[Fall 2023] ROB-GY 6203 Robot Perception Homework 2

Panayiotis Christou

Submission Deadline (No late submission): NYC Time 11:00 AM, November 29, 2023
Submission URL (must use your NYU account): https://forms.gle/tZVGVSQZ4T8CVUNN7

1. Please submit the **.pdf** generated by this LaTex file. This .pdf file will be the main document for us to grade your homework. If you wrote any code, please zip all the **code** together and **submit a single .zip file**. Name the code scripts clearly or/and make explicit reference in your written answers. Do NOT submit very large data files along with your code!

2. Please **start early**. Some of the problems in this homework can be **time-consuming**, in terms of the time to solve the problem conceptually and the time to actually compute the results. *It's guaranteed that you will NOT be able to compute all the results if you start on the date of deadline.*

3. Please typeset your report in LaTex/Overleaf. Learn how to use LaTex/Overleaf before HW deadline, it is easy because we have created this template for you! **Do NOT submit a hand-written report!** If you do, it will be rejected from grading.

4. Do not forget to update the variables "yourName" and "yourNetID".

Contents

Task 1. RANSAC Plane Fitting (4pt)

In this task, you are supposed to fit a plane in a 3D point cloud. You have to write a custom function to implement the RAndom SAmple Consensus (RANSAC) algorithm to achieve this goal.

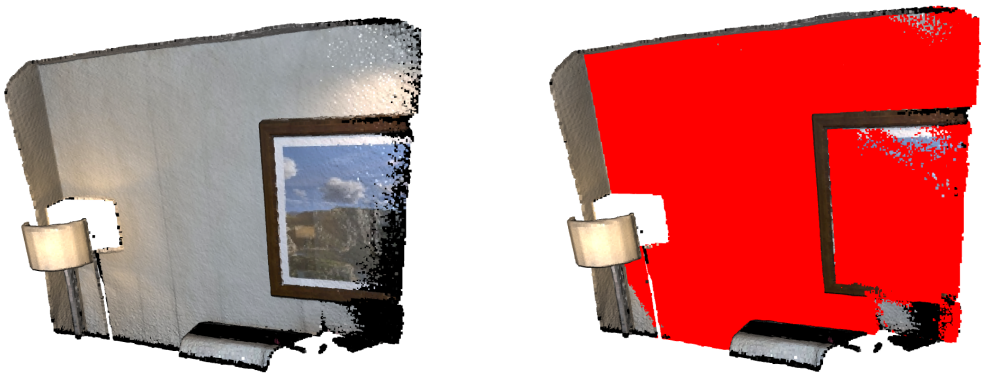

Figure 1: **Left:** Original Data, **Right:** Data with the best fit plane marked in red.

Use the following code snippet to load and visualize the demo point cloud provided by Open3D.

```
import open3d as o3d

# read demo point cloud provided by Open3D
pcd_point_cloud = o3d.data.PCDPointCloud()
pcd = o3d.io.read_point_cloud(pcd_point_cloud.path)

# function to visualize the point cloud
o3d.visualization.draw_geometries([pcd],
                                  zoom=1,
                                  front=[0.4257, -0.2125, -0.8795],
                                  lookat=[2.6172, 2.0475, 1.532],
                                  up=[-0.0694, -0.9768, 0.2024])
```

Note: If you use RANSAC API in existing libraries instead of your own implementation, you will only get 60% of the total score.

Answers:
The image solution for this problem is given at the end of this solution section.
For all the following questions I used ChatGPT.
The algorithm implementation can be found in the python file HW2Q1.py.
To implement the RANSAC algorithm, the multiple aspects of it were implemented as separate functions that get called iteratively. Aside from the main adaptive RANSAC algorithm, the file contains 3 helper functions. The fit plane function takes the 3 randomly sampled points and fits the plane by calculating the cross product between the difference of the 2nd point with the 1st one and the 3rd point with the 1st one to find the normal and then calculating the dot product of the normal with the first point and then switching its sign. The second helper function is the distance to the plane calculated by using the coordinates of the point and plugging them into the plane, getting the absolute value and dividing it by the L2 norm i.e. $distance = \frac{||(A*x+B*y+C*z+D)||}{\sqrt{A^2+B^2+C^2}}$. The third helper function is the check inlier function which loops through the cloud points and calculates their distance to the plane and increments an in-function-initialized count of the inlier points and returns the indices of those points and the count if the distance of each point is within a given threshold defined in the main RANSAC function. The main RANSAC function uses all these helper functions. First it uses an adaptive number of iterations where in each iteration, the error is calculated as $e = 1 - \frac{inliers}{total \quad points}$ and used in $N = ceil(\frac{log(1-p)}{log(1-(1-e)^{sample \quad size})})$ to calculate the new number

of iterations for the algorithm to run. The main RANSAC function defines the number of iterations N initially to be infinity and the threshold distance for a point to be an inlier as 0.025. The while loop keeps running as long as the number of iterations is higher than the sample count where the sample count is incremented at the end of each iteration. These adaptive number of iterations, incrementing sample count and constant check of the error in each iteration are the basic building blocks that make the RANSAC algorithm adaptive. The algorithm in each iteration gets all the point cloud points and converts them to a numpy array, samples 3 random indices to be used to sample 3 random points from the point cloud, calls the fit plan function with those 3 points and then calls the check inlier function. It then checks if the inliers of these plane are more than the best plane (which is initialized to none in the beginning of the function) then it rewrites the total inlier indices variable (which holds the indices of the inliers for the best plane i.e. the plane with the most inliers as of this iteration) and then rewrites the best plane with this new plane. This happens in each iteration along with the adaptive number of iterations check until the algorithm converges. The algorithm then paints all of the inliers with a red color by using the total inlier indices found in the while loop and then visualizes the point cloud. The point cloud is given below.
Check the notebook for a better idea of the sequence of the functions and the call of each function throughout the algorithm.

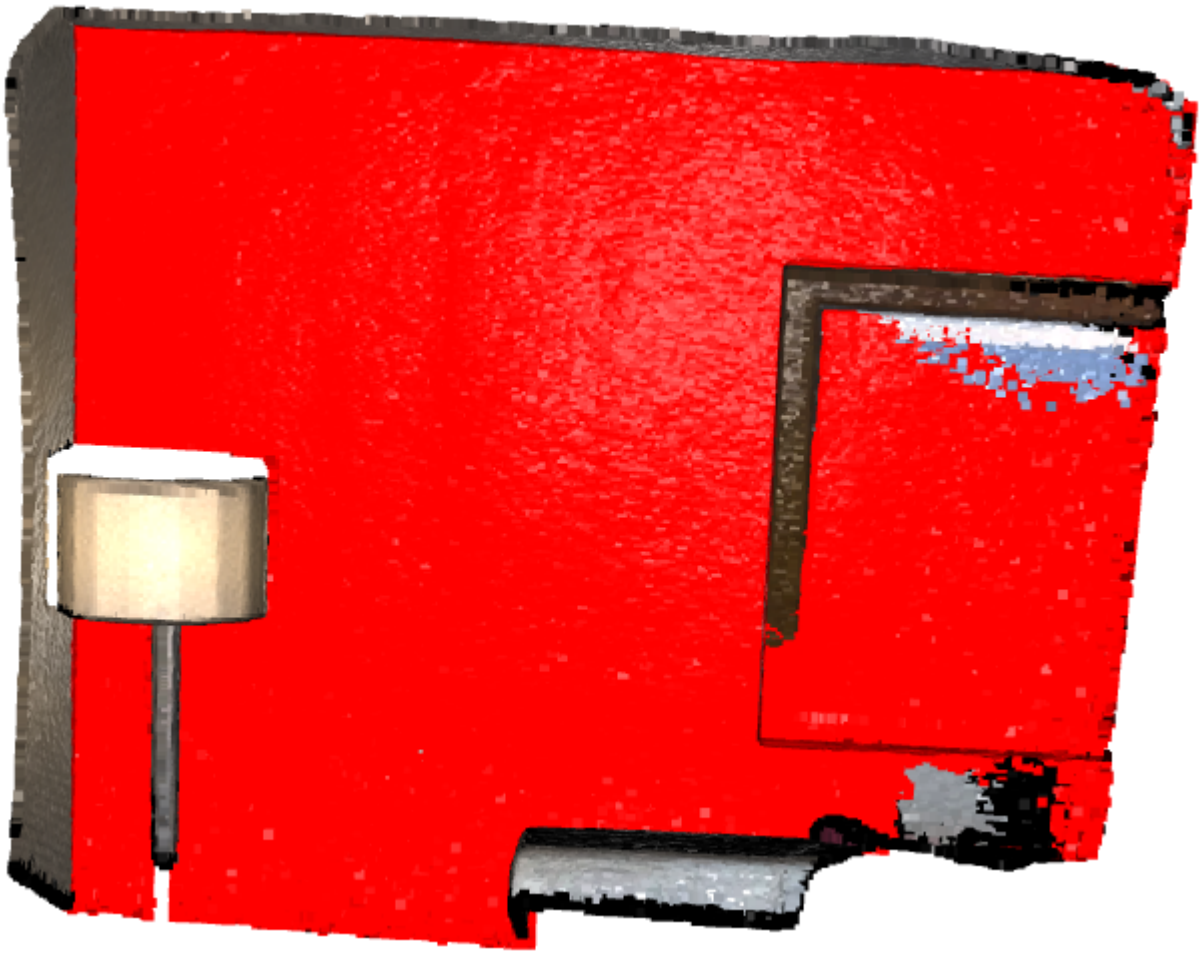

Figure 2: **Fitted RANSAC**

Task 2. ICP (4pt)

In this task, you are required to align two point clouds (source and target) using the Iterative Closest Point (ICP) algorithm discussed in class. The task consists of two parts.

a) (2pt)

In part 1, you have to load the demo point clouds provided by Open3D and align them using ICP. **Caution:** These point clouds are different from the point cloud used in the previous question. You are expected to **write a custom function to implement the ICP algorithm**. Use the following code snippet to load the demo point clouds and to visualize the registration results. You will need to pass the final 4X4 homogeneous transformation (pose) matrix obtained after the ICP refinement. Explain in detail, the process you followed to perform the ICP refinement.

```
import open3d as o3d
import copy

demo_icp_pcds = o3d.data.DemoICPPointClouds()
source = o3d.io.read_point_cloud(demo_icp_pcds.paths[0])
target = o3d.io.read_point_cloud(demo_icp_pcds.paths[1])

# Write your code here

def draw_registration_result(source, target, transformation):
    """
    param: source - source point cloud
    param: target - target point cloud
    param: transformation - 4 X 4 homogeneous transformation matrix
    """
    source_temp = copy.deepcopy(source)
    target_temp = copy.deepcopy(target)
    source_temp.paint_uniform_color([1, 0.706, 0])
    target_temp.paint_uniform_color([0, 0.651, 0.929])
    source_temp.transform(transformation)
    o3d.visualization.draw_geometries([source_temp, target_temp],
                                      zoom=0.4459,
                                      front=[0.9288, -0.2951, -0.2242],
                                      lookat=[1.6784, 2.0612, 1.4451],
                                      up=[-0.3402, -0.9189, -0.1996])
```

Answers:
To perform the ICP we first need the helper function find nearest neighbors which first takes the point cloud and converts it to a tree with KDTreeFlann and then uses a knn vector search in 3D to find the nearest neighbors which in the function call from the ICP is defined as 1 therefore its searching for the closest neighbor to that point. The ICP function first colors the source and the target points in different colors and converts the target points to a numpy array. Then it uses the transformation matrix from the tutorial from Open3D as an initializing transformation matrix. It then transforms the source with the initial transformation matrix and it initializes the cost to change threshold to 0.001, the current cost to 1000 and the previous cost to 10000. The while loop then finds the new source points by running find nearest neighbors using the source and the target and then it calculates the cloud centroids and their repositions. The centroids are found using the mean along the rows and then to calculate the reposition for the target adn the source we loop through all the points found using the find nearest function and subtract the centroid. We then find the covariance matrix using matrix multiplication of the transpose of the new target position and new source position. We then decompose the covariance matrix to U, X and V using SVD and then we get the transformation matrix by matrix multiplication between U and V. The transformation matrix is found by subtracting the result of the matrix multiplication of the rotation matrix with the source centroid from the target centroid. The current cost is calculated as the normal between the new target poistion and the transpose of the matrix multiplication between the rotation matrix and the transpose of the new source position. If the previous cost minus the current cost is higher than the cost to change threshold we set the previous cost to the current cost

and we get the new transformation matrix by horizontally stacking the rotation matrix and the transpose of the translation matrix and then we stack a [0 0 0 1] at the bottom to make it homogeneous and then update the source by transforming it with the new transform matrix and increase the current iteration count. At the end we visualize the result. The result is shown below as Fitted ICP - PART A.
The corresponding notebook for this question, both for Part A and Part B is HW2Q2.py. You can find the full code implementation in the notebook.

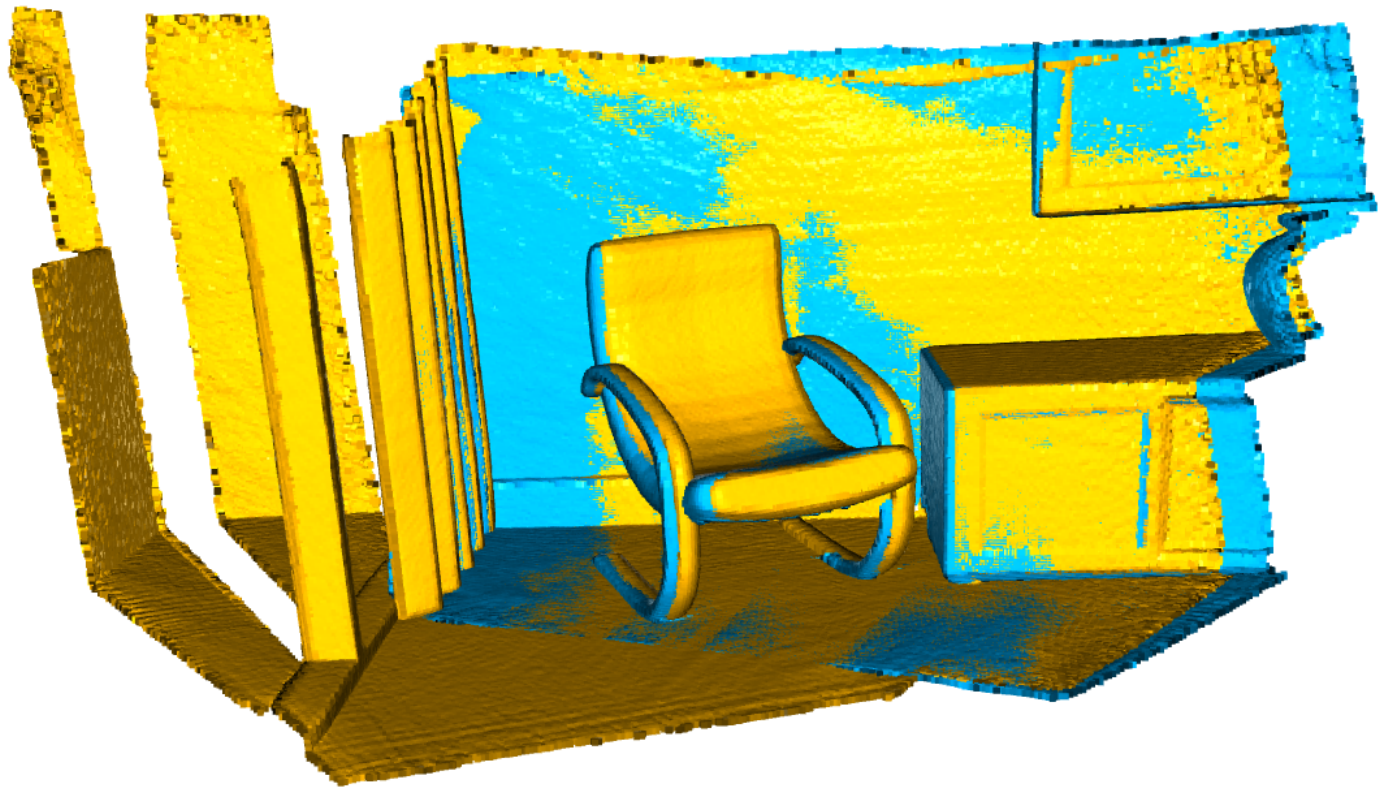

Figure 3: **Fitted ICP - PART A**

b) (2pt)

You have been given two point clouds from the KITTI dataset, one of the benchmark datasets used in the self-driving domain. *The point clouds are located in the data/Task2 folder under the overleaf project: https://www.overleaf.com/read/fzhnnqsqrnbz*. Repeat part 1 using these two point clouds. Compare the results from part 1 with the results from part 2. Are the point clouds in part 2 aligning properly? If no, explain why. Provide the visualizations for both parts in your answer.

Note: If you use an ICP API in existing libraries instead of your own implementation, you will only get 60% of the total score.

Answers:
The algorithm used here is the same as in the previous part and it converged, evident from the majority of the points being on top of each other. But unlike Part A, Part B is not fitted completely and is most likely because the source and the target do not have 1 to 1 correspondence and at the same time they do not have the same amount of points. It is also possible that the KITTI point clouds require different initialization matrices and may even contain enough outliers where the convergence of the algorithm is hindered. This leads to inconsistencies when fitting the data which is evident from the visualization given below.

Figure 4: **Fitted ICP - PART B**

Task 3. F-matrix and Relative Pose (4pt)

All the raw pictures needed for this problem are provided in the `data/Task3` folder under the overleaf project: https://www.overleaf.com/read/fzhnnqsqrnbz. You may or may not need to use all of them in your problem solving process.

a) (2pt)

Estimate the fundamental matrix between `left.jpg` and `right.jpg`.

Tips: The Aruco tags are generated using Aruco's 6×6 dictionary. Although you don't have to use these tags.

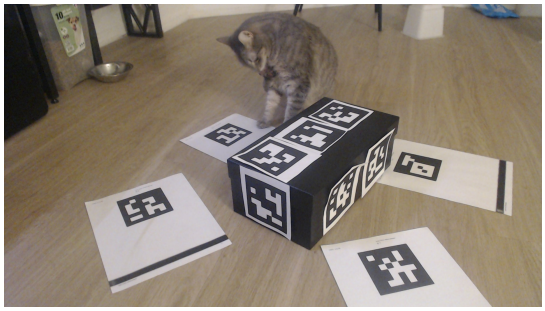

Figure 5: left.jpg

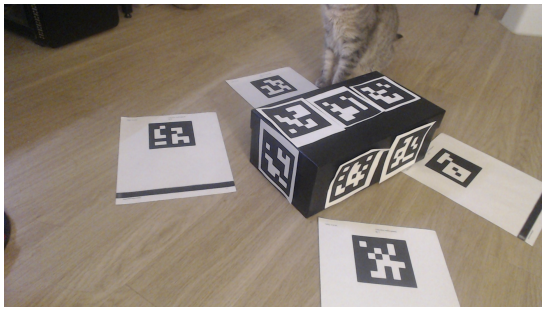

Figure 6: right.jpg

Answers:
The code for all parts of question 3 can be found in the python file HW2Q3.py. Refer to that for the actual implementation.
To find the fundamental matrix we first convert both pictures in grayscale and then use ORB to detect keypoints and compute the descriptors. We then initialize a BF matches and find the matches between the 2 images and using those we extra the matched keypoints between the 2 images. We use those and a RANSAC algorithm with open cv2's findFundamentalMat function to find the fundamental matrix given below:

Fundamental matrix:

$$\begin{bmatrix} 4.57160620e-07 & 1.83036087e-06 & -1.33186839e-03 \\ -1.81160668e-06 & 8.34476433e-07 & 1.46204082e-03 \\ 1.18490888e-04 & -2.37679649e-03 & 1.00000000e+00 \end{bmatrix}$$

b) (1pt)

Draw epipolar lines in both images. You don't need to explain the process. Just provide the visualization.

Answers:
The epipolar lines are given below in Figure 7. Each line in both images passes through the same point and if you stacked them on top of each other, the point the lines meet outside the image is the epipolar center of the images.

c) (1pt)

Find the relative pose (R and t) between the two images, expressed in the left image's frame. Before you give the solution, answer the following two questions

1. Can you directly use the F-matrix you estimated in a) to acquire R and t without calculating any other quantity?

2. If yes, please describe the process. If no, what other quantity/matrix do you need to calculate to solve this problem?

Answers:
1. You cannot find R and t using only the fundamental matrix.

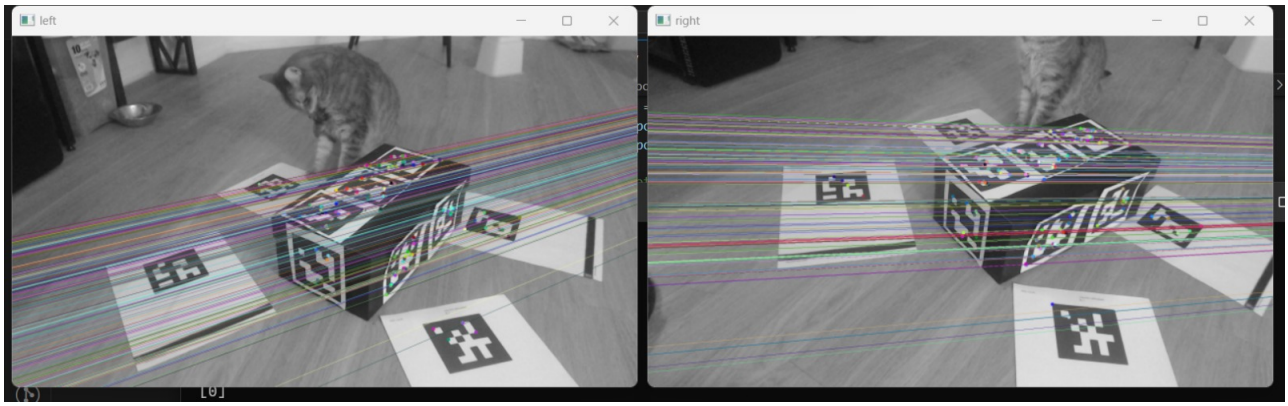

Figure 7: **Visualized Epipolar Lines**

2. First you need to use the checkerboard images to do camera calibration and get the camera matrix K and the distortion parameters. Then you need to perform a dot product between the fundamental matrix and the camera matrix and then a dot product between the transpose of the camera matrix and the aforementioned result to get the essential matrix E. We then use open cv2's recover pose with the matched keypoints, the essential matrix and the camera matrix to find R and t. The calibration to get the camera matrix was done the same way as homework 1 so I won't repeat the process. For the detailed implementation see the python file HW2Q3.py.

Camera Matrix (K):

$$\begin{bmatrix} 1.40936609e+03 & 0.00000000e+00 & 9.90752053e+02 \\ 0.00000000e+00 & 1.41351032e+03 & 5.89506906e+02 \\ 0.00000000e+00 & 0.00000000e+00 & 1.00000000e+00 \end{bmatrix}$$

Distortion Parameters (k1, k2):

$$\begin{bmatrix} 0.03350148653581941 & -0.35689743006611035 \end{bmatrix}$$

Rotation Matrix (R):

$$\begin{bmatrix} 0.95346003 & -0.26298782 & 0.1474835 \\ 0.28037189 & 0.95324416 & -0.11277046 \\ -0.11093053 & 0.14887236 & 0.98261459 \end{bmatrix}$$

Translation Vector (t):

$$\begin{bmatrix} -0.95858195 \\ 0.02913408 \\ -0.28332287 \end{bmatrix}$$

Task 4. Object Tracking (4pt)

Given a short video sequence, persistently track the entities in said sequence across time until it goes out of frame, or the sequence terminates. You can find the aforementioned sequence in the *data/Task4 folder of this Overleaf project*. You are free to use any tracking algorithm or pre-trained model for the task. With your implementation, answer the following two questions:

1. Can you explain in detail, the process or algorithm you used to perform the tracking?

2. Additionally, can you provide a few example frames from your resulting sequence with the proper visualization to demonstrate the efficacy of your implementation?

Note: By *tracking*, it means that you should be able to return the bounding box or centroid coordinate(s) of the entities in the video across the entire sequence.

Tip 1: For the second part of the question, you should provide an example via a few adjacent frames so that the tracking performance is obvious. You should also use consistent labelling or color coding for your visualization corresponding to each unique entity for consistency if possible.

Tip 2: You should yield something like the following for your implementation and submission.

Figure 8: Frame t_1

Figure 9: Frame t_2

Answers:
1. We first use cv2.dnn.readNet to read the model chosen for tracking. For this question I used yolov3 from ultralytics which is pretrained for detection and tracking. The configuration file and the weights were downloaded from online and will be given along with the code. The code is the python file HW2Q4.py. The file yolo3.txt contains all the classes available in the yolov3 model and we read that to get those. We define a helper function to draw the bounding box that draws a rectangle around the object by getting the objects left bottom corner and defining the width and the height of the bounding box and also includes the label on top. The function also has an input for confidence but it has not been implemented, it is there should I choose to have it displayed as well. We then open the video using open cv2's VideoCapture and break it down frame by frame and using open cv2's dnn.blobFromImage we get one image at a time and we construct a 4 dimensional tensor by doing mean subtraction, scaling and channel reordering and then we feed to the neural network for inference. The network gives us the class ids and the confidences. We then initialize lists to hold the class ids, the confidences and the bounding boxes for each image. We then loop through all outputs of the network and for whichever one we got more than 0.5 probability that it's true, we define a bounding box for it and append it to the bounding box list and we also append the confidence we got for it in the confidences list and its class id in the class ids list. We then use non maximum suppression to remove the extra bounding boxes and then loop through the bounding boxes list and draw them on each frame. We then use open cv2's VideoWriter to write the frames into an output called outpy also given with the rest of the code. The video replay after doing detection and tracking is slower but it gets the job done.
Below you can see 3 adjacent frames (not literally one after the other so you can see the cars move). The function to capture frames is in the code and you capture by pressing the button c and the frame is saved in a frame folder created if not already existing with the name framen where n is the number of the capture. It allows for multiple frames to be captured at different time intervals at the discretion of the user. You can see that as the cars are moving the bounding boxes remain on the cars and detect and track them properly. It has some trouble with the truck and the person when they are not clearly visible or are in the distance and it can't properly detect their features and it may end up either not labeling them or mislabeling them. But the model is consistent in tracking the cars. Newer

yolo models do not suffer from the aforementioned difficulties to the same extent.

2.

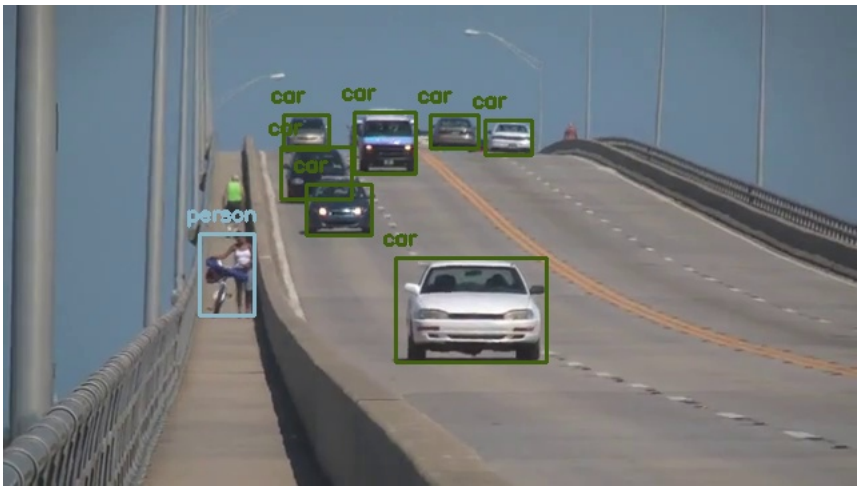

Figure 10: **Tracking Sequence - Frame 1**

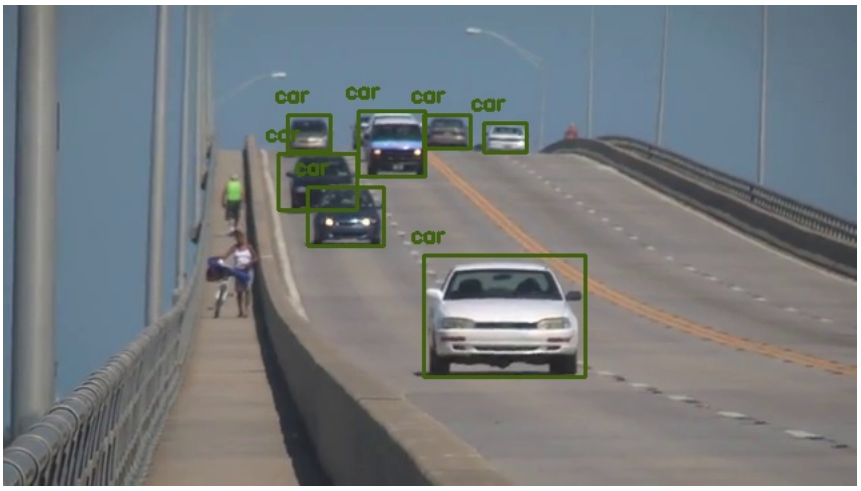

Figure 11: **Tracking Sequence - Frame 2**

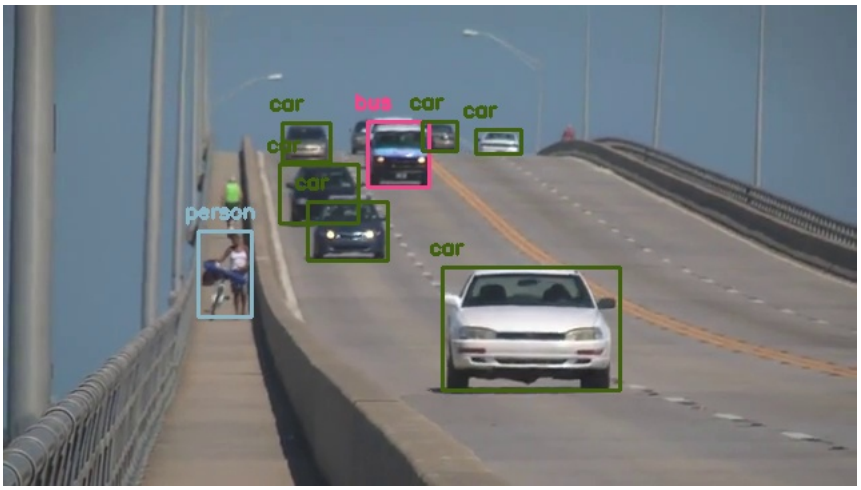

Figure 12: **Tracking Sequence - Frame 3**

Task 5. Skiptrace (4pt)

Sherlock needs a team to defeat Professor Moriarty. Irene Adler recommended 3 reliable associates and provided 3 pictures of their last known whereabouts. Sherlock just needs to know their identities to be able to track them down.

Sherlock has a database of surveillance photos around NYC. He knows that these three associates definitely appear in these surveillance photos once.

Could you use these 3 query pictures provided by Irene Adler to figure out the names of pictures that contain our persons of interest? After you **obtain the picture names**, **show these pictures** to us in your report, and comment on the possibility of them defeating Professor Moriarty!

Tip 1: This question can be time consuming and memory intensive. To give you some perspective of what to expect, I tested it on my laptop with 16GB of RAM and a Ryzen 9 5900HS CPU (roughly equivalent to a Intel Core i7-11370H or i7-11375H) and it took about 50 mins to finish the whole thing, so please start early.

Tip 2: Refrain from using `np.vstack()/np.hstack()/np.concatenate()` too often. Numpy array is designed in a way so that frequently resizing them will be very time/memory consuming. Consider other options in Python should such a need to concatenate arrays arise.

Answers:
We can indeed use those 3 query pictures to find the names of the pictures that contain our persons of interest. We can do that using VLAD (Vector of Locally Aggregated Descriptors). For the implementation for this question I referenced the project from TA Irving and this paper as well as ChatGPT. In order to run the code in an efficient manner I uploaded to NYU HPC and run it through a Jupyter Notebook using 128 GB of memory, 8 CPUs and 1 V100 GPU. The code uses SuperPoint for inference to get the descriptors and then performs VLAD. The paper first defined 3 helper functions, the read image function that reads an image from the path, converts it to grayscale and then to a tensor. The conversion to a tensor is being done by the convert to tensor helper function which uses torch and the get descriptor super point function reads an image, does inference to get the prediction and extracts the descriptor that is then transposed and converted to a numpy array. The superpoint implementation is straight from the Github report and the weights as well. I included the code for superpoint in the jupyter file since I couldn't get the python file to work. The extract features helper function loops through all the pictures and uses the previous helper function to extract their descriptors and append them into a list of descriptors which is then converted to a numpy array. This function is then used to extract the descriptors of the whole database. the descriptors are then clustered in 16 clusters with mini batch k means for more efficient memory utilization and to prevent the kernel from dying and we perform this only once since the query images are found identical in the search database. The VLAD function gets the descriptor of its input image and predicts its cluster and then gets its centroids and labels for all the clusters and the total number of clusters. It then loops through all the clusters, makes sure there is at least 1 descriptor in each cluster and sums all the differences between all the other clusters. It then performs square-root and L2 normalization. The program then loops through all the images and using superpoint gets their descriptor and using that descriptor, it feeds it into VLAD to get their corresponding VLAD feature and then saves that into a list and saves the name of the image in another list. The list is then converted to a ball tree data structure for more efficient searching. We then loop through all the query images, get their descriptor, then their VLAD feature and then we look in the ball tree for the entry with 0 distance from the VLAD feature of the query since the images are identical and the distance should be 0 for the match. We then display the match for each query. The matches are shown below along with their names.

Query 1 Match - 'dr5rsn1w5bcx-dr5rsn1tuppy-cds-22ecab6d4a71bfcf-20160901-1157-7368.jpg'
Query 2 Match - 'dr5rsn1tgm9f-dr5rsn1tggkn-cds-22ecab6d4a71bfcf-20160901-1157-7188.jpg'
Query 3 Match - 'dr5rsn0yet6m-dr5rsn0ympkm-cds-3670f40ef7987d5a-20161022-1109-395.jpg'

The 3 persons of interest would provide strong emotional support and misdirection to help fight Professor Moriarty.

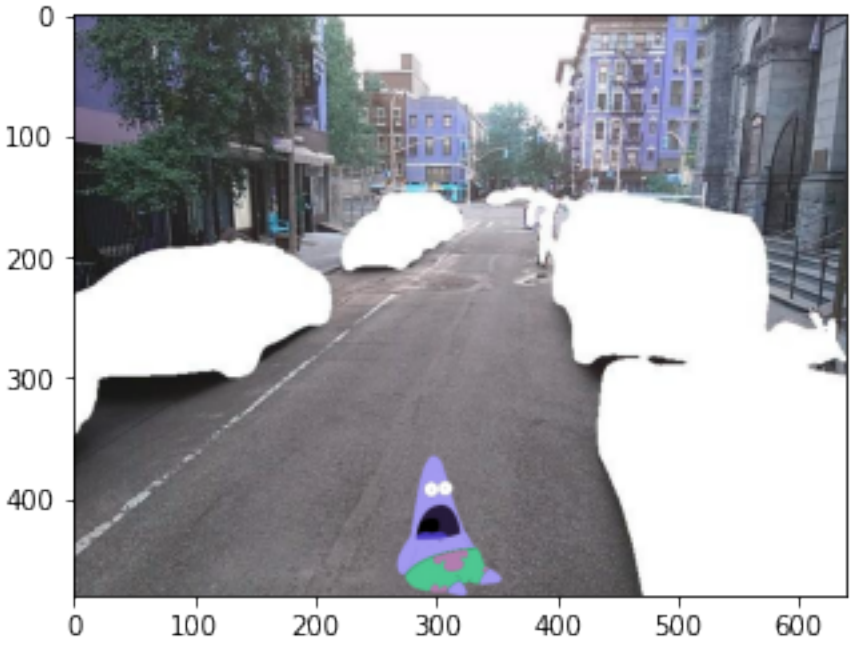

Figure 13: **Query 1 - Matched Image**

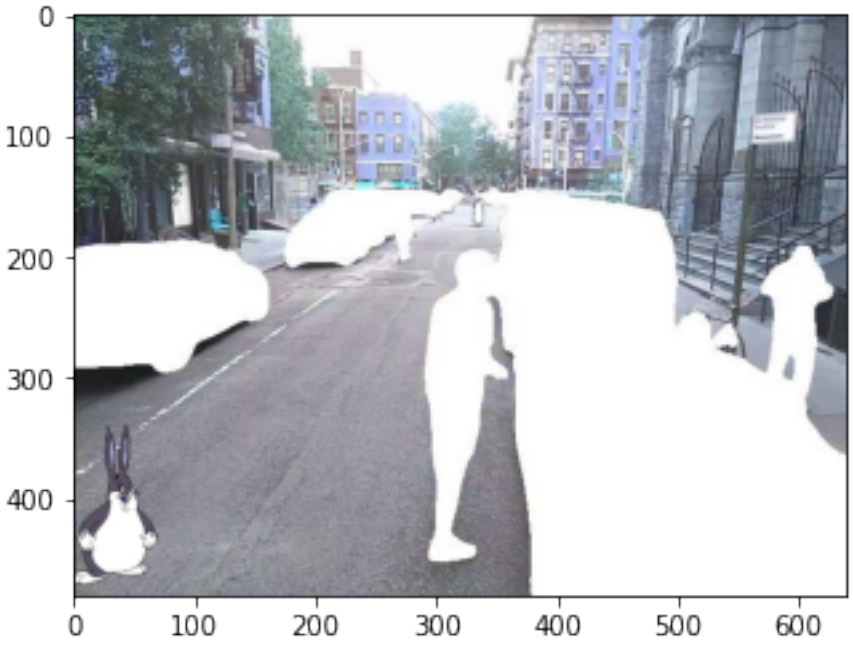

Figure 14: **Query 2 - Matched Image**

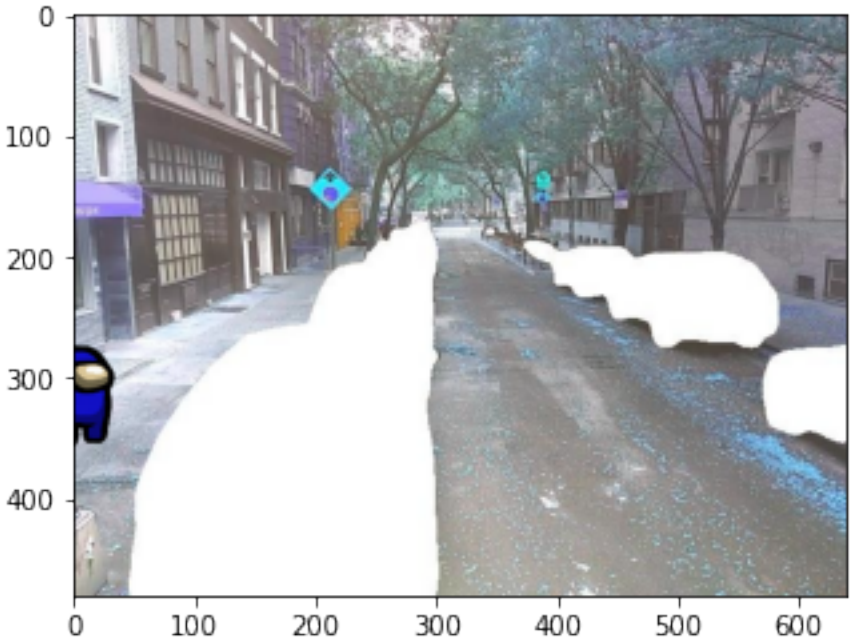

Figure 15: **Query 3 - Matched Image**