Classic Monocular SfM Pipeline

Overview

This implementation performs classic, feature-based monocular Structure-from-Motion (SfM) on a short, carefully selected subsequence of the Middlebury TempleRing views. It estimates a relative camera motion between consecutive frames, chains those motions into a trajectory, triangulates sparse 3D points from inlier correspondences, filters points that are physically impossible, and exports a colored point cloud plus a trajectory CSV. When enabled, it also uses the dataset's calibrated motion magnitude to assign a per-step scale to the otherwise scale-ambiguous monocular translation.

1) Data inputs and the projection convention

The dataset provides, for each image, a camera intrinsic matrix K and extrinsics R, t packaged so that a 3D world point projects into an image using a standard pinhole model. The program reads these values directly from `templeR_par.txt` and uses them consistently wherever it needs ground-truth relative motion or camera-center baselines for view selection.

A calibrated camera maps a homogeneous 3D world point $\mathbf{X}$ to a homogeneous image point $\mathbf{x}$ by:

$$\mathbf{x} \sim \mathbf{K}\,[\mathbf{R} \mid \mathbf{t}]\,\mathbf{X}$$

The "$\sim$" indicates equality up to a scale factor, because homogeneous image coordinates are converted back to pixels by dividing by the third coordinate. This convention matches how the program constructs projection matrices and how it interprets the dataset-provided R and t.

2) Parsing `templeR_par.txt` exactly as implemented

The parser reads an integer n and then reads n records. Each record contains an image name and 21 floating-point values. Those 21 values are split into:

- $K \in \mathbb{R}^{3\times3}$ from the first 9 values
- $R \in \mathbb{R}^{3\times3}$ from the next 9 values
- $t \in \mathbb{R}^{3\times1}$ from the final 3 values

No normalization, re-scaling, or re-interpretation is applied during parsing. Downstream computations assume these matrices are already consistent with the dataset's projection model.

3) Camera centers for view selection (baseline computation)

Before estimating motion from image content, the program selects a contiguous run of views that are likely to match well. For that, it computes a camera center per view from the dataset extrinsics, using the standard relationship for a world-to-camera pose:

$$\mathbf{X}_c = \mathbf{R}\mathbf{X}_w + \mathbf{t}$$

The camera center $\mathbf{C}$ is the world point that maps to the camera origin $\mathbf{0}$ in camera coordinates. Setting $\mathbf{X}_c = \mathbf{0}$ and solving for $\mathbf{X}_w$ yields:

$$\mathbf{C} = -\mathbf{R}^\top \mathbf{t}$$

The program computes a baseline length between consecutive views as the Euclidean distance between consecutive camera centers:

$$b_i = \|\mathbf{C}_{i+1} - \mathbf{C}_i\|$$

This baseline is used only for selecting a stable run (and later, a different baseline is used for optional scaling of estimated translation).

4) Selecting a stable consecutive run using two signals

The run-selection logic combines two independent checks between consecutive dataset entries:

1. **Longitude step** from `templeR_ang.txt`, used as a proxy for viewpoint change.
2. **Baseline** b_i, used as a proxy for physical camera displacement.

A pair of consecutive indices is considered acceptable when both are below thresholds:

$$|\Delta \text{lon}_i| < \tau_{\text{lon}}$$

$$b_i < \tau_b$$

The program scans the full boolean sequence of acceptable edges and picks the longest contiguous segment. The reconstruction then uses the first `-frames` views from that segment, and motion is estimated only between consecutive selected frames, with no skipping.

5) Feature extraction: SIFT keypoints and descriptors

For each selected image, the program converts it to grayscale and extracts SIFT features, producing:

- Keypoints $\{\mathbf{x}_i\}$, each a 2D location in pixels with additional metadata
- Descriptors $\{\mathbf{d}_i\}$, numeric fingerprints used for matching

SIFT itself encapsulates its internal scoring and normalization; the implementation treats it as a black box that outputs reliable distinctive points for matching.

6) Matching: k-nearest neighbors and Lowe ratio test

Between each consecutive pair of frames, descriptors are matched using a brute-force L2 matcher, requesting the two closest candidates in the second image for each descriptor in the first image. A match is accepted if the closest candidate is sufficiently better than the second closest candidate, which reduces ambiguous matches in repeated textures.

The ratio test can be summarized as:

$$\frac{d_1}{d_2} < \alpha$$

where d_1 is the descriptor distance to the best match, d_2 is the distance to the second-best match, and $\alpha = 0.75$ in this program.

The output of this step is a set of tentative correspondences $(\mathbf{x}_k, \mathbf{x}'_k)$ in pixel coordinates. These contain outliers and must be filtered geometrically.

7) Essential matrix estimation with RANSAC (`findEssentialMat`)

Using the intrinsic matrix K, the program estimates an essential matrix E from the pixel correspondences. The essential matrix encodes the epipolar geometry of two calibrated views. In normalized coordinates $\hat{\mathbf{x}} = K^{-1}\mathbf{x}$, correct matches satisfy:

$$\hat{\mathbf{x}}_k'^{\top}\,\mathbf{E}\,\hat{\mathbf{x}}_k = 0$$

Because many tentative matches may be incorrect, the estimation is wrapped in RANSAC. Conceptually, RANSAC repeatedly samples minimal subsets, fits a candidate E, and measures how many matches agree with it. Agreement is evaluated by an epipolar residual; a simple form is:

$$r_k = \left|\hat{\mathbf{x}}_k'^{\top}\mathbf{E}\hat{\mathbf{x}}_k\right|$$

Matches are classified as inliers when the residual is below a threshold:

$$r_k < \tau$$

The returned inlier mask from `findEssentialMat` is applied immediately to filter correspondences. The program records this inlier count per pair as `inliers_E`.

8) Recovering relative pose (`recoverPose`) and its interpretation

From the estimated essential matrix E and the RANSAC-filtered correspondences, the program recovers a relative rotation R_{21} and a translation direction t_{21}. The essential matrix has the structural relationship:

$$\mathbf{E} = [\mathbf{t}]_\times\,\mathbf{R}$$

where $[\mathbf{t}]_\times$ is the cross-product matrix:

$$[\mathbf{t}]_\times = \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix}$$

The recovered motion is interpreted as mapping 3D points from camera-1 coordinates into camera-2 coordinates:

$$\mathbf{X}_2 = \mathbf{R}_{21}\mathbf{X}_1 + \mathbf{t}_{21}$$

A key monocular limitation applies: the magnitude of $\mathbf{t}_{21}$ is unknown. Multiplying $\mathbf{t}_{21}$ by any positive scalar yields an epipolar geometry consistent with the same image correspondences if the reconstructed 3D points are scaled accordingly:

$$\mathbf{t}_{21} \equiv s\,\mathbf{t}_{21} \quad \text{for any } s > 0$$

`recoverPose` also outputs a second inlier mask (`pose_mask`) that reflects additional internal checks (including cheirality consistency). The program applies this mask and records the surviving inlier count as `inliers_pose`.

9) Ground-truth relative motion for evaluation, exactly as computed

For each pair, the program computes a ground-truth relative motion from the dataset-provided extrinsics of the two images. With world-to-camera extrinsics (R_i, t_i) and (R_j, t_j), the relative rotation is computed as:

$$\mathbf{R}_{ji} = \mathbf{R}_j\mathbf{R}_i^{\top}$$

The corresponding relative translation in the same convention is:

$$\mathbf{t}_{ji} = \mathbf{t}_j - \mathbf{R}_{ji}\mathbf{t}_i$$

These expressions match the program's `R_gt` and `t_gt`. They are used for computing angular errors and optionally defining a per-step translation scale.

10) Optional per-step scale using ground-truth baseline (`-use_gt_scale`)

When `-use_gt_scale` is enabled, the program computes a baseline length from the ground-truth relative translation vector:

$$b = \|\mathbf{t}_{gt}\|$$

It then scales the recovered translation direction from `recoverPose` by that baseline:

$$\mathbf{t}_{scaled} = b\,\mathbf{t}_{21}$$

This injects known scale from the dataset so the estimated trajectory can be expressed in comparable units per step for evaluation and visualization. When `-use_gt_scale` is not enabled, the program keeps $\mathbf{t}_{scaled} = \mathbf{t}_{21}$, producing a trajectory in arbitrary units.

11) Per-pair accuracy metrics as recorded in `pair_metrics.csv`

11.1 Rotation error from trace-to-angle

The program computes a rotation difference:

$$\mathbf{R}_{err} = \mathbf{R}_{est}\mathbf{R}_{gt}^{\top}$$

It converts that difference into a single angle using the standard trace relationship for 3D rotations:

$$\theta = \cos^{-1}\left(\frac{\mathrm{trace}(\mathbf{R}_{err}) - 1}{2}\right)$$

This yields the reported `rot_err_deg` per pair.

11.2 Translation direction error (angle between vectors)

The program compares the direction of the estimated translation to the direction of the ground-truth relative translation:

$$\phi = \cos^{-1}\left(\frac{\mathbf{t}_{est}^{\top}\mathbf{t}_{gt}}{\|\mathbf{t}_{est}\|\,\|\mathbf{t}_{gt}\|}\right)$$

This yields `t_dir_err_deg`. Because it uses a direction-only comparison, it remains meaningful even when monocular scale is unknown.

12) Pose accumulation: updating camera-to-world poses exactly as implemented

The program stores a list `poses` of pairs (R_{wc}, t_{wc}), described in the code as "camera→world pose accumulation." It starts at identity pose for the first camera:

$$\mathbf{R}_{wc}^{(0)} = \mathbf{I}, \quad \mathbf{t}_{wc}^{(0)} = \mathbf{0}$$

Each step provides an estimated relative transform mapping camera i to camera $i+1$:

$$\mathbf{X}_{i+1} = \mathbf{R}_{21}\mathbf{X}_i + \mathbf{t}$$

To update a camera-to-world pose using this relative motion, the program inverts the relative transform. For rigid transforms, inversion uses:

$$\mathbf{R}^{-1} = \mathbf{R}^{\top}$$

$$\mathbf{t}^{-1} = -\mathbf{R}^{\top}\mathbf{t}$$

Then it composes the current camera-to-world transform with this inverse using:

$$\mathbf{R}_{new} = \mathbf{R}_a\mathbf{R}_b$$

$$\mathbf{t}_{new} = \mathbf{R}_a\mathbf{t}_b + \mathbf{t}_a$$

This matches the program's transform update rule and chains transforms so each camera pose is expressed in the shared reference frame defined by the first camera.

13) Triangulation exactly as performed: $P_1 = K[I|0]$, $P_2 = K[R|t]$

For each pair, after filtering matches with `pose_mask`, the program triangulates using a two-view setup where camera 1 is treated as the origin and camera 2 uses the recovered relative pose. It constructs:

$$\mathbf{P}_1 = \mathbf{K}\,[\mathbf{I} \mid \mathbf{0}]$$

$$\mathbf{P}_2 = \mathbf{K}\,[\mathbf{R}_{21} \mid \mathbf{t}]$$

Triangulation seeks a homogeneous 3D point $\mathbf{X}$ consistent with both projections. A common way to express the constraints is the cross-product form:

$$\mathbf{x} \times (\mathbf{P}\mathbf{X}) = \mathbf{0}$$

After solving, homogeneous coordinates are converted to 3D by division:

$$\mathbf{X}_{3D} = \left(\frac{X_1}{X_4}, \frac{X_2}{X_4}, \frac{X_3}{X_4}\right)$$

14) Cheirality filtering exactly as implemented (positive depth in both cameras)

Triangulated points can be numerically unstable or correspond to the wrong side of the camera. The program enforces a physical constraint: points must lie in front of both cameras. For camera 1, this is:

$$z_1 = (\mathbf{X}_{c1})_z > 0$$

For camera 2, the point is transformed using the same relative pose used in triangulation:

$$\mathbf{X}_{c2} = \mathbf{R}_{21}\mathbf{X}_{c1} + \mathbf{t}$$

and must satisfy:

$$z_2 = (\mathbf{X}_{c2})_z > 0$$

Only points satisfying both are kept.

15) Mapping points from the pair frame into the global world frame

Triangulation yields $\mathbf{X}_{c1}$ in camera-i coordinates for that pair. The program then converts these to the global world coordinates using the stored camera-to-world pose for that camera:

$$\mathbf{X}_w = \mathbf{R}_{wc}^{(i)}\,\mathbf{X}_{ci} + \mathbf{t}_{wc}^{(i)}$$

All exported 3D points live in this shared world frame.

16) Coloring points from image evidence

For visualization, the program assigns a color to each 3D point by sampling the pixel color at the corresponding feature location in the first image of the pair. This ties the sparse 3D cloud to recognizable texture. It does not change geometry.

17) Output trajectory CSV: what is written

The program writes one row per selected frame, storing:

- `frame` index within the selected run
- `image` filename
- `x, y, z` from the accumulated translations stored in `poses`
- `lat, lon` from `templeR_ang.txt`

The `lat, lon` values are not used in the reconstruction itself; they are included for inspection and for verifying that the selected run changes smoothly.

18) Why results can drift even when per-pair metrics look good

This implementation estimates motion only between consecutive frames and chains those motions without a global refinement step. Even small per-pair errors, when repeatedly composed, can accumulate into drift. Classical SfM pipelines often add bundle adjustment, which refines all poses and 3D points by minimizing reprojection error across all observations. That refinement is absent here; the program instead reduces difficulty by selecting a stable run and triangulating only from locally consistent inliers.

A standard bundle-adjustment objective (not implemented in this file) is:

$$\min_{\{\mathbf{R}_i,\mathbf{t}_i,\mathbf{X}_k\}} \sum_{i,k} \rho\Big(\|\mathbf{x}_{ik} - \pi(\mathbf{K}[\mathbf{R}_i \mid \mathbf{t}_i]\mathbf{X}_k)\|^2\Big)$$

This expression is included only to clarify what "global refinement" typically means in the same problem setting.

19) Summary of the exact per-iteration flow (one pair $i \to i+1$)

For each consecutive pair, the program performs, in this order:

1. Match descriptors with 2-NN and apply ratio test to form `pts1, pts2`.

2. Estimate E with RANSAC: `findEssentialMat(pts1, pts2, K, ...)` yielding an inlier mask.
3. Filter correspondences to `pts1_in, pts2_in`.
4. Recover R_{21}, t_{21} and `pose_mask` with `recoverPose(E, pts1_in, pts2_in, K)`.
5. Filter again to `pts1_pose, pts2_pose`.
6. Compute ground-truth `R_gt, t_gt` from dataset extrinsics for evaluation.
7. Optionally scale translation: `t_scaled = t_21 * ||t_gt||`.
8. Invert (R_{21}, t_{scaled}) and compose into the accumulated (R_{wc}, t_{wc}) pose list.
9. Triangulate points with $P_1 = K[I|0]$, $P_2 = K[R_{21}|t]$.
10. Cheirality filter: positive depth in both cameras.
11. Transform points into the global world frame using `poses[i]`.
12. Sample colors from the first image and append to the exported point set.