AR/VR/XR Assignment #3 – Structure From Motion

1. Submission

- Assignment due: May 22 (11:59 pm)
- Individual assignment
- Up to 3 page summary write-up with resulting visualization (no more than 3 pages).
- Use Python 3
- You will complete the following functions:
    - hw4.py
    - feature.py
        - MatchSIFT
        - EstimateE
        - EstimateE_RANSAC
        - BuildFeatureTrack
    - camera_pose.py
        - GetCameraPoseFromE
        - Triangulation
        - EvaluateCheirality
        - EstimateCameraPose
    - pnp.py
        - PnP
        - PnP_RANSAC
        - ComputePoseJacobian
        - PnP_nl
    - reconstruction.py
        - FindMissingReconstruction
        - Triangulation_nl
        - ComputePointJacobian
        - SetupBundleAdjustment

        - MeasureReprojection
        - UpdatePosePoint
        - RunBundleAdjustment
    - utils.py
    - Quaternion2Rotation
    - Rotation2Quaternion
- You are not allowed to use computer vision related package functions unless ex- plicitly mentioned here. Please consult with TAs if you are not sure about the list of allowed functions.
- You can use the following script to install the dependencies:

  pip install -r requirements.txt
- The visualization code for the final result is provided. We you finish and run the pipeline, you will find output/cameras_i.ply and output/points_i.ply, which are the camera coordinate frames and reconstructed points, respectively. You can use MeshLab (https://www.meshlab.net) to open them.

2 Overview

Figure 1: You will implement structure from motion to reconstruct camera pose and 3D points.

In this assignment, you will use a set of images to reconstruct the 3D scene and camera poses by implementing structure from motion algorithm. This will include feature matching, camera pose estimation using fundamental matrix, camera registration, triangulation, and bundle adjustment. A nonlinear optimization is always followed by the initial estimate by linear least squares solution. The pseudo-code can be found in Algorithm 1.

Algorithm 1 Structure from Motion

```
Build feature track                                          ▷ BuildFeatureTrack
Estimate first two camera poses                              ▷ EstimateCameraPose
Initialize pose set P
for i = 2, ···, N-1 do
    Estimate new camera pose                                 ▷ PnP, PnP_nl
    P = P ∪ P_i
    for j < i do
        Find new points to reconstruct                       ▷ FindMissingReconstruction
        Triangulate point                                    ▷ Triangulation, Triangulation_nl
        Filter out point based on cheirality                 ▷ EvaluateCheirality
        Update 3D point
    end for
    Run bundle adjustment                                    ▷ RunBundleAdjustment
end for
```

3 Building Feature Track

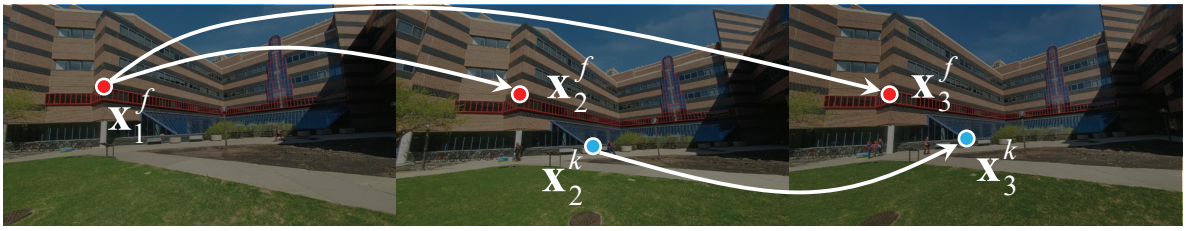


Figure 2: Given a set of images, you will build a tensor `track` that specifies the matches across views.

Given a set of images, $\texttt{Im} \in \mathbb{R}^{N\times H\times W\times 3}$, you will build a tensor $\texttt{track} \in \mathbb{R}^{N\times F\times 2}$ that specifies the matches across all views where N is the number of images, and F is the total number of features. Consider the f^{th} feature point (x, y) in the i^{th} image is stored in `track[i,f,:]`. If the point is matched to a point in the j^{th} image, the correspondence is stored in `track[j,f,:]`, i.e.,

$$\texttt{track[j,f,:]} = \begin{cases} \mathbf{x}_j & \text{if } \mathbf{x}_i \leftrightarrow \mathbf{x}_j \\ -1 & \text{otherwise} \end{cases}, \tag{1}$$

where $\mathbf{x}_i \leftrightarrow \mathbf{x}_j$ indicates a point correspondence between two views.

For instance, in Figure 2, $\mathbf{x}_1^f$ is matched to $\mathbf{x}_2^f$ and $\mathbf{x}_3^f$, and therefore, `track[1,f,:]` $= \mathbf{x}_1^f$, `track[2,f,:]` $= \mathbf{x}_2^f$, and `track[3,f,:]` $= \mathbf{x}_3^f$. On the other hand, $\mathbf{x}_2^k$ is only matched to $\mathbf{x}_3^k$, and therefore, `track[1,k,:]` = -1, `track[2,k,:]` $= \mathbf{x}_2^k$, and `track[3,k,:]` $= \mathbf{x}_3^k$. To build the `track`, you need to aggregate all pairwise matches. The pseudo-code can be found in Algorithm 2

Algorithm 2 BuildFeatureTrack

for `i` $= 0, \cdots, N-1$ **do**
  Extract SIFT descriptor of the i^{th} image, `Im[i]`
end for
for `i` $= 0, \cdots, N-1$ **do**
  Initialize `track_i` = -1$^{N\times F\times 2}$
  for `j` $= i+1, \cdots, N-1$ **do**
    Match features between the i^{th} and j^{th} images ▷ `MatchSIFT`
    Normalize coordinate by multiplying the inverse of intrinsics.
    Find inliner matches using essential matrix ▷ `EstimateE_RANSAC`
    Update `track_i` using the inlier matches.
  end for
  Remove features in `track_i` that have not been matched for $i+1, \cdots, N$.
  `track` = `track` $\cup$ `track_i`
end for

Figure 3: You can visualize epipolar lines for each image to validate the fundamental matrix.

```
def EstimateE(x1, x2):
  ...
  return E
```

Input: `x1` $\in \mathbb{R}^{n\times 2}$ and `x2` $\in \mathbb{R}^{n\times 2}$ are the set of correspondences.
Output: `E` $\in \mathbb{R}^{3\times 3}$ is an essential matrix.
Description: The essential matrix must be a rank 2 matrix and its singular values must be $(1, 1, 0)$. Use SVD to set the singular values.

```
def EstimateE_RANSAC(x1, x2, ransac_n_iter, ransac_thr):
  ...
  return E, inlier
```

Input: `x1` $\in \mathbb{R}^{n\times 2}$ and `x2` $\in \mathbb{R}^{n\times 2}$ are the set of correspondences. `ransac_n_iter` and `ransac_thr` are the number of iterations and the error threshold for RANSAC.
Output: `E` $\in \mathbb{R}^{3\times 3}$ is an essential matrix, and `inlier` $\in \mathbb{Z}^k$ is the set of inlier indices where k is the number of inliers.
Description: You will implement 8-point algorithm (`EstimateE`) with RANSAC to find robust estimate of `E`. You can visualize the filtered inliers based on RANSAC as shown in Figure 3 (in order to visualize in pixel space, you have to *unnormalize* the normalized coordinates by multiplying the intrinsic parameter.).

Structure from Motion

```
def BuildFeatureTrack(Im, K):
  ...
  return track
```

Input: `Im` $\in \mathbb{R}^{N\times H\times W\times 3}$ are the set of images where N is the number of images, and H, W are height and width of images. `K` $\in \mathbb{R}^{3\times 3}$ is the intrisic parameter.
Output: `track` $\in \mathbb{R}^{N\times F\times 2}$ is a feature tensor where F is the number of total features.
Note: `track` is expected to include only inlier matches. The coordinates of `track` are expected to normalized by the intrinsic parameters `K`, which can improve numerical stability. You will use SIFT matching function that you implemented for HW1:

```
def MatchSIFT(loc1, desc1, loc2, desc2):
  ...
  return x1, x2, ind1
```

This SIFT matching includes ratio test and bidirectional consistency check. Unlike HW1, you will return not only `x1`, `x2` $\in \mathbb{R}^{m\times 2}$ but also `ind1` $\in \mathbb{Z}^m$, the indices of `x1` in `loc1`, i.e., `loc1[ind1,:] = x1`. m is the number of matches.

4 Estimating Camera Pose

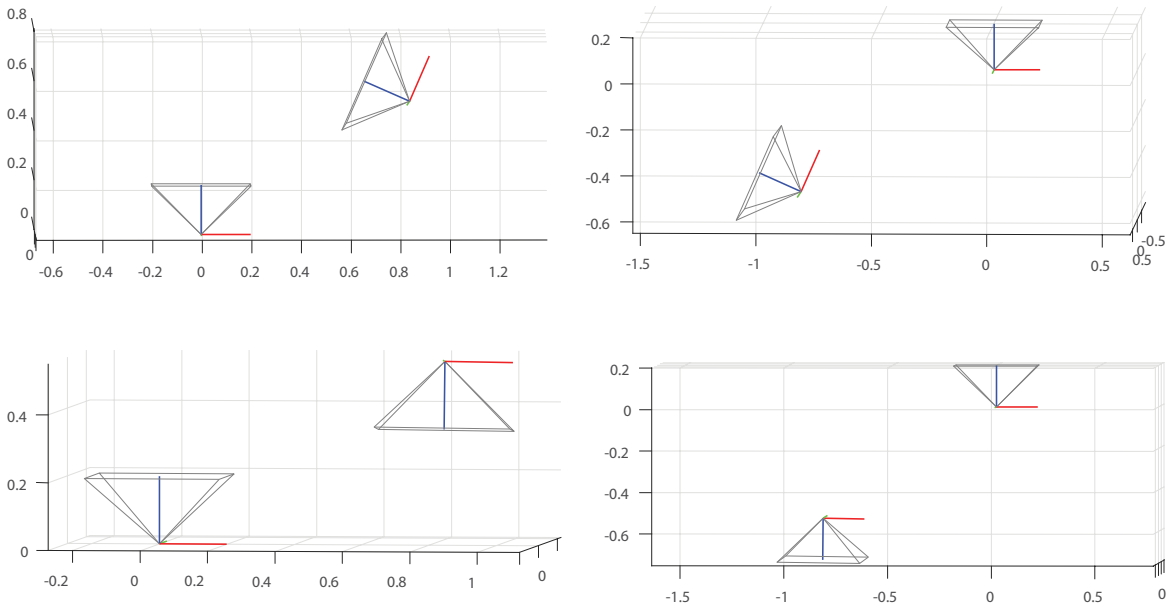


Figure 4: Four configurations of camera pose from a fundamental matrix.

Given an essential matrix, you can compute find four configurations of camera poses as shown in Figure 4. Given the four camera configurations, you will find the best configuration by taking into account cheirality, i.e., the triangulated point must be in front of both cameras. Pseudo-code of pose estimation can be found in Algorithm 3

Algorithm 3 EstimateCameraPose

1: Compute essential matrix given tracks ▷ `EstimateE_RANSAC`
2: Estimate four configurations of poses ▷ `GetCameraPoseFromE`
3: **for** `i` $= 0, 1, 2, 3$ **do**
4: Triangulate points using for each configuration ▷ `Triangulation`
5: Evaluate cheirality for each configuration ▷ `EvaluateCheirality`
6: **end for**

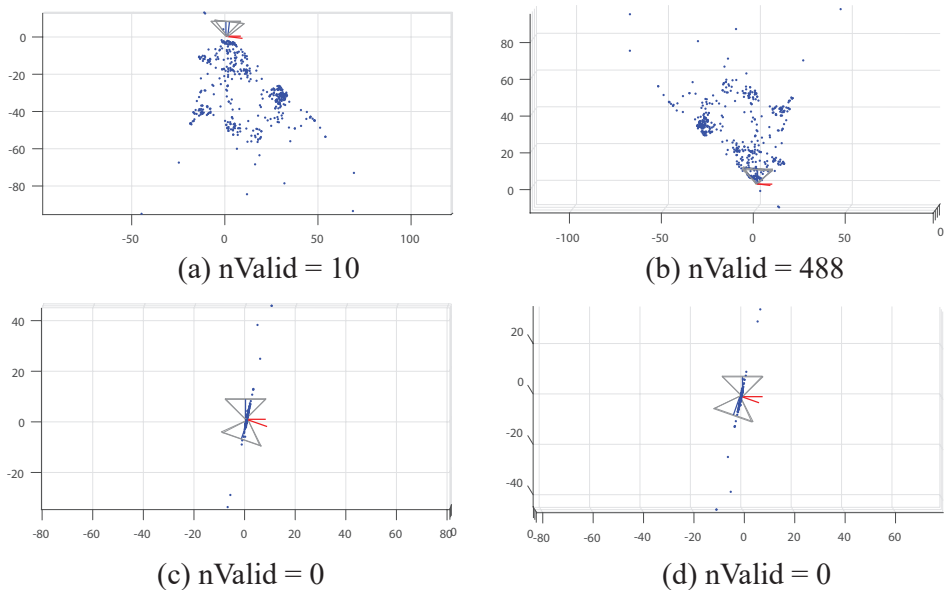


Figure 5: Visualization of camera configurations and their point clouds. (b) produces the maximum valid points that satisfy cheirality condition.

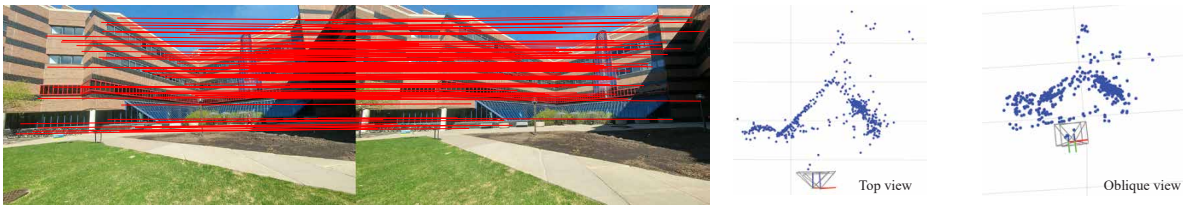
```
def GetCameraPoseFromE(E):
  ...
  return R_set, C_set
```

Input: `E` $\in \mathbb{R}^{3\times3}$ is rank 2 essential matrix.
Output: `R_set` $\in \mathbb{R}^{4\times3\times3}$ is the set of four rotation matrices, and `C_set` $\in \mathbb{R}^{4\times3}$ is the set of four camera centers.
Description: Given an essential matrix, you can find four configurations of rotation and camera center.

```
def Triangulation(P1, P2, track1, track2):
  ...
  return X
```

Input: `P1`, `P2` $\in \mathbb{R}^{3\times4}$ are two camera projection matrices, and `track1`, `track2` $\in \mathbb{R}^{n\times2}$ are point correspondences from the two poses. n is the number of 2D points.
Output: `X` $\in \mathbb{R}^{n\times3}$ is the set of 3D points. For the invalid point matches (`track1`=-1 or `track2`=-1), the corresponding row of `X` can be set to -1 to indicate invalid reconstruction.
Description: You will use the linear triangulation method to triangulation the point.

Structure from Motion



(a) Inlier matches (b) 3D camera pose

Figure 6: Camera pose estimation.

```
def EvaluateCheirality(P1, P2, X):
  ...
  return valid_index
```

Input: `P1`, `P2` $\in \mathbb{R}^{3\times4}$ are two camera projection matrices, and `X` $\in \mathbb{R}^{n\times3}$ are the set of 3D points.

Output: `valid_index` $\in \{0,1\}^n$ is the binary vector indicating the cheirality condition, i.e., one if the point is in front of both cameras, and zero otherwise. You may visualize the camera and point cloud to validate cheirality as shown in Figure 5.

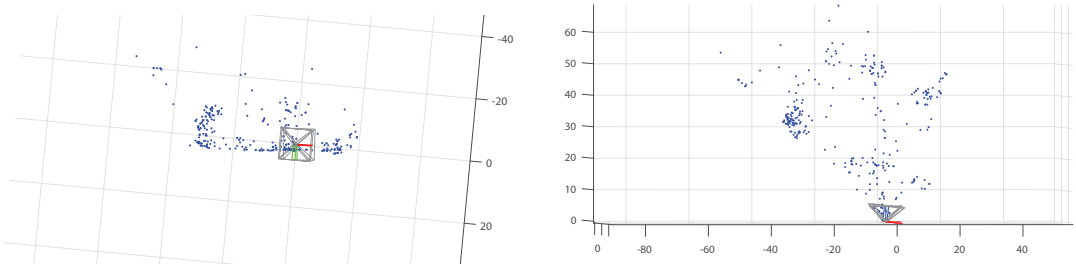
```
def EstimateCameraPose(track1, track2):
  ...
  return R, C, X
```

Input: `track1`, `track2` $\in \mathbb{R}^{F\times2}$ are correspondences.

Output: `R` $\in SO(3)$ is the rotation matrix (orthogonal matrix), `C` $\in \mathbb{R}^3$ is the camera center, and `X` $\in \mathbb{R}^{F\times3}$ is the set of 3D reconstructed points.

Description: You will return the best pose configuration of which the number of valid 3D points is maximum.

5 Perspective-n-Point algorithm



(a) Front view (b) Top view

Figure 7: You will implement a perspective-n-point algorithm to register a new image given 3D reconstruction of points.

Given 3D reconstruction of points, you will register a new image using 3D-2D correspondences using a linear perspective-n-point algorithm with RANSAC. This linear estimate will be refined by minimizing the reprojection error:

$$\underset{\mathbf{p}}{\text{minimize}} \quad \sum_{i}^{n} \|f(\mathbf{p}, \mathbf{X}_i) - \mathbf{b}_i\|^2, \tag{2}$$

where $\mathbf{p}$ is the vectorized camera pose made of the camera center and quaternion, i.e., $\mathbf{p} = \begin{bmatrix} \mathbf{C}^\mathsf{T} & \mathbf{q}^\mathsf{T} \end{bmatrix}^\mathsf{T}$. $f(\mathbf{p}, \mathbf{X}_i)$ is the projection of the i^{th} 3D point $\mathbf{X}_i$ onto the camera $\mathbf{p}$, i.e.,

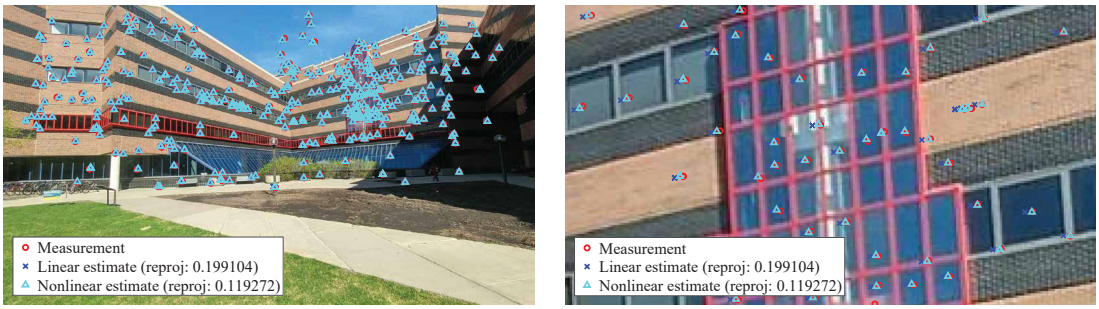
$$f(\mathbf{p}, \mathbf{X}_i) = \begin{bmatrix} u_i/w_i \\ v_i/w_i \end{bmatrix}, \qquad \text{where} \quad \begin{bmatrix} u_i \\ v_i \\ w_i \end{bmatrix} = \mathbf{R} \begin{bmatrix} \mathbf{I}_3 & -\mathbf{C} \end{bmatrix} \begin{bmatrix} \mathbf{X}_i \\ 1 \end{bmatrix} \tag{3}$$

where $\mathbf{X}_i \in \mathbb{R}^3$ is the i^{th} 3D point. $\mathbf{b}_i$ is the i^{th} 2D point from `track`, i.e., $\mathbf{b}_i = \begin{bmatrix} x_i & y_i \end{bmatrix}^\mathsf{T}$.

Equation (2) can be minimized by updating the camera pose $\mathbf{p}$ using the Levenberg-Marquardt method:

$$\mathbf{p} = \mathbf{p} + \Delta\mathbf{p}, \qquad \text{where } \Delta\mathbf{p} = \left(\sum_{i}^{n} \frac{\partial f_i}{\partial \mathbf{p}}^\mathsf{T} \frac{\partial f_i}{\partial \mathbf{p}} + \lambda \mathbf{I} \right)^{-1} \sum_{i}^{n} \frac{\partial f_i}{\partial \mathbf{p}}^\mathsf{T} (\mathbf{b}_i - f_i), \tag{4}$$

where $\frac{\partial f_i}{\partial \mathbf{p}} \in \mathbb{R}^{2\times 7}$ is the pose Jacobian for the i^{th} point (we denote $f(\mathbf{p}, \mathbf{X}_i)$ by f_i by an abuse of notation.). λ is the damping parameter, and you can try $\lambda \in [0, 10]$.



(a) Reprojection error (b) Close up

Figure 8: Nonlinear refinement reduces the reprojection error (0.19→0.11).

```
def PnP(X, x):
  ...
  return R, C
```

Input: `X` $\in \mathbb{R}^{n\times 3}$ is the set of reconstructed 3D points, and `x` $\in \mathbb{R}^{n\times 2}$ is the 2D points of the new image where n is the number of points.
Output: `R` $\in SO(3)$ is the rotation matrix (orthogonal matrix), `C` $\in \mathbb{R}^3$ is the camera center.
Description: You will implement linear perspective-n-point algorithm.

```
def PnP_RANSAC(X, x, ransac_n_iter, ransac_thr):
  ...
  return R, C, inlier
```

Input: `X` $\in \mathbb{R}^{n\times 3}$ is the set of reconstructed 3D points, and `x` $\in \mathbb{R}^{n\times 2}$ is the 2D points of the new image where n is the number of points. `ransac_n_iter` and `ransac_thr` are the number of iterations and threshold for RANSAC.
Output: `R` $\in SO(3)$ is the rotation matrix (orthogonal matrix), `C` $\in \mathbb{R}^3$ is the camera center, and `inlier` $\in \{0,1\}^{n\times 1}$ is indicator of inliers, i.e., one if the point is one of inliers, and zero otherwise.
Description: You will estimate pose using `PnP` with RANSAC. An inlier is the one that has a smaller error than the threshold and satisfies cheirality.

Structure from Motion

```
def Rotation2Quaternion(R):
  ...
  return q
```
Input: `R` $\in SO(3)$ is rotation matrix.
Output: `q` $\in \mathbb{R}^4$ is unit quaternion.
Note: You need to normalize `q` before return.

```
def Quaternion2Rotation(q):
  ...
  return R
```
Input: `q` $\in \mathbb{R}^4$ is unit quaternion.
Output: `R` $\in SO(3)$ is rotation matrix.
Note: `R` must be an orthogonal matrix.

```
def ComputePoseJacobian(p, X):
  ...
  return dfdp
```
Input: `p` $\in \mathbb{R}^7$ is the camera pose made of camera center ($\mathbb{R}^3$) and quaternion ($\mathbb{R}^4$). `X` $\in \mathbb{R}^3$ is the 3D point.
Output: `dfdp` $\in \mathbb{R}^{2\times 7}$ is the pose Jacobian, i.e., $\partial f_i / \partial \mathbf{p}$.

```
def PnP_nl(R, C, X, x):
  ...
  return R_refined, C_refined
```
Input: `R` $\in SO(3)$ and `C` $\in \mathbb{R}^3$ are the refined rotation and camera center by `PnP`. `X` $\in \mathbb{R}^{n\times 3}$ is the 3D points, and `x` $\in \mathbb{R}^{n\times 3}$ is the 2D point where n is the number of points.
Output: `R_refined` $\in SO(3)$ and `C_refined` $\in \mathbb{R}^3$ are the refined rotation and camera center via nonlinear optimization.
Note: You will use `ComputePoseJacobian` to update the pose. The pose update via Equation (6) does not enforce the quaternion to be unit length. After update, you have to enforce the unit length, i.e., $\mathbf{q} = \frac{\mathbf{q}}{\|\mathbf{q}\|}$. After optimization, you can visualize the reduction of reprojection error (0.19→0.11) as shown in Figure 8.

6 Reconstructing 3D Points

Given a newly registered image, you will reconstruct 3D points that have not reconstructed yet. You find the points that will be newly added and reconstruct the points using `Triangulation`. These linearly estimated points will be refined by minimizing the reprojection error for each point:

$$\underset{\mathbf{X}_j}{\text{minimize}} \quad \sum_{k=1}^{2} \|f(\mathbf{p}_k, \mathbf{X}_j) - \mathbf{b}_{k,j}\|^2, \tag{5}$$

where $\mathbf{X}_j$ is the j^{th} 3D point, $\mathbf{p}_k$ is the k^{th} camera pose for triangulation, and $\mathbf{b}_{k,j}$ is the j^{th} 2D point on the k^{th} image from `track`. We consider $k = 1, 2$ where a point is triangulated by two camera poses.

Equation (5) can be minimized by updating the camera pose $\mathbf{X}$ using the Levenberg-Marquardt method:

$$\mathbf{X}_j = \mathbf{X}_j + \Delta\mathbf{X}_j \quad \text{where } \Delta\mathbf{X}_j = \sum_{k=1}^{2} \left(\frac{\partial f_{k,j}}{\partial \mathbf{X}}^{\mathsf{T}} \frac{\partial f_{k,j}}{\partial \mathbf{X}} + \lambda \mathbf{I} \right)^{-1} \frac{\partial f_{k,j}}{\partial \mathbf{X}}^{\mathsf{T}} (\mathbf{b}_{k,j} - f_{k,j}), \tag{6}$$

where $\frac{\partial f_{k,j}}{\partial \mathbf{X}} \in \mathbb{R}^{2\times 3}$ is the point Jacobian for the j^{th} point (we denote $f(\mathbf{p}_k, \mathbf{X}_j)$ by $f_{k,j}$ by an abuse of notation.). λ is the damping parameter, and you can try $\lambda \in [0, 10]$.

```
def FindMissingReconstruction(X, track_i):
  ...
  return new_point
```

Input: $\texttt{X} \in \mathbb{R}^{F\times 3}$ is the 3D points, and $\texttt{track\_i} \in \mathbb{R}^{F\times 2}$ is the 2D points of the newly registered image.
Output: $\texttt{new\_point} \in \{0,1\}^{F\times 1}$ is a set of indicators that indicate new points.
Note: The new points are the points that are valid for the new image and are not reconstructed yet, i.e.,

$$\texttt{new\_point}_i = \begin{cases} 1 & \text{if } \texttt{X}_i \neq -1, \ \ \texttt{track\_i}_i \neq -1 \\ 0 & \text{otherwise} \end{cases}, \tag{7}$$

where $\texttt{X}_i$ and $\texttt{track\_i}_i$ are the i^{th} 3D and 2D points.

```
def ComputePointJacobian(X, p):
  ...
  return dfdX
```

Input: $\texttt{X} \in \mathbb{R}^3$ is the 3D point, and $\texttt{p} \in \mathbb{R}^7$ is the camera pose made of camera center ($\mathbb{R}^3$) and quaternion ($\mathbb{R}^4$).
Output: $\texttt{dfdX} \in \mathbb{R}^{2\times 3}$ is the point Jacobian, i.e., $\partial f_{k,j}/\partial \mathbf{X}$.

```
def Triangulation_nl(X, P1, P2, x1, x2):
  ...
  return X_new
```

Input: $\texttt{X} \in \mathbb{R}^{n\times 3}$ is 3D points, $\texttt{P1}, \texttt{P2} \in \mathbb{R}^{3\times 4}$ are two camera projection matrices, and $\texttt{x1}, \texttt{x2} \in \mathbb{R}^{n\times 2}$ are point correspondences from the two poses.
Output: $\texttt{X\_new} \in \mathbb{R}^{n\times 3}$ is the set of refined 3D points.

7 Running Bundle Adjustment

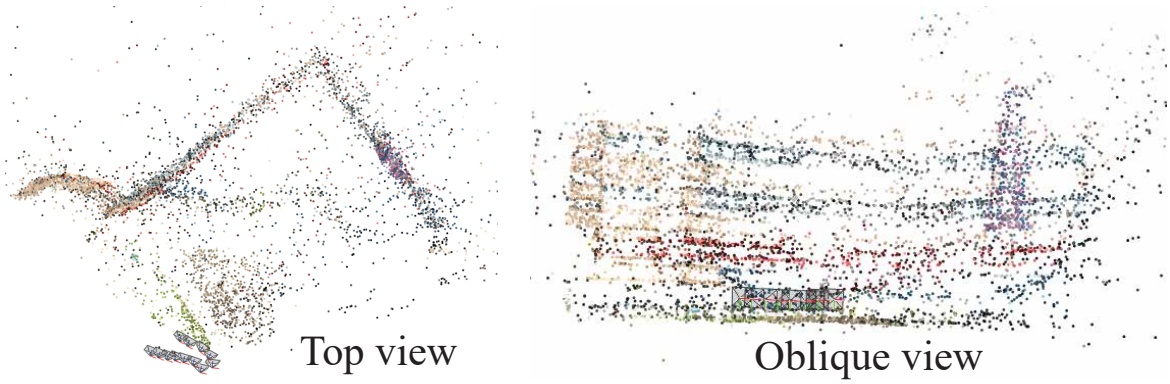


Figure 9: You will reconstruct all images and 3D points using structure from motion.

You will use a nonlinear least squares optimization to refine the camera pose and reconstructed points all together by minimizing the reprojection error:

$$\underset{\{\mathbf{p}_k\}\{\mathbf{X}_j\}}{\text{minimize}} \quad \sum_{k=1}^{K}\sum_{j=1}^{J} s_{k,j}\|f(\mathbf{p}_k, \mathbf{X}_j) - \mathbf{b}_{k,j}\|^2, \tag{8}$$

where K is the number of images, and J is the number of 3D points. $s_{k,j} = \{0, 1\}$ is the visibility indicator, i.e., one if the j^{th} point is visible to the k^{th} camera, and zero otherwise:

$$s_{k,j} = \begin{cases} 1 & \text{if } \mathbf{b}_{k,j} \neq -1 \\ 0 & \text{otherwise.} \end{cases} \tag{9}$$

You will use `scipy.optimize.least_squares` to optimize Equation (8). The pseudo-code is found in Algorithm 4.

Algorithm 4 RunBundleAdjustment

1: Setup bundle adjustment. ▷ `SetupBundleAdjustment`
2: Run sparse bundle adjustment.
3: Update the poses and 3D points. ▷ `UpdatePosePoint`

Structure from Motion

```
def SetupBundleAdjustment(P, X, track):
  ...
  return z, b, S, camera_index, point_index
```

Input: `P` $\in \mathbb{R}^{K\times 3\times 4}$ is the set of reconstructed camera poses, `X` $\in \mathbb{R}^{J\times 3}$ is the set of reconstructed 3D points, `track` $\in \mathbb{R}^{K\times J\times 2}$ is the tracks for the reconstructed cameras.
Output: `z` $\in \mathbb{R}^{7K+3J}$ is the optimization variable that is made of all camera poses and 3D points, i.e.,

$$\mathtt{z} = \begin{bmatrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_K \\ \mathbf{X}_1 \\ \vdots \\ \mathbf{X}_F \end{bmatrix} \tag{10}$$

where $\mathbf{p}_i = \begin{bmatrix} \mathbf{C}_i^\mathsf{T} & \mathbf{q}_i^\mathsf{T} \end{bmatrix}^\mathsf{T} \in \mathbb{R}^7$ is the i^{th} camera pose (center and quaternion), and $\mathbf{X}_j \in \mathbb{R}^3$ is the j^{th} 3D point.

`b` $\in \mathbb{R}^{2M}$ is the 2D points in `track` where M is the number of 2D visible points, i.e., not all points in `track` are visible (only $s_{k,j} = 1$ should be included.).

`S` $\in \{0,1\}^{2M\times(7K+3J)}$ is a sparse indicator matrix that indicates the locations of Jacobian computation. Jacobian is only valid when $s_{k,j} = 1$. Consider the m^{th} measurement made by the projection of the j^{th} 3D point ($\mathbf{X}_j$) onto the k^{th} camera ($\mathbf{p}_k$) in Equation (8). The pose Jacobian entries correspond to `S[2m:2m+1, 7k:7k+6]=1` and point Jacobian entries are `S[2m:2m+1, 7K+3*j:7K+3*j+2]=1`.

Note: For the first two camera poses, you do not want to update through the Jacobian, i.e., `S[2m:2m+1, 7k:7k+6]=0` for `m=1,2`.

`camera_index` $\in \mathbb{Z}^M$ specifies the index of camera for each measurement, i.e., `camera_index`=k, and `point_index` $\in \mathbb{Z}^M$ specifies the index of 3D point for each measurement, i.e., `point_index`=j.

Structure from Motion

```
def MeasureReprojection(z, b, n_cameras, n_points, camera_index, point_index):
  ...
  return err
```

Input: `z` $\in \mathbb{R}^{21KJ}$ is the optimization variable, `b` $\in \mathbb{R}^{2M}$ is the 2D measured points, `n_cameras` and `n_points` are the number of cameras and 3D points, respectively, and `camera_index`, `point_index` $\in \mathbb{Z}^M$ are the indices of camera and 3D point.
Output: `err` $\in \mathbb{R}^{2M}$ is the reprojection error, i.e., the aggregate of all reprojection errors.
Description: This function can be called during optimization (`least_squares`) to evaluate the reprojection error, e.g.,

```
res = least_squares(MeasureReprojection, z0, jac_sparsity=S, verbose=2,
                    x_scale='jac', ftol=1e-4, method='trf',
                    args=(b, n_cameras, n_points,
                    camera_index, point_index))
```

```
def UpdatePosePoint(z, n_cameras, n_points):
  ...
  return P_new, X_new
```

Input: `z` $\in \mathbb{R}^{21KJ}$ is the optimization variable, and `n_cameras` and `n_points` are the number of cameras and 3D points, respectively.
Output: `P_new` $\in \mathbb{R}^{K\times 3\times 4}$ is the set of refined camera poses, `X_new` $\in \mathbb{R}^{J\times 3}$ is the refined 3D points.

```
def RunBundleAdjustment(P, X, track):
  ...
  return P_new, X_new
```

Input: `P` $\in \mathbb{R}^{K\times 3\times 4}$ is the set of reconstructed camera poses, `X` $\in \mathbb{R}^{J\times 3}$ is the set of reconstructed 3D points, `track` $\in \mathbb{R}^{K\times J\times 2}$ is the tracks for the reconstructed cameras.
Output: `P_new` $\in \mathbb{R}^{K\times 3\times 4}$ is the set of refined camera poses, `X_new` $\in \mathbb{R}^{J\times 3}$ is the refined 3D points. You can use `MeasureReprojection` to measure the reprojection error before and after bundle adjustment to see the error reduction.