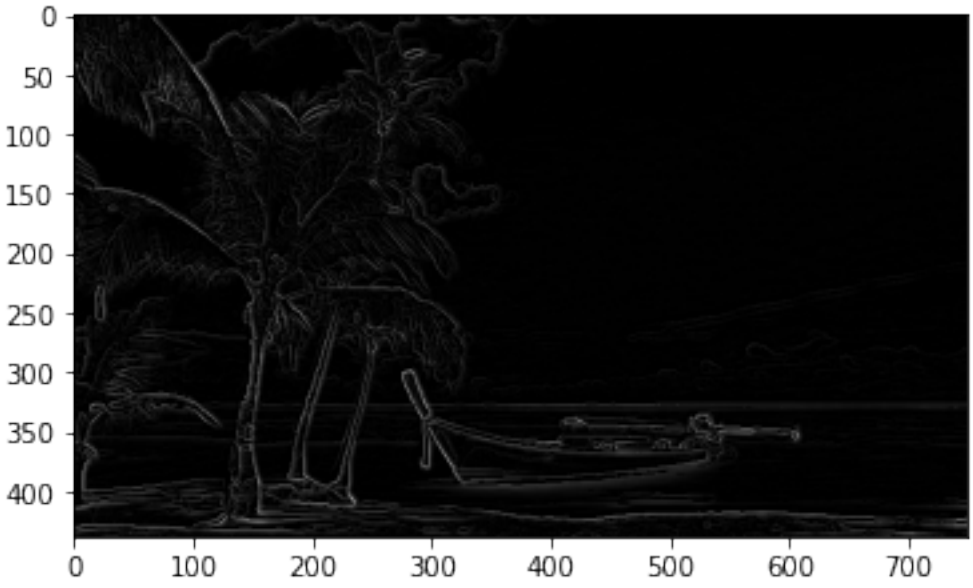




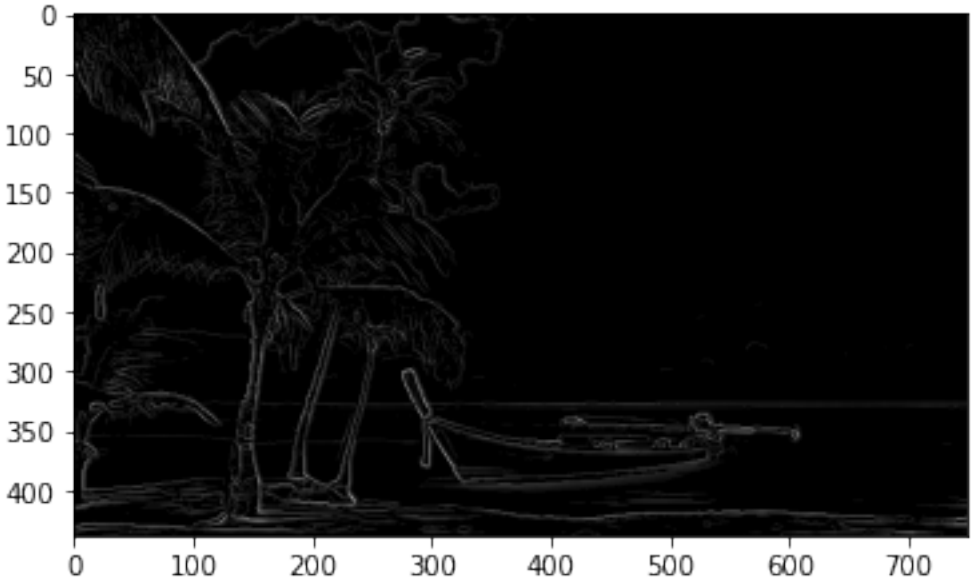
Now get the magnitude of the gradient descents:



After that we apply Non-Maximum Suppression:



Then we can apply double thresh hold to better increase the accuracy:



We can do this all by in-built open-cv function which results in the image below:

Question 2

In this question we are going to implement Hough transform to detect straight lines.

The Hough transform is a feature extraction technique used in image analysis, computer vision, and digital image processing. The purpose of the technique is to find imperfect instances of objects within a certain class of shapes by a voting procedure.

The main idea for the HT is as follows:

- For each line L, there is a unique line L perpendicular to L which passes through the origin.
- L has a unique distance and angle from the horizontal axis of the image. This angle and distance define a point in the parameter space, sometimes known as Hough space.
- A point in image space has an infinite number of lines that could pass through it, each with a unique distance and angle.
- This set of lines corresponds to a sinusoidal function in parameter space. Two points on a line in image space correspond to two sinusoids which cross at a point in parameter space.
- That point in parameter space corresponds to that line in image space, and all sinusoids corresponding to points on that line will pass through that point.

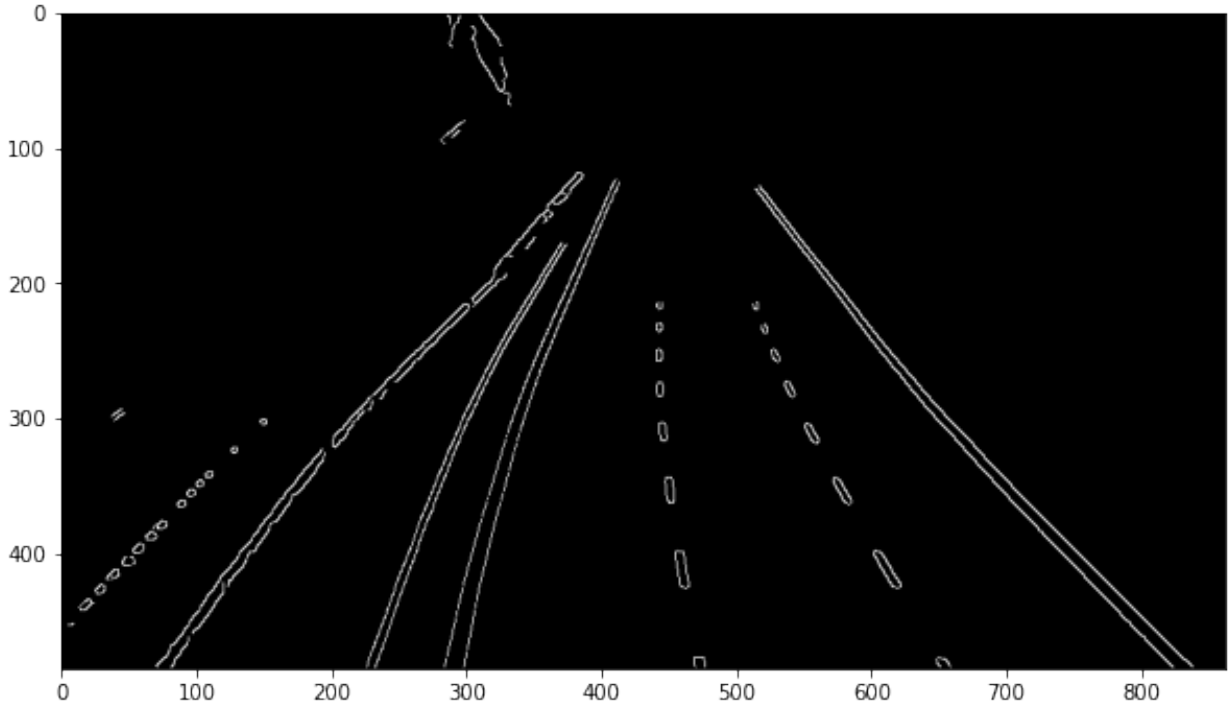
The real solution to implement this algorithm is to quantize the parameter space by using a 2D array of counters, where the array coordinates represent the parameters of the line; this is commonly known as an accumulator array.

The HT method for finding lines in images generally consists of the following three stages:

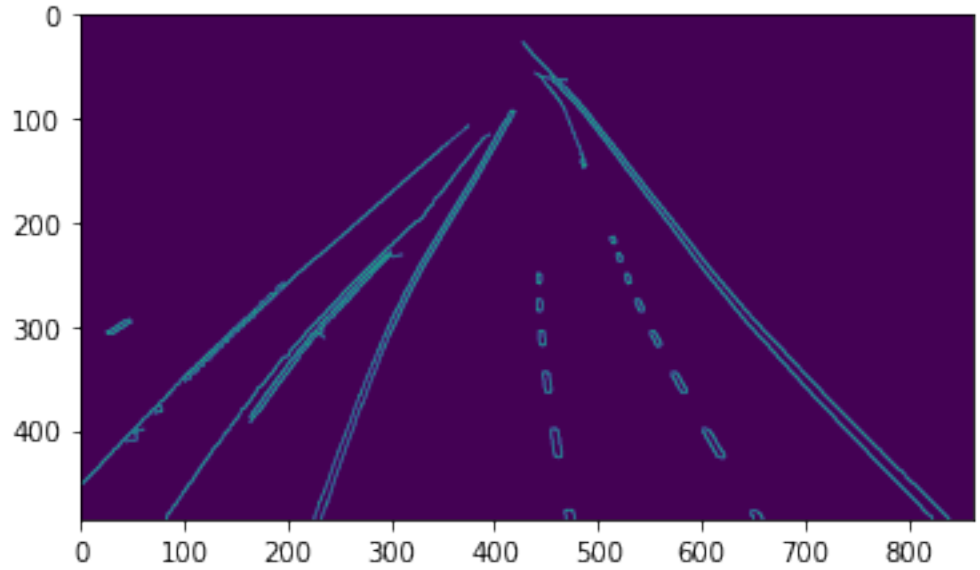
- Perform accumulation on the accumulator array using the binary edge image.
- Find peak values in the accumulator array
- Verify that the peaks found correspond to legitimate lines, rather than noise.

First, we plot the image and find its corresponding edges:

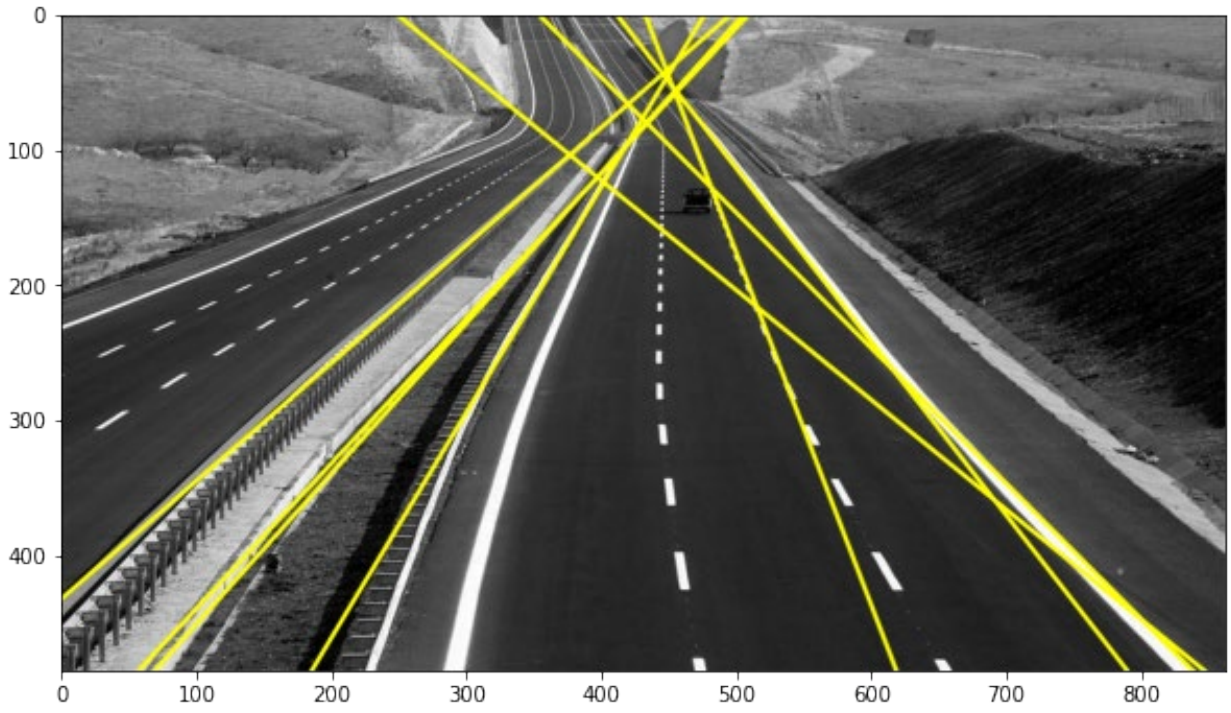
First Image:



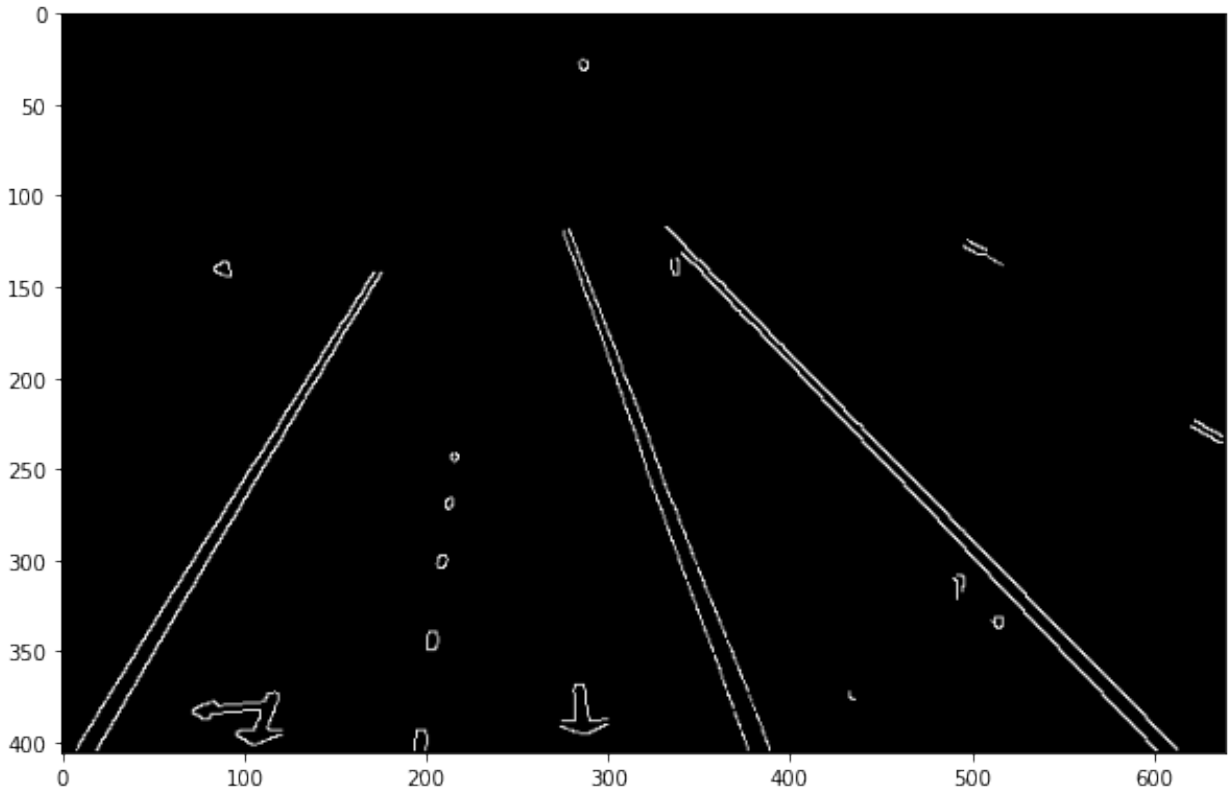
And in Built Canny edge will produce below result:

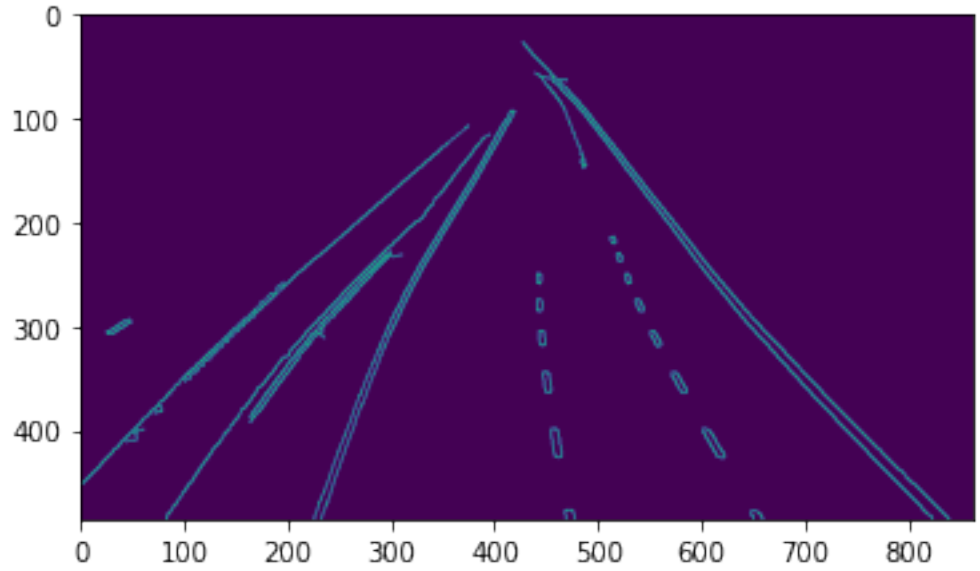


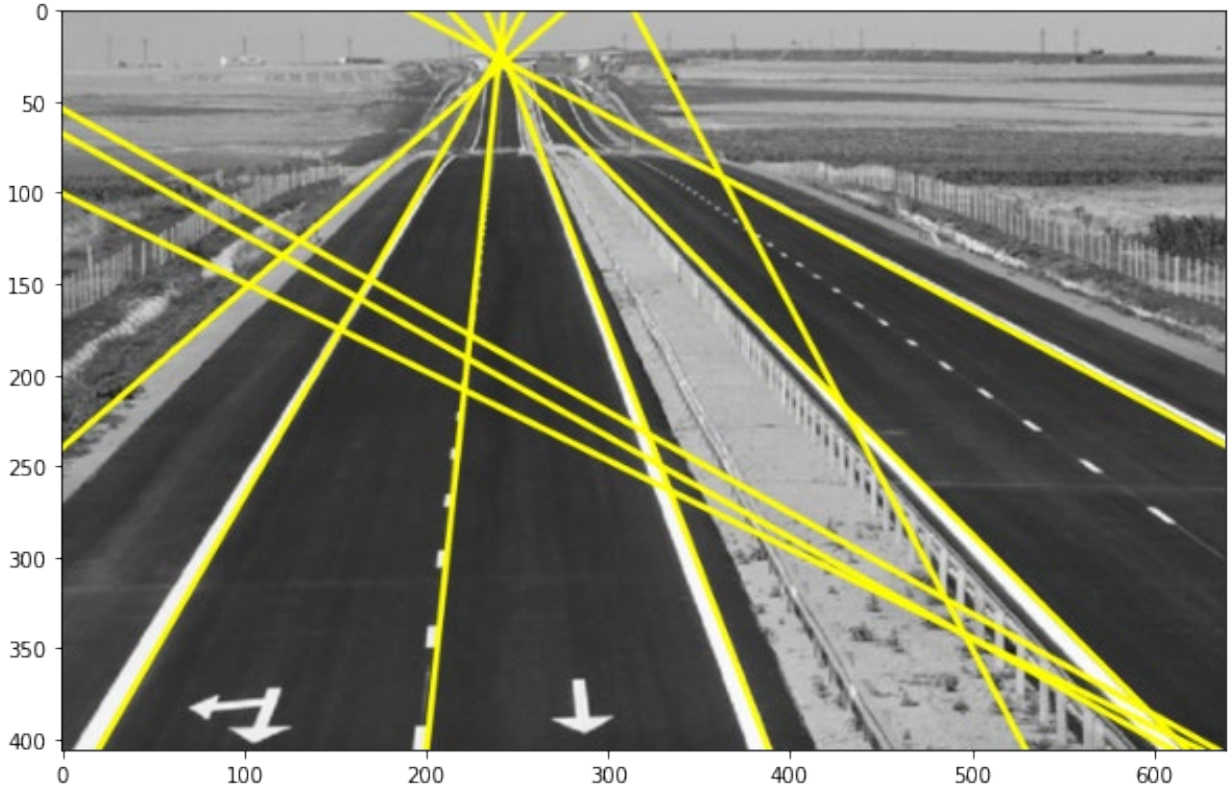
After finding line we have:



For second image we have:




0
100
200
300
400
0
100
200
300
400
500
600
700
800


0
50
100
150
200
250
300
350
400
0
100
200
300
400
500
600

Question 3

In this question we are going to find the best similar faces to each other.

First, we divide the dataset into test and train with equal size.

We then calculate the key points and their related descriptor for each individual image. For this purpose, we calculated the Euclidean distance between descriptors of the two image and if they had 20 key points with less than 150 distance, they will be assumed matched. This is our threshold.

Then the best similar image has been found and showed in Jupiter notebook.

Then will run the same algorithm on test data with same threshold that we found by Running it for several times. We get the first image of each person with its 14 best similar image and the calculate the accuracy.

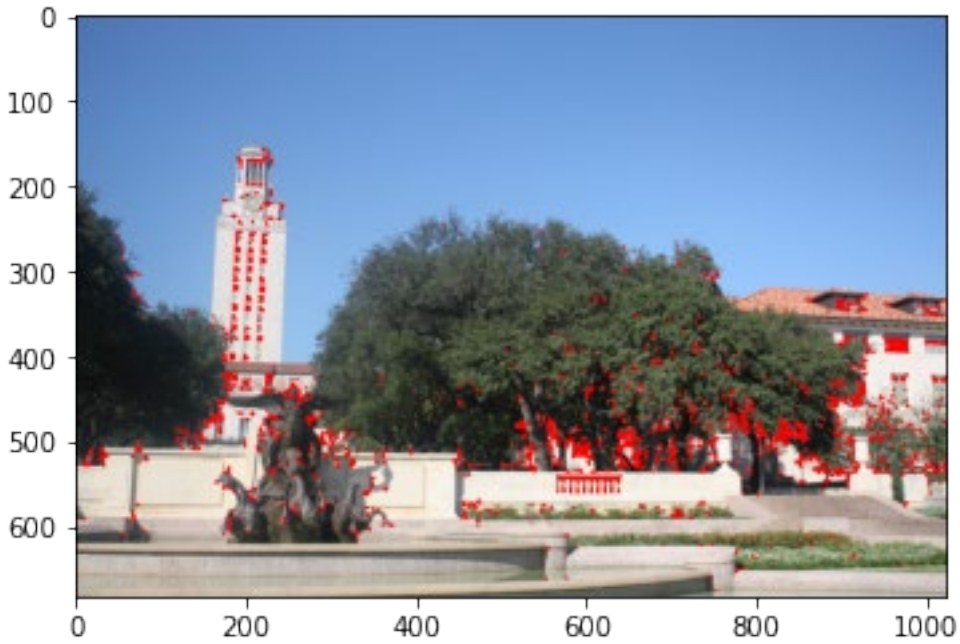
Accuracy was calculated as 1%. I don't know why but probably the threshold was not fine enough.

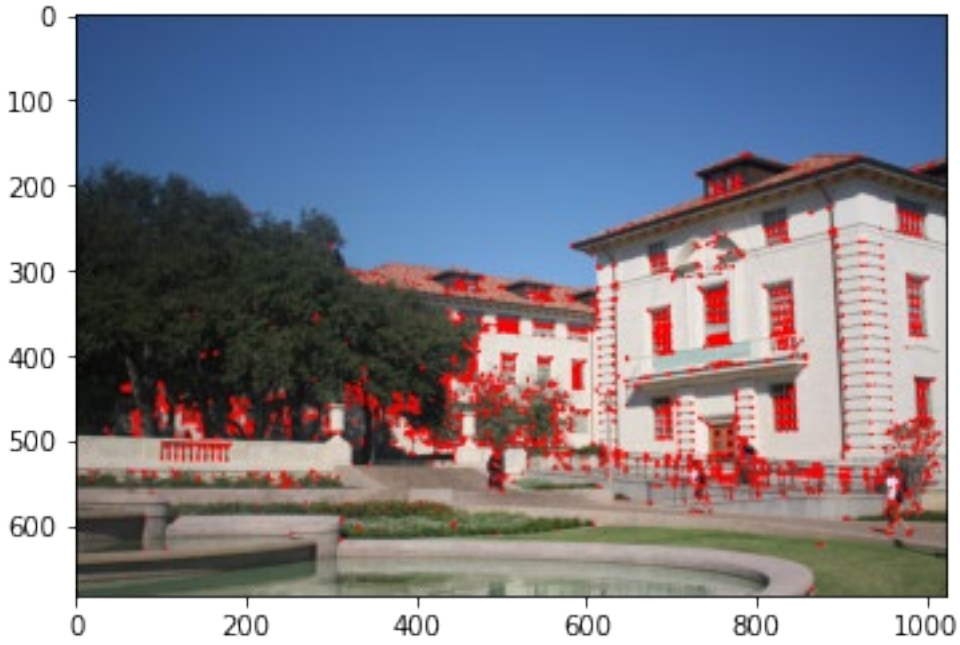
Since the outputs were large (375 best image, fitting and etc..) the outputs are shown in the Jupiter notebook itself.

Question 4

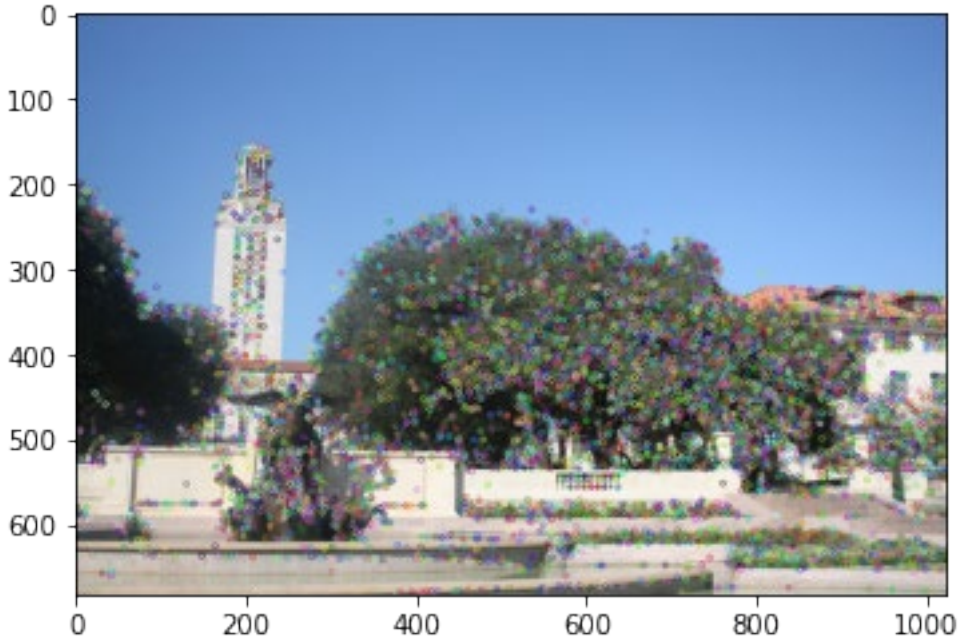
In this question we are going to stich 2 image as a panorama image using SIFT features.

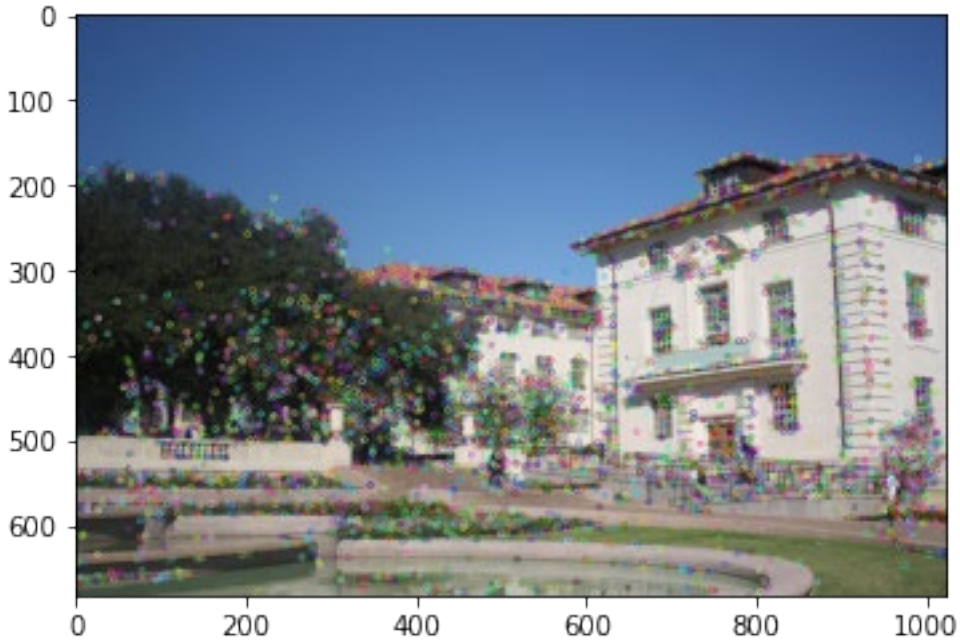
First we are going to detect feature points by Harris algorithm:





Second, We did the same using SIFT and showed the feature points.

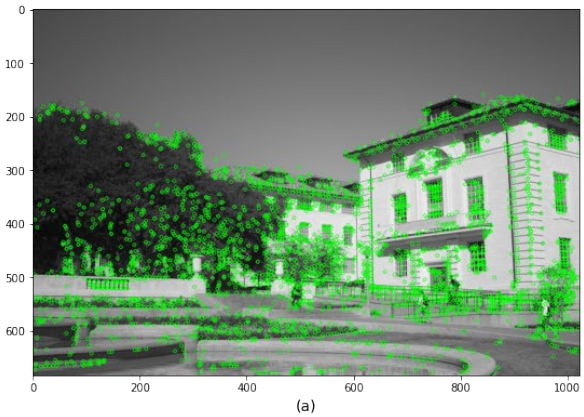


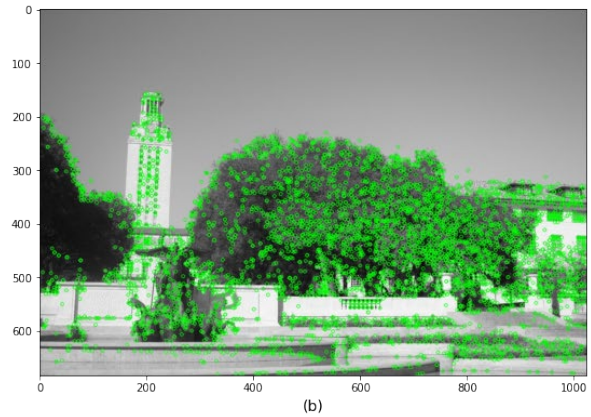


Now we want to match the features.

First we read and plot the images:

Second, we get the key points and descriptors for each key point using SIFT algorithms and plot them:






```
Found keypoints in right: 4280
Found keypoints in left: 5346
```

Third, we should math the key points and find nearest key points.

To do so we implement a function to calculate the Euclidean distance between all the descriptors of the key points in 2 images and then used a Threshold of 130. Means every distance which is lower than 130 will be calculated as matched points.

With this implementation we found 1514 matched points.

We then find the correspondence matrix by adding each 2 points to a list. The first point is from image one and the second point is the matched point to first point in second image. So, we should have a 1514*4 matrix:

```
(1514, 4)
[[   9.41284084  581.62542725  454.49749756  616.42559814]
 [   9.41284084  581.62542725  486.83786011  617.13635254]
 [   9.41284084  581.62542725  508.86181641  585.27722168]
 ...
 [ 980.49682617  350.24014282  251.0741272   630.0982666 ]
 [1002.46795654  594.9005127   196.50958252  459.47033691]
 [1017.65014648  562.42895508  544.44140625  608.38446045]]
```

Now we should implement the RANSAC algorithm. We did this algorithm twice, once to calculate the Homograph matrix and second for Affine matrix.

Steps of the algorithm is demonstrated in the code so we just see the results. We ran the algorithm using 0.6 threshold with 1000 max iteration.

After running we get the below matrixes:

For homograph matrix we have:

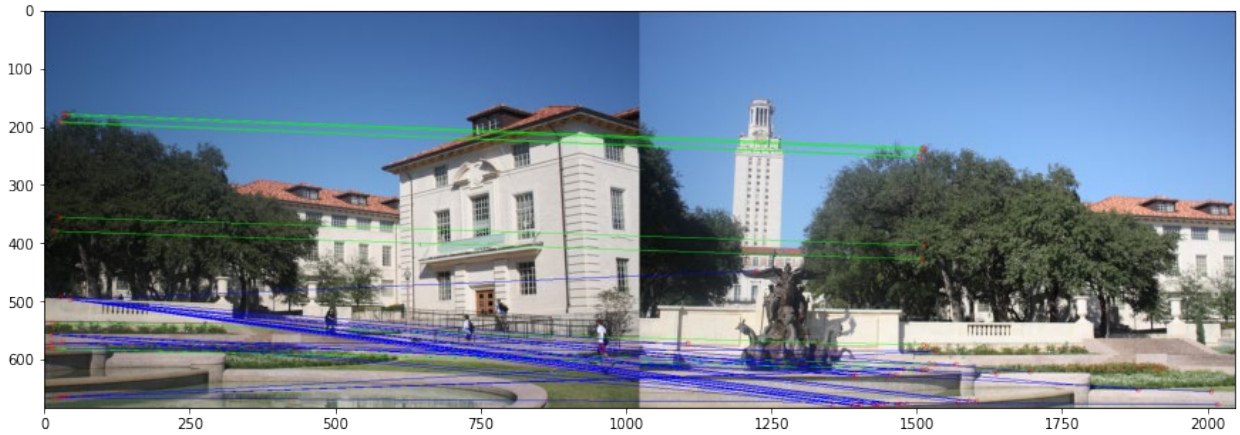
```
Final homography:
[[ 7.75724000e-01  7.09479952e-02  4.39553498e+02]
 [-1.41714805e-01  9.47008860e-01  7.00664038e+01]
 [-2.21163640e-04  1.05844796e-05  1.00000000e+00]]
Final inliers count:  689
```

And for Affine matrix we have:

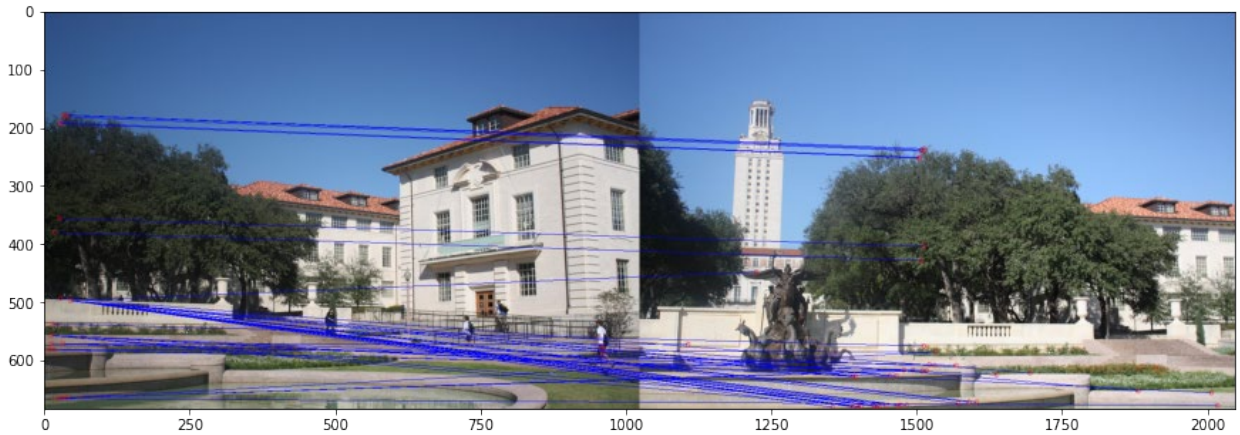
```
Final homography:
[[ 1.03363335e+00  7.54868751e-02  4.08103991e+02]
 [-3.65164695e-02  1.01845386e+00  3.52019583e+01]
 [ 1.73472348e-18  0.00000000e+00  1.00000000e+00]]
Final inliers count:  462
```

So to match inliers in images:

For hemograph matrix we have:



And for Affine matrix we have:



Then we can stich images together and build our Panamera:

For homograph matrix we have:

For Affine we have: