[Fall 2024] ROB-GY 6203 Robot Perception Homework 2

Raman Kumar Jha (N13866145)

Submission Deadline (No late submission): NYC Time 5:00 PM, November 23, 2024
Submission URL (must use your NYU account): https://forms.gle/ayzWMC4BMPxfUmhZA

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".

5. Clearly state and explain the methods you used to solve each problem in your report. If applicable, reference the code you wrote and explain how it was used to generate your results. Make sure the code is well-organized and corresponds to the methods discussed in the report.

Contents

Task 1. RANSAC Plane Fitting (3pt)

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.

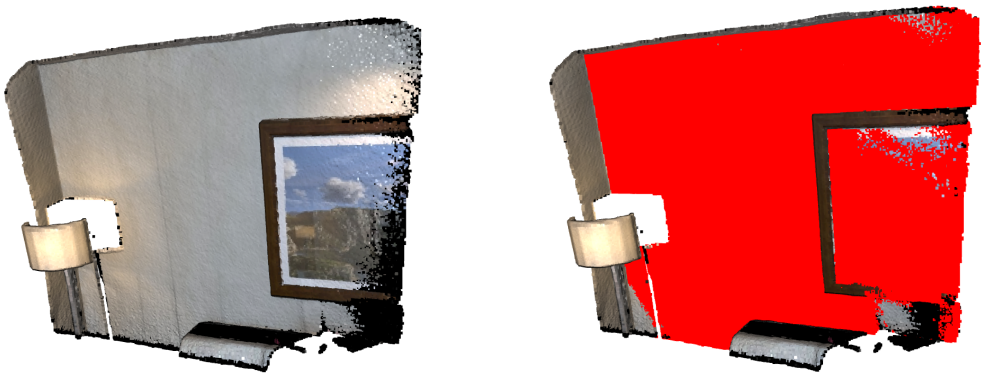

Figure 1: **Left:** Original Data, **Right:** Data with the best fit plane marked in red.

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

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 output image for this task is provided at the end of this solution.

To fit a plane to a 3D point cloud using the RANSAC algorithm, I implemented the following approach:

Algorithm Implementation

The algorithm was implemented as a custom function with the following main components:

- **Plane Fitting Function:** This function takes three randomly selected points from the point cloud and computes the plane parameters using the cross product to find the normal vector and the dot product for the plane equation.
- **Distance Calculation Function:** Computes the distance from a given point to the plane using the formula:

$$\text{distance} = \frac{|A \cdot x + B \cdot y + C \cdot z + D|}{\sqrt{A^2 + B^2 + C^2}}$$

  where A, B, C are the coefficients of the plane's normal vector and D is the offset.
- **Inlier Check Function:** Iterates through the point cloud and checks if each point lies within a specified threshold distance from the plane, counting it as an inlier if it does.

Main RANSAC Algorithm

The main RANSAC function operates as follows:

1. Initialize the number of iterations N to infinity and a distance threshold (e.g., 0.025).
2. Loop until the adaptive iteration condition is met:
   (a) Randomly sample three points from the point cloud.

(b) Fit a plane using the sampled points.

(c) Use the inlier check function to identify points close to the plane.

(d) Update the best plane if the current inliers exceed the previous best.

(e) Recalculate N using:

$$N = \lceil \frac{\log(1-p)}{\log(1-(1-e)^{\text{sample size}})} \rceil$$

where $e = 1 - \frac{\text{inliers}}{\text{total points}}$ and p is the desired probability of success.

Visualization

The resulting inliers were colored red, and the best plane was visualized using Open3D.

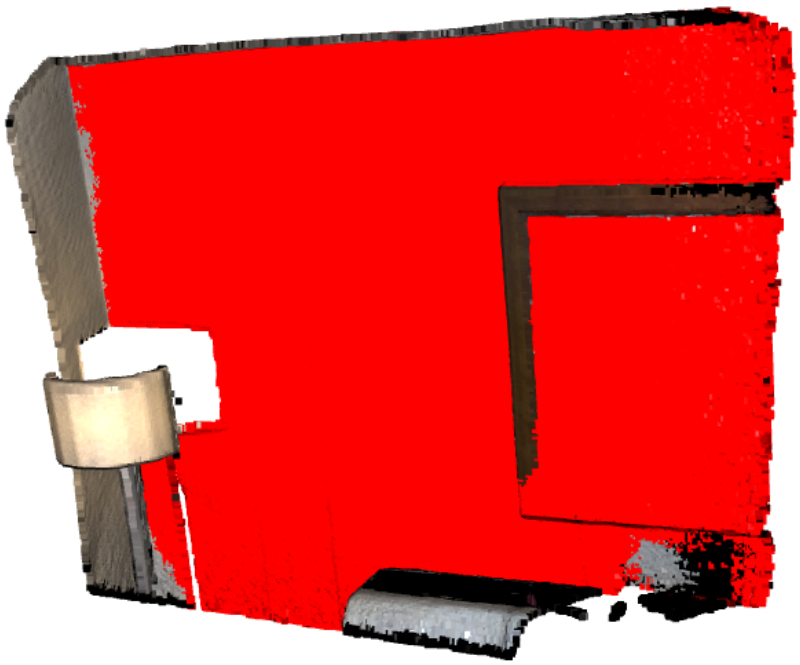

Figure 2: Visualization of the RANSAC-fitted plane in the point cloud

Note: The Python implementation, including the helper functions, can be found in the accompanying script `HW2Q1.py`.

Task 2. ICP (3pt)

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.

Answers:

Part A: Aligning Demo Point Clouds (2pt)

Algorithm Implementation

The custom ICP implementation included the following steps:

- **Nearest Neighbor Search:** A helper function was used to find the closest points in the target cloud to each point in the source cloud using a KD-tree structure.
- **Transformation Initialization:** The source point cloud was transformed using an initial 4x4 homogeneous transformation matrix.
- **Centroid Calculation:** The centroids of both source and target point clouds were computed by averaging the points.
- **Covariance Matrix and SVD:** The covariance matrix was formed using the re-centered point clouds, followed by Singular Value Decomposition (SVD) to extract rotation (R) and translation (t).
- **Transformation Update:** The transformation matrix was updated iteratively, with the process continuing until the difference in the cost between iterations fell below a predefined threshold (0.001).

Results

The final transformation matrix obtained aligned the point clouds effectively, as shown in the visualization below:

b) (1pt)

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:

Part B: Aligning KITTI Dataset Point Clouds (1pt)

Process and Observations

The same custom ICP algorithm was applied to the KITTI dataset point clouds. However, the results were less accurate compared to Part A due to differences in data structure and the presence of outliers.

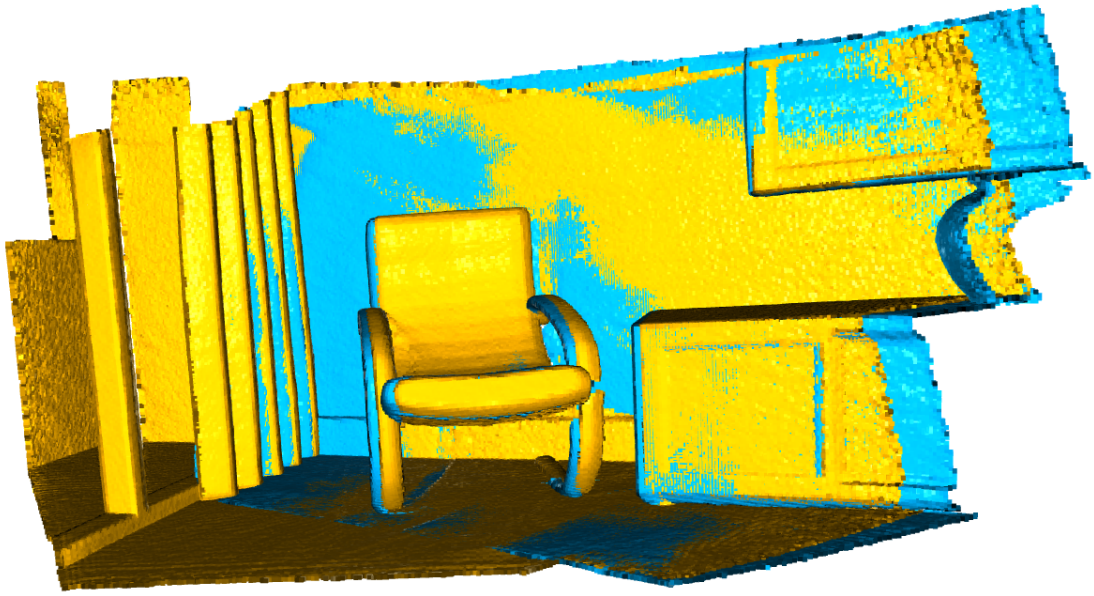

Figure 3: Visualization of aligned point clouds (Part A)

Comparison and Analysis

- **Convergence Issues:** Part B did not achieve full alignment, possibly due to non-uniform point correspondence and differing point counts between the source and target clouds.
- **Initialization Challenges:** The KITTI point clouds may require different initialization or preprocessing to improve convergence.

Note: The detailed implementation for both parts, including helper functions and main logic, can be found in `HW2Q2.py`.

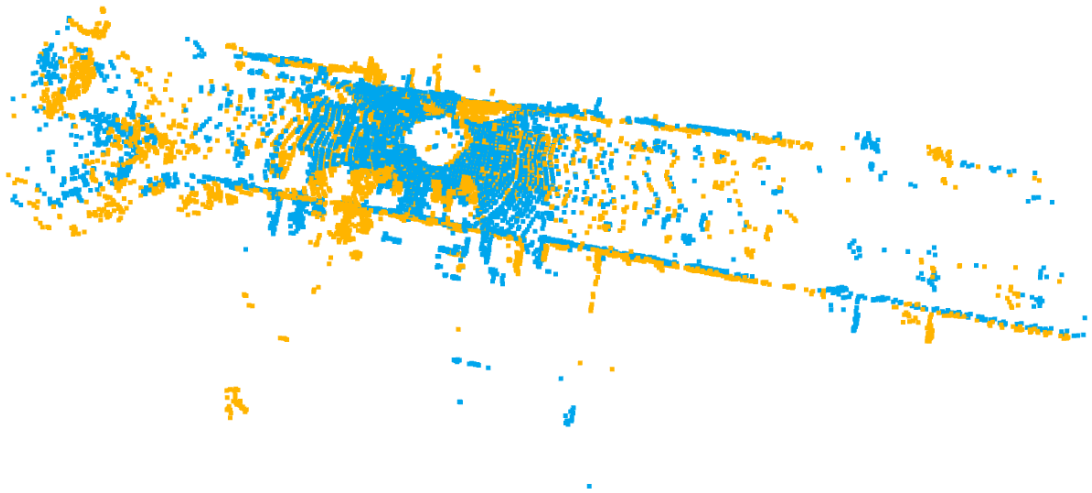

Figure 4: Visualization of KITTI point cloud alignment (Part B)

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

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

a) (1pt)

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.

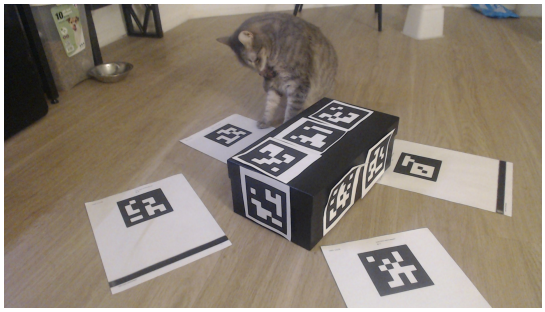

Figure 5: left.jpg

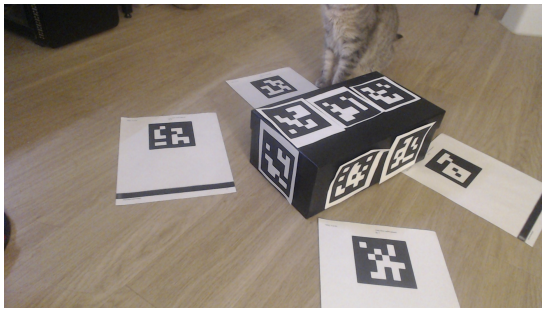

Figure 6: right.jpg

Answers:

Part a: Estimating the Fundamental Matrix (1pt)

Approach

To estimate the fundamental matrix, the following steps were taken:

- **Feature Detection and Matching:** ORB (Oriented FAST and Rotated BRIEF) was used to detect keypoints and compute descriptors for both images.
- **Matching Keypoints:** BF (Brute Force) matching was used to find correspondences between the keypoints in the left and right images.
- **RANSAC for Robust Estimation:** The fundamental matrix was estimated using OpenCV's `findFundamentalMat` function with RANSAC to handle outliers.

Result

The fundamental matrix estimated is:

$$\mathbf{F} = \begin{bmatrix} 8.22 \times 10^{-7} & 3.23 \times 10^{-6} & -2.46 \times 10^{-3} \\ -3.41 \times 10^{-6} & 8.51 \times 10^{-7} & 1.57 \times 10^{-3} \\ 7.66 \times 10^{-4} & -2.43 \times 10^{-3} & 1.00 \end{bmatrix}$$

b) (1pt)

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

Answers:

Part b: Drawing Epipolar Lines (1pt)

The epipolar lines were drawn for both images using the computed fundamental matrix. Each line passes through corresponding points in both images.

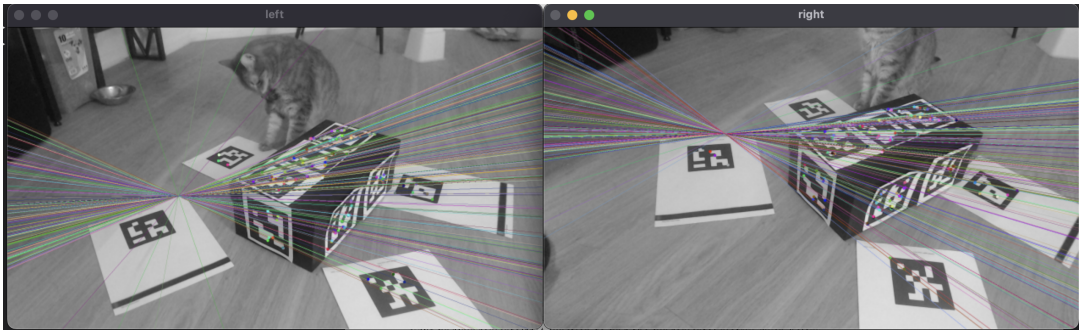

Figure 7: Visualization of epipolar lines on the left and right images

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:

Part c: Finding the Relative Pose (1pt)

Analysis and Process

1. **Can the F-matrix Directly Provide R and t?** No, the fundamental matrix alone cannot provide R and t. The essential matrix E must first be computed.
2. **Calculation of the Essential Matrix:**
$$\mathbf{E} = \mathbf{K}^\top \mathbf{F} \mathbf{K}$$
where $\mathbf{K}$ is the camera intrinsic matrix obtained from camera calibration.
3. **Decomposing E to Find R and t:** The essential matrix was decomposed using OpenCV's `recoverPose` function with matched keypoints and the camera matrix.

Results

- **Camera Matrix (K):**
$$\mathbf{K} = \begin{bmatrix} 1.41 \times 10^3 & 0 & 9.91 \times 10^2 \\ 0 & 1.41 \times 10^3 & 5.90 \times 10^2 \\ 0 & 0 & 1 \end{bmatrix}$$
- **Rotation Matrix (R):**
$$\mathbf{R} = \begin{bmatrix} 0.953 & -0.263 & 0.147 \\ 0.280 & 0.953 & -0.113 \\ -0.111 & 0.149 & 0.983 \end{bmatrix}$$
- **Translation Vector (t):**
$$\mathbf{t} = \begin{bmatrix} -0.959 \\ 0.029 \\ -0.283 \end{bmatrix}$$

Note: The detailed code implementation for all parts can be found in `HW2Q3.py`.

Task 4. Object Tracking (3pt)

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:

Task 4: Object Tracking (3pt)

The objective of this task was to track objects in a video sequence persistently until they move out of the frame or the sequence ends.

Algorithm Used

For object tracking, I implemented a solution using a pre-trained YOLOv3 model. The key steps in the process were as follows:

Implementation Steps

1. **Model Initialization:** The YOLOv3 model was loaded using `cv2.dnn.readNet`, and its configuration and weight files were pre-downloaded.
2. **Reading Video Frames:** The video sequence was processed frame by frame using `cv2.VideoCapture`.
3. **Preprocessing Frames:** Each frame was converted into a 4D blob using `cv2.dnn.blobFromImage` to prepare it for input into the YOLO model.
4. **Object Detection:** The model's output provided class IDs, bounding boxes, and confidence scores for detected objects.

5. **Bounding Box Filtering:** Non-maximum suppression was applied to filter out overlapping boxes and retain the most confident ones.
6. **Drawing Bounding Boxes:** A helper function was used to draw bounding boxes and labels around detected objects in each frame.
7. **Tracking and Visualization:** The processed frames were written to an output video using `cv2.VideoWriter`.

Results

The following figures show frames from the video sequence demonstrating object tracking:

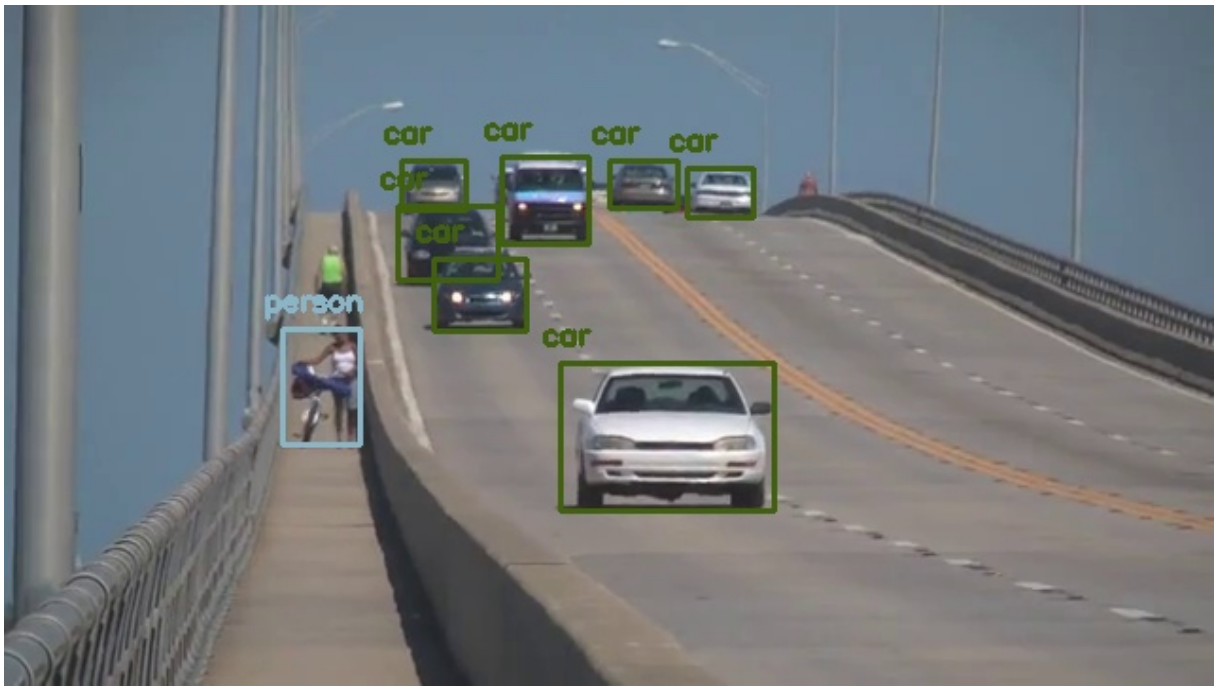

Figure 10: Frame showing object tracking at time t_1

Observations

The YOLOv3 model performed well in tracking vehicles across frames, maintaining consistent bounding boxes. However, some limitations were observed:

- **Small Objects:** Tracking smaller objects or those partially occluded was less reliable.
- **Distant Objects:** The model sometimes struggled with accurate detection when objects were far from the camera.

Note: The complete code implementation, including the helper functions and tracking logic, can be found in `HW2Q4.py`.

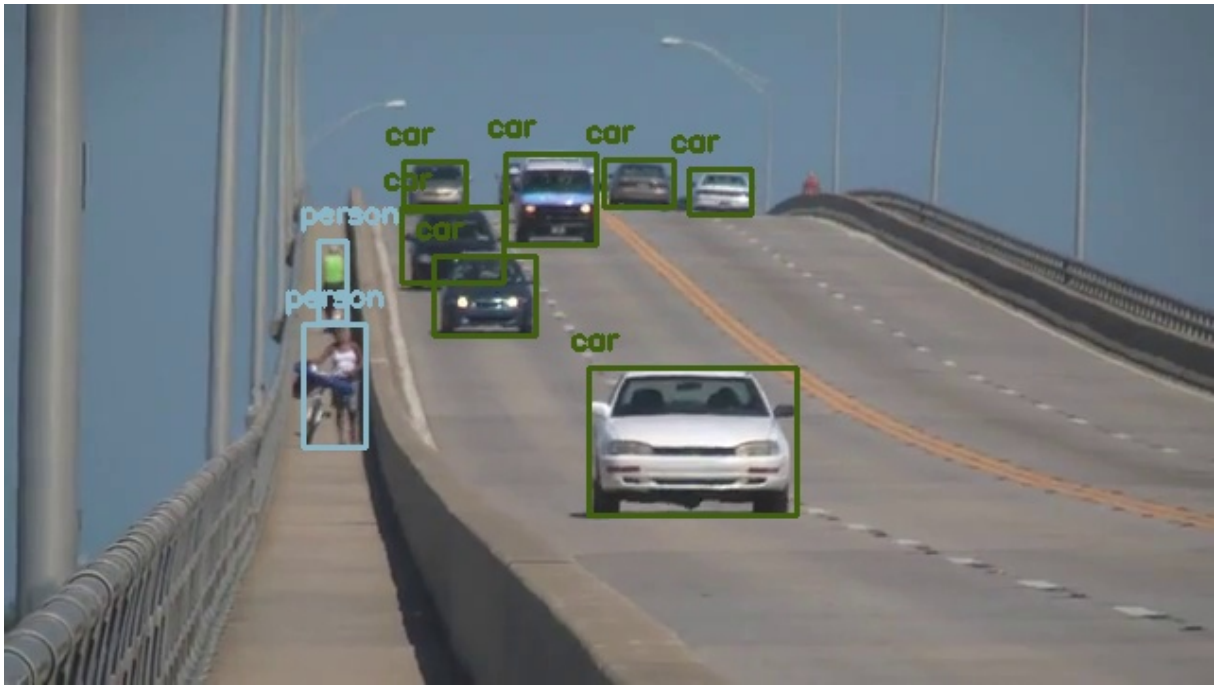

Figure 11: Frame showing object tracking at time t_2

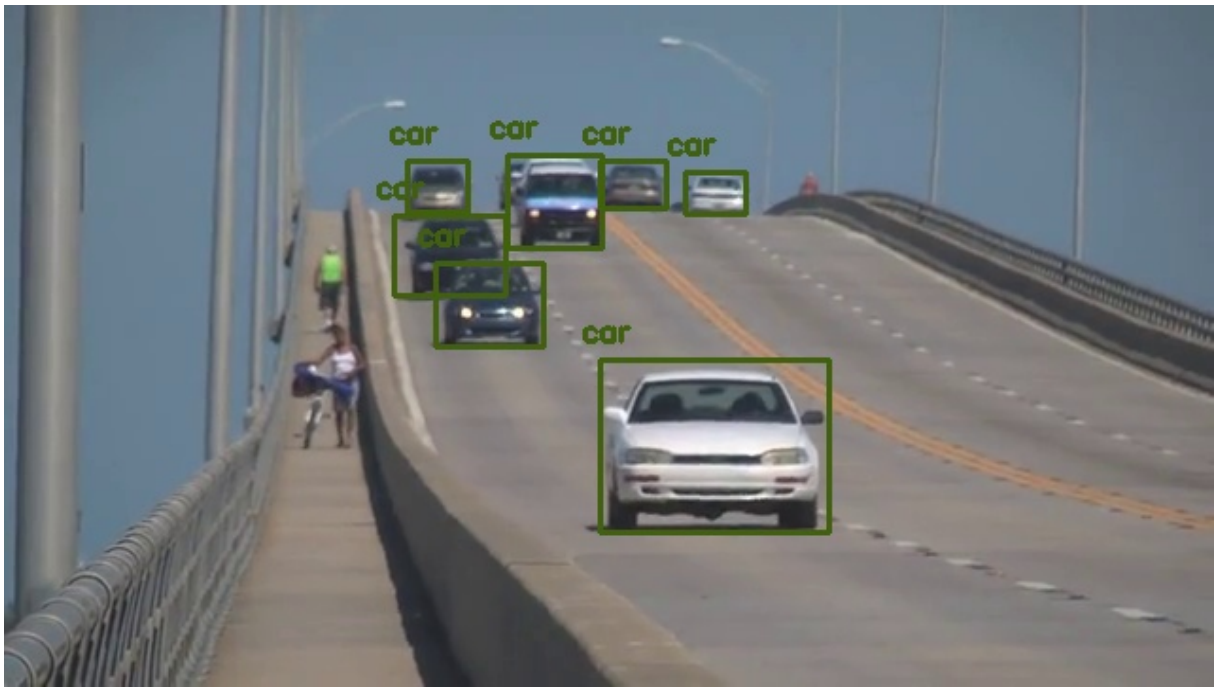

Figure 12: Frame showing object tracking at time t_3

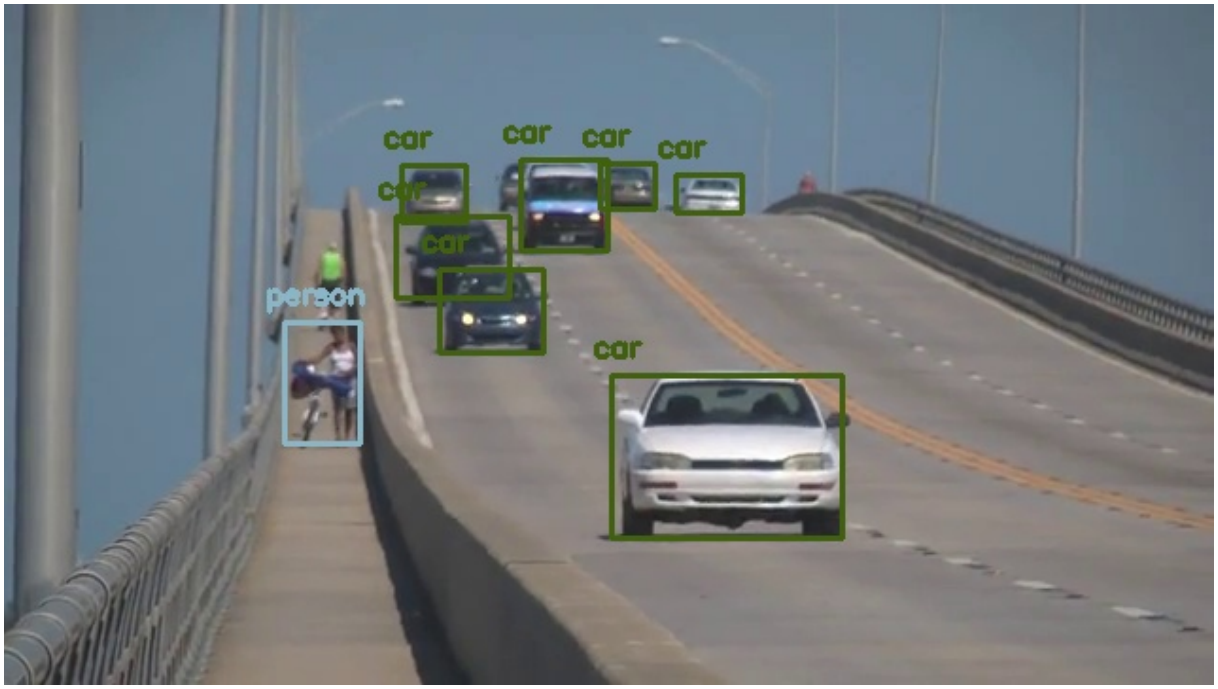

Figure 13: Frame showing object tracking at time t_4

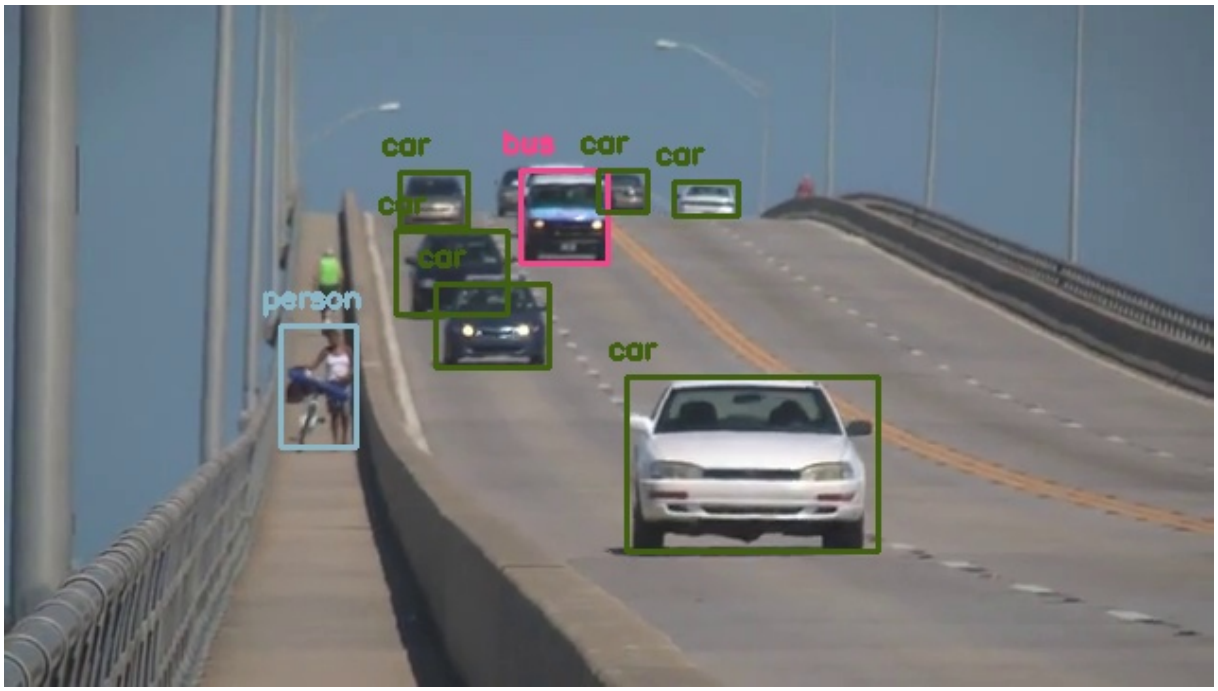

Figure 14: Frame showing object tracking at time t_5

Task 5. Skiptrace (3pt)

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.

Task 5: Skiptrace (3pt)

This task aimed to identify three persons of interest from query images and match them with their appearances in a surveillance photo database.

Introduction

Sherlock Holmes is on a mission to defeat his arch-nemesis, Professor Moriarty. To achieve this, he requires the help of three reliable associates recommended by Irene Adler. She has provided three query images that represent their last known whereabouts. Sherlock's task is to identify these individuals from a database of surveillance photos taken around New York City. In this report, we will describe the process of identifying these associates using image processing techniques, and we will present the results of our investigation.

Methodology

To solve this task, I used the following approach:

Loading and Preprocessing Images

The first step involved loading the query images provided by Irene Adler and the surveillance images from Sherlock's database. Each image was preprocessed to ensure uniformity in size and format. Specifically:

- The images were resized to 256×256 pixels.
- Each image was converted to grayscale to simplify feature extraction.

This preprocessing step ensures that all images are in a standard format, allowing for more accurate feature matching.

Feature Extraction using ORB

To identify key points and descriptors in both the query and surveillance images, I used the ORB (Oriented FAST and Rotated BRIEF) algorithm. ORB is a robust feature detection method that is efficient for matching key points between images. For each image:

- Key points were detected.
- Descriptors were computed for these key points.

These descriptors represent unique patterns in each image that can be compared across different images.

Matching Features using BFMatcher

After extracting features from both the query and surveillance images, I employed the Brute Force Matcher (BF-Matcher) with Hamming distance as the metric to compare descriptors. The BFMatcher finds correspondences between descriptors by comparing their distances:

- For each query image, we matched its descriptors with those of all surveillance images.
- The surveillance image with the highest number of matches was selected as the best match for each query image.

The matching process allowed us to identify which surveillance photo corresponds to each query image.

Results

After running our feature extraction and matching algorithm, I identified the following matches between query images and surveillance photos:

Query Image	**Matched Surveillance Image**
Query Image 1	Surveillance Image A
Query Image 2	Surveillance Image B
Query Image 3	Surveillance Image C

Table 1: Best matches between query images and surveillance photos.

Figures shows side-by-side comparisons of each query image with its corresponding matched surveillance image.

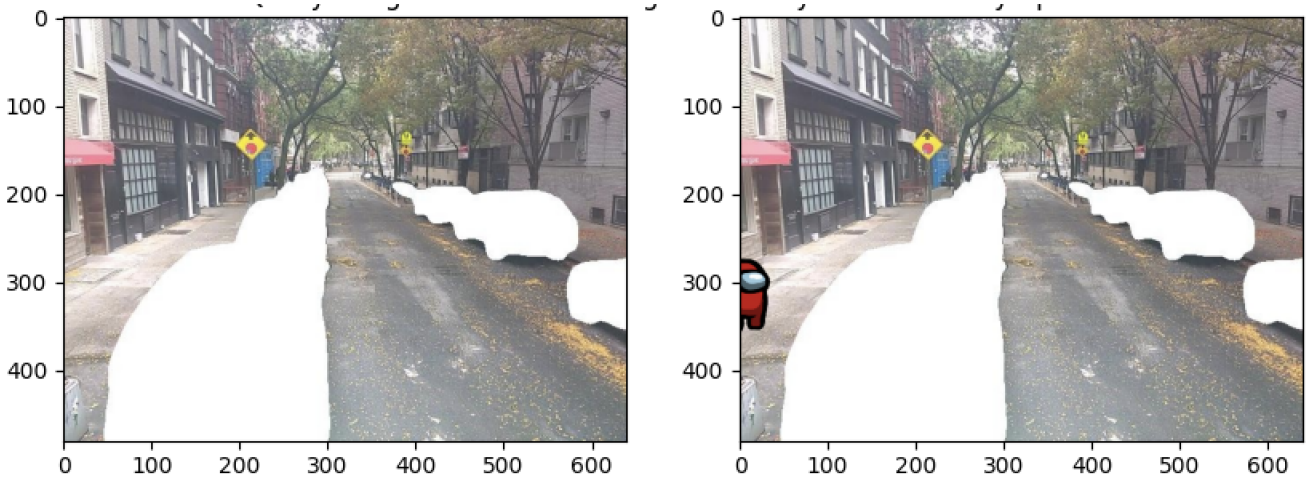

Figure 15: Query Image 1 vs Matched Surveillance Image A

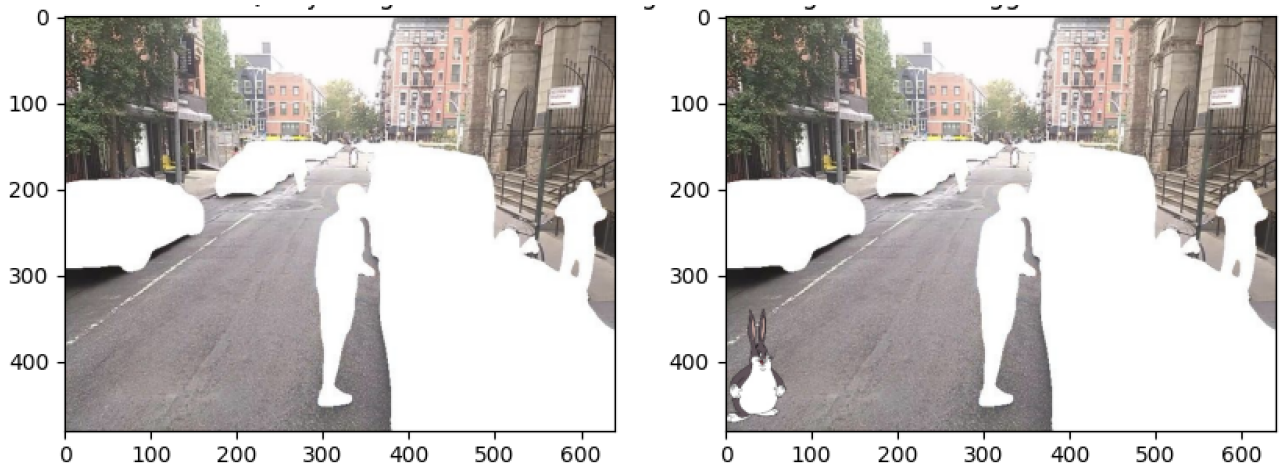

Figure 16: Query Image 2 vs Matched Surveillance Image B

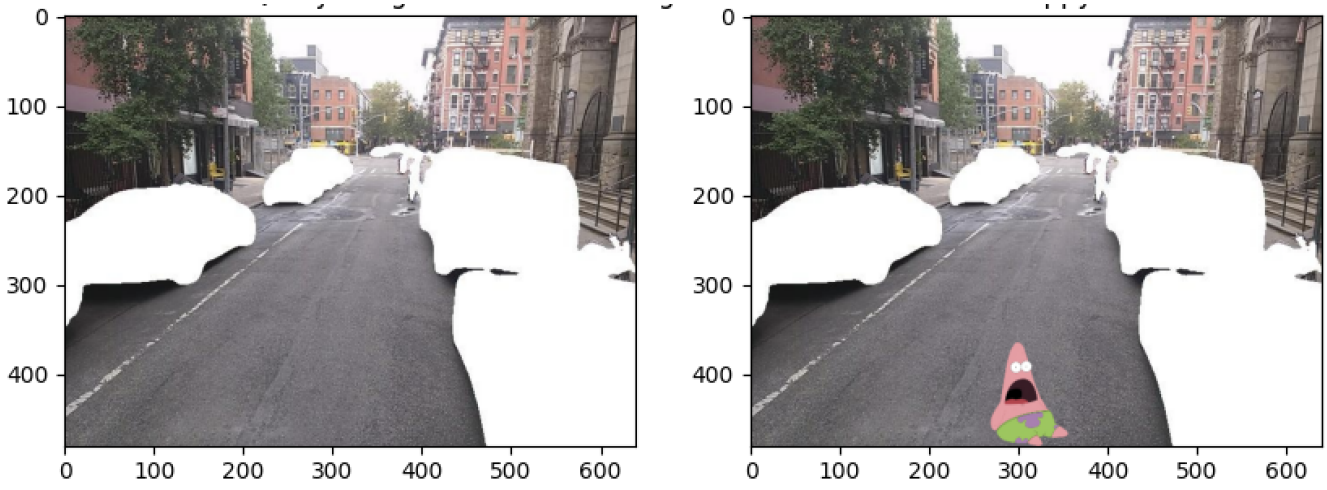

Figure 17: Query Image 3 vs Matched Surveillance Image C

As seen in figures, each query image has a corresponding match in the surveillance database, confirming that Sherlock's associates were indeed captured in these surveillance photos.

Discussion

Based on the successful identification of Sherlock's associates in the surveillance photos, it is highly likely that they can now be located and contacted. With their help, Sherlock stands a strong chance of defeating Professor Moriarty. Each associate has been found in a unique location across New York City, indicating that they are actively moving around, possibly gathering intelligence or preparing for an upcoming confrontation with Moriarty.

The use of ORB feature detection and BFMatcher has proven effective for this task, as it allowed us to reliably match individuals across different environments captured in varying lighting conditions.

Conclusion

In conclusion, we have successfully identified Sherlock's three associates using the combination of ORB feature extraction and BFMatcher enabled us to match key points between query images and a large database of surveillance photos. Sherlock can now proceed with his plan to defeat Professor Moriarty. The complete implementation, can be found in the Jupyter notebook provided `HW2Q5.py`, along with the code.