EECS C106B Project 3: Multi-View 3D Reconstruction *

Due date: March 23rd at 11:59pm

You will develop implement algorithms to perform two-view feature matching and reconstruction, and multi-view feature-matching and reconstruction. You will also implement a triangulation procedure to find the 3D locations of feature points given observations in two images with known poses. You will evaluate your approach by measuring the re-projection error of your reconstruction. For extra credit, you may implement a feature matching scheme for learning-based wireframe representations of images.

Contents

*Developed by Haozhi Qi and Amay Saxena, Spring 2021. Part of the dataset is supported by Yichao Zhou and Xili Dai

1 Theory

1.1 Camera Model

Camera Parameters. Without loss of generality, we will define the first camera's frame as the world coordinate frame. We denote the relative rotation and translation of the k-th camera (with respect to the first camera) as $\mathbf{R}_k \in \mathbb{R}^{3\times3}$, $\mathbf{t}_k \in \mathbb{R}^{3\times1}$. And we will use $\mathbf{P}_k \in \mathbb{R}^{3\times4}$ and $\mathbf{K}_k \in \mathbb{R}^{3\times3}$ as the projection matrix and the intrinsic matrix for the k-th camera.

Projection. Let $\mathbf{X} \in \mathbb{R}^4$ be a point in the 3D space (in the homogeneous coordinate, which means the fourth coordinate is 1), and $\mathbf{x} \in \mathbb{R}^3$ be its projection on the camera coordinate system (also in the homogeneous coordinate). We have the following equation:

$$\lambda_k \mathbf{x}_k = \mathbf{P}_k \mathbf{X} = \mathbf{K}_k[\mathbf{R}_k|\mathbf{t}_k]\mathbf{X} \tag{1}$$

where λ_k is the scale coefficient to guarantee the last coordinate of $\mathbf{x}_k$ to be 1.

1.2 Fundamental Matrix and Essential Matrix

Now consider the setting of two-view reconstructions. The fundamental matrix $\mathbf{F} \in \mathbb{R}^{3\times3}$ (or the essential matrix $\mathbf{E} \in \mathbb{R}^{3\times3}$) is a matrix which relates corresponding points in the two view images. Let $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^3$ (both in the homogeneous coordinate) be corresponding points in an image pair, then the following equation holds:

$$\mathbf{x}_2^T \mathbf{F} \mathbf{x}_1 = 0 \tag{2}$$

And essential matrix relates calibrated corresponding image point, and it can be defined from the fundamental matrix as follows:

$$\mathbf{E} = \mathbf{K}_2^T \mathbf{F} \mathbf{K}_1 \tag{3}$$

The essential matrix relates with the relative camera pose as the following:

$$\mathbf{E} = \widehat{\mathbf{t}}\mathbf{R} \tag{4}$$

where $\widehat{\mathbf{t}}$ is the skew-symmetric matrix corresponding to the translation vector $\mathbf{t}$. Since we consider the relationship between the first and second view, equation 6 actually means $\mathbf{E} = \widehat{\mathbf{t}}_2\mathbf{R}_2$ in a verbose form.

1.3 Eight-Point Algorithm

This section will describe algorithm 5.1 from the vision textbook [1].

From equation 6, we know if we can estimate the essential matrix $\mathbf{E}$, then we can decompose it to translation and rotation matrix. Now we introduce the eight-point algorithm, which can be used to estimate the fundamental matrix or the essential matrix from multiple point correspondences.

From equation 2 and 3, we have the following relationship:

$$(\mathbf{K}_2^{-1}\mathbf{x}_2)^T \ \mathbf{E} \ (\mathbf{K}_1^{-1}\mathbf{x}_1) = 0. \tag{5}$$

The $\mathbf{K}_i^{-1}\mathbf{x}_i$ term can be viewed as a calibrated version of image points (assuming the focal length is 1, no skew, and no offsets) and we will use them (denote as $\tilde{\mathbf{x}}_i = \mathbf{K}_i^{-1}\mathbf{x}_i$) to solve for the essential matrix $\mathbf{E}$.

Now assume we have N matched points, and we use $\tilde{\mathbf{x}}_i^j = (\tilde{x}_i^j, \tilde{y}_i^j)$ to denote the coordinate of the j-th point in the i-th image. Then equation 5 can be written to:

$$\begin{bmatrix} \tilde{x}_2^j & \tilde{y}_2^j & 1 \end{bmatrix} \begin{bmatrix} e_{11} & e_{12} & e_{13} \\ e_{21} & e_{22} & e_{23} \\ e_{31} & e_{32} & e_{33} \end{bmatrix} \begin{bmatrix} \tilde{x}_1^j \\ \tilde{y}_1^j \\ 1 \end{bmatrix} = 0. \tag{6}$$

This means each of the N point will give one constraint of the solution for the $\mathbf{E}$ matrix. And we can rewrite the N constraint as the following linear system form:

$$\mathbf{A}\mathbf{E}^s = 0$$

, where $\mathbf{E^s}$ is formed from the entries of $\mathbf{E}$ stacked to a 9 dimensional vector column-wise:

$$\mathbf{E}^s = [e_{11}, e_{21}, e_{31}, e_{12}, e_{22}, e_{32}, e_{13}, e_{23}, e_{33}]^T \tag{7}$$

and $\mathbf{A} \in \mathbb{R}^{N\times 9}$. In particular, the i-th row of $\mathbf{A}$ is:

$$A_i = [\tilde{x}_1^i\tilde{x}_2^i \ , \ \tilde{x}_1^i\tilde{y}_2^i \ , \ \tilde{x}_1^i \ , \ \tilde{y}_1^i\tilde{x}_2^i \ , \ \tilde{y}_1^i\tilde{y}_2^i \quad \tilde{y}_1^i \quad \tilde{x}_2^i \quad \tilde{y}_2^i \quad 1] \tag{8}$$

The linear system $\mathbf{A}\mathbf{E}^s = 0$ will have an exact non-zero solution if $\text{rank}(\mathbf{A}) = 8$, which would happens if the point correspondences are exact. But usually due to noise, $\text{rank}(\mathbf{A}) > 8$. In that case, there is no exact solution to the linear system and an approximate solution has to be found. An approximate solution can be found by solving the following optimization problem:

$$\begin{aligned} &\min_f ||\mathbf{A}\mathbf{E}^s||_2 \\ &\text{s.t. } ||\mathbf{E}^s||_2 = 1 \end{aligned} \tag{9}$$

, which can be solved by using the singular value decomposition of the matrix A.

1.3.1 Finding linear system solution by SVD.

We show that the solution of the above optimization problem can be calculated using SVD: Let $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$ be the SVD of $\mathbf{A}$. Then let $\mathbf{y} = \mathbf{V}^T\mathbf{E}^s$. Note that since $\mathbf{V}$ is a rotation matrix (so it preserves norm), $||\mathbf{y}||_2 = 1$. Therefore, the above optimization problem can be written as follows:

$$\begin{aligned} &\min_f ||\mathbf{U}\mathbf{\Sigma}\mathbf{y}||_2^2 = ||\mathbf{\Sigma}\mathbf{y}||_2^2 \\ &\text{s.t. } ||\mathbf{y}||_2 = 1 \end{aligned} . \tag{10}$$

Then we can write the objective function $||\mathbf{\Sigma}\mathbf{y}||_2^2 = \sum \sigma_i^2 y_i^2 \geq \sigma_9^2 \sum y_i^2 = \sigma_9^2$, where $\sigma_1 \geq \cdots \geq \sigma_9$ are the singular values of $\mathbf{A}$. And the equality holds if and only if $y_9 = \pm 1$, which means $\mathbf{E}^s = \mathbf{V}\mathbf{y}$. So $\mathbf{E}^s$ is the last column vector of $\mathbf{V}$. Here we intentionally skip the solution corresponding to $y_9 = -1$. This does not affect our final solution because the sign of the $\mathbf{E}$ matrix is inherently unknown, which will be incorporated in the four-fold ambiguity discussion in the next section.

1.4 Extraction of Rotation and Translation Matrix

After we find the solution $\mathbf{E}^s$, we can unstack it back to the $\mathbf{E}$.

It can be shown (see Chapter 5.2 of [1] for the detailed derivation) that the recovered relative pose for the second camera matrices are:

$$(\widehat{\mathbf{t}}, \mathbf{R}) = (\mathbf{U}\mathbf{R}_z(\frac{\pi}{2})\mathbf{\Sigma}\mathbf{U}^T, \mathbf{U}\mathbf{R}_z^T(\frac{\pi}{2})\mathbf{V}^T) \tag{11}$$

$$(\widehat{\mathbf{t}}, \mathbf{R}) = (\mathbf{U}\mathbf{R}_z(\frac{\pi}{2})\mathbf{\Sigma}\mathbf{U}^T, \mathbf{U}\mathbf{R}_z^T(-\frac{\pi}{2})\mathbf{V}^T) \tag{12}$$

$$(\widehat{\mathbf{t}}, \mathbf{R}) = (\mathbf{U}\mathbf{R}_z(-\frac{\pi}{2})\mathbf{\Sigma}\mathbf{U}^T, \mathbf{U}\mathbf{R}_z^T(\frac{\pi}{2})\mathbf{V}^T) \tag{13}$$

$$(\widehat{\mathbf{t}}, \mathbf{R}) = (\mathbf{U}\mathbf{R}_z(-\frac{\pi}{2})\mathbf{\Sigma}\mathbf{U}^T, \mathbf{U}\mathbf{R}_z^T(-\frac{\pi}{2})\mathbf{V}^T) \tag{14}$$

, $\mathbf{\Sigma} = \text{diag}(1, 1, 0)$, $\mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$, and $\mathbf{E} = \mathbf{U}\mathbf{S}\mathbf{V}^T$ be the SVD of the essential matrix.

1.5 Triangulation

Triangulation refers to the estimation of the 3D position of a point when the corresponding points in the two image planes are given as well as the parameters of the camera. In particular, assume $\mathbf{x}_1$ and $\mathbf{x}_2$ are two corresponding points in the image plane and the camera matrices are $\mathbf{P}_1$ and $\mathbf{P}_2$ respectively. We want to find the 3D point $\mathbf{X} = (X, Y, Z, 1)$ such that $\lambda_1 \mathbf{x}_1 = \mathbf{P}_1 X$ and $\lambda_2 \mathbf{x}_2 = \mathbf{P}_2 X$. This is equivalent to:

$$\lambda \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{15}$$

This implies:

$$x_i = \frac{p_{11}X + p_{12}Y + p_{13}Z + p_{14}}{p_{31}X + p_{32}Y + p_{33}Z + p_{34}} \tag{16}$$

$$y_i = \frac{p_{21}X + p_{22}Y + p_{23}Z + p_{24}}{p_{31}X + p_{32}Y + p_{33}Z + p_{34}} \tag{17}$$

It can be further turned into

$$(p_{11} - p_{31}x_i)X + (p_{12} - p_{32}x_i)Y + (p_{13} - p_{33}x_i)Z = p_{34}x_i - p_{14} \tag{18}$$

$$(p_{21} - p_{31}x_i)X + (p_{22} - p_{32}x_i)Y + (p_{23} - p_{33}x_i)Z = p_{34}y_i - p_{24} \tag{19}$$

Because we have $i = 1, 2$, there are four equations and three unknowns. So the above system can be turned into a linear system and can be solved by SVD (using the method in section 1.3.1).

1.6 Re-projection error

We have now spoken about how to infer the structure of a scene (the 3D locations of the point features written in the world frame, which is typically taken to be the reference frame of camera 1), and the motion of the camera (the transformation $(R_i, T_i) \in SE(3)$ between camera frame 1 and camera frame i for each i), at least in the case of two views. Now, we will talk about how to measure the quality of this reconstruction. This is typically quantified by the "reprojection error".

Say we have estimated the locations $\{f_i^{(1)}\}_{i=1}^m \subset \mathbb{R}^3$ of m features in the reference frame of camera 1. We have also estimated the transforms (R_i, T_i) for $i = 1, \cdots, n$ for each of the n cameras. Say that the image observation of feature i in camera j is given by $z_{ij} \in \mathbb{R}^2$. This is the original set of image-features that we used to come up with our reconstruction.

Next, let $\pi_i : \mathbb{R}^3 \to \mathbb{R}^2$ be the function that projects a point written in the reference frame of camera 1 into the image plane of camera i. This is just the composition of a transformation into the reference frame of camera i followed by a pinhole projection using the camera matrix K. Note that to transform a point $p \in \mathbb{R}^3$ from camera frame 1 to camera frame i, we need to do $p' = R_i p + T_i$.

Finally, we can define the *total re-projection error* of the system to be

$$r = \sum_{i=1}^{n} \sum_{j=1}^{m} \left\| z_{ij} - \pi_i \left(f_j^{(1)} \right) \right\|^2$$

Geometrically, this is the total distance between the predicted location of the image measurements (assuming our reconstruction is correct) and the actual measurements. If this number is small, then our reconstruction is consistent with the measurements, and hence we are justified in trusting it. In the next section, we will introduce the factorization algorithm for multi-view re-construction, in which we will use the re-projection error to terminate the algorithm. i.e. we will terminate when the re-projection error becomes small enough.

Note that we can also come up with metrics other than *total* re-projection error, such as the average reprojection error over all measurements, or average re-projection error over image frames.

1.7 Factorization Algorithm

Now we can successfully recover R_2 and T_2 from the first two views. Next, we describe how to use information from more than one views. This section will describe algorithm 8.1 from the vision textbook [1]. Please refer to the relevant section of the textbook for more details.

In this algorithm, we will iteratively: 1) estimate the 3D location of a point; 2) estimate the camera rotation and translation using the depth information. Recall that in class, we learnt the relationship $\lambda_i^j \mathbf{x}_i^j = \lambda_1^j \mathbf{R}_i \mathbf{x}_1^j + \mathbf{t}_i$ (start from here, we will assume the camera is calibrated to avoid writing $\tilde{\mathbf{x}}$ for every equation). We can multiply both sides by $\widehat{\mathbf{x}_i^j}$ so that we will have the following equation $\frac{1}{\lambda_1^j}\widehat{\mathbf{x}_i^j}\mathbf{t}_i + \widehat{\mathbf{x}_i^j}\mathbf{R}_i\mathbf{x}_1^j = 0$. We define $\alpha_j = 1/\lambda_1^j$ for notation convenience. So the final equation is:

$$\alpha^j \widehat{\mathbf{x}_i^j}\mathbf{t}_i + \widehat{\mathbf{x}_i^j}\mathbf{R}_i\mathbf{x}_1^j = 0 \tag{20}$$

And we already have the camera rotation and translation of $\mathbf{R}_2$ and $\mathbf{t}_2$ from the eight-point algorithm. we can calculate an initial estimation of α^j for each of the point by the following equation:

$$\alpha^j = -\frac{(\widehat{\mathbf{x}_2^j}\mathbf{t}_2)^T \widehat{\mathbf{x}_2^j}\mathbf{R}_2\mathbf{x}_1^j}{||\widehat{\mathbf{x}_2^j}\mathbf{t}_2||^2} \tag{21}$$

Using some linear algebra tricks (see the explanation of *Kronecker product* in Appendix A of MaSKS), we can re-write the equation 20 for each of the N points as follow:

$$\begin{bmatrix} \mathbf{x}_1^{1T} \otimes \widehat{\mathbf{x}_i^1} & \alpha^1 \widehat{\mathbf{x}_i^1} \\ \mathbf{x}_1^{2T} \otimes \widehat{\mathbf{x}_i^2} & \alpha^2 \widehat{\mathbf{x}_i^2} \\ \vdots & \vdots \\ \mathbf{x}_1^{nT} \otimes \widehat{\mathbf{x}_i^n} & \alpha^n \widehat{\mathbf{x}_i^n} \end{bmatrix} \begin{bmatrix} \mathbf{R}_i^s \\ \mathbf{t}_i \end{bmatrix} = 0 \in \mathbb{R}^{3n} \tag{22}$$

, where $\mathbf{R}_i^s$ denotes the stacked vector (column-wise, same order as $\mathbf{E}^s$ above) of rotation matrix $\mathbf{R}_i$, and $\otimes$ means the Kronecker product of the matrix. In the above, the only unknowns are $\mathbf{R}_i$ and $\mathbf{t}_i$. This is again a linear system as in equation 6, and we can use the technique introduced in 1.3.1 to solve it.

Once we have a solution, we can unstack the solution vector $\tilde{\mathbf{R}}_i^s$ into a resulting 3x3 matrix $\tilde{\mathbf{R}}_i$. However, this may not be in the space of $SO(3)$, so we need to project it onto the correct space. Let $\tilde{\mathbf{R}}_i = \mathbf{U}_i\mathbf{S}_i\mathbf{V}_i^T$ be the SVD of $\tilde{\mathbf{R}}_i$. Then the solution of equation 22 in $SO(3) \times \mathbb{R}^3$ is:

$$\mathbf{R}_i = \text{sign}(\det(\mathbf{U}_i\mathbf{V}_i^T))\mathbf{U}_i\mathbf{V}_i^T \tag{23}$$

$$\mathbf{t}_i = \frac{\text{sign}(\det(\mathbf{U}_i\mathbf{V}_i^T))}{\sqrt[3]{\det(\mathbf{S}_i)}}\tilde{\mathbf{t}}_i \tag{24}$$

Recall that we only use two views to compute the α. Now we can compute the refined depth value by utilizing information from more views using the following equation:

$$\alpha^j = -\frac{\sum_{i=2}^m (\widehat{\mathbf{x}_i^j}\mathbf{t}_i)^\top \widehat{\mathbf{x}_i^j}\mathbf{R}_i\mathbf{x}_1^j}{\sum_{i=2}^m ||\widehat{\mathbf{x}_i^j}\mathbf{t}_i||^2} \tag{25}$$

At this point, we are equipped with all the equations to implement the factorization algorithm for recovering all the rotation and translation matrix. The whole algorithm can be summarized as follows:

1. Initialization.
   (a) Compute $\mathbf{R}_2$ and $\mathbf{t}_2$ using the eight-point algorithm from the first two views.
   (b) Compute α^j using equation 21.
   (c) Remember that we recover the depth only up to a scale. Thus we can choose to use the first point as a reference point: (let $\alpha^1 = 1$) by normalizing the α: $\alpha^j = \alpha^j/\alpha^1$

2. Compute $\mathbf{R}_i$ and $\mathbf{t}_i$ using equation 22 and 23.
3. Compute the refined α^j using equation 25. Again normalize it by using $\alpha^j = \alpha^j/\alpha^1$ and $\mathbf{t}_i = \alpha^1\mathbf{t}_i$ (we need to adjust the scale of $\mathbf{t}$ because we need equation 20 holds).
4. Using the new $\alpha^j = 1/\lambda_1^j$, $\mathbf{R}_i$, $\mathbf{t}_i$, compute the re-projection error for all views.
5. If the error is larger than a given threshold, go to step 2 and iterate again with the new values of α^j.

In the above algorithm, you can either decide when to terminate by setting a threshold on the total reprojection error, or you can terminate after a set number of iterations, which you have determined to be sufficient for convergence.

1.8 Image Feature Matching

In the previous sections, we talked about how to do reconstruction given point correspondences. However, in reality, the point correspondence is unknown and we need to estimate that. Normally, this pipeline is divided into three parts:

1. Detect the keypoints and get the features of the keypoints. There are a huge amount of research in this field: Harris, SIFT, SURF, ORB. In this project, we recommend you using SIFT to do feature detection and description. Read https://docs.opencv.org/master/da/df5/tutorial_py_sift_intro.html to learn the concept of SIFT and how to implement a SIFT feature extractor.
2. Feature Matching: After you get the feature vector of some candidates keypoints from each image, you need to find which point in the first image matches to which point in the second image. Normally this is done by a simple k-nearest neighbour method.
3. Since the observations are usually noisy, we need to run an extra step to filter out the false positives. Normally this is done by Random sample consensus (RANSAC). RANSAC is an iterative algorithm to estimate parameters by random sampling of observed data. Since we only need a few correct matched points to calculate the homography / fundamental matrix between images, we can randomly sample several points and compute the transformation. After that, we will count the inliers and outliers for this specific transformation. After doing this procedure several times, we could find a good homography / fundamental matrix in terms of the largest number of inliers. Using that, we can then filter out all the outliers and consider the remaining one as correct matched points. Read this document to learn how RANSAC is used in practical applications: https://docs.opencv.org/master/d1/de0/tutorial_py_feature_homography.html and https://www.youtube.com/watch?v=MQbJn6b7zmc&ab_channel=The106Channel (from 25:56). Note that in the first document, the final goal is to recover a homography between images. However, you are only required to learn how to use the `cv2.findHomography` function to implement outlier removal. The `cv2.findFundamentalMat` function works similarly, and can also be used for outlier removal. It is up to you to pick which one better suits your purpose. You may also choose to switch between the two based on which dataset you are working with.

1.8.1 Two-view correspondences

Given two images, we can find point-feature correspondences between them using the pipeline described above. i.e. we first extract features and descriptors in each image, come up with a preliminary set of matches using nearest neighbours on the descriptor vectors, and then use an outlier rejection technique like RANSAC to refine the set of matches to a set of good matches.

1.8.2 Multiple-view correspondences

However, recall that in order to use the factorization algorithm to perform multi-view reconstruction, we need to find features that are visible in all images, not just any two of them. One way to do this is through the use of *sequential feature tracks*.

Feature tracks are sequences of image features from consecutive images which are assumed to arise from the same point in 3D space. Say we have a sequence of n images $I_1, \cdots, I_n$. First, we extract keypoint features in each image.

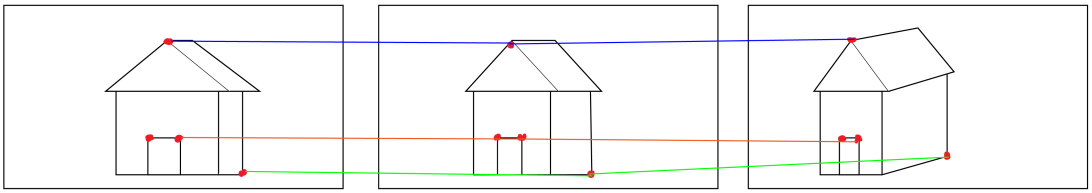


Figure 1: Sequential feature tracks of length 3. These tracks are extracted by first finding correspondences between consecutive frames and then tracing these correspondences across multiple frames. The feature points connected by blue, red and green lines respectively are form three separate feature tracks.

Then, we find feature correspondences between *consecutive* image frames I_k, I_{k+1}. Any two feature-points that are matched with each other are assumed to belong to the same point in 3D space. Let $f_i^{(j)}$ be the jth feature found in the ith image. Say that we find f_k^p matches with f_{k+1}^q (from matching between I_k and I_{k+1}) and feature f_{k+1}^q matches with f_{k+2}^r (from matching between I_{k+1} and I_{k+2}). Then we assign $\{f_k^p, f_{k+1}^q, f_{k+2}^r\}$ to the same *feature track*, and assume that these features are all observations of the same point in 3D space. This is then a feature track of length 3 (since it spans 3 images). If it is found that feature f_{k+2}^r matches with some feature in I_{k+3}, then this new feature will also be added to this feature track, and the track will become length 4. So, to find features that are visible in all n images, we must seek feature tracks of length n with span starting from the first image. See figure (1).

2 The Data

In this project you will get to play around with two separate datasets of images.

1. The "city" dataset contains image sequences taken in a synthetic outdoor urban environment. For this dataset, we will have access to ground-truth point-features *and* ground-truth pairwise correspondences between any two images. Since we have ground-truth features, you will be able to directly test 3D reconstruction algorithms in this dataset without having to implement any feature-extraction or matching pipeline yourself.
2. The "house" dataset consists of a sequence of 10 images taken of a real dollhouse. Since these are real images, there is no ground-truth feature-correspondence available for it. To work with this dataset, you will implement your own feature extraction and matching algorithms using OpenCV.

3 Project Tasks

You will implement a number of 3D reconstruction and feature matching tasks. In the first two tasks, you will be given a set of matched points from two images (e.g. the correspondence is ground-truth) from the city dataset. Based on that, you will need to implement the eight-point algorithm and the factorization algorithm. Since the correspondence is accurate, you can use them to debug your algorithm and ensure your implementation is correct.

After that, you need to implement your own feature detection and matching algorithm on the "house" dataset. Then you can run the same 3D reconstruction algorithm as above to see whether your feature matching is reasonably good.

At this point, you may wondering why we use two different datasets for different tasks. Now, you need to try to run the same pipeline on the city dataset again. Describe your findings. Does the above pipeline still work in the city dataset? If not, why?

Finally, you will need to implement the 3D reconstruction again. This time the keypoints are still given, however, they are slightly perturbed and the correspondence is not given. Implement your own feature descriptor and matching process to get a good reconstruction result.

3.1 Two-View 3D Reconstruction using given matched points

Preliminary. Firstly, you need to make sure you understand the format of input data.

Try to run `python main.py --dataset city --images data/city/000_0.jpg data/city/001_0.jpg`. The output should be:

The color of each dot indicating the id for each keypoints. The image coordinate is $(0, 0)$ at the left-top corner. Each keypoint is specified by their horizontal and vertical coordinates (x, y).

Your first task is to implement the `eight_point_algorithm` and `triangulation` functions in `reconstruct.py` to recover the relative pose of the two cameras. After implementing these two functions, you should be able to recover the correct camera pose as well as the 3D positions of each keypoint.

1. The `eight_point_algorithm` function takes as input corresponding lists of keypoint feature locations in two images along with the camera matrix K of the camera. It should then use the eight-point algorithm to return the four possible relative poses (R, T) between the two cameras. This is algorithm 5.1 from the textbook *An Invitation to 3D Vision* (MaSKS).
2. The `triangulation` function takes as input lists of corresponding feature points between two images, along with candidate poses (R_i, T_i), and the camera matrix K. It then computes the 3D coordinates of each corresponding pair of points in the first camera's reference frame for each of the candidate poses. Then, it returns that pose (R_j, T_j) which assigns a positive depth to every feature point (i.e. that puts all feature points in front of the camera). This is a way to filter out the candidate poses generated by the eight-point algorithm to find the physically possible pose (which is hence the only correct one).

Now we can verify the correctness of your reconstruction algorithm. Firstly, we will do it by verifying the correctness by re-projection. Since you already have the 3D positions (with respect to the first camera's coordinate frame), you can transform it to the second camera's coordinate frame and project it to the image space.

Implement the `visualize_reprojection` function in `utils/vis.py` to **Show the projected points and the original point in camera 2's image space and calculate the average reprojection error of these points.**

Our expectation is that after running

```
python main.py --dataset city --images data/city/000_0.jpg data/city/001_0.jpg
```

it will output a 3D plots of the keypoint features in the scene.

3.2 Multiple-View 3D Reconstruction using given matched points

Your second task is to implement the `factorization_algorithm` in `reconstruct.py`. Use the following command:

```
python main.py --dataset city --images data/city/*0.jpg
```

The function is expected to return the rotation and translation matrix for all views. Again, in this task, you are required to verify the correctness by reprojecting your recovered 3D points to the second, third, and the fourth image. Visualize and report the average reprojection errors.

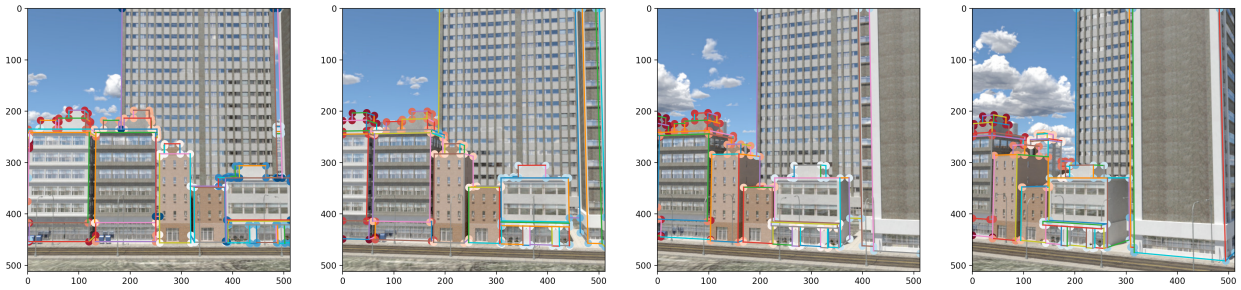


Figure 2: An example of the wireframe representation.

3.3 Multiple-View Reconstruction using detected features and keypoints

In the previous tasks, the correspondence points are given, which simplify the task a lot. In this section, you will detect the keypoints and do feature matching by yourselves. You are going to implement the following functions in `correspondence.py`.

1. `extract_features`: This function should use an OpenCV feature extractor (of type `cv2.Feature2D`) to extract keypoint feature locations and feature descriptors for each keypoint. Refer to the OpenCV tutorials on doing feature extraction linked in section 1.7.

2. `match_two_images`. This function takes in a list of keypoints and feature descriptors from two separate images and finds correspondences between them by running a feature matching routine. Once again, refer to the tutorials linked in section 1.7. Here, you should use OpenCV functions to first find nearest neighbours matches between the two lists of features, and then refine these matches using an outlier rejection scheme such as RANSAC. We urge you to seek out OpenCV documentation and other sources for more information. Note that you can use either fundamental matrix-based RANSAC or homography-based RANSAC. It is up to you to pick between the two and to specify the RANSAC parameters. Note that you may also choose to switch between the two based on which dataset you are working with. You should make it clear in the report what you are choosing and why.

3. `match_n_images`. This function finds multiple-view feature matches in the form of feature-tracks. Here, we seek to extract features from a list of input images, and then locate features that correspond to points in the scene that are visible in all images. Read the comments in the python file for more information.

Once you have implemented your feature matching code, you can test out your reconstruction algorithms using these extracted features on the house dataset. To test two-view reconstruction (using just the eight-point algorithm) run the code with the `--sift` flag and two images from the dataset.

```
python main.py --dataset house --images data/house/000.pgm data/house/003.pgm --sift
```

Likewise, you can test your multi-view reconstruction by specifying more images from the dataset. Once again, you should visualize the re-projection errors of your reconstruction and discuss them in your report.

3.4 Extra Credit: Wireframe Matching

The SIFT features we have been using in this project are local image features, and hence only hold information about small image patches. In highly structured scenes however, we may ask if we can take advantage of more higher-level features that can capture the global structure of the scene. In this extra credit task, you will play around with one such representation, the wireframe.

Wireframe. We call the collection of straight lines and their intersections (called junctions) as wireframe, as shown in Figure 2. Instead of focusing on features within a small local region, this kind of representation focuses on

the global scene structure and is usually more robust. The wireframe detection problem is usually done using a learning based approach [2].

In this task, we are going to provide the wireframes of each image (you are not required to do the wireframe detection), however, the correspondence between different images is unknown. Can you find the correspondence by utilizing both the geometric and semantic information?

Read the comments of `load_city_raw_wireframes` function in `utils/dataset.py` and the `vis_2d` function in `utils/vis.py`. This function provides you the junctions and lines for each image in the city dataset. And the correct label is specified in the `junc_ids` variable.

You task is to get the correspondence between the junctions in different images. Then, you can check how well you did by checking to see if the junction IDs of your candidate correspondences match. If they match, then they were indeed images of the same junction, and so that was a correct correspondence detection. In the final report, you should report your accuracy.

Here are a few hints and suggestions that you could try. Note that none of these are guaranteed to be full solutions, and we expect that you will have to do some novel work in order to get good results.

1. You can try to compute standard SIFT (or other) feature descriptors around the junctions and then use the same matching process as before. Note that in the tutorials linked above, we generally make a call to `sift.detectAndCompute`, which returns both detected keypoints and their descriptors in the same step. However, OpenCV also provides a method `compute` for its feature extractors, which takes in a list of keypoints and returns a list of feature descriptors for those keypoints. You could try to use this to treat the wireframe junctions directly as your keypoints.
2. You can try to leverage the connectivity information between points. For instance, if you have candidate matches $f_1 \leftrightarrow f_2$ and $g_1 \leftrightarrow g_2$, but (f_1, g_1) are connected by a line in the image 1 wireframe and (f_2, g_2) are not connected, then this set of matches is likely to be incorrect. Can you leverage more complex incidence relations between the graphs to find matches than simple pairwise connectivity?
3. You could leverage the work we have already done by extracting dense SIFT-like feature points and estimating the homography or fundamental matrix with them. Then, use this homography or fundamental matrix to filter matches amongst the wireframe junctions.

Since this task is extra credit, we have not provided you with much starter code. We have given you a function stub to fill in in `correspondence.py`, but this function is not called anywhere, so you will have to write code to call this function, make any necessary visualizations, and evaluate your performance yourself. You should document all of this in your report.

4 Deliverables

To demonstrate that your implementation works, deliver the following in a report. The purpose of these project reports are to gradually scale up to a full conference-style paper, which you'll be writing for your final project. Please format your report using the IEEE two-column conference template. Column suggestions do not account for the figures.

1. Abstract: An abstract, of at most 150 words, describing what you did and what results you achieved. Conveying information concisely is a difficult skill, and we want you to practice it here.
2. Methods:
   (a) Describe any implementation details you deem necessary to understand your implementations of the eight-point algorithm, the factorization algorithm, and the two-view triangulation algorithm. Note that you do not need to explain each algorithm in detail (i.e. you do not need to repeat anything that is in this document). For instance, you should state the termination criteria you use for factorization algorithm. ($<$ 1 column)
   (b) Describe how you computed and measured the re-projection error of your reconstruction. ($\sim$ 1 column)

(c) Describe the parameters of your two-view feature matching algorithm. What kind of feature extractor did you use? Did you use homography estimation or fundmental matrix estimation for RANSAC-based outlier rejection? Note that you can vary your outlier rejection strategy based on the datatset. When would you use one versus the other? State the parameters you chose to use for RANSAC. ($\sim$ 1 column)

(d) Describe your algorithm for computing feature tracks for multi-view correspondence. ($<$ 1 column)

(e) (Bonus) Describe your algorithm for finding correspondences between wireframes. ($\sim$ 1-2 columns)

3. Experimental Results ($<$ 1 column + figures):

(a) In Task 3.1, provide the plot of the reconstructed 3D points (at least from 3 different viewpoints) as well as the image showing both the reprojected points and the original points. Describe any experiments you ran to better understand the quality of your reconstruction.

(b) In Task 3.2, provide the plot of the reconstructed 3D points (at lease from 3 different viewpoints) as well as the image showing both the reprojected points and the original points. Describe any experiments you ran to better understand the quality of your reconstruction.

(c) In Task 3.3, show the results of your detected SIFT features, the matching results before RANSAC and after RANSAC. After that, show the plot of the reconstructed 3D points (at least from 3 different viewpoints).

(d) (Bonus) Demonstrate the functionality of your wireframe matching algorithm through visualizations and numerical accuracy.

4. Discussion ($\sim$ 2 columns):

(a) How well does your feature extraction scheme work? What kind of features does the SIFT extractor find? Are the features generally well distributed on the object of interest? In what kinds of scenes would SIFT be a good choice of feature extractor, and in what kinds of scenes might it fail?

(b) How well does your feature matching scheme work? Are you successful in eliminating all false-positive matches while still keeping a healthy number of good matches? How many good matches are you generally able to find for the house dataset in both the two-view and multi-view case? Discuss any failure cases that you have identified, and describe when they tend to happen.

(c) How effective is RANSAC at eliminating false-positive matches?

(d) Discuss the quality of your reconstruction in terms of the re-projection error and the plots of the re-projected feature points.

(e) Does the iteration in the factorization algorithm help in reducing reprojection error? Discuss this in the context of ground truth features and noisy hand-extracted SIFT features.

(f) How does the quality of the reconstruction vary between the ground-truth wireframe keypoints in the city dataset versus the manually-extracted SIFT keypoints in the house dataset?

(g) You have seen dealt with two kinds of feature extractors in this project: one is the SIFT keypoint feature extractor, and the other is a learning-based feature extractor that generates wire-frames from an input image. Discuss under what circumstances you might want to use one versus the other.

(h) (Bonus) What challenges did you encounter in designing your algorithm for finding wireframe correspondences? How did you address them? Discuss the performance of your feature matching algorithm.

5. Bibliography:

(a) Cite any sources you used from outside of this course to help you with this project, including any online sources or tutorials you consulted.

6. Appendix:

(a) Future Improvements: This is the first time this project is being assigned as part of 106B, so feedback is highly encouraged. How can the lab documentation and starter code be improved? We are especially looking for any comments you have regarding the pedagogical success of the assignment.

(b) Github Link: Share your private github repo with us at the email ID `berkeley.ee106b@gmail.com` and provide a link to it in your report.

(c) (Bonus) What do you think the learning goals of this assignment are? How effective was this assignment at fostering those learning goals for you?

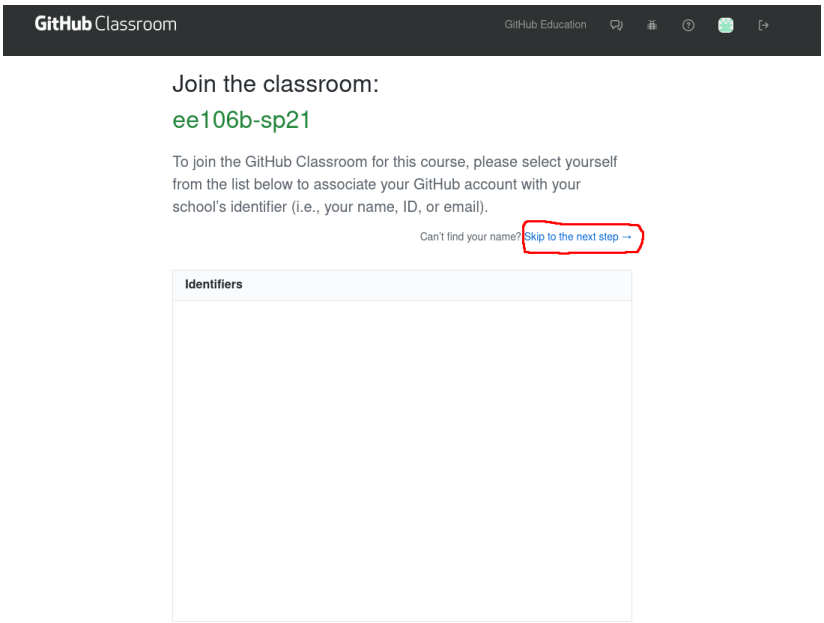
7. Page Limit: To prepare you for conference-style papers (and to protect the graders' time), reports are **limited to 6 pages**, not including the bibliography or appendix.

   (a) While we are not enforcing a figure limit, we expect your figures to be concise, informative, and readable, with detailed captions for each.

5 Getting Started

5.1 Github Classroom

For subsequent projects in this class, we will be using GitHub Classroom. To access the starter code, simply head over to the project 3 assignment page to accept the assignment. If you are asked to associate yourself with a school identifier, just click "Skip to the next step"



You should now be asked to create a team or join from a list of existing teams. If one of your teammates has already created a team, you should join that team instead of creating a new one. **After you have joined a team, you will not be able to switch teams by yourself. If you make a mistake or something else comes up that requires you to switch teams, let us know.** That's it! Your team should now have a private project repository located at https://github.com/ucb-ee106-classrooms/project-3-your-team-name. Let us know if anything went wrong.

5.2 Pulling starter code updates

If there are any updates to the starter code that you wish to pull you may do so with the commands

```
cd private-repo
git remote add public https://github.com/ucb-ee106/proj3_pkg.git
git pull public main
git push origin main
```

5.3 Environment Setup

In previous projects, you should already have your `pip` properly installed. Then install the following packages using `pip`:

```
pip install opencv-python
pip install opencv-contrib-python
```

5.4 Starter Code

- `main.py` This file is the entry point of all tasks. Please read the task section for example usages.
- `reconstruct.py` This file contains the majority reconstruction algorithms you need to implement.
- `correspondence.py` This file ask you to implement the feature matching mechanism. You will implement two-view feature matching and multi-view feature-tracking in this file. See the documentation in the file for more information.
- `utils/dataset.py` This file contains function for reading images (and matched points, only for the city dataset) from `data/`. You are not required to understand how it works, but you are required to know what is the output format. Make sure you understand how to use the output by visualize it. You won't need to modify this file.
- `utils/vis.py` This file holds a host of visualization functions. You are going to implement the `vis_3d` function in this file.

5.5 Implementation Tips

- The tasks in `reconstruct.py` and those in `correspondence.py` can be done in parallel. This is because you can test your reconstruction algorithms on the ground truth features in the city dataset and hence you don't need to wait until you have implemented feature matching and tracking.
- Start setting up and reading documentation early! If you haven't worked with OpenCV before, you have a much easier time if you start reading documentation and tutorials early so that you can focus on the meat of the project later.

6 Scoring

Table 1: Point Allocation for Project 3

Section	**Points**
Page Limit	10
Figures: Quality and Readability	10
Abstract	5
Methods: Two-view reconstruction	10
Methods: Multi-view reconstruction	10
Methods: Two-view feature matching	10
Methods: Multi-view feature matching	10
(Bonus) Methods: Wireframe matching	15*
Results	25
(Bonus) Results: Wireframe matching	5*
Discussion	10
Bibliography	5
Code	5
Bonus (learning outcomes)	5*

As per Table 1, this project will be out of 110 points, with an additional 25 points possible.

7 Submission

For this project we will be using Github classrooms for submission. Simply push your code to the private repository that was created for your team. We will be able to see any changes you push to your assignment repository.

8 References

[1] Y. Ma et al. *An invitation to 3-d vision: from images to geometric models*. Springer Science & Business Media, 2012.

[2] Y. Zhou, H. Qi, and Y. Ma. "End-to-end wireframe parsing". *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019.