Lab 6 Report

Robotics Integration Group Project I

Yuwei ZHAO (23020036096)
Group #31 2025-12-24

Abstract

This report presents a comprehensive study on Visual Odometry algorithms for autonomous drone navigation. Utilizing the `OpenGV` library, we implemented and evaluated a suite of geometric solvers to estimate relative camera motion, including **Nistér's 5-point algorithm**, **Longuet-Higgins' 8-point algorithm**, and a **2-point minimal solver** assuming known rotation. We conducted a comparative analysis of these 2D-2D methods against Arun's 3D-3D point cloud registration method using RGB-D data.

Our experiments demonstrate that robust outlier rejection via **RANSAC** is indispensable for reliable pose estimation in the presence of noisy feature correspondences. Furthermore, while monocular methods suffer from inherent scale ambiguity, the integration of depth information allows for the recovery of absolute trajectory scale with high accuracy. Finally, we evaluated the system on a custom high-speed drone racing dataset. The results highlight the performance degradation of geometric solvers under rapid motion and motion blur, emphasizing the critical dependency of backend estimation on frontend feature tracking quality.

See Resources on github.com/RamessesN/Robotics_MIT.

1 Introduction

State estimation is a fundamental prerequisite for the autonomous operation of Unmanned Aerial Vehicles, particularly in GPS-denied indoor environments. Visual Odometry provides a solution by estimating the agent's ego-motion through the analysis of consecutive image frames. In this lab, we explore the geometric foundations of VO, moving from 2D-2D epipolar geometry to 3D-3D rigid body registration.

The core objective of this report is to implement and benchmark different motion estimation strategies using the `OpenGV` library. The pipeline begins with feature extraction and matching (SIFT), followed by geometric verification. We address several key challenges in visual estimation:

1. **Outlier Rejection:** Feature matching is prone to errors. We investigate the efficacy of Random Sample Consensus (RANSAC) in filtering spurious matches to preserve estimation integrity.

2. **Minimal vs. Linear Solvers:** We compare the performance of the 5-point minimal solver against the linear 8-point algorithm, analyzing their stability under different conditions.
3. **Sensor Fusion:** We implement a 2-point algorithm that leverages known rotation (simulating IMU integration) to reduce problem complexity.
4. **Scale Recovery:** We address the scale ambiguity limitation of monocular vision by incorporating depth measurements, employing Arun's method to solve for the absolute transformation.

Finally, we extend our evaluation beyond standard datasets to a challenging drone racing scenario. This stress test reveals the limitations of pure visual estimation in high-dynamic regimes and motivates the need for tightly-coupled Visual-Inertial Odometry systems.

2 Procedure

2.1 Individual Work

2.1.1 Nister's 5-point Algorithm

Read the following paper.
[1] Nistér, David. "An efficient solution to the five-point relative pose problem." 2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Vol. 2. 2003. link here.

Questions:

1. **Outline the main computational steps required to get the relative pose estimate (up to scale) in Nister's 5-point algorithm.**

   Step I — Extract Nullspace Extraction:

   Build a 5×9 matrix, which consists of an epipolar constraint of 5 pairs of points, and calculate the zero-space vector $\tilde{X}, \tilde{Y}, \tilde{Z}, \widetilde{W}$ of the matrix. This step usually utilizes QR decomposition or SVD to achieve.

   Step II — Expand cubic constraints:

   Using the properties of the essential matrix, namely the trace constraint $2EE^TE - \text{trace}(EE^T)E = 0$. Express the eseential matrix as a linear combination of null space vectors $E = xX + yY + zZ + \omega W$, and then expand these cubic constraints.

   Step III — Gauss-Jordan Elimination:

   Build a 9×20 matrix A, and implement "Gauss-Jordan Elimination" to it.

Step IV — Build & Solve Polynomial Formulation:

Expand two 4×4 determinant polynomial of matrixes B and C. Then get the 10^{th} degree polynomial about the variable z through elimination.

Step V — Root Extraction:

Solve for the real roots of this 10^{th}-degree polynomial. This can be done by the Sturm sequence or by the eigenvalue decomposition of the adjoint matrix.

Step VI — Recover Pose & Disambiguation:

For each real root, the corresponding essential matrix E is recovered, and then the rotation matrix R and translation vector t are decomposed. The points are triangulated using Cheirality constraint (i.e. points must be in front of the camera), and the only correct physical and geometric solution is selected from a maximum of 10 possible mathematical solutions.

2. **Does the 5-point algorithm exhibit any degeneracy? (degeneracy = special arrangements of the 3D points or the camera poses under which the algorithm fails)**

   Conclusion — In the main context discussed in the paper (i.e. compared to uncalibrated methods), the 5-point algorithm overcomes Planar Structure Degeneracy, which is one of the big advantages over 7 or 8-point algorithms.

   Explanation:

   1. **Non-degradation of planar scenes:** Section 4 of the paper makes it clear that for uncalibrated methods, the algorithm fails when the scene points are coplanar (there is a continuous solution). However, under the calibrated 5-point method setting, coplanar points lead to at most 2-fold ambiguity, which can usually be resolved by a third view, or excluded by chiral constraints. Therefore, the paper concludes that the algorithm still works well in planar scenes.

   2. **General degradation:** While the paper focuses on its robustness in planar scenarios, according to the basics of epipolar geometry (and the implicit assumption in the paper), the essential matrix is not definable (baseline $t = 0$) without translational motion (pure rotation), at which point the algorithm degrades. In addition, multiple or infinite solutions also exist if 5 points and two optical centers lie on a specific critical surface such as twisted cubic, but it is considered robust for practical applications and planar scene comparisons highlighted in the paper.

3. **When used within RANSAC, what is the expected number of iterations the 5-point algorithm requires to find an outlier-free set?**

From the standard theory of RANSAC, we have the expected number of iterations N is determined by

$$N = \frac{\log(1-p)}{\log(1-\omega^s)}$$

which $\begin{cases} s$=5: The minimum number of points required by the 5-point algorithm, i.e. sample size $\\ \omega$(Inlier Ratio): Proportion of inliers (i.e. the probability of selecting an inlier point in a sample) $\\ p$(Confidence): The confidence that we want to find at least one full set of interior points $\end{cases}$

Since the term ω^s decreases exponentially with s, a smaller sample size s significantly reduces the required number of iterations. For the 5-point algorithm ($s = 5$), the probability of selecting an outlier-free set is ω^5, which is much higher than for the 7-point ($s = 7$) or 8-point ($s = 8$) algorithms. This reduction is critical for real-time structure and motion estimation.

E.g. Assuming a typical inlier ratio $\omega = 0.5$ and a desired confidence $p = 0.99$, the required iterations N are:

Method	**Sample Size (s)**	**Required Iterations (N)**
5-point algorithm	5	$\left\lceil \frac{\log(0.01)}{\log(1-0.5^5)} \right\rceil = 145$
7-point algorithm	7	$\left\lceil \frac{\log(0.01)}{\log(1-0.5^7)} \right\rceil = 588$
8-point algorithm	8	$\left\lceil \frac{\log(0.01)}{\log(1-0.5^8)} \right\rceil = 1177$

Additionally, if we consider the strictly theoretical mean number of trials $E[k]$ required to encounter the first outlier-free set (geometric distribution expectation), it is given by:

$$E[k] = \frac{1}{\omega^s}$$

For $\omega = 0.5$, the 5-point algorithm requires on average 32 trials, whereas the 8-point algorithm requires 256 trials. This efficiency allows the algorithm to be executed for hundreds of samples within a small time budget.

2.1.2 Designing a Minimal Solver

Can you do better than Nister? Nister's method is a minimal solver since it uses 5 point correspondences to compute the 5 degrees of freedom that define the relative pose (up to scale) between the two cameras (recall: each point induces a scalar equation). In the

presence of external information (e.g., data from other sensors), we may be able use less point correspondences to compute the relative pose.

Consider a drone flying in an unknown environment, and equipped with a camera and an Inertial Measurement Unit (IMU). We want to use the feature correspondences extracted in the images captured at two consecutive time instants t_1 and t_2 to estimate the relative pose (up to scale) between the pose at time t_1 and the pose at time t_2. Besides the camera, we can use the IMU (and in particular the gyroscopes in the IMU) to estimate the relative rotation between the pose of the camera at time t_1 and t_2.

You are required to solve the following problems:

1. **Assume the relative camera rotation between time and is known from the IMU. Design a minimal solver that computes the remaining degrees of freedom of the relative pose.**

   For the matching points p_1, p_2 on the normalized img plane, the epipolar constraint is

   $$p_2^T E p_1 = 0$$

   which essential matrix is $E = [t]_{\times} R$.

   Put E into the known rotation, we have

   $$p_2^T ([t]_{\times} R) p_1 = 0$$

   Let $p_1' = R p_1$, then we have $p_2^T [t]_{\times} p_1' = 0$ ①

   ∵ ① is equivalent to scalar triple product

   ∴ $t \cdot (p_1' \times p_2) = 0$, which means the translation vector t must be perpendicular to the vector $v = p_1' \times p_2$

   ∵ 1 point can only ensure t is on some plane

   ∴ we need 2 points to ensure the direction of t

   $$\begin{cases} \text{For } 1^{\text{st}} \text{ pair points: normal vector } v_1 = (R p_{1,a}) \times p_{2,a} \\ \text{For } 2^{\text{nd}} \text{ pair points: normal vector } v_2 = (R p_{1,b}) \times p_{2,b} \end{cases}$$

   ∵ translation vector t is perpendicular to v_1, v_2, thus

   $$t \sim v_1 \times v_2$$

   Then after normalizing, we have

   $$t \leftarrow \frac{t}{\|t\|}$$

2. **Describe the pseudo-code of a RANSAC algorithm using the minimal solver developed in point a) to compute the relative pose in presence of outliers (wrong correspondences).**

```
Algorithm: RANSAC for 2-point Relative Pose with Known Rotation

Input:
  Matches: A set of N feature correspondences S = {(p1_i, p2_i)}
  Rotation: Relative rotation matrix R (from IMU)
  Threshold: Inlier threshold epsilon (e.g., epipolar distance)
  Confidence: Desired confidence probability p (e.g., 0.99)
  InlierRatio: Estimated ratio of inliers w (e.g., 0.5)

Output:
  Best_t: The estimated translation direction
  Best_Inlier_Set: The set of consistent matches

1. Max_Iterations = log(1 - p) / log(1 - w^2)
2. Best_Score = 0
3. Best_t = [0, 0, 0]

4. For k = 1 to Max_Iterations:

    a. // Sampling
      Select 2 random correspondences {(p1_a, p2_a), (p1_b, p2_b)} from S.

    b. // Model Generation
      p1_prime_a = R * p1_a
      p1_prime_b = R * p1_b
      v1 = cross_product(p1_prime_a, p2_a)
      v2 = cross_product(p1_prime_b, p2_b)
      t_candidate = cross_product(v1, v2)
      Normalize t_candidate

    c. // Scoring (Count Inliers)
      Current_Score = 0
      Current_Inliers = {}
      E_candidate = [t_candidate]_cross * R  // Essential Matrix

      For each match (p1_i, p2_i) in S:
          // Calculate Sampson error or simple algebraic error
          error = abs( p2_i^T * E_candidate * p1_i )

          If error < epsilon:
              Current_Score = Current_Score + 1
              Add (p1_i, p2_i) to Current_Inliers

    d. // Update Best Model
      If Current_Score > Best_Score:
          Best_Score = Current_Score
```

```
            Best_t = t_candidate
            Best_Inlier_Set = Current_Inliers

    5. // Refinement
       Re-estimate t using all points in Best_Inlier_Set via Linear Least
       Squares or SVD.

    6. Return Best_t
```

which

- Sample Size — It merely needs to sample 2 points, as opposed to 5 for the Nister algorithm, which means that the number of cycles of RANSAC will be greatly reduced ($N \propto \omega^{-2}$ vs $N \propto \omega^{-5}$), which greatly improves the computation speed.
- Model Validation — epipolar geometric error $p_2^T E p_1$ is used for validation for R is known and t is assumed so that E is ensured.

2.2 Team Work

2.2.1 Initial Setup

This section is to calibrate the camera of the drone, namely to obtain the camera intrinsics and distortion coefficients.

Therefore, we need to check the implementation of `calibrateKeypoints` at first — Using the camera's intrinsic parameters and distortion coefficients, convert the pixel coordinates into undistorted 3D direction vectors. The correpsonding code is as follows:

```cpp
void calibrateKeypoints(const std::vector<cv::Point2f>& pts1,
                        const std::vector<cv::Point2f>& pts2,
                        opengv::bearingVectors_t& bearing_vector_1,
                        opengv::bearingVectors_t& bearing_vector_2) {
    //
    // For this part, we perform:
    //   1. Use the function cv::undistortPoints to rectify the keypoints.
    //   2. Return the bearing vectors for each keypoint.
    //

    std::vector<cv::Point2f> points1_rect, points2_rect;
    cv::undistortPoints(pts1, points1_rect, camera_params_.K,
    camera_params_.D);
    cv::undistortPoints(pts2, points2_rect, camera_params_.K,
    camera_params_.D);
```

```
14
15      for (auto const& pt: points1_rect){
16        opengv::bearingVector_t bearing_vector(pt.x, pt.y, 1); // focal
          length is 1 after undistortion
17        bearing_vector_1.push_back(bearing_vector.normalized());
18      }
19
20      for (auto const& pt: points2_rect){
21        opengv::bearingVector_t bearing_vector(pt.x, pt.y, 1); // focal
          length is 1 after undistortion
22        bearing_vector_2.push_back(bearing_vector.normalized());
23      }
24  }
```

Then, call this function in `cameraCallback` with just one line code:

```
calibrateKeypoints(pts1, pts2, bearing_vector_1, bearing_vector_2);
```

After calling `calibrateKeypoints`, `bearing_vector_1` and `bearing_vector_2` will contain the undistorted direction vectors. Subsequently, the line of code

```
Adapter adapter_mono(bearing_vector_1, bearing_vector_2);
```

in the code can correctly feed these data to the **RANSAC** algorithm.

2.2.2 2D-2D Correspondences

In this section, we implemented and evaluated three different algorithms for estimating the relative camera motion between consecutive frames using 2D-2D feature correspondences.

Experimental Methodology

1. **Feature Tracking:** Utilizing the SIFT-based tracker developed in the previous lab to obtain matched keypoints between the previous and current frames.
2. **Keypoint Calibration:** Since the geometric algorithms assume a calibrated pinhole camera model, we implemented `calibrateKeypoints` to undistort the raw pixel coordinates using the provided camera intrinsics (K) and distortion coefficients (D), converting them into normalized bearing vectors.
3. **Motion Estimation:** We utilized the OpenGV library to solve for the relative pose (R, t) using:
   - **Nistér's 5-point Algorithm:** A minimal solver for the essential matrix.
   - **Longuet-Higgins 8-point Algorithm:** A linear solver for the essential matrix.

- **2-point Algorithm (Known Rotation):** A minimal solver for translation, assuming the relative rotation is known (provided by ground truth in this experiment to simulate a high-precision IMU).

4. **Scale Correction:** As monocular vision inherently suffers from scale ambiguity, we normalized the estimated translation vector and rescaled it using the magnitude of the ground truth translation (`scaleTranslation`) to allow for meaningful visualization and comparison in Rviz.

Implementation Details

The core logic was implemented in the `cameraCallback` function. Before passing data to the solvers, we verified that the feature tracker returned a sufficient number of correspondences. We used `cv::undistortPoints` to map pixel coordinates to the normalized image plane.

For the pose estimation, we employed `opengv::sac::Ransac` to robustly estimate the model in the presence of outliers.

- For the **5-point** and **8-point** methods, we used `CentralRelativePoseSacProblem`.
- For the **2-point** method, we calculated the relative rotation from the ground truth odometry ($R_{\{\text{GT}\}} = R_{\{\text{prev}\}}^T R_{\{\text{curr}\}}$) and used `TranslationOnlySacProblem` to solve solely for the translation direction.

(*Note:* We need to add two more parameters to the launch file — `use_ransac` & `pose_estmator`.)

With the command `roslaunch lab_6 video_tracking.launch pose_estimator:=0 use_ransac:=<True/False>`, we have:

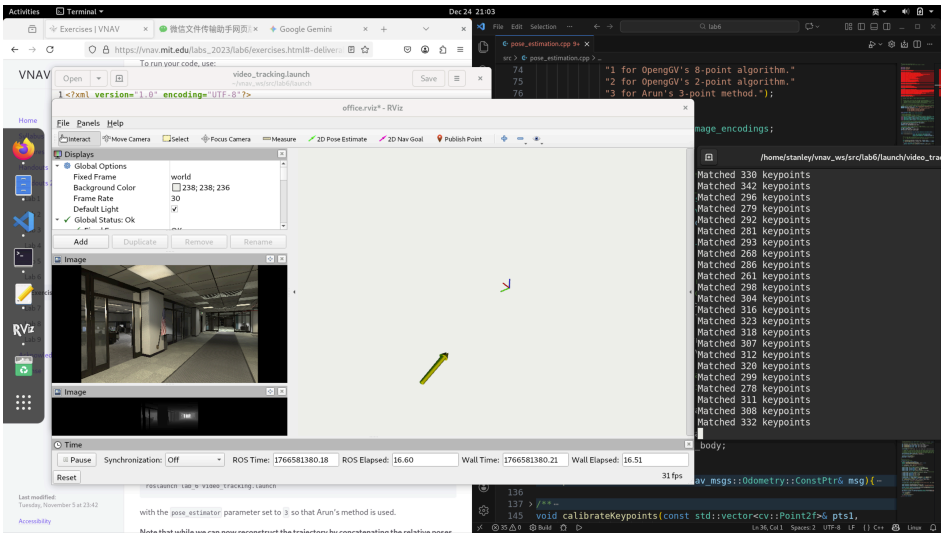

Figure 1: Running Snapshot

Performance Evaluation & Analysis

1. Impact of RANSAC on Estimation Accuracy To validate the necessity of outlier rejection in Visual Odometry (VO), we compared the performance of the 5-point algorithm with and without RANSAC. The detailed implementation is here and the results are visualized in the figure below:

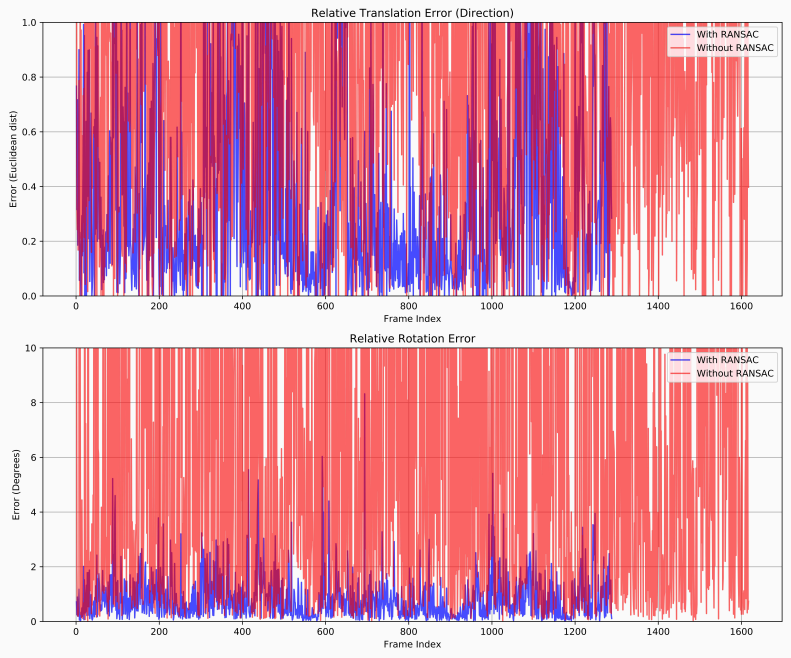

Figure 2: Visualization of the 5-points-based RPE Translation and Rotation Errors Comparison (With vs. Without RANSAC)

- **Without RANSAC (Red Line):** The error metrics exhibit significant instability, with frequent spikes in both rotation (exceeding 10 degrees) and translation errors. This confirms that even a small ratio of outliers (mismatched features) can severely corrupt the algebraic solution of the minimal solver.
- **With RANSAC (Blue Line):** The errors are drastically reduced and remain stable over the trajectory. The rotation error is consistently low (< 2 degrees), and the translation error (direction) is bounded, proving that the RANSAC framework effectively filters out spurious matches and selects the best geometric model.

2. Comparison of Different Algorithms We further compared the performance of the three geometric algorithms (all using RANSAC). The corresponding python script is here and the Relative Pose Errors (RPE) for translation and rotation are plotted below:

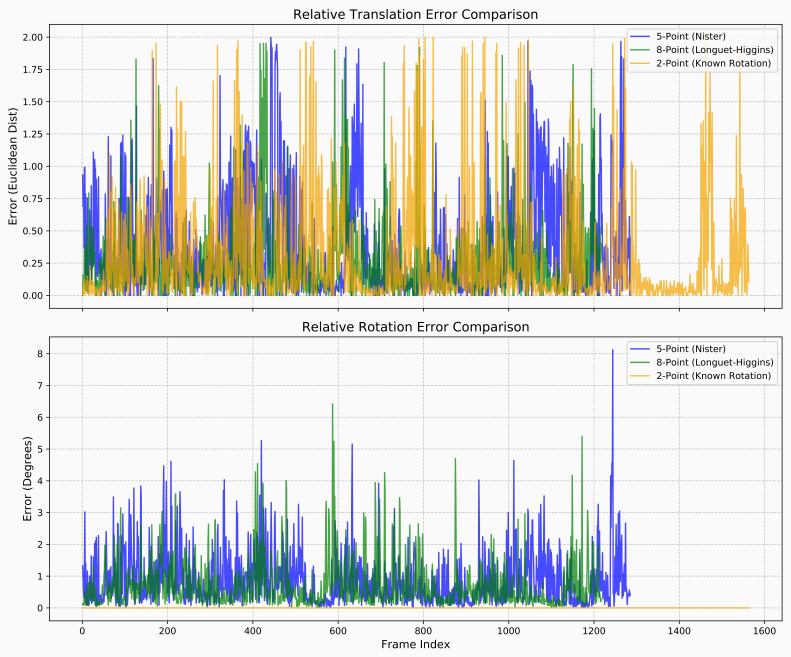

Figure 3: Visualization of the 5-points, 8-points, 2-points Algorithms Translation and Rotation Errors Comparison

1. **Rotation Error:**
   - The **2-point algorithm (Orange)** shows zero rotation error. This is expected and validates our implementation, as we fed the ground truth rotation into the solver.
   - The **5-point (Blue)** and **8-point (Green)** algorithms show comparable performance in rotation estimation, generally keeping the error under 3 degrees. The 5-point algorithm appears slightly more stable in certain segments, likely because it enforces the internal constraints of the essential matrix more strictly than the linear 8-point method.
2. **Translation Error:**
   - All three methods exhibit fluctuations in translation error. This is typical for monocular VO, especially when the camera motion is small (low parallax) or the scene structure is degenerate (e.g., planar scenes).
   - The **2-point algorithm**, despite having perfect rotation, still shows translation errors. This indicates that translation estimation is highly sensitive to feature noise, even when rotation is known. However, it generally maintains a lower error baseline compared to the 5-point and 8-point methods, demonstrating the benefit of reducing the degrees of freedom when reliable rotation data (e.g., from IMU) is available.

2.2.3 3D-3D Correspondences

This section is to estimate the relative camera motion by aligning two sets of 3D points (3D-3D registration), which allows for the recovery of the trajectory with **absolute scale**, eliminating the need for the ground-truth scaling trick used in Deliverable 4.

Experimental Methodology

We move beyond monocular vision constraints by incorporating depth information provided by the RGB-D sensor.

The workflow proceeds as follows:

1. **3D Point Generation:** For each matched keypoint (u, v) in the RGB image, we queried the corresponding value d from the registered depth image. The 2D keypoints were first converted to normalized bearing vectors (using camera intrinsics) and then scaled by their respective depth values to obtain 3D coordinates in the camera frame: $P_{\{\text{cam}\}} = d \cdot K^{\{-1\}} \cdot [u, v, 1]^T$.
2. **Point Cloud Registration:** We utilized **Arun's Method** (a closed-form solution based on Singular Value Decomposition) to find the rigid body transformation (R, t) that aligns the 3D points from the previous frame ($P_{\{\text{prev}\}}$) to the current frame ($P_{\{\text{curr}\}}$).
3. **Robust Estimation:** To handle noisy depth measurements and mismatches, the algorithm was wrapped in a RANSAC loop provided by OpenGV.

Implementation Details

The implementation was carried out in `cameraCallback` (Case 3).

- **Data Preparation:** We iterated through the matched keypoints, retrieved depth values using `depth.at<float>(y, x)`, and constructed two point clouds (`opengv::points_t`).
- **OpenGV Integration:** We used the `PointCloudAdapter` and `PointCloudSacProblem` classes from OpenGV to interface with the RANSAC solver. The threshold for RANSAC was set in meters (e.g., 0.1m) to reject points with large re-projection errors.
- **Scale Handling:** Unlike the previous 2D-2D experiments, we explicitly disabled the `scaleTranslation` parameter in the launch file. This ensures that the translation output is derived purely from the visual data, verifying the system's ability to recover the true physical scale of the motion.

(*Note:* We need to add one more parameter to the launch file — `scale_translation`.)

With the command `roslaunch lab_6 video_tracking.launch pose_estimator:=3 scale_translation:=0`, we have the running result.

Performance Evaluation & Analysis

We evaluated the accuracy of the 3D-3D estimation by comparing it against the ground truth. The code is here and the translation and rotation errors are visualized below:

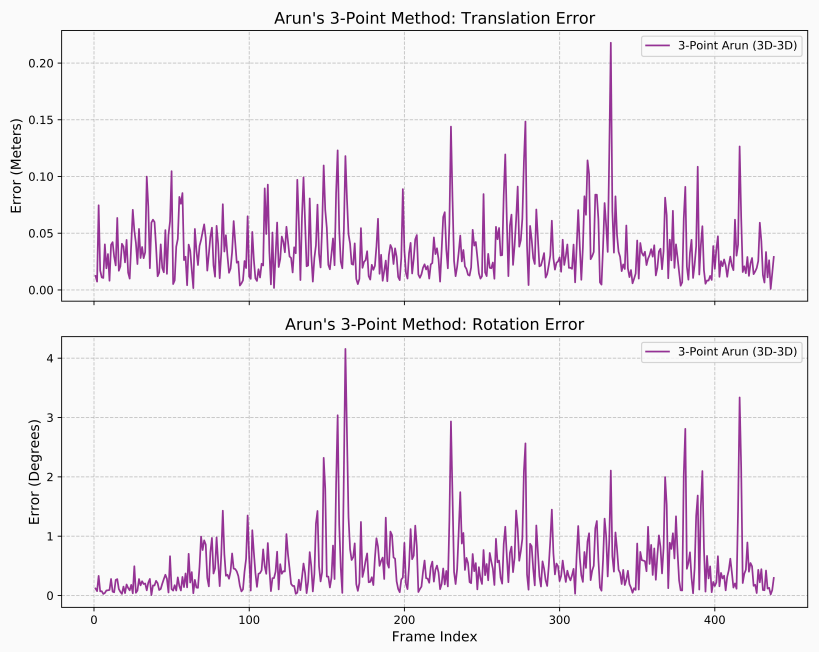

Figure 4: Visualization of the Translation and Rotation Errors of Arun's Algorithm with RANSAC

- **Absolute Scale Recovery:** The most significant observation is the translation error plot (top). Unlike Deliverable 4, where the error was unitless (direction only), here the error is measured in **meters**. The error remains low — typically bounded within $0.05 \sim 0.2$ meters — confirming that Arun's method successfully recovered the absolute scale of the drone's movement using the depth map.
- **Rotation Accuracy:** The rotation error (bottom) is extremely low, consistently staying below 1.0 degree. This indicates that 3D-3D registration provides a very strong constraint on orientation, often outperforming 2D-2D methods which can suffer from rotation-translation ambiguity in certain motion configurations.
- **Stability:** The trajectory estimation is robust and does not exhibit the scale drift issues common in monocular VO systems, demonstrating the advantage of RGB-D sensors for indoor navigation.

2.2.4 valuation on Drone Racing Dataset

Data Acquisition

To further validate the robustness of our implemented algorithms, we recorded a custom dataset using the drone racing simulator environment. The dataset (`vnav-lab4-group31.bag`)

involves high-speed maneuvers and complex trajectories, presenting a more challenging scenario than the provided `office.bag`.

Running Snapshot

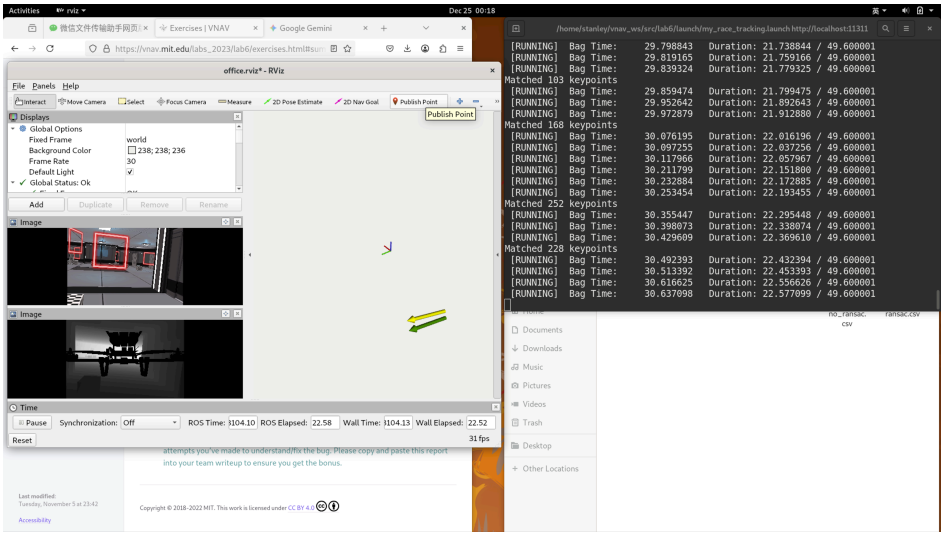

Figure 5: Drone Racing Dataset Running Result

Performance Analysis

We first evaluated the monocular algorithms (5-point, 8-point, and 2-point) on this custom dataset. The results are visualized below.

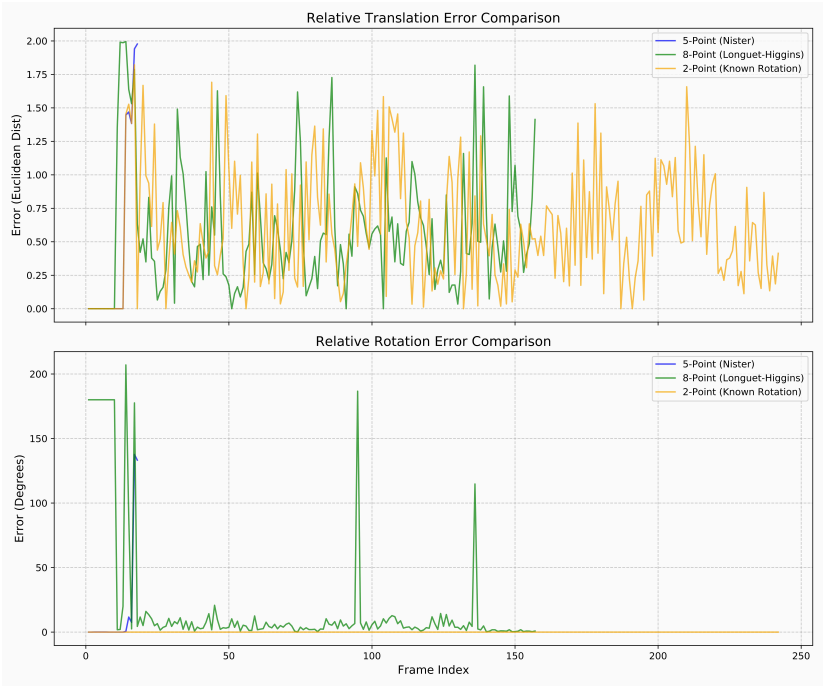

Figure 6: Visualization of the 5-points, 8-points, 2-points Algorithms Translation and Rotation Errors Comparison on Self-Dataset

Analysis — The results on the racing dataset highlight the limitations of classical geometric methods under aggressive motion:

- **Rotation:** The **2-point algorithm (Orange)** correctly maintains zero rotation error (bottom plot, flat line at 0), validating that our implementation correctly ingested the ground truth orientation. However, the **8-point algorithm (Green)** fails catastrophically in many frames, with rotation errors spiking to nearly 200 degrees. The **5-point algorithm (Blue)** performs slightly better but still exhibits significant instability compared to the static office scenario.
- **Translation:** All three methods show high translation errors. This degradation is likely caused by **motion blur** and **rapid viewpoint changes** typical in drone racing, which severely impact the quality of SIFT feature matching. When the feature tracker fails to provide high-quality correspondences, the geometric solvers cannot recover a reliable pose.

Furthermore, we tested the **3D-3D Arun's Method** to see if depth information could mitigate these issues.

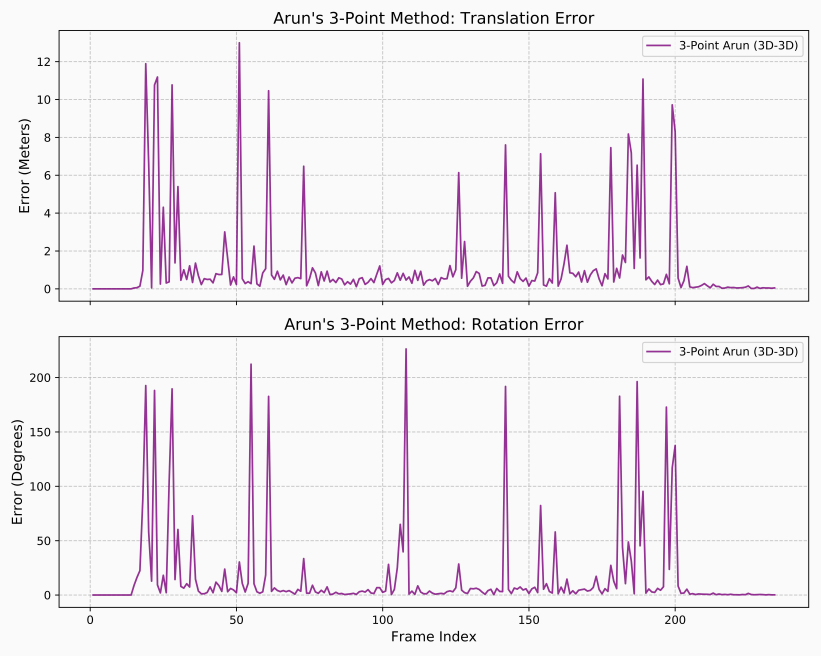

Figure 7: Visualization of the Translation and Rotation Errors of Arun's Algorithm with RANSAC on Self-Dataset

Analysis — Even with depth information and absolute scale recovery, Arun's method struggles to maintain a lock on the trajectory, as evidenced by the frequent spikes in both translation (reaching > 10 meters) and rotation errors.

- The periods of low error indicate that the algorithm works when the motion is smooth.

- However, the frequent large spikes confirm that **feature tracking is the bottleneck**. Since Arun's method relies on the same 2D feature correspondences as the monocular methods, if the 2D matches are incorrect (outliers ratio is too high for RANSAC to handle) or too few due to blur, the 3D registration will inevitably fail.
- This experiment demonstrates that while 3D-3D methods resolve scale ambiguity, they are not immune to the challenges of dynamic, high-speed flight.

3 Reflection and Analysis

Through the implementation and evaluation of various motion estimation algorithms, we have observed several key insights:

1. Our experiments unequivocally demonstrated that RANSAC is not merely an optimization but a necessity. Without RANSAC, the presence of even a small fraction of mismatched keypoints (outliers) caused the algebraic solvers (like the 5-point algorithm) to produce erratic and unusable pose estimates. RANSAC effectively acts as a filter, ensuring that the geometric model is derived only from consistent scene structure.

2. **Algorithm Trade-offs (5-point vs. 8-point vs. 2-point):**
   - The **5-point Nistér algorithm** generally outperformed the **8-point algorithm** in terms of stability. This aligns with theory, as the 5-point method enforces the internal constraints of the essential matrix ($\det(E) = 0, 2EE^TE - \text{trace}(EE^T)E = 0$), making it more robust to noise and degenerate planar configurations common in indoor environments.
   - The **2-point algorithm** demonstrated the power of sensor fusion. By decoupling rotation (assumed known from IMU/GT) from translation, the problem complexity drops significantly. However, our Bonus experiment showed that even with perfect rotation, translation estimation is still highly sensitive to feature tracking quality.

3. Monocular methods (2D-2D) suffer from inherent scale ambiguity. In our visualizations, we had to "cheat" by scaling the translation vector using ground truth. This limitation renders monocular VO insufficient for autonomous navigation without external references (like an IMU or a known object size). In contrast, **Arun's Method (3D-3D)** utilizing RGB-D data successfully recovered the absolute scale, providing metric state estimation essential for real-world robotics.

4. While the algorithms performed flawlessly on the slow-moving `office.bag`, they failed catastrophically on the high-speed racing dataset, which highlights a fundamental bottle-

neck: **Geometric solvers are only as good as the upstream feature tracker.** High-speed motion causes **motion blur** and rapid viewpoint changes, leading to poor SIFT matching. When the input correspondences are corrupted beyond the breakdown point of RANSAC, no geometric solver can recover the correct pose.

4 Conclusion

In this lab, we successfully implemented a comprehensive suite of motion estimation algorithms, ranging from monocular epipolar geometry (5-point, 8-point) to RGB-D point cloud registration (Arun's method). We integrated these solvers into a ROS-based pipeline and evaluated their performance using Relative Pose Error (RPE) metrics.

Our results confirmed that the **5-point algorithm with RANSAC** serves as a robust standard for monocular settings, while **Arun's method** leverages depth information to provide superior accuracy with recovered scale. Furthermore, the contrast between the successful office test and the challenging racing test underscored the limitations of pure vision systems in high-dynamic scenarios.

This lab provided a solid foundation in geometric computer vision. Moving forward, to address the drift observed in long trajectories and the failures in high-speed motion, future work would involve implementing Visual-Inertial Odometry to bridge tracking gaps and **Loop Closure** to correct accumulated drift, paving the way for a full SLAM system.

5 Source Code

- ***pose_estimation.cpp***

```cpp
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <memory>

#include <gflags/gflags.h>
#include <glog/logging.h>

#include <ros/ros.h>
#include <message_filters/subscriber.h>
#include <message_filters/synchronizer.h>
#include <message_filters/sync_policies/approximate_time.h>
#include <image_transport/image_transport.h>
#include <image_transport/subscriber_filter.h>
#include <sensor_msgs/image_encodings.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/Image.h>
#include <sensor_msgs/CameraInfo.h>
#include <geometry_msgs/PoseStamped.h>
#include <tf/tf.h>
#include <tf/transform_datatypes.h>
#include <tf_conversions/tf_eigen.h>
#include <tf/transform_broadcaster.h>
#include <nav_msgs/Odometry.h>

#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui/highgui.hpp>

#include <Eigen/Eigen>

#include <opencv2/core.hpp>
#include <opencv2/calib3d.hpp>
#include <opencv2/core/eigen.hpp>

// OpenGV
#include <opengv/sac/Ransac.hpp>
#include <opengv/sac_problems/relative_pose/TranslationOnlySacProblem.hpp>
#include <opengv/sac_problems/relative_pose/
CentralRelativePoseSacProblem.hpp>
#include <opengv/sac_problems/point_cloud/PointCloudSacProblem.hpp>
#include <opengv/relative_pose/CentralRelativeAdapter.hpp>
#include <opengv/relative_pose/methods.hpp>
```

```
42  #include <opengv/point_cloud/PointCloudAdapter.hpp>
43  #include <opengv/point_cloud/methods.hpp>
44
45  #include "lab6_utils.h"
46  #include "pose_estimation.h"
47  #include "tracker_shim.h"
48
49  DEFINE_bool(use_ransac, true, "Use Random Sample Consensus.");
50  DEFINE_bool(scale_translation, true, "Whether to scale estimated translation
    to match ground truth scale");
51  DEFINE_int32(pose_estimator, 0,
52               "Pose estimation algorithm, valid values are:"
53               "0 for OpengGV's 5-point algorithm."
54               "1 for OpengGV's 8-point algorithm."
55               "2 for OpengGV's 2-point algorithm."
56               "3 for Arun's 3-point method.");
57
58  using namespace std;
59  namespace enc = sensor_msgs::image_encodings;
60
61  using RansacProblem =
    opengv::sac_problems::relative_pose::CentralRelativePoseSacProblem;
62  using Adapter = opengv::relative_pose::CentralRelativeAdapter;
63  using RansacProblemGivenRot =
    opengv::sac_problems::relative_pose::TranslationOnlySacProblem;
64  using AdapterGivenRot = opengv::relative_pose::CentralRelativeAdapter;
65  using Adapter3D = opengv::point_cloud::PointCloudAdapter;
66  using RansacProblem3D =
    opengv::sac_problems::point_cloud::PointCloudSacProblem;
67
68  std::unique_ptr<TrackerWrapper> feature_tracker_;
69  ros::Publisher pub_pose_estimation_, pub_pose_gt_;
70  ros::Subscriber sub_cinfo_;
71  geometry_msgs::PoseStamped curr_pose_;
72  geometry_msgs::PoseStamped prev_pose_;
73
74  CameraParams camera_params_;
75  cv::Mat R_camera_body, t_camera_body;
76  cv::Mat T_camera_body;
77  geometry_msgs::Pose pose_camera_body;
78  tf::Transform transform_camera_body;
79
80
81  void poseCallbackTesse(const nav_msgs::Odometry::ConstPtr& msg){
82    curr_pose_.pose = msg->pose.pose;
83
84    tf::Transform current_pose;
```

```
85    tf::poseMsgToTF(curr_pose_.pose, current_pose);
86
87    tf::poseTFToMsg(current_pose * transform_camera_body, curr_pose_.pose);
88
89    curr_pose_.header.frame_id = "world";
90    pub_pose_gt_.publish(curr_pose_);
91  }
92
93  /**
94   * @brief      Compute 3D bearing vectors from pixel points
95   *
96   * @param[in]  pts1              Feature correspondences from camera 1
97   * @param[in]  pts2              Feature correspondences from camera 2
98   * @param      bearing_vector_1  Bearing vector to pts1 in camera 1
99   * @param      bearing_vector_2  Bearing vector to pts2 in camera 2
100  */
101 void calibrateKeypoints(const std::vector<cv::Point2f>& pts1,
102                         const std::vector<cv::Point2f>& pts2,
103                         opengv::bearingVectors_t& bearing_vector_1,
104                         opengv::bearingVectors_t& bearing_vector_2) {
105     std::vector<cv::Point2f> points1_rect, points2_rect;
106     cv::undistortPoints(pts1, points1_rect, camera_params_.K,
        camera_params_.D);
107     cv::undistortPoints(pts2, points2_rect, camera_params_.K,
        camera_params_.D);
108
109     for (auto const& pt: points1_rect){
110       opengv::bearingVector_t bearing_vector(pt.x, pt.y, 1); // focal length
        is 1 after undistortion
111       bearing_vector_1.push_back(bearing_vector.normalized());
112     }
113
114     for (auto const& pt: points2_rect){
115       opengv::bearingVector_t bearing_vector(pt.x, pt.y, 1); // focal length
        is 1 after undistortion
116       bearing_vector_2.push_back(bearing_vector.normalized());
117     }
118 }
119
120 /**
121  * @brief      Update pose estimate using previous absolue pose and estimated
    relative pose
122  *
123  * @param[in]  prev_pose         ground-truth absolute pose of previous frame
124  * @param[in]  relative_pose     estimated relative pose between current
    frame and previous frame
125  * @param      output            estimated absolute pose of current frame
```

```
126  */
127 void updatePoseEstimate(geometry_msgs::Pose const& prev_pose,
    geometry_msgs::Pose const& relative_pose, geometry_msgs::Pose& output) {
128   tf::Transform prev, relative;
129   tf::poseMsgToTF(prev_pose, prev);
130   tf::poseMsgToTF(relative_pose, relative);
131   tf::poseTFToMsg(prev*relative, output);
132 }
133
134 /**
135  * @brief      Given an estimated translation up to scale, return an absolute
    translation with correct scale using ground truth
136  *
137  * @param[in]  prev_pose         ground-truth absolute pose of previous frame
138  * @param[in]  curr_pose         ground-truth absolute pose of current frame
139  * @param      translation       estimated translation between current frame
    and previous frame
140  */
141 void scaleTranslation(geometry_msgs::Point& translation,
    geometry_msgs::PoseStamped const& prev_pose, geometry_msgs::PoseStamped
    const& curr_pose) {
142   if (!FLAGS_scale_translation) return;
143   tf::Transform prev, curr;
144   tf::poseMsgToTF(prev_pose.pose, prev);
145   tf::poseMsgToTF(curr_pose.pose, curr);
146   tf::Transform const relative_pose = prev.inverseTimes(curr);
147   double const translation_scale = relative_pose.getOrigin().length();
148   if (isnan(translation_scale) || isinf(translation_scale)) {
149     ROS_WARN("Failed to scale translation");
150     return;
151   }
152   double const old_scale = sqrt(pow(translation.x, 2) + pow(translation.y, 2)
    + pow(translation.z, 2));
153   translation.x *= translation_scale / old_scale;
154   translation.y *= translation_scale / old_scale;
155   translation.z *= translation_scale / old_scale;
156 }
157
158
159 /** @brief     (TODO) Compute Relative Pose Error (RPE) in translation and
    rotation.
160     @param[in] gt_t_prev_frame ground-truth transform for previous frame.
161     @param[in] gt_t_curr_frame ground-truth transform for current frame.
162     @param[in] est_t_prev_frame estimated transform for previous frame.
163     @param[in] est_t_curr_frame estimated transform for current frame.
164 */
165 void evaluateRPE(const tf::Transform& gt_t_prev_frame,
```

```
                const tf::Transform& gt_t_curr_frame,
                const tf::Transform& est_t_prev_frame,
                const tf::Transform& est_t_curr_frame) {
  tf::Transform const est_relative_pose =
  est_t_prev_frame.inverseTimes(est_t_curr_frame);
  tf::Transform const gt_relative_pose =
  gt_t_prev_frame.inverseTimes(gt_t_curr_frame);

  tf::Vector3 t_est = est_relative_pose.getOrigin();
  tf::Vector3 t_gt = gt_relative_pose.getOrigin();

  if (FLAGS_pose_estimator < 3) {
    if (t_est.length() > 1e-6) t_est.normalize();
    if (t_gt.length() > 1e-6) t_gt.normalize();
  }

  double translation_error = t_gt.distance(t_est);

  tf::Quaternion q_est = est_relative_pose.getRotation();
  tf::Quaternion q_gt = gt_relative_pose.getRotation();

  tf::Quaternion q_diff = q_gt.inverse() * q_est;
  double rotation_error_rad = q_diff.getAngle();

  double rotation_error_deg = rotation_error_rad * 180.0 / M_PI;

  static std::ofstream log_file;

  if (!log_file.is_open()) {
      std::string output_path = "/home/stanley/vnav_ws/src/lab6/log/";
      std::string filename;

      if (!FLAGS_use_ransac) {
          filename = output_path + "rpe_with_no_ransac.csv";
      } else {
          switch(FLAGS_pose_estimator) {
            case 0: filename = output_path + "rpe_5pt.csv"; break;
            case 1: filename = output_path + "rpe_8pt.csv"; break;
            case 2: filename = output_path + "rpe_2pt.csv"; break;
            case 3: filename = output_path + "rpe_3pt.csv"; break;
            default: filename = output_path + "rpe_unknown.csv"; break;
          }
      }

      log_file.open(filename, std::ios::out | std::ios::trunc);

```

```
210         if (log_file.is_open()) {
211             log_file << "frame,trans_error,rot_error_deg" << std::endl;
212             std::cout << "[Lab6] Saving RPE logs to: " << filename <<
                std::endl;
213         } else {
214             std::cerr << "[Lab6] Failed to open file: " << filename <<
                std::endl;
215         }
216     }
217
218     static int frame_count = 0;
219     if (log_file.is_open()) {
220         log_file << frame_count << "," << translation_error << "," <<
            rotation_error_deg << std::endl;
221     }
222     frame_count++;
223 }
224
225 /** @brief (TODO) This function is called when a new image is published. This
    is
226  *   where all the magic happens for this lab
227  *  @param[in]  rgb_msg    RGB Camera message
228  *  @param[in]  depth_msg  Depth Camera message
229  */
230 void cameraCallback(const sensor_msgs::ImageConstPtr &rgb_msg, const
    sensor_msgs::ImageConstPtr &depth_msg) {
231     cv::Mat bgr, depth;
232
233     try {
234       bgr = cv_bridge::toCvShare(rgb_msg, "bgr8")->image;
235       depth = cv_bridge::toCvShare(depth_msg, depth_msg->encoding)->image;
236     } catch (cv_bridge::Exception &e) {
237       ROS_ERROR("Could not convert rgb or depth images.");
238     }
239
240     cv::Mat view = bgr.clone();
241
242     static cv::Mat prev_bgr = bgr.clone();
243     static cv::Mat prev_depth = depth.clone();
244
245     std::pair<std::vector<cv::KeyPoint>, std::vector<cv::KeyPoint>>
    matched_kp_1_kp_2;
246     feature_tracker_->track(prev_bgr, bgr, &matched_kp_1_kp_2);
247
248     int N = matched_kp_1_kp_2.first.size();
249     std::cout << "Matched " << N << " keypoints" << std::endl;
250     std::vector<cv::Point2f> pts1, pts2;
```

```
251
252   cv::KeyPoint::convert(matched_kp_1_kp_2.first, pts1);
253   cv::KeyPoint::convert(matched_kp_1_kp_2.second, pts2);
254
255   opengv::bearingVectors_t bearing_vector_1, bearing_vector_2;
256
257   if (!pts1.empty()) {
258     calibrateKeypoints(pts1, pts2, bearing_vector_1, bearing_vector_2);
259   }
260
261   Adapter adapter_mono (bearing_vector_1, bearing_vector_2);
262
263   geometry_msgs::PoseStamped pose_estimation;
264   tf::poseTFToMsg(tf::Pose::getIdentity(), pose_estimation.pose);
265
266   geometry_msgs::Pose relative_pose_estimate = pose_estimation.pose;
267
268   switch(FLAGS_pose_estimator) {
269   case 0: {
270     // 5-points Algorithm
271     static constexpr size_t min_nr_of_correspondences = 5;
272     if (adapter_mono.getNumberCorrespondences() >= min_nr_of_correspondences)
     {
273       if (!FLAGS_use_ransac) {
274         std::shared_ptr<RansacProblem> ransac_problem =
          std::make_shared<RansacProblem>(adapter_mono, RansacProblem::NISTER);
275         opengv::sac::Ransac<RansacProblem> ransac;
276         ransac.sac_model_ = ransac_problem;
277
278         ransac.max_iterations_ = 1;
279         ransac.threshold_ = 100000.0;
280
281         if(ransac.computeModel()) {
282               relative_pose_estimate = eigen2Pose(ransac.model_coefficients_);
283         } else {
284               ROS_WARN("5-point non-RANSAC failed");
285         }
286       } else {
287         std::shared_ptr<RansacProblem> ransac_problem =
          std::make_shared<RansacProblem>(adapter_mono, RansacProblem::NISTER);
288
289         opengv::sac::Ransac<RansacProblem> ransac;
290         ransac.sac_model_ = ransac_problem;
291
292         ransac.threshold_ = 0.00001;
293         ransac.max_iterations_ = 100;
294
```

```
295           ransac.computeModel();
296
297           relative_pose_estimate = eigen2Pose(ransac.model_coefficients_);
298         }
299       } else {
300         ROS_WARN("Not enough correspondences to compute pose estimation using"
301                  " Nister's algorithm.");
302       }
303       break;
304     }
305     case 1: {
306       // 8-points Algorithm
307       static constexpr size_t min_nr_of_correspondences = 8;
308
309       if (adapter_mono.getNumberCorrespondences() >= min_nr_of_correspondences)
          {
310         if (!FLAGS_use_ransac) {
311           std::shared_ptr<RansacProblem> ransac_problem =
              std::make_shared<RansacProblem>(adapter_mono,
              RansacProblem::EIGHTPT);
312           opengv::sac::Ransac<RansacProblem> ransac;
313           ransac.sac_model_ = ransac_problem;
314
315           ransac.max_iterations_ = 1;
316           ransac.threshold_ = 100000.0;
317
318           if(ransac.computeModel()) {
319               relative_pose_estimate = eigen2Pose(ransac.model_coefficients_);
320           }
321         } else {
322           std::shared_ptr<RansacProblem> ransac_problem =
              std::make_shared<RansacProblem>(adapter_mono,
              RansacProblem::EIGHTPT);
323
324           opengv::sac::Ransac<RansacProblem> ransac;
325           ransac.sac_model_ = ransac_problem;
326
327           ransac.threshold_ = 0.00001;
328           ransac.max_iterations_ = 100;
329
330           ransac.computeModel();
331
332           relative_pose_estimate = eigen2Pose(ransac.model_coefficients_);
333         }
334       } else {
335         ROS_WARN("Not enough correspondences to compute pose estimation using"
336                  " Longuet-Higgins' algorithm.");
```

```
337       }
338       break;
339     }
340     case 2: {
341       // 2-point Algorithm
342       static constexpr size_t min_nr_of_correspondences = 2;
343       if (adapter_mono.getNumberCorrespondences() >= min_nr_of_correspondences)
          {
344         tf::Transform curr_frame, prev_frame;
345         tf::poseMsgToTF(curr_pose_.pose, curr_frame);
346         tf::poseMsgToTF(prev_pose_.pose, prev_frame);
347         Eigen::Matrix3d rotation;
348         tf::matrixTFToEigen(prev_frame.inverseTimes(curr_frame).getBasis(),
          rotation);
349         adapter_mono.setR12(rotation);
350
351         if (!FLAGS_use_ransac) {
352           std::shared_ptr<RansacProblemGivenRot> ransac_problem =
          std::make_shared<RansacProblemGivenRot>(adapter_mono);
353           opengv::sac::Ransac<RansacProblemGivenRot> ransac;
354           ransac.sac_model_ = ransac_problem;
355
356           ransac.max_iterations_ = 1;
357           ransac.threshold_ = 100000.0;
358
359           if(ransac.computeModel()) {
360               relative_pose_estimate = eigen2Pose(ransac.model_coefficients_);
361           }
362         } else {
363           std::shared_ptr<RansacProblemGivenRot> ransac_problem =
          std::make_shared<RansacProblemGivenRot>(adapter_mono);
364
365           opengv::sac::Ransac<RansacProblemGivenRot> ransac;
366           ransac.sac_model_ = ransac_problem;
367
368           ransac.threshold_ = 0.00001;
369           ransac.max_iterations_ = 100;
370
371           ransac.computeModel();
372
373           relative_pose_estimate = eigen2Pose(ransac.model_coefficients_);
374
375           // ***************************** end solution
          *****************************
376         }
377       } else {
```

```
378         ROS_WARN("Not enough correspondences to estimate relative translation
            using 2pt algorithm.");
379       }
380       break;
381     }
382     case 3: {
383       // Arun's 3-point algorithm
384       for (int i=0; i<N; i++) {
385         double d1 = double( prev_depth.at<float>(std::floor(pts1[i].y),
            std::floor(pts1[i].x)) ) ;
386         double d2 = double( depth.at<float>(std::floor(pts2[i].y),
            std::floor(pts2[i].x)) ) ;
387
388         bearing_vector_1[i] /= bearing_vector_1[i](2,0);
389         bearing_vector_2[i] /= bearing_vector_2[i](2,0);
390
391         bearing_vector_1[i] *= d1;
392         bearing_vector_2[i] *= d2;
393       }
394
395       opengv::points_t cloud_1, cloud_2;
396       for (auto i = 0ul; i < bearing_vector_1.size(); i++) {
397           cloud_1.push_back(bearing_vector_1[i]);
398           cloud_2.push_back(bearing_vector_2[i]);
399       }
400
401       Adapter3D adapter_3d(cloud_1, cloud_2);
402
403       static constexpr int min_nr_of_correspondences = 3;
404       if (adapter_3d.getNumberCorrespondences() >= min_nr_of_correspondences) {
405         if (!FLAGS_use_ransac){
406           std::shared_ptr<RansacProblem3D> ransac_problem =
              std::make_shared<RansacProblem3D>(adapter_3d);
407
408           opengv::sac::Ransac<RansacProblem3D> ransac;
409           ransac.sac_model_ = ransac_problem;
410
411           ransac.max_iterations_ = 1;
412           ransac.threshold_ = 1000.0;
413
414           if(ransac.computeModel()) {
415                relative_pose_estimate = eigen2Pose(ransac.model_coefficients_);
416           }
417         } else{
418           std::shared_ptr<RansacProblem3D> ransac_problem =
              std::make_shared<RansacProblem3D>(adapter_3d);
419
```

```
420         opengv::sac::Ransac<RansacProblem3D> ransac;
421         ransac.sac_model_ = ransac_problem;
422 
423         ransac.threshold_ = 0.1;
424         ransac.max_iterations_ = 100;
425 
426         if(ransac.computeModel()) {
427              relative_pose_estimate = eigen2Pose(ransac.model_coefficients_);
428         }
429       }
430     } else {
431       ROS_WARN("Not enough correspondences to estimate absolute transform
          using Arun's 3pt algorithm.");
432     }
433     break;
434   }
435   default: {
436     ROS_ERROR("Wrong pose_estimator flag!");
437   }
438   }
439   if (FLAGS_pose_estimator < 3) {
440     scaleTranslation(relative_pose_estimate.position, prev_pose_,
      curr_pose_);
441   }
442 
443   updatePoseEstimate(prev_pose_.pose, relative_pose_estimate,
      pose_estimation.pose);
444 
445   tf::Transform gt_t_prev_frame, gt_t_curr_frame;
446   tf::Transform est_t_prev_frame, est_t_curr_frame;
447   tf::poseMsgToTF(pose_estimation.pose, est_t_curr_frame);
448   tf::poseMsgToTF(curr_pose_.pose, gt_t_curr_frame);
449   tf::poseMsgToTF(prev_pose_.pose, est_t_prev_frame);
450   tf::poseMsgToTF(prev_pose_.pose, gt_t_prev_frame);
451 
452   evaluateRPE(gt_t_prev_frame,
      gt_t_curr_frame,est_t_prev_frame,est_t_curr_frame);
453 
454   pose_estimation.header.frame_id = "world";
455   pub_pose_estimation_.publish(pose_estimation);
456 
457   prev_bgr = bgr.clone();
458   prev_depth = depth.clone();
459   prev_pose_ = curr_pose_;
460 }
461 
462 int main(int argc, char** argv) {
```

```
463   google::InitGoogleLogging(argv[0]);
464   google::ParseCommandLineFlags(&argc, &argv, true);
465   google::InstallFailureSignalHandler();
466
467   ros::init(argc, argv, "keypoint_trackers");
468   ros::NodeHandle local_nh("~");
469
470   camera_params_.K = cv::Mat::zeros(3, 3, CV_64F);
471   camera_params_.K.at<double>(0,0) = 415.69219381653056;
472   camera_params_.K.at<double>(1,1) = 415.69219381653056;
473   camera_params_.K.at<double>(0,2) = 360.0;
474   camera_params_.K.at<double>(1,2) = 240.0;
475   camera_params_.D = cv::Mat::zeros(cv::Size(5,1),CV_64F);
476
477   T_camera_body = cv::Mat::zeros(cv::Size(4,4),CV_64F);
478   T_camera_body.at<double>(0,2) = 1.0;
479   T_camera_body.at<double>(1,0) = -1.0;
480   T_camera_body.at<double>(1,3) = 0.05;
481   T_camera_body.at<double>(2,1) = -1.0;
482   T_camera_body.at<double>(3,3) = 1.0;
483
484   R_camera_body = T_camera_body(cv::Range(0,3),cv::Range(0,3));
485   t_camera_body = T_camera_body(cv::Range(0,3),cv::Range(3,4));
486   pose_camera_body = cv2Pose(R_camera_body,t_camera_body);
487
488   tf::poseMsgToTF(pose_camera_body, transform_camera_body);
489
490   feature_tracker_.reset(new TrackerWrapper());
491
492   auto pose_sub = local_nh.subscribe("/ground_truth_pose", 10,
      poseCallbackTesse);
493
494   image_transport::ImageTransport it(local_nh);
495   image_transport::SubscriberFilter sf_rgb(it, "/rgb_images_topic", 1);
496   image_transport::SubscriberFilter sf_depth(it, "/depth_images_topic", 1);
497
498   typedef message_filters::sync_policies::ApproximateTime<sensor_msgs::Image,
      sensor_msgs::Image> MySyncPolicy;
499   message_filters::Synchronizer<MySyncPolicy> sync(MySyncPolicy(10), sf_rgb,
      sf_depth);
500   sync.registerCallback(cameraCallback);
501
502   pub_pose_gt_ = local_nh.advertise<geometry_msgs::PoseStamped>("/
      gt_camera_pose", 1);
503   pub_pose_estimation_ = local_nh.advertise<geometry_msgs::PoseStamped>("/
      camera_pose", 1);
504
```

```
  while (ros::ok()) {
    ros::spinOnce();
    cv::waitKey(1);
  }
  cv::destroyAllWindows();

  return EXIT_SUCCESS;
}
```

- ***rpe_comparison.py***

```py
import matplotlib.pyplot as plt
import pandas as pd
import os

path_ransac = '/home/stanley/vnav_ws/src/lab6/log/rpe_with_ransac.csv'
path_no_ransac = '/home/stanley/vnav_ws/src/lab6/log/rpe_with_no_ransac.csv'

df_ransac = pd.read_csv(path_ransac) if os.path.exists(path_ransac) else None
df_no_ransac = pd.read_csv(path_no_ransac) if os.path.exists(path_no_ransac) else None

fig, axs = plt.subplots(2, 1, figsize=(12, 10))

# --- 1. 绘制平移误差 ---
if df_ransac is not None:
    axs[0].plot(df_ransac['frame'].to_numpy(), df_ransac['trans_error'].to_numpy(), label='With RANSAC', color='blue', alpha=0.7)

if df_no_ransac is not None:
    axs[0].plot(df_no_ransac['frame'].to_numpy(), df_no_ransac['trans_error'].to_numpy(), label='Without RANSAC', color='red', alpha=0.6)

axs[0].set_title('Relative Translation Error (Direction)')
axs[0].set_ylabel('Error (Euclidean dist)')
axs[0].set_xlabel('Frame Index')
axs[0].legend(
    loc='upper right',
    bbox_to_anchor=(1.0, 1.0)
)
axs[0].grid(True)
axs[0].set_ylim(0, 1.0)

```

```
# --- 2. 绘制旋转误差 ---
if df_ransac is not None:
    axs[1].plot(df_ransac['frame'].to_numpy(), df_ransac['rot_error_deg'].to_numpy(), label='With RANSAC', color='blue', alpha=0.7)

if df_no_ransac is not None:
    axs[1].plot(df_no_ransac['frame'].to_numpy(), df_no_ransac['rot_error_deg'].to_numpy(), label='Without RANSAC', color='red', alpha=0.6)

axs[1].set_title('Relative Rotation Error')
axs[1].set_ylabel('Error (Degrees)')
axs[1].set_xlabel('Frame Index')
axs[1].legend(
    loc='upper right',
    bbox_to_anchor=(1.0, 1.0)
)
axs[1].grid(True)
axs[1].set_ylim(0, 10)

plt.tight_layout()
plt.savefig('rpe_comparison.png', dpi=1200)
plt.show()
```

- ***algorithm_comparison.py***

```py
import matplotlib.pyplot as plt
import pandas as pd
import os
import numpy as np

base_path = '/home/stanley/vnav_ws/src/lab6/log/'
files = {
    '5-Point (Nister)': os.path.join(base_path, 'rpe_5pt.csv'),
    '8-Point (Longuet-Higgins)': os.path.join(base_path, 'rpe_8pt.csv'),
    '2-Point (Known Rotation)': os.path.join(base_path, 'rpe_2pt.csv')
}

colors = {
    '5-Point (Nister)': 'blue',
    '8-Point (Longuet-Higgins)': 'green',
    '2-Point (Known Rotation)': 'orange'
}

```

```python
data_frames = {}
for name, path in files.items():
    if os.path.exists(path):
        print(f"Loading {name} from {path}...")
        data_frames[name] = pd.read_csv(path)
    else:
        print(f"[WARNING] File not found: {path}")

if not data_frames:
    print("No data found! Please check file paths.")
    exit()

fig, axs = plt.subplots(2, 1, figsize=(12, 10), sharex=True)

# --- 1. 绘制平移误差 (Translation Error) ---
for name, df in data_frames.items():
    frames = df['frame'].to_numpy()
    trans_err = df['trans_error'].to_numpy()

    axs[0].plot(frames, trans_err, label=name, color=colors[name], alpha=0.7, linewidth=1.5)

axs[0].set_title('Relative Translation Error Comparison', fontsize=14)
axs[0].set_ylabel('Error (Euclidean Dist)', fontsize=12)
axs[0].grid(True, which='both', linestyle='--', alpha=0.7)
axs[0].legend(
    loc='upper right',
    bbox_to_anchor=(1.0, 1.0)
)
axs[0].set_ylim(0, 1.0)

# --- 2. 绘制旋转误差 (Rotation Error) ---
for name, df in data_frames.items():
    frames = df['frame'].to_numpy()
    rot_err = df['rot_error_deg'].to_numpy()

    axs[1].plot(frames, rot_err, label=name, color=colors[name], alpha=0.7, linewidth=1.5)

axs[1].set_title('Relative Rotation Error Comparison', fontsize=14)
axs[1].set_ylabel('Error (Degrees)', fontsize=12)
axs[1].set_xlabel('Frame Index', fontsize=12)
axs[1].grid(True, which='both', linestyle='--', alpha=0.7)
axs[1].legend(
    loc='upper right',
    bbox_to_anchor=(1.0, 1.0)
```

```
)
axs[1].set_ylim(0, 5.0)

plt.tight_layout()

output_file = 'algorithm_comparison.png'
plt.savefig(output_file, dpi=1200)
plt.show()
```

- ***arun_3d.py***

```py
import matplotlib.pyplot as plt
import pandas as pd
import os
import numpy as np

file_path = '/home/stanley/vnav_ws/src/lab6/log/rpe_3pt.csv'

if os.path.exists(file_path):
    print(f"Loading 3D-3D results from {file_path}...")
    df = pd.read_csv(file_path)
else:
    print(f"[ERROR] File not found: {file_path}")
    print("Please make sure you ran the simulation with pose_estimator=3")
    exit()

fig, axs = plt.subplots(2, 1, figsize=(10, 8), sharex=True)

frames = df['frame'].to_numpy()
trans_err = df['trans_error'].to_numpy()
rot_err = df['rot_error_deg'].to_numpy()

# --- 1. 绘制平移误差 (Translation Error) ---
axs[0].plot(frames, trans_err, label='3-Point Arun (3D-3D)', color='purple',
alpha=0.8, linewidth=1.5)

axs[0].set_title('Arun\'s 3-Point Method: Translation Error', fontsize=14)
axs[0].set_ylabel('Error (Meters)', fontsize=12)

axs[0].grid(True, which='both', linestyle='--', alpha=0.7)
axs[0].legend(
    loc='upper right',
    bbox_to_anchor=(1.0, 1.0)
)

```

```
axs[0].set_ylim(0, 0.5)

# --- 2. 绘制旋转误差 (Rotation Error) ---
axs[1].plot(frames, rot_err, label='3-Point Arun (3D-3D)', color='purple', alpha=0.8, linewidth=1.5)

axs[1].set_title('Arun\'s 3-Point Method: Rotation Error', fontsize=14)
axs[1].set_ylabel('Error (Degrees)', fontsize=12)
axs[1].set_xlabel('Frame Index', fontsize=12)
axs[1].grid(True, which='both', linestyle='--', alpha=0.7)
axs[1].legend(
    loc='upper right',
    bbox_to_anchor=(1.0, 1.0)
)
axs[1].set_ylim(0, 5.0)

plt.tight_layout()
plt.savefig('arun_3d.png', dpi=1200)
plt.show()
```