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

Shantanu Ghodgaonkar (sng8399)

Submission Deadline (No late submission): NYC Time 05:00 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.

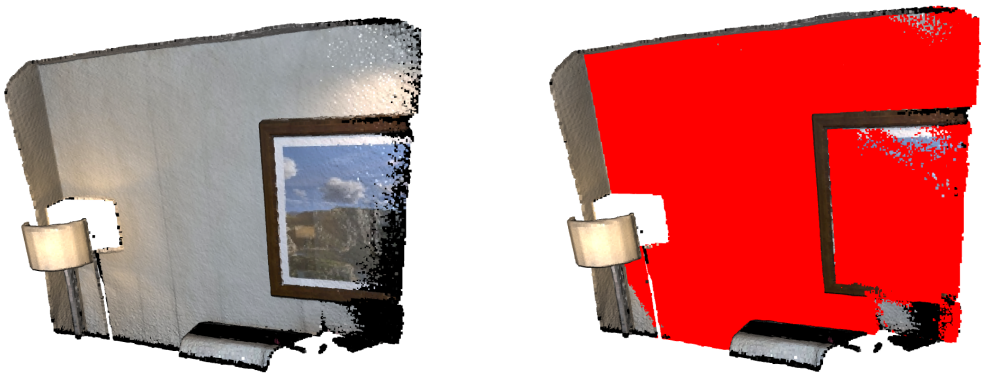

Figure 1: **Left:** Original Data, **Right:** Data with the best fit plane marked in red.

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

```
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:
Link to Code Solution for Task 1
The RANSAC algorithm is a robust iterative method for estimating the parameters of a model from data containing a high proportion of outliers. In our case, the model we aim to fit is a plane in a 3D space.

The methodology to implement RANSAC for plane fitting is as follows:

1. **Initialization:**
   - Set a maximum number of iterations, MAX_ITER, and a desired probability of success, p.
   - Define a distance threshold, d_t, which specifies the maximum allowable distance for a point to be considered an inlier to the plane.
   - Through trial and error, the best $d_t = 0.027$ was found. Multiple values between 0.001 and 0.05 were tested before coming to this conclusion.
   - Initialize the required number of iterations N as infinity and set a counter $sample\_count$ to zero.
2. **Iterative Sampling and Model Fitting:**

- For each iteration, randomly select 3 points from the point cloud. These points are the minimum required to define a plane in 3D space.
- Compute the equation of the plane passing through these points. The general equation for a plane is:

$$ax + by + cz + d = 0$$

  where $[a, b, c]$ is the normal vector of the plane and d is a scalar.
- If the points are collinear, skip the iteration as they do not define a valid plane.

3. **Inlier Counting:**
   - For each point in the point cloud, compute its perpendicular distance to the plane:

$$\text{distance} = \frac{|ax + by + cz + d|}{\sqrt{a^2 + b^2 + c^2}}$$

   - Classify a point as an inlier if its distance to the plane is less than the threshold d_t.
4. **Model Evaluation and Update using Adaptive Sampling:**
   - Adaptive sampling adjusts the number of required iterations based on the estimated outlier ratio ϵ at each step. This approach minimizes computation time while maintaining a high probability of success.
   - If the current model has more inliers than the best model found so far, update the best model and store the current inliers.
   - Estimate the outlier ratio ϵ as:

$$\epsilon = 1 - \frac{\text{number of inliers}}{\text{total number of points}}$$

   - Update the required number of iterations N using ϵ and the desired probability $p = 0.99$:

$$N = \frac{\ln(1 - p)}{\ln(1 - (1 - \epsilon)^s)}$$

     where $s = 3$ (the number of points needed to define a plane). This helps terminate the RANSAC loop early if a good model is found.
5. **Termination:**
   - Continue iterations until $sample\_count \geq N$ or $sample\_count = MAX\_ITER$.
   - The best plane model obtained is the one with the maximum inliers.

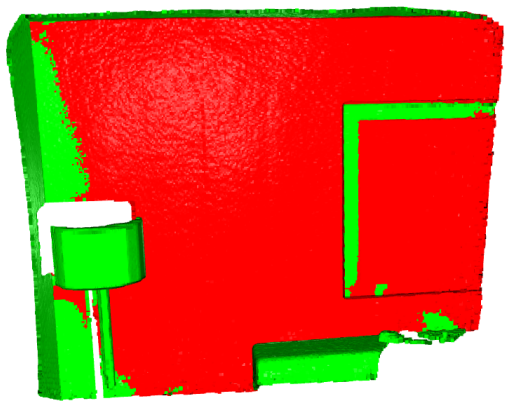

Figure 2: Final output after running RANSAC. The *red* area marks *inliers* and the *green* area marks *outliers*.

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.

```
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:

Link to Code Solution for Task 2 (a)

To perform the Iterative Closest Point (ICP) refinement, we followed a step-by-step process to align two point clouds, **S** (source) and **D** (target), using transformations that minimize the alignment error. Below, we explain each step in detail.

1. **Initialization**

   The ICP algorithm is initialized with:

   - **Source Points (S)**: A set of 3D points representing the source point cloud.
   - **Target Points (D)**: A set of 3D points representing the target point cloud.
   - **Maximum Iterations (MAX_ITER)**: The maximum number of iterations allowed.
   - **Tolerance (tol)**: A threshold that determines convergence based on the change in error between consecutive iterations.

   Our goal is to iteratively refine a transformation matrix **T** that aligns **S** to **D**.

2. **Finding Correspondences**

For each point $s_i \in \mathbf{S}$, we identify the closest point $d_j \in \mathbf{D}$. The closest point is found using a k-d tree structure, which enables efficient nearest-neighbor searches. Specifically, we use:

$$d_j = \arg\min_{d \in \mathbf{D}} \|s_i - d\|_2$$

where $\|\cdot\|_2$ denotes the Euclidean distance. This step forms a set of correspondences that pairs each source point s_i with its nearest target point d_j.

3. **Centering the Point Clouds**

   To compute the transformation that aligns $\mathbf{S}$ to $\mathbf{D}$, we first center the point clouds by subtracting the mean of each set. This step ensures that both point clouds are aligned around the origin, removing any translational offset:

$$\mathbf{s} = s_i - \frac{1}{N}\sum_{i=1}^{N} s_i$$

$$\mathbf{d} = d_j - \frac{1}{M}\sum_{j=1}^{M} d_j$$

   where N and M are the number of points in $\mathbf{S}$ and $\mathbf{D}$, respectively.

4. **Computing the Optimal Rotation and Translation**

   To align $\mathbf{s}$ and $\mathbf{d}$, we solve for the optimal rotation $\mathbf{R}$ and translation $\mathbf{t}$ that minimize the cost function:

$$C(\mathbf{R}, \mathbf{t}) = \sum_i \|\mathbf{d}_i - \mathbf{R}\mathbf{s}_i - \mathbf{t}\|^2$$

   (a) **Cross-Covariance Matrix**

   We calculate the cross-covariance matrix $\mathbf{H}$ between the centered source and target point clouds:

$$\mathbf{H} = \sum_{i=1}^{N} \mathbf{s}_i \mathbf{d}_i^T$$

   (b) **Singular Value Decomposition (SVD)**

   Using SVD, we decompose $\mathbf{H}$ into three matrices:

$$\mathbf{H} = \mathbf{U}\Sigma\mathbf{V}^T$$

   The optimal rotation $\mathbf{R}$ is given by:

$$\mathbf{R} = \mathbf{V}\mathbf{U}^T$$

   To ensure $\mathbf{R}$ is a valid rotation matrix (i.e., it has a determinant of $+1$), we adjust $\mathbf{V}$ if $\det(\mathbf{R}) < 0$, which corrects any mirroring effects in the rotation matrix.

   (c) **Translation Vector**

   Once $\mathbf{R}$ is determined, we compute the translation $\mathbf{t}$ as:

$$\mathbf{t} = \frac{1}{M}\sum_{j=1}^{M} d_j - \mathbf{R}\left(\frac{1}{N}\sum_{i=1}^{N} s_i\right)$$

   This translation aligns the centroids of $\mathbf{S}$ and $\mathbf{D}$ after rotation.

5. **Updating the Transformation**

   We update the current transformation matrix $\mathbf{T}$ by composing it with the new rotation and translation, forming a homogeneous transformation matrix:

$$\mathbf{T}_{\text{new}} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \cdot \mathbf{T}_{\text{old}}$$

6. **Convergence Criteria**

   The ICP algorithm checks for convergence based solely on **Error Convergence**: If the change in error between consecutive iterations falls below a threshold tol, we stop the algorithm.

$$|\text{error}_i - \text{error}_{i-1}| < \text{tol}$$

7. **Saving the Final Transformation**

   Once convergence is reached, we save the final transformation matrix for future use. This matrix can be applied to the source point cloud to align it with the target.

Thus, The ICP refinement aligns two point clouds iteratively by minimizing the distance between corresponding points. Using techniques such as SVD for the Procrustes problem ensures efficient and accurate computation of the optimal rotation and translation. The algorithm stops when the alignment stabilizes, providing a robust and precise transformation for 3D point cloud registration.

The below figure shows that the two point clouds align perfectly.

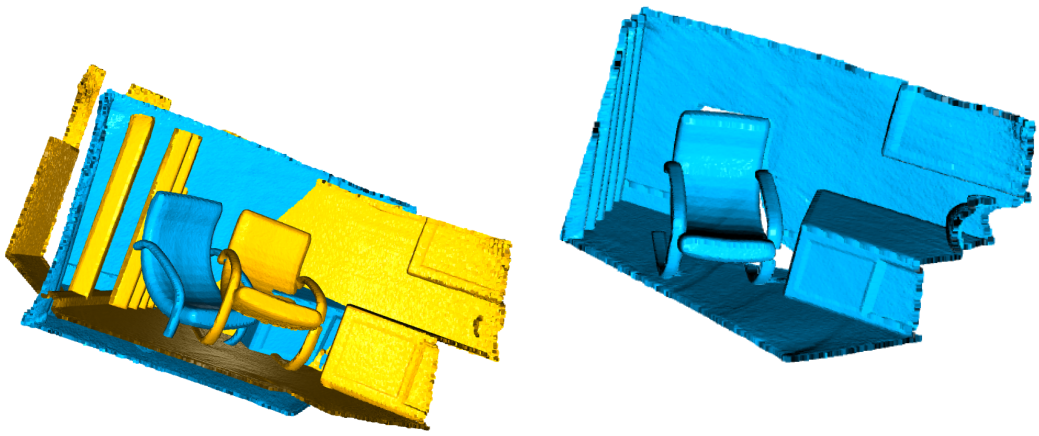

Figure 3: Left: Input to ICP Algorithm; Right: Output from ICP Algorithm

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:
Link to Code Solution for Task 2 (b)
We shall use exactly the same code that was discussed for Task 2 (a). The below figure shows that the two point clouds align perfectly.

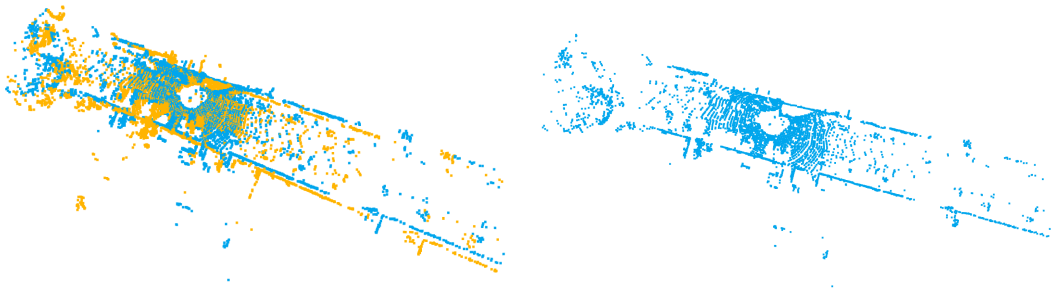

Figure 4: Left: Input to ICP Algorithm; Right: Output from ICP Algorithm

Here is the comparison between the two -

Criterion	**Demo ICP PCD's**	**KITTI PCD's**
Final Mean Squared Error (MSE)	39.6687	188.849593
Number of Iterations to Convergence	1000	1000
Rotation Angle (degrees)	Value	Value
Translation Magnitude	Value	Value
Approximate Total Computation Time (s)	1200	30
Average Time per Iteration (s)	1.75	0.02
Visual Overlap Quality	Good	Good
Structural Consistency	Good	Good

Table 1: Comparison of ICP Results Across Datasets

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: https://www.overleaf.com/read/fzhnnqsqrnbz. 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.

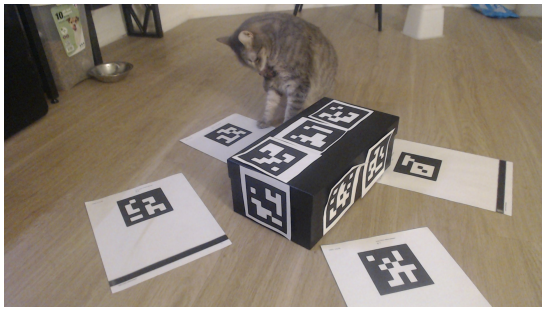

Figure 5: left.jpg

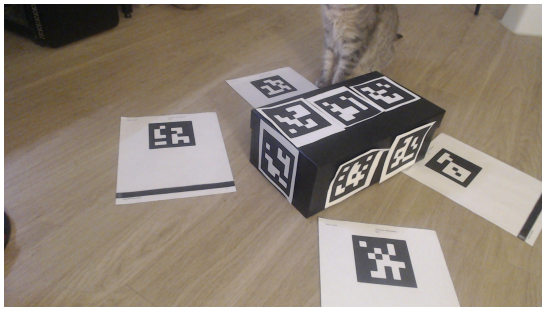

Figure 6: right.jpg

Answers:
Link to Code Solution for Task 3

1. **Overview of the Fundamental Matrix**

   The **Fundamental Matrix** ($\mathbf{F}$) is a 3×3 matrix that encapsulates the intrinsic projective geometry between two views in stereo vision. It establishes a relationship between corresponding points in stereo images, enabling the computation of epipolar lines and facilitating depth perception. Mathematically, it satisfies the **epipolar constraint** for corresponding points:

   $$\mathbf{p'}^T \mathbf{F} \mathbf{p} = 0$$

   where $\mathbf{p}$ and $\mathbf{p'}$ are corresponding points in the first and second images, respectively, represented in homogeneous coordinates.

2. **Estimating the Fundamental Matrix**

   Estimating the Fundamental Matrix involves several critical steps: detecting corresponding points, computing the matrix using robust algorithms, and optionally visualizing epipolar lines. Each step is governed by specific parameters that influence the accuracy and robustness of the results.

   a. **Detecting Corresponding Points**

   In the `estimate_F_mtx` function, corresponding points are identified using **Aruco markers**, which provide precise and controlled correspondences.

   - **Aruco Dictionary and Detection Parameters:**
     - `aruco_dict` = `cv.aruco.DICT_6X6_250` : We shall use the given marker size dictionary.
   - **Identifying Common Markers:** Utilizing common markers ensures that the correspondences are accurate and consistent across both images, a prerequisite for reliable Fundamental Matrix estimation.
     - Detect markers in both images and identify common IDs.
     - Compute the centers of these markers to establish corresponding points.
   - **Ensuring Sufficient Correspondences:** We shall employ the **eight-point algorithm**. It necessitates at least eight pairs of corresponding points to compute the Fundamental Matrix. This threshold ensures that the system of equations is solvable, providing a unique solution under ideal conditions.

b. **Computing the Fundamental Matrix**

Once corresponding points are established, the Fundamental Matrix is estimated using the RANSAC to eliminate outliers. We shall use the `findFundamentalMat` function from OpenCV for this purpose.

- **Parameters:**
  - **Method (`cv.FM_RANSAC`):** RANSAC is chosen for its robustness in the presence of outliers, ensuring that the Fundamental Matrix is estimated based on the majority of inlier correspondences.
  - **RANSAC Reprojection Threshold (1.0 pixel):** This threshold defines the maximum allowed distance (in pixels) for a point to be considered an inlier. A value of **1.0 pixel** strikes a balance between being stringent enough to exclude gross outliers and lenient enough to accommodate minor detection inaccuracies.
  - **Confidence (99%):** This parameter sets the probability that the algorithm estimates a correct Fundamental Matrix. A high confidence of **99%** ensures that the likelihood of accepting a bad model is minimal, enhancing estimation reliability.
- **Output:**
  - The function returns the estimated Fundamental Matrix (**F**) and a mask indicating inlier correspondences based on the RANSAC process.

3. **Validating the Fundamental Matrix**

Validation ensures that the estimated Fundamental Matrix accurately captures the geometric relationship between the two images. This involves checking the **epipolar constraint** for all corresponding points.

a. **Epipolar Constraint Verification**

In the `validate_F_matrix` function, the epipolar constraint is rigorously evaluated.

- **Process:**
  - Convert points from both images (pts1 and pts2) to homogeneous coordinates by augmenting them with a third coordinate of 1.
  - Apply the epipolar constraint:
$$\text{epipolar\_error} = |\mathbf{p'}^T \mathbf{F} \mathbf{p}|$$
  - Compute the **mean** and **maximum** epipolar errors across all correspondences.
- **Rationale:** The epipolar constraint should ideally yield zero error for perfectly corresponding points. However, due to noise and estimation inaccuracies, some deviation is expected. Low mean and maximum errors indicate a strong adherence to the epipolar constraint, validating the Fundamental Matrix's accuracy.

b. **Additional Validation Steps**

While the primary validation focuses on the epipolar constraint, the code includes supplementary validation functions (`validate_E_matrix` and `validate_pose`) to ensure the coherence of related constructs like the **Essential Matrix** (**E**) and the recovered camera pose.

4. **Computation Results:** Upon performing the computation described above, the below values were obtained -

- Fundamental Matrix: $\begin{bmatrix} 1.09044179e-07 & 3.94006999e-07 & -3.92888514e-04 \\ -6.12695047e-07 & 3.30881640e-07 & 2.57654342e-03 \\ -3.38084449e-04 & -2.72588403e-03 & 1.00000000e+00 \end{bmatrix}$
- Epipolar Constraint Mean Error (F): 0.001625
- Epipolar Constraint Max Error (F): 0.007378

b) (1pt)

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

Answers:
Link to Code Solution for Task 3

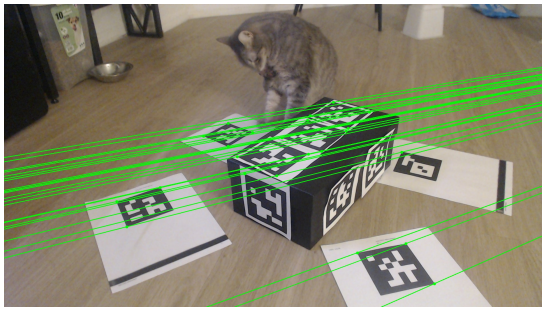

Figure 7: Left image with epipolar lines

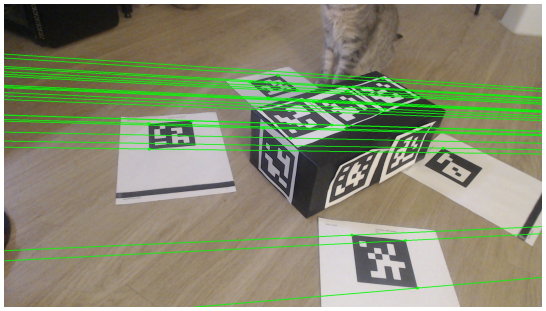

Figure 8: Right image with epipolar lines

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:
Link to Code Solution for Task 3

1. No. we cannot directly use the F-matrix that was estimated in part a). We also need the Camera Intrinsics Matrix and the Essential Matrix to perform this computation.
2. In addition to the F-matrix, we also need the camera intrinsics and the Essential Matrix. The OpenCV: Camera Calibration guide was used for this purpose. The output that we obtained was,
   - RMS Re-projection error: 2.7287908399424508
   - Camera Matrix: $\begin{bmatrix} 1.37703227e+03 & 0.00000000e+00 & 9.89500107e+02 \\ 0.00000000e+00 & 1.38159326e+03 & 5.89329411e+02 \\ 0.00000000e+00 & 0.00000000e+00 & 1.00000000e+00 \end{bmatrix}$
   - Distortion Coefficients: $\begin{bmatrix} -0.02568162 & 0.128821 & 0.00506443 & 0.00331354 & -0.28263947 \end{bmatrix}$

   As for the Essential Matrix, we shall use findEssentialMat from OpenCV that takes the undistorted correspondences from the two images and the camera intrinsics to compute the Essential Matrix. Initially, the equation $E = K^T F K$ was tried to compute the Essential Matrix, but the results were grossly erroneous. Thus, the inbuilt method was used to provide more robustness. The following parameters were used -

   - **Parameters:**
     - `pts1_norm = cv2.undistortPoints(np.expand_dims(pts1, axis=1), K, None)`: Undistorted points from the left image.
     - `pts2_norm = cv.undistortPoints(np.expand_dims(pts2, axis=1), K, None)`: Undistorted points from the right image.
     - **Method (`cv.FM_RANSAC`):** RANSAC is chosen for its robustness in the presence of outliers, ensuring that the Fundamental Matrix is estimated based on the majority of inlier correspondences.

  - **RANSAC Reprojection Threshold (1.0 pixel):** This threshold defines the maximum allowed distance (in pixels) for a point to be considered an inlier. A value of **1.0 pixel** strikes a balance between being stringent enough to exclude gross outliers and lenient enough to accommodate minor detection inaccuracies.
  - **Confidence (99%):** This parameter sets the probability that the algorithm estimates a correct Fundamental Matrix. A high confidence of **99%** ensures that the likelihood of accepting a bad model is minimal, enhancing estimation reliability.

- **Output:**
  - The function returns the Essential Matrix (**E**) and a mask indicating inlier correspondences based on the RANSAC process.

Based on this, we have obtained the following output results -

- Essential Matrix: $\begin{bmatrix} 0.0309459 & 0.1232407 & -0.0016379 \\ 0.02637071 & -0.07223439 & -0.70284236 \\ 0.17199059 & 0.67099543 & -0.06430527 \end{bmatrix}$
- Rotation Matrix (R): $\begin{bmatrix} 0.94238711 & -0.23345033 & 0.23959856 \\ 0.26076413 & 0.96129021 & -0.08901239 \\ -0.20954377 & 0.14636284 & 0.96678298 \end{bmatrix}$
- Translation Vector (t): $\begin{bmatrix} -0.98371886 \\ -0.01409948 \\ 0.17916029 \end{bmatrix}$
- Epipolar Constraint Mean Error (E): 0.001475
- Epipolar Constraint Max Error (E): 0.004721
- Singular Values of E: $\begin{bmatrix} 7.07106781e-01 & 7.07106781e-01 & 8.76240409e-17 \end{bmatrix}$
- Percentage of points with positive depth: 100.00%
- Reprojection Error in Image 1: Mean=1.4639, Max=5.0022
- Reprojection Error in Image 2: Mean=1.4385, Max=4.4987

These results are desireable as the error values are within the expected range.

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 9: Frame t_1

Figure 10: Frame t_2

Answers:
Link to Code Solution for Task 4

To solve the given problem statement, two methods have been used. Both these methods were inspired by the OpenCV: Optical Flow chapter.

1. Lucas-Kanade Optical Flow (Sparse Optical Flow) Algorithm

Step 1: Identify Feature Points

- Detect "good" points to track using Shi-Tomasi Corner detection.
- These points are usually corners or distinct features in the image.

Step 2: Formulate the Optical Flow Equation

$$I(x, y, t) = I(x + \Delta x, y + \Delta y, t + \Delta t) \tag{1}$$

Using a first-order Taylor expansion:

$$\frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v + \frac{\partial I}{\partial t} = 0 \tag{2}$$

Where:

- $I_x = \frac{\partial I}{\partial x}$: Gradient along x-axis.
- $I_y = \frac{\partial I}{\partial y}$: Gradient along y-axis.

- $I_t = \frac{\partial I}{\partial t}$: Temporal gradient (change in intensity over time).
- $u = \frac{\Delta x}{\Delta t}$: Horizontal velocity.
- $v = \frac{\Delta y}{\Delta t}$: Vertical velocity.

Step 3: Solve for Motion

- Solve the above equation using least-squares over a small window W around the pixel:

$$\begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} \sum I_x I_t \\ \sum I_y I_t \end{bmatrix} \tag{3}$$

- Solve this linear system for u and v.

Step 4: Update

- Track the new positions of the feature points in the next frame and repeat the process.

Step 5: Multi-Scale (Pyramidal) Lucas-Kanade

- To handle large motions, use a coarse-to-fine strategy:
  1. Build a pyramid of downsampled versions of the frames.
  2. Compute optical flow starting from the coarsest level.
  3. Refine flow estimates at finer scales.

Shown below are a few frames that were processed by this algorithm:

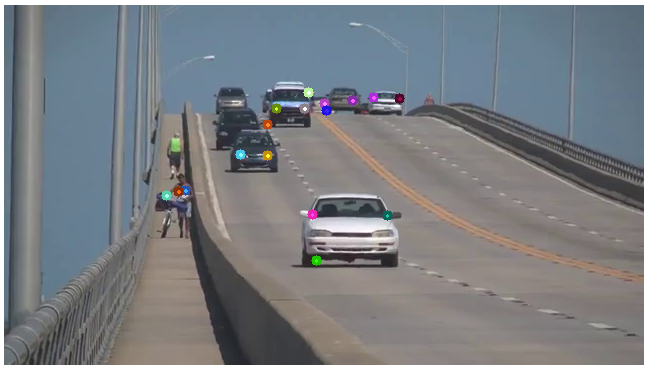

Figure 11: LK Optical Flow Frame 0

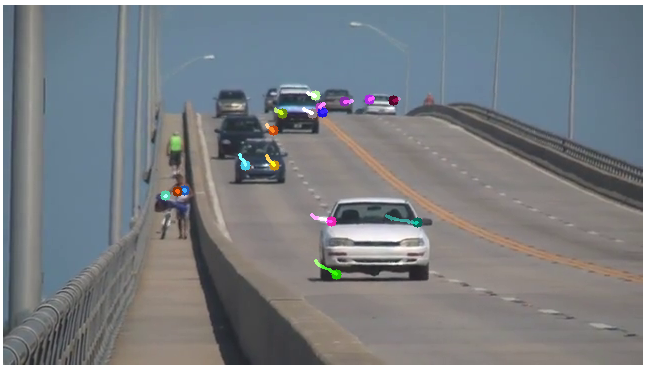

Figure 12: LK Optical Flow Frame 29

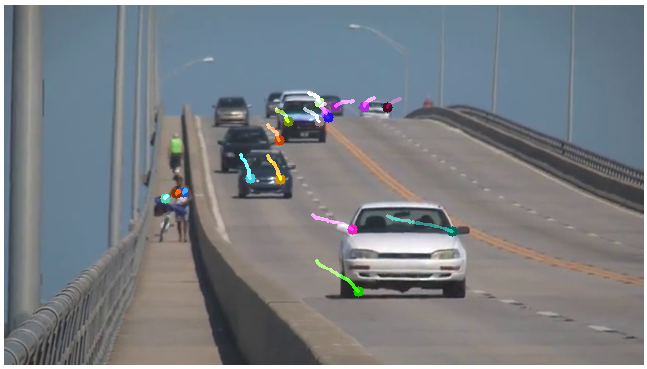

Figure 13: LK Optical Flow Frame 58

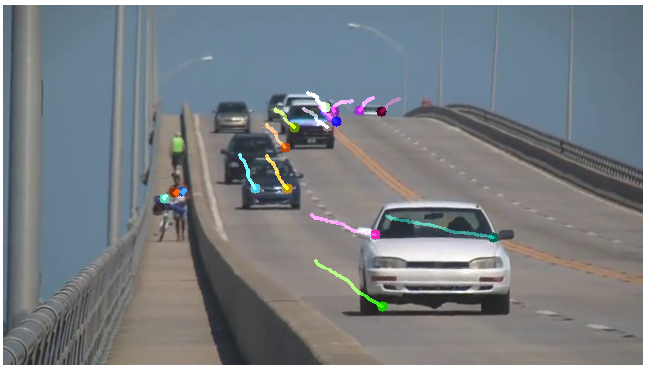

Figure 14: LK Optical Flow Frame 87

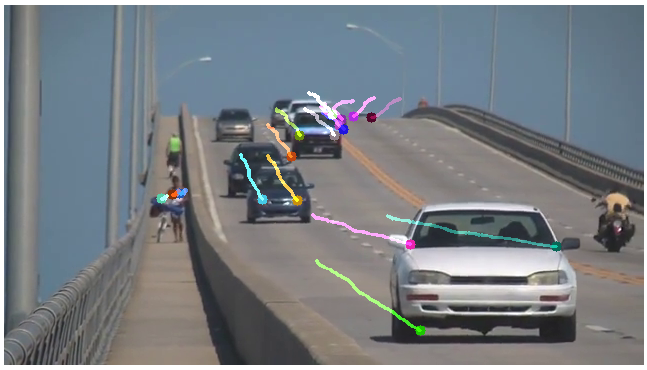

Figure 15: LK Optical Flow Frame 116

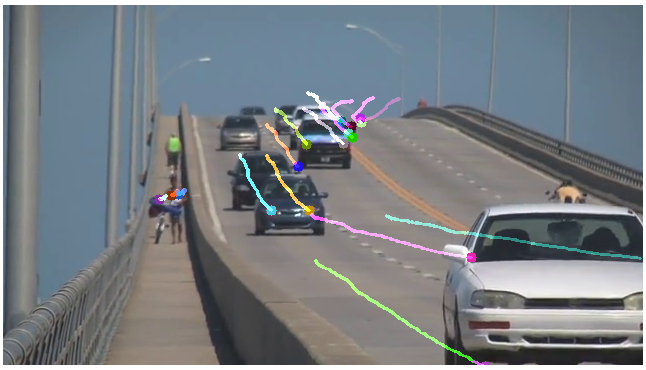

Figure 16: LK Optical Flow Frame 145

2. Farneback Dense Optical Flow Algorithm

Step 1: Model Image Neighborhoods

- Approximate pixel neighborhoods with a quadratic polynomial:

$$f(x, y) = a_0 + a_1 x + a_2 y + a_3 x^2 + a_4 xy + a_5 y^2 \tag{4}$$

- This representation encodes local intensity variations.

Step 2: Compare Neighborhoods Across Frames

- For each pixel, estimate the displacement $(\Delta x, \Delta y)$ by matching polynomial representations between consecutive frames.

Step 3: Compute Dense Optical Flow

- Use iterative refinement to estimate the displacement field.
- The flow is computed at all pixels, resulting in:
  - **Magnitude** (M): Speed of motion.
  - **Angle** (θ): Direction of motion.

Step 4: Post-Processing and Visualization

- Represent the motion field as a color-coded map:
  1. Map the flow direction (θ) to HSV **hue**.
  2. Map the flow magnitude (M) to HSV **brightness**.
  3. Convert HSV to RGB for visualization.

Shown below are a few frames that were processed by this algorithm:

Aspect	**Lucas-Kanade (Sparse)**	**Farneback (Dense)**
Tracking Targets	Tracks selected feature points	Tracks motion for all pixels
Motion Model	Assumes small motion locally	Models motion as polynomial shifts
Scale Handling	Uses pyramidal approach for large motions	Handles larger motions natively
Output	Sparse displacement vectors	Dense vector field (for all pixels)
Computation Complexity	Low (fewer points to track)	High (dense estimation for all pixels)
Applications	Object tracking, small motions	Motion analysis, flow visualization

Table 2: Key differences between Lucas-Kanade and Farneback optical flow algorithms.

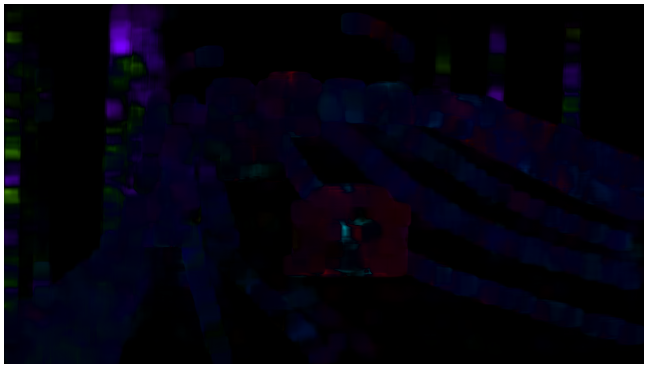

Figure 17: Dense Optical Flow Frame 0

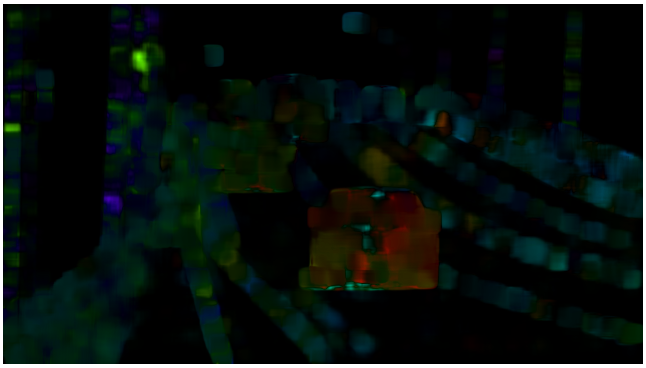

Figure 18: Dense Optical Flow Frame 29

Figure 19: Dense Optical Flow Frame 58

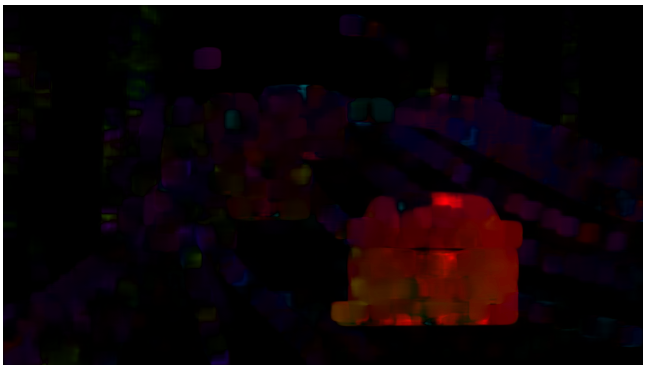

Figure 20: Dense Optical Flow Frame 87

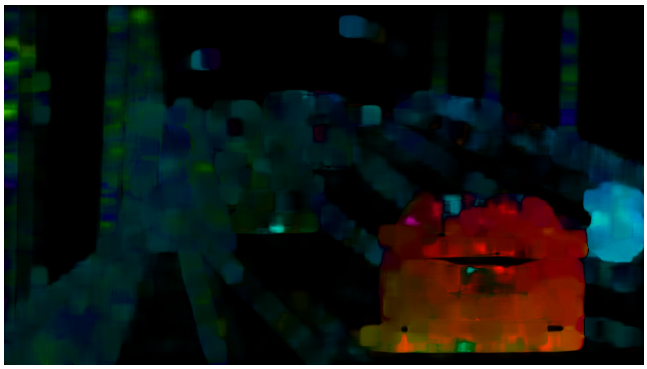

Figure 21: Dense Optical Flow Frame 116

Figure 22: Dense Optical Flow Frame 145

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.

Answers:
Link to Code Solution for Task 5

The given problem was solved using the algorithm shown below:

- **Initialization:**
  - Define directories for:
    - Query images (`query_dir`).
    - Database images (`database_dir`).
    - Output files (`output_dir`).
  - Specify the number of clusters for k-means clustering (`num_clusters`).
  - Determine the computational device (use GPU if available, otherwise CPU).
- **Collect Image Paths:**
  - Gather all image file paths from the query and database directories.
  - Filter the files to include only valid image formats (e.g., '.jpg', '.png', '.jpeg').
- **Feature Extraction:**
  - For each image, extract SIFT (Scale-Invariant Feature Transform) descriptors:
    - Load the image in grayscale.
    - Use a SIFT detector to compute keypoints and their descriptors.
    - If the image cannot be loaded, return `None`.
- **VLAD Encoding:**
  - Compute the VLAD (Vector of Locally Aggregated Descriptors) encoding for the extracted descriptors:
    1. Compute distances between descriptors and k-means centroids.
    2. Assign each descriptor to its nearest centroid.
    3. Calculate residuals (differences) between descriptors and their assigned centroids.
    4. Aggregate the residuals per cluster.
    5. Normalize the final VLAD vector using L2 normalization.

- **Descriptor Management:**
  - Load or compute descriptors for all database images:
    - If a pre-computed descriptor file exists, load it.
    - Otherwise, extract descriptors for each database image, save them, and return.
- **K-Means Clustering:**
  - Load or compute k-means centroids for clustering descriptors:
    - If a pre-computed k-means file exists, load it.
    - Otherwise, fit a MiniBatchKMeans model on all database descriptors, save the centroids, and return.
- **VLAD Descriptor Management:**
  - Load or compute VLAD vectors for all database images:
    - If a pre-computed VLAD file exists, load it.
    - Otherwise, compute VLAD vectors for each database image using the k-means centroids, save them, and return.
- **Query Processing:**
  - For each query image:
    1. Extract SIFT descriptors.
    2. Compute the VLAD vector using the k-means centroids.
- **Matching Query to Database:**
  - Compare each query's VLAD vector to the database VLAD vectors:
    1. Compute cosine similarity between the query VLAD vector and all database VLAD vectors.
    2. Filter matches based on a similarity threshold.
    3. Sort the matches by similarity in descending order.
- **Running the Process:**
  1. Load or compute SIFT descriptors for all database images.
  2. Perform k-means clustering to compute centroids from database descriptors.
  3. Load or compute VLAD vectors for all database images.
  4. For each query image:
     - Extract SIFT descriptors and compute its VLAD vector.
     - Find matches between the query VLAD vector and database VLAD vectors.
- **Output Results:**
  - Save results to output files (if necessary).
  - Display the matched database images with similarity scores for each query image.

The following pictures were obtained:

Figure 23: Query Image 1

Figure 24: Database Search Result 1

Figure 25: Query Image 2

Figure 26: Database Search Result 2

Figure 27: Query Image 3

Figure 28: Database Search Result 3