Project 2
Buildings built in minutes

Sarvesh Surendran Nair
Master's of Science
Robotics Engineering
Worcester Polytechnic Institute
Email: ssnair@wpiedu

Samruddhi Naukudkar
Master's of Science
Robotics Engineering
Worcester Polytechnic Institute
Email: snaukudkar@wpi.edu

PHASE 1 - STRUCTURE FROM MOTION (SFM)

I. INTRODUCTION

In this project, we have implemented a classical Structure from Motion (SfM) pipeline to reconstruct a 3D scene and estimate the camera poses from a set of 2D images. Our approach follows a structured methodology, incorporating multiple steps to ensure accurate reconstruction.

Our implementation consists of the following key stages:

1) **Feature Matching and Outlier Rejection:** We extracted SIFT features and used RANSAC to eliminate outliers, ensuring robust feature correspondences between image pairs.
2) **Fundamental Matrix Estimation:** Using the normalized 8-point algorithm, we computed the fundamental matrix and enforced the rank-2 constraint via Singular Value Decomposition (SVD).
3) **Essential Matrix Computation:** Given the intrinsic camera parameters, we derived the essential matrix from the fundamental matrix and enforced the required singular value constraints.
4) **Camera Pose Extraction:** We extracted four possible camera poses from the essential matrix by computing SVD and applying the predefined transformations.
5) **Triangulation and Cheirality Check:** We implemented linear triangulation to compute the 3D points and determined the correct camera pose by enforcing the cheirality condition.
6) **Non-Linear Triangulation:** To improve the accuracy of 3D point estimates, we refined the initial triangulation using non-linear optimization to minimize reprojection error.
7) **Perspective-n-Point (PnP) with RANSAC:** We estimated the camera poses for additional views using PnP and applied RANSAC to filter out incorrect correspondences.
8) **Bundle Adjustment:** Finally, we optimized both the 3D structure and camera poses using sparse bundle adjustment to minimize reprojection error and refine the reconstruction.

Each of these components plays a critical role in ensuring an accurate and consistent 3D reconstruction. The following sections detail the mathematical foundations, implementation, and experimental results of each step in our pipeline.

II. FUNDAMENTAL MATRIX ESTIMATION

The fundamental matrix $\mathbf{F}$ plays a crucial role in epipolar geometry as it defines the geometric relationship between two images taken from different viewpoints. Given a pair of corresponding points (x, x') in two images, the fundamental matrix satisfies the epipolar constraint:

$$x'^T \mathbf{F} x = 0 \tag{1}$$

where $x = [u, v, 1]^T$ and $x' = [u', v', 1]^T$ are the homogeneous coordinates of the matched points in the two images.

A. Epipolar Geometry

Epipolar geometry is the intrinsic projective geometry between two views. It is independent of the scene structure and is determined solely by the camera parameters and relative pose. Each point in the first image corresponds to an epipolar line in the second image, restricting the search space for correspondence.

Figure 1 illustrates the epipolar geometry, showing the corresponding epipolar lines and the epipoles where these lines intersect.

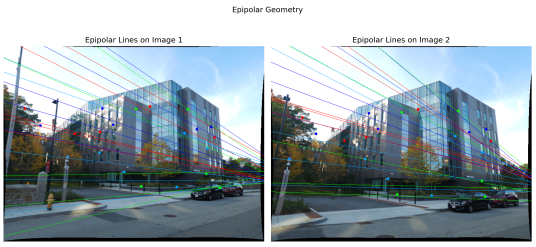

Fig. 1: Epipolar Geometry: A 3D point X is projected onto two images, generating epipolar lines.

B. Computing the Fundamental Matrix

We implemented the normalized 8-point algorithm in `EstimateFundamentalMatrix.py` to compute the fundamental matrix. The algorithm follows these steps:

1) **Normalization of points:** To improve numerical stability, we normalize the image coordinates by translating them to have zero mean and scaling them so that the average distance from the origin is $\sqrt{2}$.
2) **Constructing the constraint matrix:** Using the normalized correspondences, we build a system of linear equations of the form $\mathbf{A}f = 0$, where $\mathbf{A}$ is an $N \times 9$ matrix (with $N \geq 8$ correspondences).
3) **Solving using SVD:** We compute the singular value decomposition (SVD) of $\mathbf{A}$, and the fundamental matrix is obtained as the singular vector corresponding to the smallest singular value.
4) **Enforcing rank-2 constraint:** Since the fundamental matrix should have rank 2, we set the smallest singular value to zero and reconstruct $\mathbf{F}$.
5) **Denormalization:** Finally, we transform the fundamental matrix back to the original coordinate system using the inverse of the normalization transformation.

C. Outlier Rejection using RANSAC

Feature matching is prone to noise and incorrect correspondences. To address this, we implemented RANSAC-based outlier rejection in `GetInliersRANSAC.py`. The RANSAC algorithm follows these steps:

1) Randomly sample 8 correspondences and estimate a fundamental matrix.
2) Compute the epipolar distances for all correspondences and classify inliers as those with distances below a threshold.
3) Repeat for multiple iterations, selecting the fundamental matrix with the maximum number of inliers.
4) Recompute $\mathbf{F}$ using all inliers to obtain a refined estimate.

Figure 2 visualizes the feature correspondences before and after applying RANSAC.

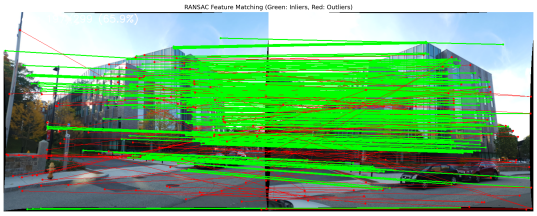

Fig. 2: Feature matches after RANSAC. Green lines indicate inlier matches, while red lines indicate rejected outliers.

D. Implementation Details

Our implementation follows the structure:

- **`EstimateFundamentalMatrix.py`:** Implements the normalized 8-point algorithm.
- **`GetInliersRANSAC.py`:** Implements RANSAC-based fundamental matrix estimation.

After applying this method, we obtain a robust fundamental matrix that accurately models the epipolar constraints between two views.

III. Estimating the Essential Matrix from the Fundamental Matrix

Once the fundamental matrix F has been computed, we can estimate the essential matrix E, which encodes the relative pose (rotation and translation) between two cameras. The essential matrix is defined as:

$$E = K^T F K \tag{2}$$

where K is the camera intrinsic matrix. Unlike the fundamental matrix, which relates points in pixel coordinates, the essential matrix operates in normalized camera coordinates, removing the dependence on intrinsic parameters.

A. Properties of the Essential Matrix

The essential matrix must satisfy certain constraints:

- It is a 3×3 matrix of rank 2.
- Its two nonzero singular values must be equal, while the third singular value must be zero.

To enforce these constraints, we apply Singular Value Decomposition (SVD) and reconstruct E by setting the singular values to $[1, 1, 0]$.

B. Implementation Details

We implemented the computation of the essential matrix in `EssentialMatrixFromFundamentalMatrix.py`, following these steps:

1) Compute the initial essential matrix using:

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

2) Perform Singular Value Decomposition (SVD):

$$E = U S V^T \tag{4}$$

3) Replace the singular values with $[1, 1, 0]$ to enforce the rank-2 constraint:

$$E_{\text{corrected}} = U \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} V^T \tag{5}$$

C. Experimental Results

After implementing the essential matrix computation, we verified its correctness by checking the singular values. The expected values were $[s, s, 0]$, confirming that the matrix met the required constraints. The essential matrix plays a crucial role in recovering the relative camera pose, which is discussed in the next section.

IV. Extracting Camera Pose from the Essential Matrix

Once the essential matrix E is computed, we extract the possible camera poses, which consist of the rotation matrix R and the camera center C. Since the essential matrix encodes the relative transformation between two camera views, we can derive four possible (R, C) combinations.

A. Computing the Camera Pose

To extract the camera pose, we first perform Singular Value Decomposition (SVD) on the essential matrix:

$$E = USV^T \tag{6}$$

where U and V are orthonormal matrices, and S is a diagonal matrix. From U and V, we define the following matrix:

$$W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{7}$$

Using W, we compute two possible rotation matrices:

$$R_1 = UWV^T, \quad R_2 = UW^TV^T \tag{8}$$

Similarly, the translation vector (up to scale) is extracted from U as:

$$C_1 = U[:, 2], \quad C_2 = -U[:, 2] \tag{9}$$

Thus, we obtain four possible camera poses:

1) $(R_1, +C_1)$
2) $(R_1, -C_1)$
3) $(R_2, +C_2)$
4) $(R_2, -C_2)$

B. Implementation Details

We implemented this step in `ExtractCameraPose.py` using the following procedure:

1) Compute the SVD of E.
2) Construct the matrix W.
3) Compute the two possible rotation matrices R_1 and R_2.
4) Compute the two possible translation vectors C_1 and C_2.
5) Ensure that R has a determinant of +1, as required for a valid rotation matrix.

C. Experimental Results

After extracting the four possible camera poses, we verify that all rotation matrices have a determinant of +1. These candidate poses are later evaluated using triangulation and the cheirality condition to determine the correct camera pose.

The next section discusses the process of triangulating 3D points and selecting the best camera pose using the cheirality check.

V. Triangulation Check for Cheirality Condition

Once we have computed the four possible camera poses from the essential matrix, we need to determine which pose correctly represents the real-world configuration. This is done by triangulating 3D points from 2D correspondences and applying the cheirality condition to identify the correct camera pose.

A. Linear Triangulation

Triangulation is the process of estimating 3D world points given two camera poses and their corresponding 2D feature matches. Since each 2D point corresponds to a back-projected ray in space, the goal is to find the optimal 3D point where these rays intersect.

Given two projection matrices P_1 and P_2 for two camera views, the relationship between a 3D point X and its 2D projections is given by:

$$x_1 = P_1X, \quad x_2 = P_2X \tag{10}$$

where x_1 and x_2 are the homogeneous 2D coordinates in the respective images. By stacking the constraints for multiple points, we obtain a system of equations that can be solved using Singular Value Decomposition (SVD) to estimate the best-fit 3D points.

We implemented linear triangulation in `LinearTriangulation.py` by:

1) Constructing the projection matrices P_1 and P_2.
2) Forming a linear system based on the epipolar constraints.
3) Solving for the homogeneous 3D point using SVD.
4) Converting homogeneous coordinates to inhomogeneous coordinates.

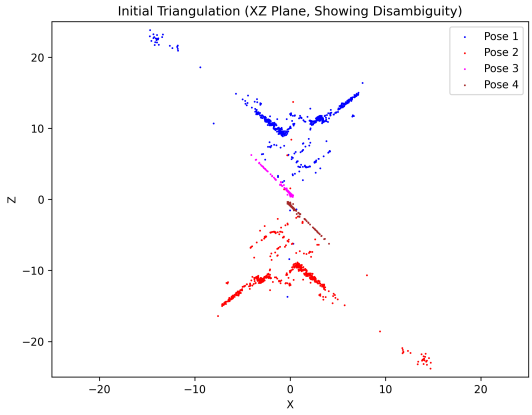

Fig. 3: Initial triangulation plot with disambiguity, showing all four possible camera poses.

While linear triangulation provides an initial estimate of 3D points, it does not account for errors in image measurements. The next step is to determine which of the four candidate camera poses is correct by applying the cheirality condition.

B. Disambiguate Camera Pose

Since we obtain four possible camera poses from the essential matrix, we need to select the correct one. The correct pose should satisfy the cheirality condition, which states that the triangulated 3D points must lie in front of both cameras.

For a given camera pose (R, C), a 3D point X is in front of the camera if:

$$r_3^T(X - C) > 0 \tag{11}$$

where r_3 is the third row of the rotation matrix R, and C is the camera center. The correct camera pose is the one that results in the maximum number of points satisfying this condition.

We implemented this in `DisambiguateCameraPose.py` by:

1) Triangulating points using each of the four camera poses.
2) Counting the number of points that satisfy the cheirality condition.
3) Selecting the pose with the highest number of valid 3D points.

Once the correct camera pose is determined, we refine the triangulated points using non-linear triangulation.

C. Non-Linear Triangulation

Linear triangulation minimizes algebraic error, which does not necessarily correspond to an accurate reprojection of the points in the image. To improve the accuracy of the 3D points, we refine them by minimizing the reprojection error, which is the squared distance between the observed 2D points and their projections.

The reprojection error for a triangulated 3D point X is defined as:

$$\sum_{j=1}^{2}\left(u_j - \frac{p_{j1}^T X}{p_{j3}^T X}\right)^2 + \left(v_j - \frac{p_{j2}^T X}{p_{j3}^T X}\right)^2 \tag{12}$$

where (u_j, v_j) are the observed image points, and p_{ji} are the rows of the projection matrix. Since this function is non-linear, we solve it using non-linear least squares optimization.

We implemented non-linear triangulation in `NonLinearTriangulation.py` by:

1) Using the linearly triangulated points as an initial estimate.
2) Optimizing the 3D points by minimizing the reprojection error.
3) Using the `scipy.optimize.least_squares` function for optimization.

This process significantly improves the accuracy of the reconstructed 3D structure.

To further verify the accuracy of the triangulated points, we compare their projections onto the original image with the detected feature points.

D. Reprojection Verification

To evaluate the accuracy of the reconstructed 3D points, we project them back into the image and compare them with the detected feature points. A well-triangulated 3D point should project close to its corresponding 2D point in the original image.

Figure 5 shows the projection of the linearly triangulated points onto the original image, while Figure 6 shows the projection of the non-linearly triangulated points.

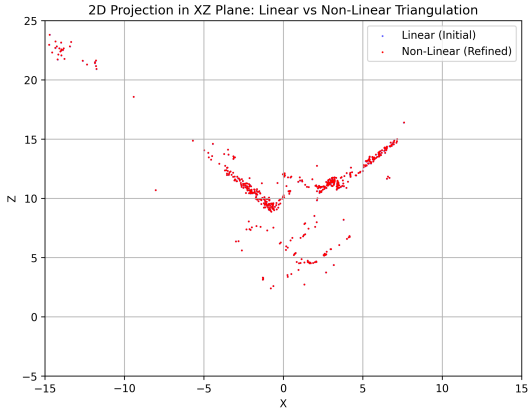

Fig. 4: Comparison between linear and non-linear triangulation. Note: Due to the refinement of points in non-linear triangulation, the linear points are almost perfectly superimposed, making only the non-linear points visible.

Fig. 5: Reprojection of linearly triangulated points (red) vs. original points (green) on the original image.

E. Final Explanation

By implementing both linear and non-linear triangulation, we successfully reconstruct 3D points from 2D correspondences. The cheirality check ensures that we select the correct camera pose, and the non-linear refinement significantly improves accuracy. These reconstructed 3D points will be used in the next step for estimating new camera poses and refining the entire structure using bundle adjustment.

VI. Perspective-n-Point (PnP)

Once we have reconstructed a set of 3D points using triangulation, we can estimate the camera poses for additional images using the Perspective-n-Point (PnP) algorithm. Given a set of 3D world points and their corresponding 2D image projections, PnP determines the 6-degree-of-freedom (6-DoF) pose of the camera.

In our implementation, we use three variations of PnP:

1) Linear PnP
2) PnP with RANSAC
3) Non-Linear PnP

Fig. 6: Reprojection of non-linearly triangulated points (red) vs. original points (green) on the original image. The non-linearly optimized points align better with the detected features.

A. Linear PnP

Linear PnP estimates the camera pose by solving a set of equations derived from the projection equation:

$$x = PX \tag{13}$$

where x is the 2D image point, X is the corresponding 3D world point, and P is the camera projection matrix. Expanding P in terms of rotation R and translation C, we get:

$$x = K[R \mid -RC]X \tag{14}$$

Given N point correspondences, we construct a linear system of equations of the form:

$$Ap = 0 \tag{15}$$

where A is a $2N \times 12$ matrix formed using the correspondences, and p is a vector containing the elements of P. The solution is obtained using Singular Value Decomposition (SVD).

Since the estimated R may not be a valid rotation matrix (i.e., it may not be orthonormal), we enforce the orthogonality constraint by performing another SVD on R and reconstructing it.

We implemented linear PnP in `LinearPnP.py` by:

1) Constructing the linear system from 2D-3D correspondences.
2) Solving for P using SVD.
3) Extracting R and C from P.
4) Ensuring R is a valid rotation matrix by enforcing orthogonality.

However, since linear PnP is sensitive to noise and outliers, we apply RANSAC to make the estimation more robust.

B. PnP with RANSAC

PnP with RANSAC improves the robustness of camera pose estimation by filtering out incorrect correspondences. The RANSAC algorithm follows these steps:

1) Randomly select a minimal set of 3D-2D correspondences.
2) Estimate the camera pose using linear PnP.
3) Compute the reprojection error for all correspondences.
4) Classify points as inliers if their error is below a threshold.
5) Repeat for multiple iterations and select the pose with the maximum inliers.
6) Recompute the final pose using all inliers.

This ensures that the estimated camera pose is less affected by outliers.

Fig. 7: Reprojection of points using Linear PnP (red) vs. original points (green) on image 3.

Although RANSAC helps filter out bad correspondences, Linear PnP still suffers from sensitivity to measurement noise. To further refine the camera pose, we apply non-linear optimization.

C. Non-Linear PnP

Non-Linear PnP refines the estimated camera pose by minimizing the reprojection error:

$$\sum_{i=1}^{N} \left(u_i - \frac{p_1^T X_i}{p_3^T X_i} \right)^2 + \left(v_i - \frac{p_2^T X_i}{p_3^T X_i} \right)^2 \tag{16}$$

where (u_i, v_i) are the observed 2D image points, and p_j are the rows of the projection matrix.

Since this function is non-linear, we solve it using non-linear least squares optimization with the Levenberg-Marquardt algorithm.

We implemented non-linear PnP in `NonLinearPnP.py` by:

1) Converting the rotation matrix to a quaternion representation.
2) Using the initial pose estimated from PnP-RANSAC as a starting point.

3) Minimizing the reprojection error using `scipy.optimize.least_squares`.

Fig. 8: Reprojection of points using Non-Linear PnP (red) vs. original points (green) on image 3.

D. Comparison of Linear vs. Non-Linear PnP

To visualize the difference between Linear and Non-Linear PnP, we compare their 2D projections on the XZ plane. The improved accuracy of Non-Linear PnP is evident, as it better aligns with the observed feature points.

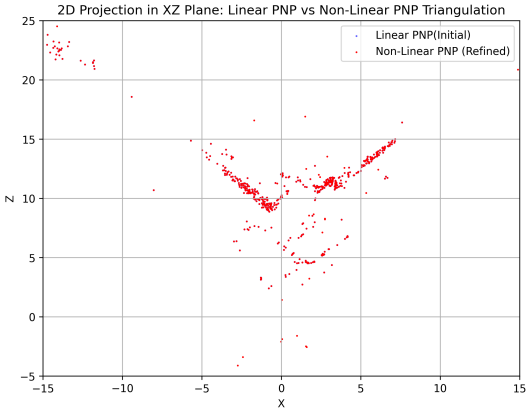

Fig. 9: Comparison of 2D projections: Linear PnP (blue) vs. Non-Linear PnP (red). Note: Since the non-linear points refine the linear estimates, they get superimposed, making only the non-linear points visible.

E. Final Camera Poses and Reconstructed Points

After estimating the camera poses for all five images, we visualize the reconstructed points and their alignment with the camera positions. The estimated camera poses are generally accurate, but we observed that the third camera pose was not corrected well due to incorrect correspondences.

F. Final Explanation

In this section, we implemented three variations of PnP for camera pose estimation. While Linear PnP provides an initial estimate, RANSAC improves robustness, and Non-Linear PnP refines the accuracy by minimizing reprojection error. The final camera poses are used in the next step for refining the entire structure using Bundle Adjustment.

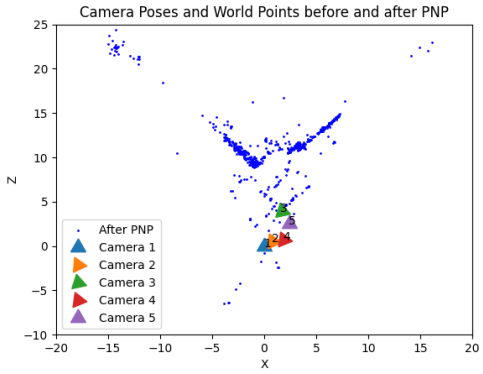

Fig. 10: Reconstructed points and estimated camera poses for all five images. The third camera pose was not corrected well, leading to slight misalignment.

VII. Bundle Adjustment

After estimating the camera poses and triangulating the 3D points, we refine both the 3D structure and camera parameters using Bundle Adjustment (BA). Bundle Adjustment is a non-linear optimization technique that minimizes the reprojection error across all views by simultaneously optimizing:

- The 3D world points.
- The camera poses (rotation and translation).

Since errors accumulate at each stage of the pipeline, Bundle Adjustment significantly improves the accuracy and consistency of the reconstruction.

A. Visibility Matrix

Bundle Adjustment requires a visibility matrix, which keeps track of which 3D points are visible in which images. A point is considered visible if it has been successfully matched in at least two views.

Initially, we faced challenges because of the way we were storing 3D points and their corresponding 2D matches. Instead of maintaining a structured mapping, we were handling points in arrays, leading to inconsistencies when associating 3D points with images.

To solve this, we:

1) Stored 3D points in a dictionary, where each key represented a unique point, and the value contained the corresponding 3D coordinates.
2) Stored 2D points separately for each image in another dictionary.
3) Used a dictionary for matches to correctly associate 3D points across multiple views.

This corrected our indexing issues and allowed us to build an accurate visibility matrix.

Given I images and J 3D points, we define the visibility matrix V as:

$$V_{ij} = \begin{cases} 1, & \text{if the } j\text{-th point is visible in the } i\text{-th image} \\ 0, & \text{otherwise} \end{cases} \tag{17}$$

We implemented this step in `BuildVisibilityMatrix.py`, where:

- Each row corresponds to a 3D point.
- Each column corresponds to a camera/image.
- Entries are set to 1 if the point is visible in that image, otherwise 0.

This matrix is crucial for BA, as it helps define which points contribute to the optimization.

B. Bundle Adjustment Optimization

Bundle Adjustment refines the 3D structure and camera parameters by minimizing the overall reprojection error:

$$\sum_{i=1}^{I}\sum_{j=1}^{J} V_{ij}\left[\left(u_{ij} - \frac{P_{j1}^T X_j}{P_{j3}^T X_j}\right)^2 + \left(v_{ij} - \frac{P_{j2}^T X_j}{P_{j3}^T X_j}\right)^2\right] \tag{18}$$

where:

- (u_{ij}, v_{ij}) are the observed 2D coordinates in image i.
- X_j is the 3D world point.
- P_{ji} are the rows of the projection matrix for camera i.

We implemented Bundle Adjustment in `BundleAdjustment.py`, following these steps:

1) Converted rotation matrices to Euler angles for parameterization.
2) Constructed a sparse Jacobian matrix to speed up optimization.
3) Used `scipy.optimize.least_squares` with the Levenberg-Marquardt method to optimize the parameters.
4) Updated the refined camera poses and 3D points after convergence.

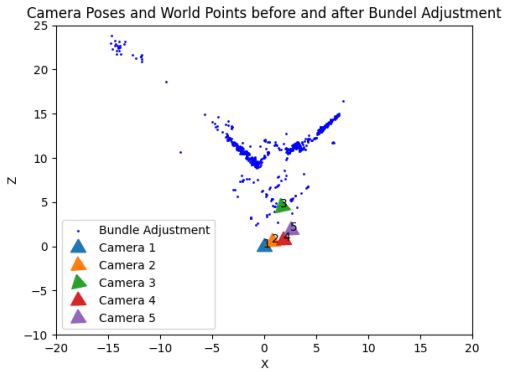

Fig. 11: Final camera poses and refined 3D points after Bundle Adjustment.

C. Explanation and Impact of Bundle Adjustment

Before Bundle Adjustment, errors in estimated camera poses and 3D points led to misalignment in the reconstructed structure. The primary issues were:

- Drift in camera poses due to error accumulation.
- Inaccurate 3D points due to noisy correspondences.
- Incorrect association of 3D points to images.

Bundle Adjustment corrected these by:

1) Optimizing all camera parameters simultaneously, reducing pose drift.
2) Adjusting the 3D points to minimize reprojection error.
3) Ensuring that points are accurately associated with their corresponding image projections.

After applying Bundle Adjustment, we observed:

- Improved alignment between estimated and actual camera poses.
- More consistent 3D structure with reduced noise.
- Lower reprojection error, ensuring better visual accuracy.

This step completes the Structure from Motion (SfM) pipeline, resulting in a robust and optimized 3D reconstruction.

VIII. Conclusion

In this project, we successfully implemented a classical Structure from Motion (SfM) pipeline to reconstruct a 3D scene and estimate camera poses from multiple images. Our implementation followed a structured approach, incorporating feature matching, epipolar geometry, triangulation, PnP, and Bundle Adjustment to refine the reconstruction.

A. Challenges and Solutions

Throughout the implementation, we faced several challenges:

- **Incorrect storage of 3D points:** Initially, 3D points were not correctly mapped to their respective images, which led to misalignment. This was resolved by structuring data in dictionaries to maintain proper correspondences.
- **Camera pose drift:** Accumulated errors in estimated camera poses led to drift in reconstruction. Applying Bundle Adjustment successfully corrected these errors.
- **Noisy correspondences in PnP:** The presence of outliers affected pose estimation, which was addressed using RANSAC to select only the most consistent inliers.

B. Impact of Bundle Adjustment

Applying Bundle Adjustment at the final stage of the pipeline significantly improved the reconstruction quality. It reduced overall reprojection error, corrected misalignments, and ensured consistency across multiple views.

Through this project, we have demonstrated a robust approach to classical SfM, following fundamental principles of multi-view geometry. The combination of triangulation, PnP, and Bundle Adjustment resulted in an accurate and refined 3D reconstruction. This pipeline lays the foundation for more advanced techniques, such as NeRF-based reconstructions,

which leverage deep learning to generate dense, photorealistic 3D models.

The final output confirms the effectiveness of classical SfM in reconstructing 3D structures from multiple images, emphasizing the importance of geometric constraints and optimization in computer vision.

REFERENCES

[1] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, 2nd ed., Cambridge University Press, 2003. Available: https://www.cambridge.org/core/books/multiple-view-geometry-in-computer-vision/2A5D2F5B8ACAAF66D5A0F0D7CBB06638

[2] N. Snavely, S. M. Seitz, and R. Szeliski, *Photo Tourism: Exploring Photo Collections in 3D*, ACM Transactions on Graphics (TOG), vol. 25, no. 3, pp. 835–846, 2006. Available: https://dl.acm.org/doi/10.1145/1141911.1141964

[3] S. Agarwal, N. Snavely, I. Simon, S. M. Seitz, and R. Szeliski, *Building Rome in a Day*, Communications of the ACM, vol. 54, no. 10, pp. 105–112, 2011. Available: https://dl.acm.org/doi/10.1145/2001269.2001293

[4] J. L. Schönberger and J.-M. Frahm, *Structure-from-Motion Revisited*, In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 4104–4113, 2016. Available: https://ieeexplore.ieee.org/document/7780814

[5] S. Ullman, *The Interpretation of Structure from Motion*, Proceedings of the Royal Society of London. Series B. Biological Sciences, vol. 203, no. 1153, pp. 405–426, 1979. Available: https://royalsocietypublishing.org/doi/10.1098/rspb.1979.0006

[6] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzgibbon, *Bundle Adjustment—A Modern Synthesis*, In Proceedings of the International Workshop on Vision Algorithms, pp. 298–372, 1999. Available: https://link.springer.com/chapter/10.1007/3-540-44480-7_21

[7] W. Chen, S. Kumar, and F. Yu, *Uncertainty-Driven Dense Two-View Structure from Motion*, arXiv preprint arXiv:2302.00523, 2023. Available: https://arxiv.org/abs/2302.00523

[8] Z. Jiang, H. Taira, N. Miyashita, and M. Okutomi, *VIO-Aided Structure from Motion Under Challenging Environments*, arXiv preprint arXiv:2101.09657, 2021. Available: https://arxiv.org/abs/2101.09657