# COMP5422: Deep 2D and 3D Visual Scene Understanding

**Homework Assignment 2**

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Release Date: Apr. 07, 2025

# The HKUST Academic Honor Code

Honesty and integrity are central to the academic work of HKUST. Students of the University must observe and uphold the highest standards of academic integrity and honesty in all the work they do throughout their program of study. As members of the University community, students have the responsibility to help maintain the academic reputation of HKUST in its academic endeavors.

## 1 Introduction

In this homework, we will have two sub-problems. In the first sub-problem, we will implement a method for the reconstruction of a 3D scene based on two given camera views and their corresponding camera intrinsic parameters. In the second sub-problem, we will implement a monocular visual odometry algorithm that takes a monocular video as input, and outputs a camera position trajectory of the whole video sequence. The implemented functions in the first sub-problem will also be used in the second sub-problem. The submission deadline for this homework is **May. 06, 2025**. All the data needed for this homework assignment can be downloaded from this shared link. Please start as early as possible, and feel free to discuss with us if you have any questions in your design and implementation.

This assignment should be solved individually. You are allowed to discuss ideas on possible solutions, but without collaboration or sharing of solutions. Please, do not directly copy existing code from anywhere. We will check assignments for duplicates. See below for more details about this homework.

## 2 Two-View 3D Scene Reconstruction

### 2.1 Overview

In this part, you will begin by implementing a method (i.e., 8 point algorithm) introduced in class to estimate the fundamental matrix from corresponding 2D points in two images (Section 2.2). Next, given the fundamental matrix and calibrated intrinsics (which will be provided) you will compute the essential matrix and use this to compute a 3D metric reconstruction from 2D correspondences using triangulation (Section 2.3). Then, you will implement a method to automatically match points taking advantage of epipolar constraints and finally make a 3D visualization of the results (Section 2.4).

### 2.2 Fundamental matrix estimation

In this section, you will explore a method to estimate the fundamental matrix given a pair of images. In the `data/` directory, you will find two images (see Figure 1) from the KITTI dataset[1], which is widely used to for various tasks in self-driving, including outdoor 3D reconstruction and SLAM.

[1] `https://www.cvlibs.net/datasets/kitti/index.php`

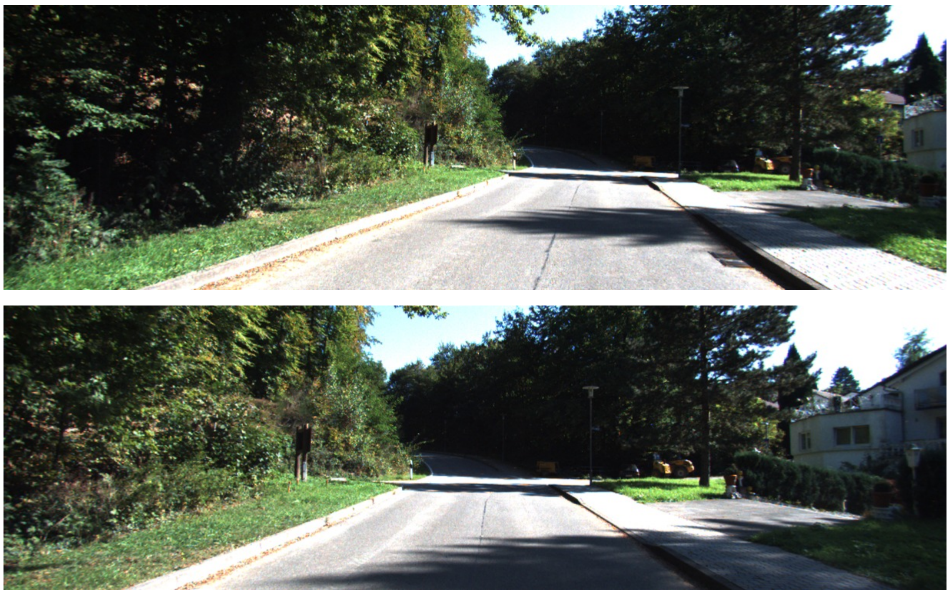
Figure 1: KITTI two-view images for this assignment

## The Eight Point Algorithm

The 8-point algorithm (discussed in class, and detailed in Section 10.1 of Forsyth & Ponce) is arguably the simplest method for estimating the fundamental matrix. You will need to first utilize a SIFT feature decriptor for detection and matching of interest of points to find a set of 2D point correspondences. And then you will implement the 8 point algorithm to solve out the fundamental matrix for the given two image views.

**Q 2.2.1 (10 points)** You will write a function to perform the detection of points of interest and the matching of detected points between two images, in order to find a set of matched 2D points. In this part, you are allowed to use existing OpenCV implementations (the `SIFT` descriptor for key point detection, and the `BFMatcher` or `FlannMatcher` for point matching) for coding your function. You can also try skimage's Ransac function to obtain a more reliable set of inliers from all the matched points. An example usage of the RANSAC algorithm is shown below:

```
from skimage.measure import ransac
from skimage.transform import AffineTransform
_, inliers = ransac((src_pts, dst_pts), AffineTransform,
                    min_samples=4, residual_threshold=8, max_trials=10000)
src_pts = src_pts[inliers]
dst_pts = dst_pts[inliers]
```

The implemented function by you will input two images, and return a set of matched 2D points. You will visualize the detected keypoints and also draw top-100 matched points using the function of `drawMatches` from OpenCV. An example of the visualization is shown in Fig. 2.

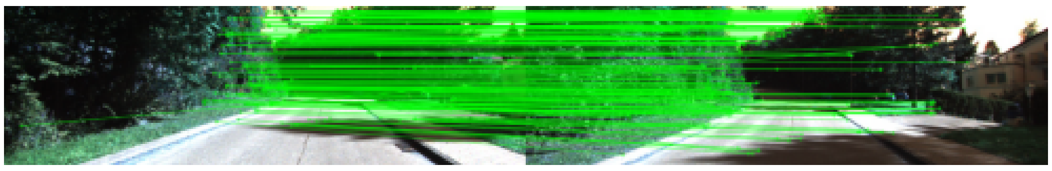

Figure 2: Visualization of detected key points by SIFT and drawing of top-100 matches of points.

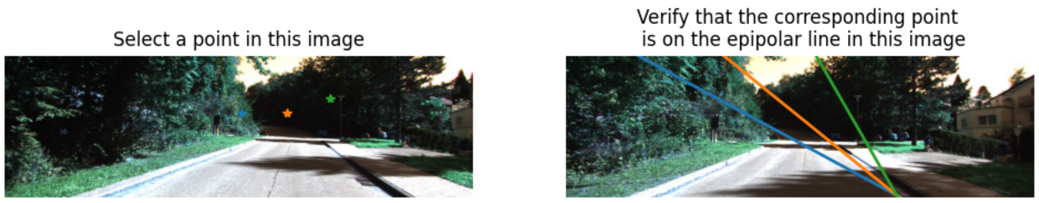
Select a point in this image
Verify that the corresponding point is on the epipolar line in this image

Figure 3: `displayEpipolarF` in `helper.py` creates a GUI for visualizing epipolar lines.

**Output**: Save your matched 2D points into `q2.2_1.npz`.
**In your write-up**: You may input the visualization of your results (i.e., Fig 2) into a PDF file (with a name 'results.pdf').

**Q 2.2.2 (10 points)** Finish the function `eightpoint` in `submission.py`. Make sure you follow the signature for this portion of the assignment:

```
F = eightpoint(pts1, pts2, M)
```

where `pts1` and `pts2` are $N \times 2$ matrices corresponding to the $(x, y)$ coordinates of the $N$ points in the first and second image repectively. $M$ is a scale parameter. Following the details provided in Section 10.1 of Forsyth & Ponce), we have the following steps:

- You should scale the data for coordinate normalization, by dividing each coordinate with $M$ (the maximum of the image's width and height). After computing $\mathbf{F}$, you will have to "unscale" the fundamental matrix.

  *Hint*: If $x_{\text{normalized}} = Tx$, then $F_{\text{unnormalized}} = T^T\mathbf{F}T$.

  You must enforce the singularity condition of the $\mathbf{F}$ before unscaling.

- You may find it helpful to refine the solution by using local minimization. This probably won't fix a completely broken solution, but may make a good solution better by locally minimizing a geometric cost function.

  For this we have provided a helper function `refineF` in `helper.py` taking in `F` and the two sets of points, which you can call from `eightpoint` before unscaling `F`.

- Remember that the $x$-coordinate of a point in the image is its column entry, and $y$-coordinate is the row entry. Also note that eight-point is just a figurative name, it just means that you need *at least* 8 points; your algorithm should use an over-determined system ($N > 8$ points).

- To visualize the correctness of your estimated `F`, we provide `displayEpipolarF` function in `helper.py`, which takes in `F`, and the two images. This GUI lets you select a point in one of the images and visualize the corresponding epipolar line in the other image (Figure 3).
- **Output**: Save your matrix `F`, scale `M` to the file `q2.2_2.npz`.

  **In your write-up**: Write your recovered `F` and include an image of some example outputs of `displayEpipolarF` into a pdf file (with a name 'results.pdf').

## 2.3 Metric 3D Reconstruction

You will compute the camera matrices and triangulate the 2D points to obtain the 3D scene structure. To obtain the Euclidean scene structure, first convert the fundamental matrix $\mathbf{F}$ to an essential matrix $\mathbf{E}$. Examine the lecture slides and the reference textbook to find out how to do this when the internal camera calibration matrices $\mathbf{K}_1$ and $\mathbf{K}_2$ are known (in our case, $\mathbf{K}_1$ and $\mathbf{K}_2$ are the same because the two image frames are from the same monocular KITTI video captured by the same camera); these are provided in `data/intrinsics4Recon.npz`.

**Q 2.3.1 (5 points)** Write a function to compute the essential matrix $\mathbf{E}$ given $\mathbf{F}, \mathbf{K}_1$ and $\mathbf{K}_2$ with the signature:

```
E = essentialMatrix(F, K1, K2)
```

**In your write-up**: Write down your estimated $\mathbf{E}$ using $\mathbf{F}$ from the eight-point algorithm in results.pdf. Given an essential matrix, it is possible to retrieve the projective camera matrices $\mathbf{M}_1$ and $\mathbf{M}_2$ from it. Assuming $\mathbf{M}_1$ is fixed at $[\mathbf{I}, 0]$, $\mathbf{M}_2$ can be retrieved up to a scale and four-fold rotation ambiguity. For details on recovering $\mathbf{M}_2$, see section 7.2 in Szeliski. We have provided you with the function `camera2` in `helper.py` to recover the four possible $\mathbf{M}_2$ matrices given $\mathbf{E}$.

**Note**: The $\mathbf{M}_1$ and $\mathbf{M}_2$ here are projection matrices of the form: $\mathbf{M}_1 = [\mathbf{I}|0]$ and $\mathbf{M}_2 = [\mathbf{R}|\mathbf{t}]$.

**Q 2.3.2 (10 points)** Using the above, write a function to triangulate a set of 2D coordinates in the image to a set of 3D points with the signature:

```
[P, err] = triangulate(C1, pts1, C2, pts2)
```

where `pts1` and `pts2` are the $N \times 2$ matrices with the 2D image coordinates of matched points and `P` is an $N \times 3$ matrix with the corresponding 3D points per row. It should be noted that `C1` and `C2` are $3 \times 4$ camera matrices, and you obtain them multiply the given intrinsics matrices with your solution for the projection matrices $\mathbf{M}_1$ and $\mathbf{M}_2$ respectively, as $\mathbf{C}_1 = \mathbf{K}_1\mathbf{M}_1 = \mathbf{K}_1[\mathbf{I}|0]$ and $\mathbf{C}_2 = \mathbf{K}_2\mathbf{M}_2 = \mathbf{K}_2[\mathbf{R}|\mathbf{t}]$. In the implementation, you assume that $\mathbf{C}_1$ and $\mathbf{C}_2$ are given as inputs of the function.

Various methods exist for triangulation, and probably the most familiar for you is based on least squares (see Szeliski Chapter 7 if you want to learn about other methods). For each point $i$, we want to solve for 3D coordinates $P_i = [x_i, y_i, z_i]^T$ , such that when they are projected back to the two images, they are close to the original 2D points. To project the 3D coordinates back to 2D images, we first write $P_i$ in homogeneous coordinates, and compute $\mathbf{C}_1 P_i$ and $\mathbf{C}_2 P_i$ to obtain the 2D homogeneous coordinates projected to camera 1 and camera 2, respectively. For each point $i$, we can write this problem as follows:

$$\mathbf{A}_i P_i = 0,$$

where $\mathbf{A}_i$ is a $4 \times 4$ matrix, and $P_i$ is a $4 \times 1$ vector of the 3D coordinates in the homogeneous form. Then, you can obtain the homogeneous least-squares solution (discussed in class) to solve for each $P_i$.

Once you have implemented triangulation, check the performance by looking at the reprojection error:

$$\texttt{err} = \sum_i \|p_{1i}, \hat{p}_{1i}\|^2 + \|p_{2i}, \hat{p}_{2i}\|^2$$

where $p_{1i}$ and $p_{2i}$ are matched 2D points; $\hat{p}_{1i} = \text{Proj}(\mathbf{C}_1, P_i)$ and $\hat{p}_{2i} = \text{Proj}(\mathbf{C}_2, P_i)$, are projected 2D points.

**Q2.3.3 (10 points)** Write a script `findM2.py` to obtain the correct `M2` from `M2s` (the four possible M2 matrices from `helper.py` mentioned in Q2.3.1) by testing the four solutions through triangulations (implemented in Q2.3.2). As $\mathbf{M}_1 = [\mathbf{I}|0]$ and $\mathbf{M}_2 = [\mathbf{R}|\mathbf{t}]$, a correct M2 triangulates a 3D point that is in front of both cameras. Considering possible noisy 2D matches, the best M2 is obtained when it triangulates the most 3D points in front of both cameras. You can utilize this constraint to implement your script to find the best M2. You will also need the 2D point correspondences that you obtained from a previous step (see Q2.2.1).

**Output**: Save the correct `M2`, the corresponding `C2`, and the correctly triangulated 3D points `P` to `q2.3_3.npz`.

## 2.4 3D Visualization

You will now create a 3D visualization of our given KITTI images. By treating our two images as a stereo-pair, we can triangulate corresponding points in each image, and render their 3D locations.

**Q2.4.1 (15 points)** Implement a function with the signature:

```
[x2, y2] = epipolarCorrespondence(im1, im2, F, x1, y1)
```

This function takes in the $x$ and $y$ coordinates of a pixel on `im1` and your fundamental matrix $\mathbf{F}$, and returns the coordinates of the pixel on `im2` which correspond to the input point. The match is obtained by computing the similarity of a small window around the $(x_1, y_1)$ coordinates in `im1` to various windows around possible matches in the `im2` and returning the closest.

Instead of searching for the matching point at every possible location in `im2`, we can use $\mathbf{F}$ and simply search over the set of pixels that lie along the epipolar line (recall that the epipolar line passes through a single point in `im2` which corresponds to the point $(x_1, y_1)$ in `im1`.

There are various possible ways to compute the window similarity. For this assignment, simple methods such as the Euclidean or Manhattan distances between the intensity of the pixels should suffice. See Szeliski Chapter 11, on stereo matching, for a brief overview of these and other methods. Some implementation notes are give below:

- Experiment with various window sizes.
- It may help to use a Gaussian weighting of the window, so that the center has greater influence than the periphery.
- Since the two images only differ by a small amount, it might be beneficial to consider matches for which the distance from $(x_1, y_1)$ to $(x_2, y_2)$ is small.

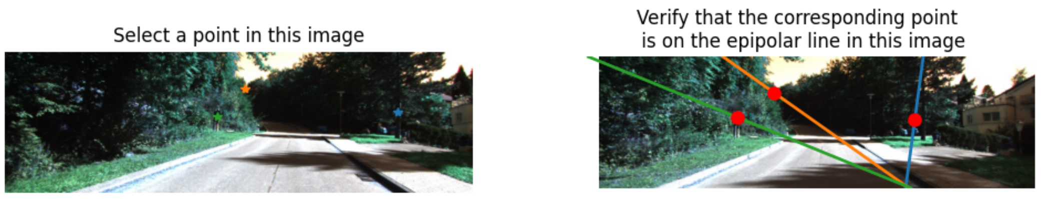
Select a point in this image
Verify that the corresponding point is on the epipolar line in this image

Figure 4: The `epipolarMatchGUI` shows the corresponding point found by calling `epipolarCorrespondence`.

To test your `epipolarCorrespondence`, we provide a helper function `epipolarMatchGUI` in `helper.py`, which takes in the fundamental matrix of the two images. This GUI allows you to click on a point in `im1`, and will use your function to display the corresponding point in `im2`. See Figure 4.

It is not necessary for your matcher to get *every* possible point right, but it should get easy points (such as those with distinctive, corner-like windows). It should also be good enough to render an intelligible representation in the next question.

**Output**: Save the matrix `F`, points `pts1` and `pts2` which you used to generate the screenshot to the file `q2.4_1.npz`.

**In your write-up**: Include a screenshot of `epipolarMatchGUI` with some detected correspondences in 'results.pdf'.

**Q2.4.2 (10 points)** Included in this homework is a file `data/VisPts.npz`, where each line contains a pair of coordinates (x, y). Now, we can determine the 3D location of these point correspondences using the triangulate function. These 3D point locations can then be plotted using the Matplotlib or plotly package. Write a script `visualize.py`, which loads the necessary files from `data/` to generate the 3D reconstruction using scatter. An example is shown in Figure 5.

**Output**: Again, save the matrix `F`, `C1`, and `C2` which you used to generate the screenshots to the file `q2.4_2.npz`.

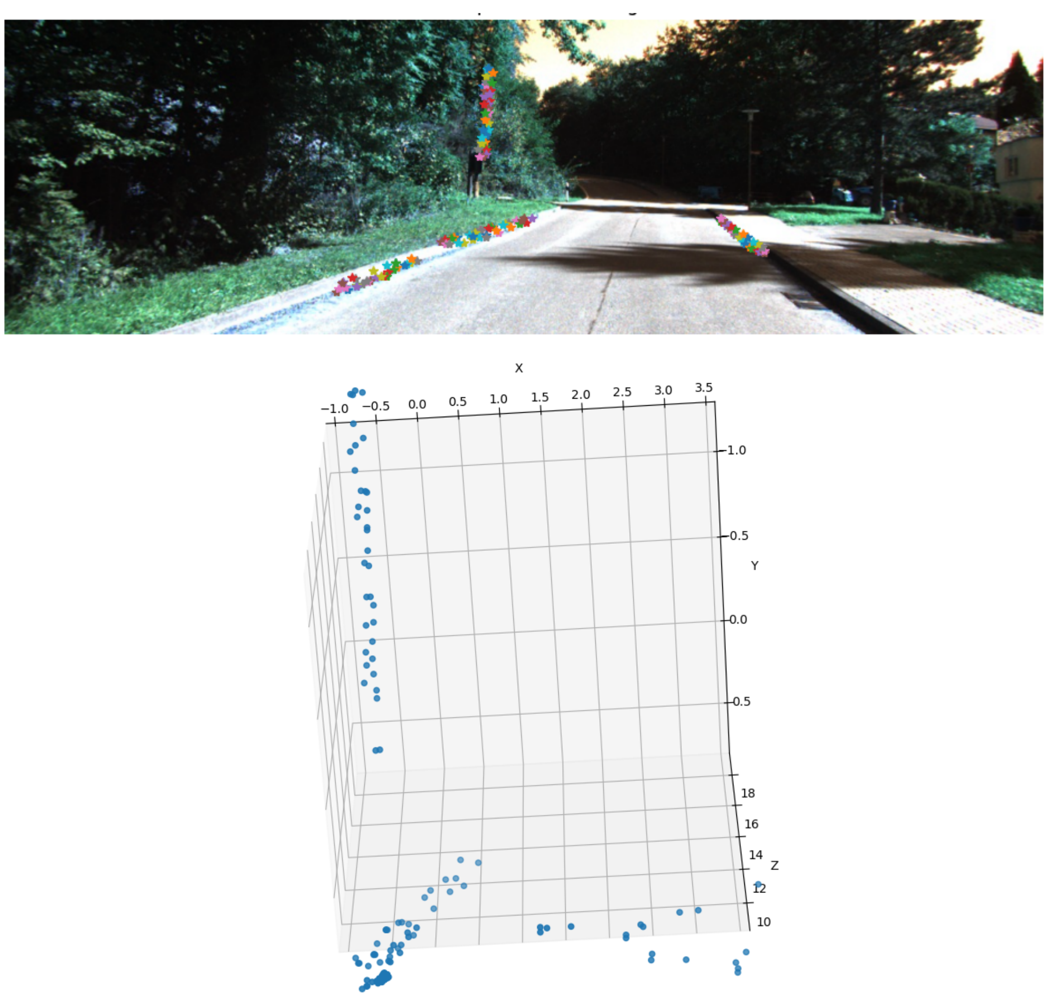

Figure 5: An example of a set of sampled 2D points for visualization and the corresponding 3D point cloud visualization.

**In your write-up**: Take a few screenshots of the 3D visualization so that the outline of the scene structure is clearly visible, and include them in the 'results.pdf'.

## 3 Monocular Visual Odometry

In this section, you will learn to implement an estimation of the camera motion (i.e., the rotation, and the translation) between each pair of consecutive frames in a video sequence captured by a monocular camera. When you can estimate a relative motion between each pair of camera frames, you can generate a camera trajectory for the whole video sequence. This task is well-known as visual odometry. As we consider a monocular video as input, it is a typical monocular visual odometry problem. You will also utilize the functions implemented in the previous section for the coding tasks of this section.

**Q3.1 (25 points)** In this task, you will implement a function to perform a decomposition

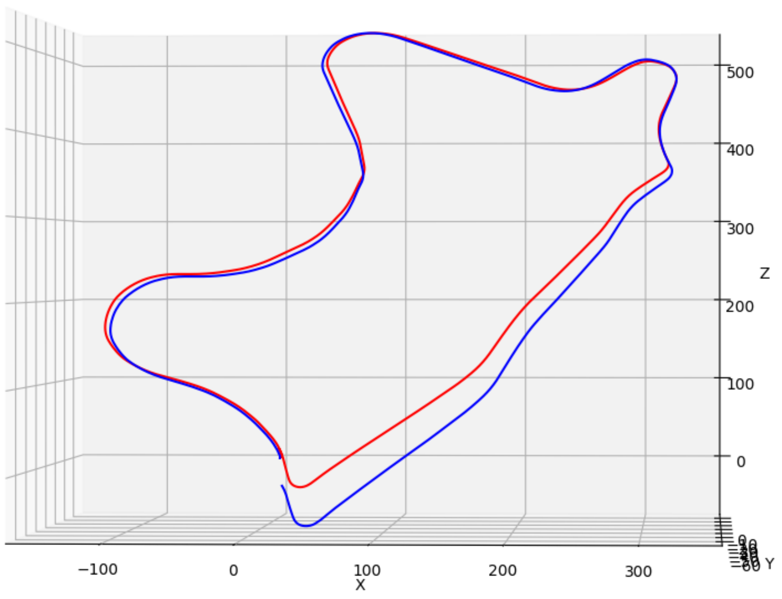

Figure 6: A visualization of the estimated and the GT camera pose trajectories.

of the Essential Matrix **E** to recover Rotation and Translation, with a signature as:

```
[R, t] = RTRecovery(im1, im2, k1, k2),
```

where **R** and **t** are the rotation and translation between the two views, respectively; `im1` and `im2` are the two consecutive frames. In this function, you will first need to use the 8-point algorithm implemented in the previous 3D reconstruction section for you to estimate the fundamental matrix **F**, and then convert the fundamental matrix to the essential matrix based on the camera intrinsic $\mathbf{K}_1$ and $\mathbf{K}_2$, based on the function `essentialMatrix` implemented in Q2.3.1. As the two views are from the same camera, $\mathbf{K}_1$ and $\mathbf{K}_2$ are the same in our case. The camera intrinsics are provided in `data/Intrinsic4Recon.npz`.

**Q3.2 (30 points)** In this part, you will implement a visual odometry function with a signature as follows:

```
trajectory = visualOdometry(datafolder, GT_Pose, plot=True),
```

where `trajectory` is the returned camera trajectory of the whole video sequence; `GT_Pose` is the provided ground-truth (GT) camera poses for all the video frames of the video sequence (`data/GTPoses.npz`). `Plot=True` indicates that we will implement to draw the estimated and the GT 3D camera trajectories in the same plot, which can help to easily observe the error of the estimated camera trajectory. An example of visualized trajectories is shown in Fig. 6. Some important notes for the implementation of the visual odometry are illustrated below:

- You need to estimate the relative camera motion of each pair of two consecutive image frames in the video, which can be performed in a loop in the function. The first frame can be defined as $[\mathbf{I}|\mathbf{0}]$. You accumulate your rotation and translation from the second frame to the last frame, and you will generate a camera trajectory in the world coordinate system.

- As the fundamental/essential matrix is defined up to a scale, you will need to calibrate the scale of the estimated trajectory based on the GT trajectory. You could calculate the scale using the GT translations of each pair of frames. We provide a function `getAbsoluteScale` in `helper.py` for you to calculate the scale factor for each pair of frames. Then, you can use this scale to multiply the estimated translation at each frame (there is no need to align the rotation) to produce the final scale-calibrated camera pose trajectory, and you will be able to obtain an absolute position (an absolute translation to the origin) for each camera frame using the following formula:

$$\mathbf{t}_i = \mathbf{t}_{i-1} + \text{scale} * (\mathbf{R}_{i-1->i} * \mathbf{t}_{i-1->i}), \tag{1}$$

  where $\mathbf{t}_i$ indicates the absolute position (translation) of the $i$-th frame; $\mathbf{R}_{i-1->i}$ and $\mathbf{R}_{i-1->i}$ mean the relative rotation and translation between the $(i-1)$-th and the $i$-th frame, respectively. And then, you are able to visualize the camera positions of the whole video sequence as a trajectory shown in Fig. 6.

**Output**: Save the generated scale-aligned camera trajectory data (i.e., the trajectory returned the function `visualOdometry`) into `q3_2.npz`.
**In your write-up**: Include a plot that draws both the estimated and the GT camera trajectories as shown in Fig. 6 into 'results.pdf'.

## 4 Submission

Once you finished the homework assignment, create a submission bundle using

```
python3 bundle.py homework [YOUR UST ID],
```

and then submit the zip file on canvas. Please remember to put your output .pdf file and .npz files under the folder `./homework`. Please note that the maximum file size for your submission is 20MB.