CIS 5800, Machine Perception, Spring 2025
Homework 3C (Coding)
Due: Wednesday Mar. 26 2025, 11:59pm ET

Instructions

- This is the coding part for Homework 3 and worth **100 points**
- You need to check the released package containing **python files** and fill in all the sections where ##### STUDENT CODE START ##### is indicated. You can run **main.ipynb** to debug and correct your implementation.
- Start early! Please post your questions on Ed Discussion (using the appropriate thread category) or come to office hours!

Submission

- For this coding part, you need to submit **6 Python files** (Please refer to to Sec. 1.1 for details) in total to the Gradescope.

3D Reconstruction from two 2D images

Your friend Satsok took some pictures of a castle he visited last summer, and he would love to turn them into a 3D miniature model of the castle. Having heard about your freshly learned computer vision skills, he gives you two pictures (as shown in figure 1) and challenges you to find out where he was standing when he took them and what the 3D scene looks like. He gives you the following details:

- You need to recover the 3D transformation (R, T) between the two views, such that $P_1, P_2 \in R^3$ describe the same scene point in frame 1 and 2, and $\boxed{P_2 = RP_1 + T}$.

We will use the following notations:

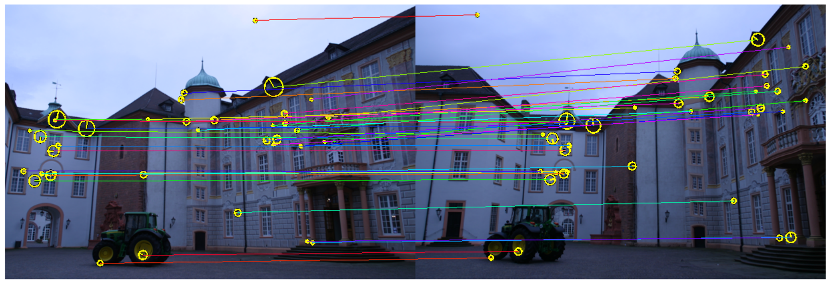

Figure 1: The two images we will use in this problem. A few SIFT matches are shown on top of the images.

- The notation $\mathbf{x} = (x, y, 1)$ (homogeneous metric coordinates) will be used for calibrated projections:

  $\boxed{P_1 = \lambda_1 \mathbf{x_1}, \quad P_2 = \lambda_2 \mathbf{x_2}}$

- The notation $\mathbf{u} = (u, v, 1)$ (homogeneous pixel coordinates) will be used for uncalibrated projections:

  $\boxed{\mathbf{u_1} \sim K\mathbf{x_1}, \quad \mathbf{u_2} \sim K\mathbf{x_2}}$

  $\boxed{\mathbf{u} \sim K\mathbf{x} \; \Leftrightarrow \; u = fx + u_0, \; v = fy + v_0}$

- epi($\mathbf{x_1}$) is the epipolar line in camera 2 corresponding to $\mathbf{x_1}$.

- He took the two pictures with no zoom and he turned the auto-focus off: the two views share the same matrix of intrinsics $K = \begin{pmatrix} f & 0 & u_0 \\ 0 & f & v_0 \\ 0 & 0 & 1 \end{pmatrix}$ with $f = 552$ and $(u_0, v_0) = (307.5, 205)$.

- Template code is available on the website along with the data/images. The code is split up into a main Jupyter notebook file along with several modules/files where you will implement the core functions used in the notebook.

- The python packages you will need to install are numpy, matplotlib, opencv-python, opencv-contrib-python, and jupyterlab.

1.1 What to submit

Submit to Gradescope only the following files:

- `lse.py`

- `plot_epi.py`

- `pose.py`
- `ransac.py`
- `recon3d.py`
- `show_reproj.py`

1.2 Estimation of the essential matrix (54 pts)

You can attempt to reconstruct the scene from the SIFT matchings in the two images (figure 1). The SIFT descriptors are simply computed and matched using OpenCV. Although all the calls to OpenCV for SIFT extraction and matching are already done for you, you might want to take a quick look at OpenCV's tutorials here. Also, quickly read the whole script to understand the overall pipeline.

1. **Least-squares estimation** Complete the function `least_squares_estimation` in `lse.py` so that it takes two $N \times 3$ matrices of matched, calibrated points `X1,X2` as input and estimates E using the SVD decomposition method as in the 8-point algorithm described in the lecture notes and the "E-matrix" handout. Don't forget to project the E you obtain onto the space of essential matrices by redecomposing `[U,S,Vt] = np.svd(E)` and returning $U\text{diag}(1,1,0)V^\top$.

2. **RANSAC estimation** The estimation of E can be made much more robust by selecting sets of points that reach a common agreement: in this section you will implement a basic RANSAC algorithm to eliminate outliers (spurious matchings) and obtain a better estimate of E.

   (a) One iteration of the algorithm uses the code from the previous question to estimate the essential matrix E based on a random set of 8 correspondences, and then evaluates how good E is for the other pairs $(\mathbf{x_1}, \mathbf{x_2})$. If E is perfect, for all pairs $(\mathbf{x_1}, \mathbf{x_2})$, $\mathbf{x_2}$ lies exactly on the epipolar $\text{epi}(\mathbf{x_1})$ computed from E and $\mathbf{x_1}$. Therefore, we will use the distances to the epipolars as an error measure. The distance of a matching point to the epipolar is the following:

   $$d(\mathbf{x_2}, \text{epi}(\mathbf{x_1}))^2 = \frac{(\mathbf{x_2}^T E \mathbf{x_1})^2}{\|\widehat{\mathbf{e_3}} E \mathbf{x_1}\|^2}.$$

   where $\mathbf{e_3} = (0,0,1)^T$ and $\hat{\mathbf{e_3}}$ is the skew-symmetric matrix of $\mathbf{e_3}$.

   (b) Complete the function `ransac_estimator` in `ransac.py` that takes a set of matching points and estimates `E` using RANSAC. The algorithm steps are the following:

   - Pick a random set of 8 pairs, estimate E using them and compute the individual residuals for all the other pairs $(\mathbf{x_1}, \mathbf{x_2})$: $d(\mathbf{x_2}, \text{epi}(\mathbf{x_1}))^2 + d(\mathbf{x_1}, \text{epi}(\mathbf{x_2}))^2$.

- Count how many residuals are lower than $\varepsilon = 10^{-4}$ (consensus set), and if this count is the largest so far, store the current estimate of `E` as the best estimate so far,
- Iterate as many times as needed according to the probability of failure

3. **Drawing the epipolar lines** You can now use the essential matrix to plot epipolar lines in the images, as shown in figure 2. Note that E defines epipolars for *calibrated* points $(\mathbf{x_1}, \mathbf{x_2})$. For the uncalibrated points, we showed in class there is a perfectly identical relationship by substituting the **fundamental matrix** F to E:

$$\mathbf{x_2}^T E \mathbf{x_1} = \mathbf{u_2}^T \underbrace{K^{-T} E K^{-1}}_{F} \mathbf{u_1} = 0 \Leftrightarrow \mathbf{u_1} \in \text{epi}_F(\mathbf{u_2})$$

   Complete the function `plot_epipolar_lines` in `plot_epi.py` that draws the epipolar lines for a set of matching image points (uncalibrated pixel coordinates). Note that you just need to fill in the equations, the "drawing" part is already done for you.

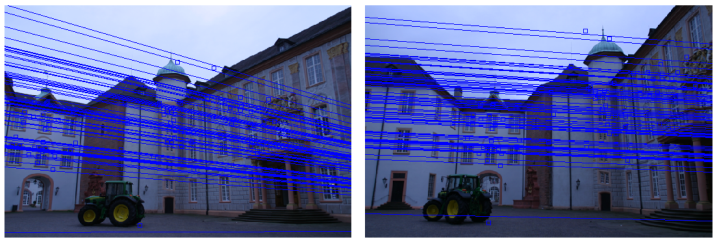

Figure 2: Some epipolar lines in both images: for a given $\mathbf{x_1}$, the matching point $\mathbf{x_2}$ should be as close as possible to epi$(\mathbf{x_1})$ and vice versa.

1.3 Pose recovery and 3D reconstruction (46 pts)

It is now time to recover the transformation R, T between the two cameras. For a given estimate of E, remember that there are two possible solutions for $(\widehat{T}, R)$ (twisted pair ambiguity):

$$\begin{aligned}(\widehat{T}_1, R_1) =& (U R_Z(\frac{\pi}{2}) \Sigma U^T, U R_Z(\frac{\pi}{2})^T V^T) \\ (\widehat{T}_2, R_2) =& (U R_Z(-\frac{\pi}{2}) \Sigma U^T, U R_Z(-\frac{\pi}{2})^T V^T)\end{aligned}$$

where (U, Σ, V) is the SVD decomposition of E and $R_Z(\frac{\pi}{2}) = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$.

1. The third column of U has associated hat matrix $UR_Z(+\frac{\pi}{2})U'$. Also the opposite of the third colum of U has associated hat matrix $UR_Z(-\frac{\pi}{2})U'$. Therefore, in the two solutions above, you can output T instead of its hat matrix and simply use the third column of U and its opposite for T_1 and T_2.

2. The sign switch between $R_Z(+\frac{\pi}{2})$ and $R_Z(-\frac{\pi}{2})$ inside the decompositions of the solutions is equivalent to a left-multiplication of the solutions by $R_T(\pi)$ where T is the third column of U.

3. Complete the function `pose_canidates_from_e` in `pose.py` that takes E as an input and returns four possible solutions for (R, T):

   - the twisted pair for E,
   - *and* the twisted pair for $-E$. This is because if we find E that satisfies all the epipolar constraints $\mathbf{x_1}^T E\mathbf{x_2} = 0$ (part 1.2), the opposite matrix $-E$ also verifies the constraint since the right-hand side is zero, and we don't know a priori whether E or $-E$ is the essential matrix we are looking for.

   ADDENDUM NOTE: Only perform SVD on the single, given E and denote the result U, Σ, V^T. Then, use the decomposition in the following order (ordering is purely enforced for autograding purposes):

   (a) T from U, $R = UR_z(\frac{\pi}{2})^T V^T$
   (b) T from U, $R = UR_z(-\frac{\pi}{2})^T V^T$
   (c) T from $-U$, $R = UR_z(\frac{\pi}{2})^T V^T$
   (d) T from $-U$, $R = UR_z(-\frac{\pi}{2})^T V^T$

4. **Triangulation** Given a candidate (R_i, T_i), we can now attempt to reconstruct all the points from the pairs (x_1, x_2), and check whether (R_i, T_i) is the correct transformation (we need to pick one out of the four candidates): we will pick the candidate (R_i, T_i) that has the highest number of reconstructed points *in front of both cameras*. Consider one of the matchings (x_1, x_2): we can reconstruct the 3D point by computing the intersection of the two rays coming from camera 1 and 2:

   $$\lambda_2 \mathbf{x_2} = \lambda_1 R\mathbf{x_1} + T,$$

   In practice, the equality doesn't hold (the two rays don't intersect perfectly) and you need to do a least square estimation of λ_1, λ_2.

   For a given pair $(\mathbf{x_1}, \mathbf{x_2})$, formulate the linear system such that $c\lambda_1\lambda_2$ is its solution. Complete the part of `reconstruct3D` in `recon3d.py` that loops through the four transformation candidates (T, R) and all pairs $(\mathbf{x_1}, \mathbf{x_2})$, and solves for the depths λ_1, λ_2.

   The selection of the right (T, R) out of the four possibilities is already coded for you but you should take a look at it and make sure you understand it: we simply count for each candidate (T_i, R_i) how many points are in front of both cameras (namely both $\lambda_1, \lambda_2 > 0$) and keep the transformation that achieves the maximum number.

5. You can now run the whole pipeline in the notebook to estimate the transformation (T, R) between the two cameras and the relative positions of the 3D points. Figure 3 shows an example of output from `plot_reconstruction`, shown in the (X, Z) plane (we're on top of the scene, looking down at it like a street map). Note that all depths and the translation (baseline) are up to a scale.

6. Complete the function `show_reprojections` in `show_reproj.py` to compare image points in a given camera to the reprojection of images points from the other camera. All you need to do in this function is apply the camera projection model (and be careful about the expression of the 3D transform from 1 to 2 and from 2 to 1).

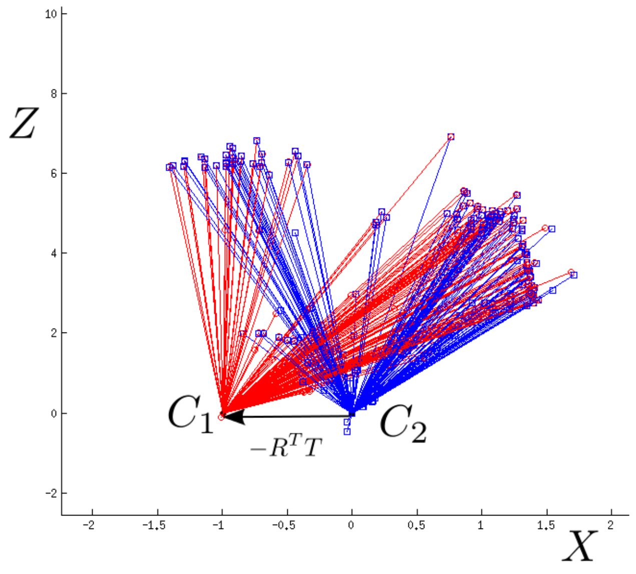

Figure 3: Top view of the the scene, coordinate are expressed in the frame of camera 2 (notice its center is at $0, 0$).