Monocular Visual Odometry

Feiyu Chen

2019 Mar 20th
Final Project of EECS-432 Advanced Computer Vision
Teacher: Prof. Ying Wu
Github: https://github.com/felixchenfy/Monocular-Visual-Odometry

Contents

1. Introduction

1.1 Concept of Monocular Visual Odometry

What is visual odometry (VO)? Imagine you are driving a car through the street. You can know that your car is moving forward by seeing the landscape moving backward outside the window. This is an example of visual odometry, where you deduce the movement from the visual information. Here "odometry" means movement, and "visual" means images and video.

The task of Monocular VO is to compute the trajectory of a moving camera from an image sequence (i.e. video) taken by this monocular camera (which is the same type as the one on our laptop). The main idea of the VO algorithms is to build a map of the environment by triangulating 3D map points, and then compare the current image with the map to know where the camera is.

Application: Visual odometry is the basic component of a visual SLAM system. It can be used for the localization of mobile robots, flying quadcopters, and autonomous driving. The technology of VO can also be used for the task of Structure from Motion, which is to construct the 3D structure of the scene from a set of images.

1.2 Overview of my project

In this project, I implemented a monocular visual odometry system from "scratch". Here scratch means that I used the basic functions from OpenCV and g2o library to code up the whole system, instead of calling some high-level library tools. Monocular VO is a difficult task. The VO system I implemented could only recover the camera trajectory from videos taken under some simple scenarios.

The implemented VO system was tested on a video from the New Tsukuba Stereo Database[1]. **Fig. 1.1** displays 10 snapshots of the video in chronological order. As you can see from the images, the camera is first moving forward (1~4), then turning right (5~8), then moving forward again (9~10).

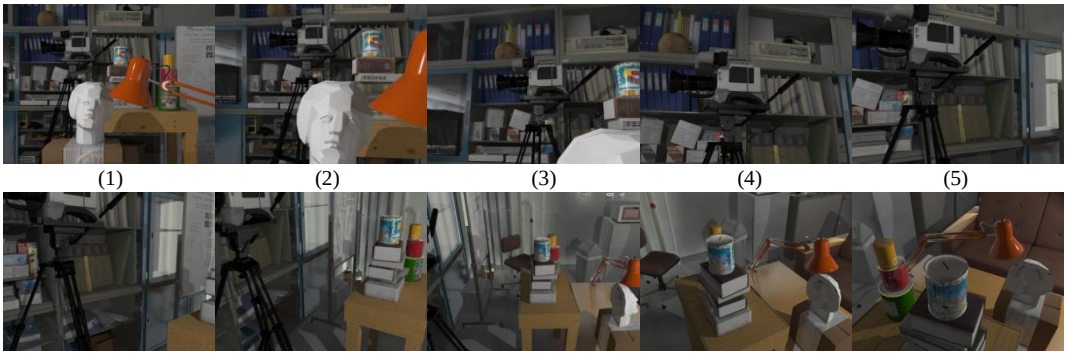
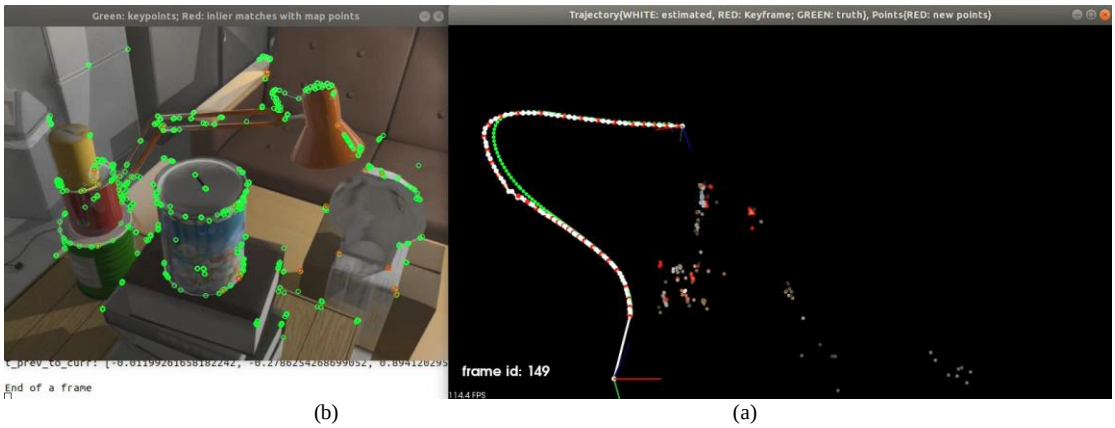


Fig. 1.1. Ten snapshots taken from the video in chronological order. The camera is first moving forward, then turning right, and moving forward again.

The corresponding camera trajectory of the above images is shown below in **Fig. 1.2**(b). The green line is the ground truth trajectory of the camera, and the white line is the trajectory computed by the visual odometry system. As seen from the figure, the estimation result is quite close to the ground truth, which demonstrates that my VO system does work. The video animations are put on my Github[2].



(b) (a)

Fig. 1.2. The camera trajectory estimated from video. (a) Current view of the camera, where small circles are the image keypoints. (b) Camera trajectory and map points. Green line is the ground truth, and the while line is the estimated camera trajectory. Red dots on green line represent the keyframes.

[1] http://cvlab.cs.tsukuba.ac.jp/
[2] https://github.com/felixchenfy/Monocular-Visual-Odometry

1.3 Snapshots of resultant trajectory

A sequence of resultant images is shown in **Fig. 1.3** and **Fig. 1.4** below to display the whole process of estimating trajectory from video:

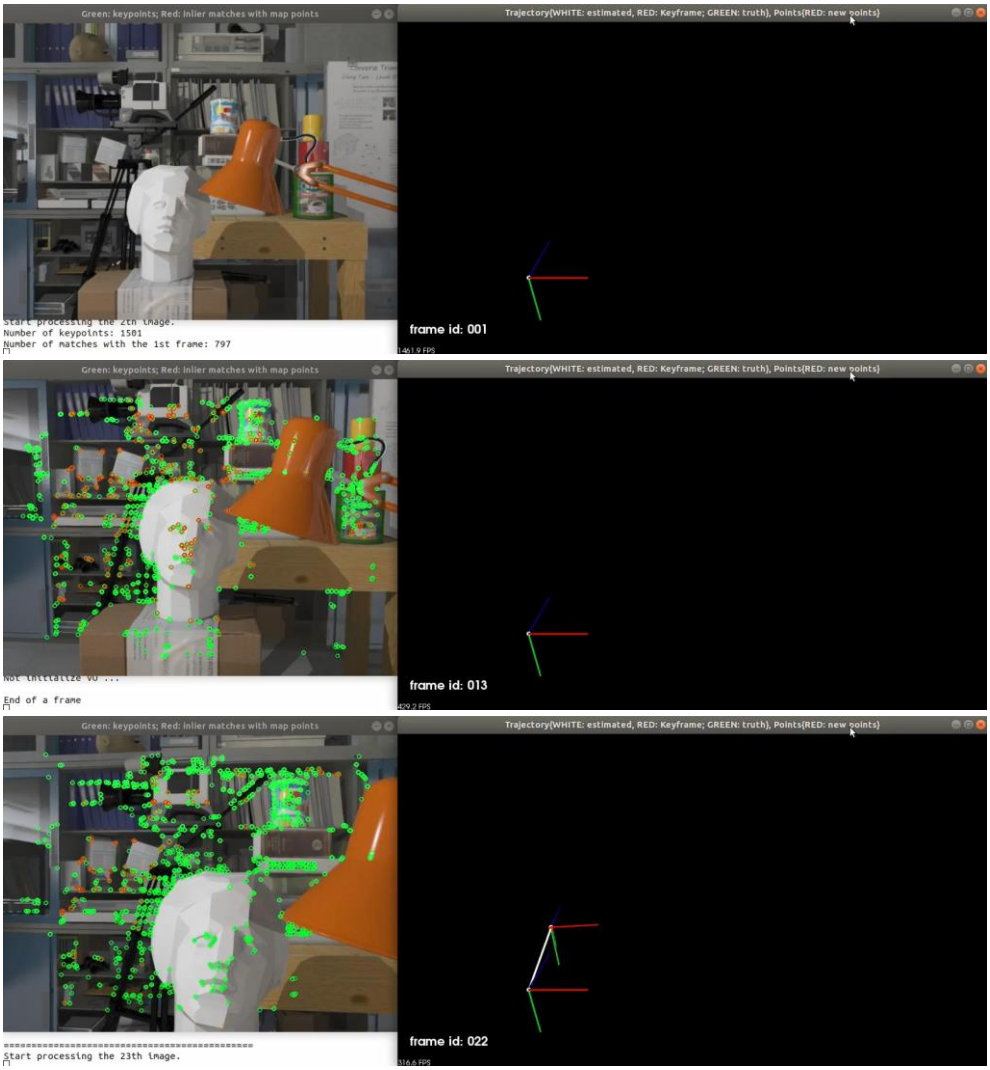
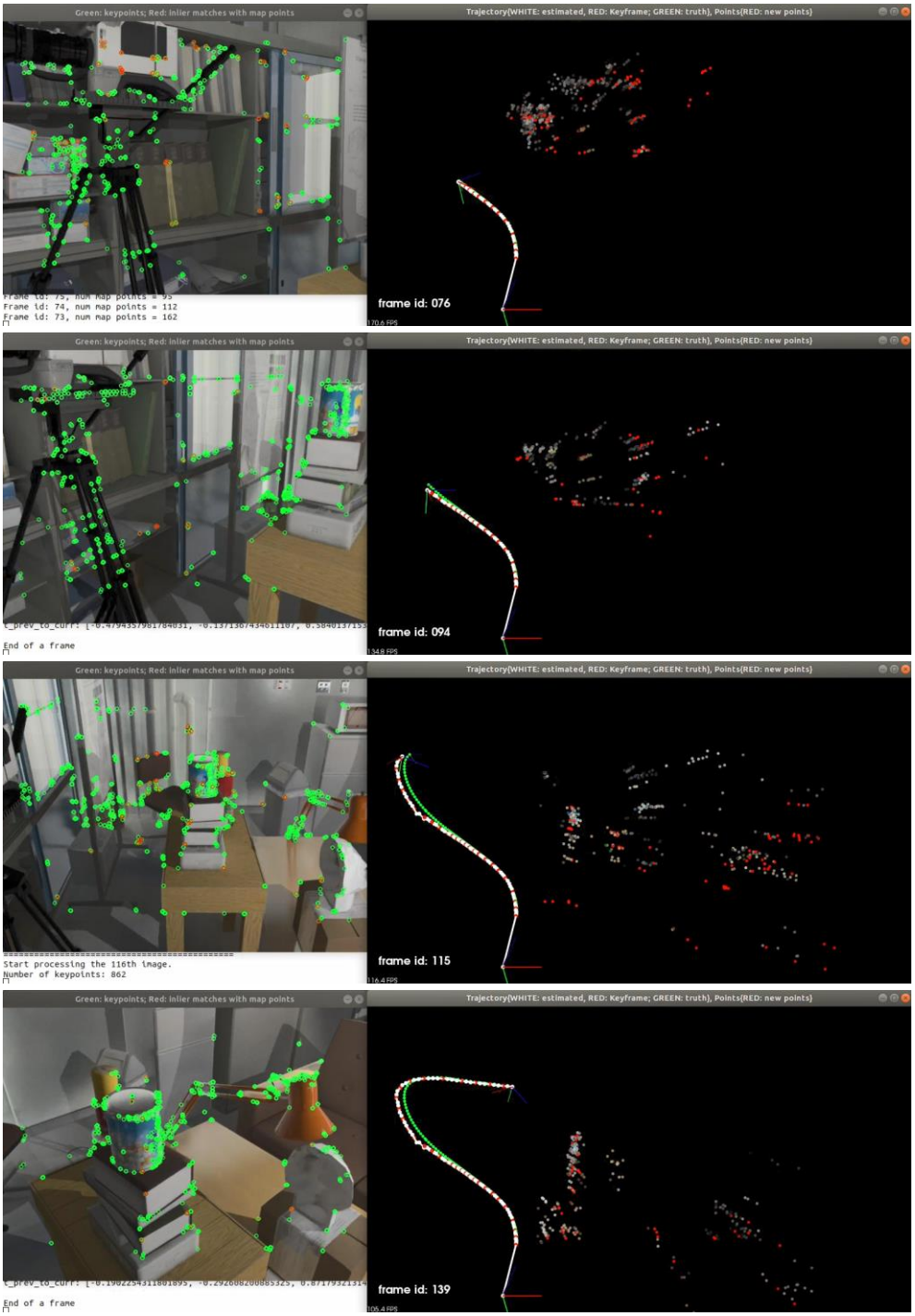


Fig. 1.3. Initialization stage of the monocular visual odometry. In frame 22, the initialization succeeds, which then returns the first estimated camera pose.



(a) (b)

Fig. 1.4. Tracking stage of the visual odometry. (a) The left image is the current video frame. (b) The image on the right displays the current map points and the estimated camera trajectory.

1.4 Notations

Some abbreviations and notations used in this project are listed below.

VO: Visual odometry.

Frame: (1) an image taken by the camera, (2) or coordinate, such as world frame.

Pos: Position of an object under some coordinate, such as (x, y, z).

Pose: Position and orientation of an object, such as (x, y, z) + Euler angles.

Pose of a frame: The pose of the camera when the camera shoots this frame.

World frame: It is coincident with the pose of the 1st frame in video.

1.5 Structure of this report

In Chapter 2, I will introduce some basic algorithms in computer vision and multi-view geometry that are used for my project.

In Chapter 3, detailed workflow of the monocular visual odometry is introduced.

The result is discussed in Chapter 4. The software structure, conclusion, and references are described in Chapter 5, 6, and 7.

2. Basic algorithms

In this chapter, I will describe the basic algorithms of computer vision and multi-view geometry that are used for my monocular VO project, as well as the settings of main parameters. After looking through these basic algorithms, it would be more fluent to read Chapter 3, which is the workflow of monocular VO implemented in this project.

The algorithms and settings are mainly referenced from the Slambook[3] and the ORB-SLAM[4] and ORB-SLAM2[5] papers.

2.1 Feature extraction

The ORB detector and descriptor are adopted for detecting keypoints and extracting descriptors. ORB is based on FAST detector and BRIEF descriptor. It has a fast computation speed (about 10x faster than SURF) and a good performance for detecting keypoints and extracting salient local features.

Here I used a grid sampling strategy to uniformly extract at most 1500 keypoints from each video frame.

2.2 Keypoints matching

Two methods are tested for matching keypoints.

The first method is from Prof. Lowe's SIFT paper[6] in 2004. Suppose a point p_{Ai} in image A is most similar to p_{Bj} in image B and second similar to p_{Bk} in image B.

[3] 14 Lectures on Visual SLAM: From Theory to Practice
[4] ORB-SLAM: a versatile and accurate monocular SLAM system
[5] ORB-SLAM2: An open-source slam system for monocular, stereo, and rgb-d cameras
[6] Distinctive Image Features from Scale-Invariant Keypoints.

If $\frac{dist(p_{Ai},p_{Bj})}{dist(p_{Ai},p_{Bk})} <$ threshold, then p_{Ai} is matched with p_{Bj}. The threshold is usually set between 0.6 to 0.8. Higher threshold gives more matches as well as a higher ratio of wrong matches.

The second method is from the Slambook. First find the minimal distance d_{min} among all pairs of points. With the same notation as in last paragraph, the point p_{Ai} is matched with p_{Bj} if $dist(p_{Ai}, p_{Bj}) < \max(30, 2 \times d_{\min})$, where 30 and 2 are two hyperparameters.

After experiments, the second method from Slambook is a little bit better when comparing the number of matched points and correctly matched points to method one. The result is shown in **Fig. 2.1**.

Fig. 2.2. Keypoints matching result between frame 1 and frame 15 of the video by using the criteria in Slambook.

2.3 Epipolar geometry

Main knowledge of Epipolar geometry includes Epipolar constraint, Essential matrix (E), Homography matrix (H). This part references the lecture notes taught in the class and the Chapter 7 of Slambook.

2.4 Decomposition of E and H

After estimating E and H from the matched keypoints of two frames, we need to decompose them to get the relative camera pose between two frames. By satisfying the criteria that matched keypoints should be in front of the camera, the decomposition returns the following results:

(1) For **Essential matrix**, it returns only one 1 result.

(2) For **Homography matrix**, it returns 2 results, and both are theoratically correct. Here, I simply chose the result whose corresponding keypoints plane (we know that Homography assumes all points to be on a same plane) is more parallel to the camera's image plane. More advanced techniques should be implemented in future work to make the system more robust and generalized.

2.5 Symmetric Transfer Error of H

Suppose we have computed Homography matrix H_{21} from the matched points of two frames I_A and I_B. Now we want to compute the symmetric transfer error, which is basically projecting point from I_A to I_B and projecting point from I_B to I_A, and compute the sum of errors. The purpose of this step is: If Homography's error is smaller than Essential matrix, we will choose the decomposition result of H, and vice versa.

Let's first compute the error for one pair of points, which is p_1 from I_A, and p_2 from I_B. Their positions represented in the homogeneous coordinate of the image are:

$$p_1 = (u_1, v_1, 1)$$

$$p_2 = (u_2, v_2, 2)$$

After projecting p_1 onto I_B and projecting p_2 onto I_A, we get the error:

$$error_H = dist(p_2, H_{21}p_1) + dist(p_1, H_{21}^{-1}p_2)$$

The total error is the sum of errors over all inlier matched points.

2.6 Symmetric Transfer Error of E

Suppose we have computed Essential matrix E_{21} from the matched points of two frames I_A and I_B. The calculation of symmetric transfer error is similar to H.

We first compute the fundamental matrix, where K is the camera intrinsics:

$$F_{21} = K^{-T} E_{21} K^{-1}$$

Then, the Epipolar constraint is:

$$p_2^T F_{21} p_1 = 0$$

(1) By Projecting p_1 onto I_B, we get the Epipolar line $L_2: a_2 x + b_2 y + c_2 = 0$:

$$[a_2, b_2, c_2]^T = F_{21} p_1$$

The corresponding error is the distance between p_2 and L_2:

$$error_2 = (a_2 u_2 + b_2 v_2 + c_2)^2 / (a_2^2 + b_2^2)$$

(2) By Projecting p_2 onto I_A, we get the Epipolar line $L_1: a_1 x + b_1 y + c_1 = 0$:

$$[a_1, b_1, c_1]^T = F_{21}^T p_2$$

The corresponding error is the distance between p_1 and L_1:

$$error_1 = (a_1 u_1 + b_1 v_1 + c_1)^2 / (a_1^2 + b_1^2)$$

The total error is the sum of $error_1 + error_2$, and then sum over all the inlier matched points.

2.7 Others

Other important algorithms include:

- RANSAC: RANdom SAmple Consensus. It's used for estimating a model's parameters from samples with many outliers, such as wrong matches here.

- PnP: Perspective-n-Point. The input of the algorithms is a set of map points with known 3D positions, and their 2D projected positions on a camera video. The output is the pose of this camera frame.
- Triangulation: Given two known camera poses and a pair of matched points in both images, the triangulation estimates the 3D position of this point.
- Lie algebra: It's used for representing the camera pose by a vector of 6 numbers, which is easier for taking derivative and doing optimization of camera poses without considering any constraint. (There would be 10 constraints if representing the camera pose using a 4x4 matrix.)

3. Workflow of Monocular Visual Odometry

3.1 Overview of the workflow

The monocular visual odometry that I implemented has 4 components: (1) Initialization, (2) tracking, (3) local map, (4) and optimization. A summary of the four components is shown in **Fig. 3.1**.

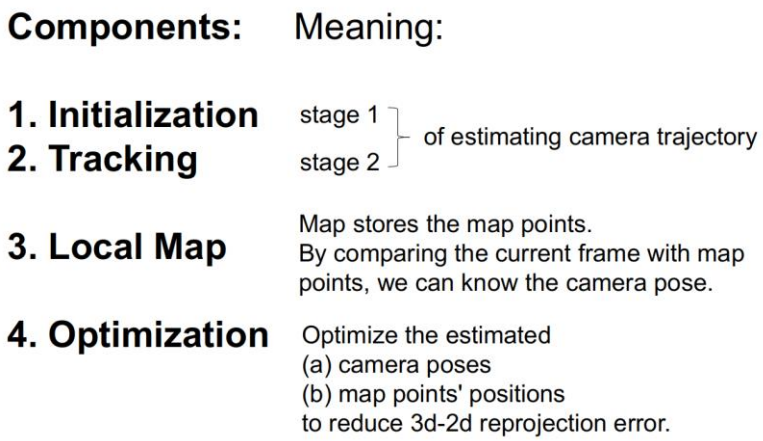


Fig. 3.1. Four components of the monocular visual odometry.

Initialization and **tracking** are the two stages of estimating camera trajectory. The initialization stage first estimates the camera pose of a frame and settles down the depth scale, and then the tracking stage keeps on estimating the camera poses for the rest of the video frames.

Local map stores the map points that are near the camera. By comparing the map points with a video frame, we can deduce the pose of this frame. This process is similar to the scenario that a person looks at the map and knows where s/he is.

Optimization is the process of improving the accuracy of the estimated camera poses and map points' positions, by reducing the 3D-2D reprojection error of each pair of matched keypoints.

Details about these 4 components are illustrated in the rest of this chapter.

3.2 Initialization

3.2.1 Overview

The first component of monocular VO is Initialization.

Given a video, let's set frame 1 of the video to be at the original of the world coordinate. In the stage of initialization, we want to estimate the pose of frame 2. However, if it fails to compute the pose of frame 2, we'll try the frame 3, 4, etc., until the initialization is completed at frame K.

An overview of what initialization does is shown in **Fig. 3.2**. The input of the system is the video frame 1 to frame K. The output is the pose of frame K, as well as the map points triangulated from frame 1 and frame K.

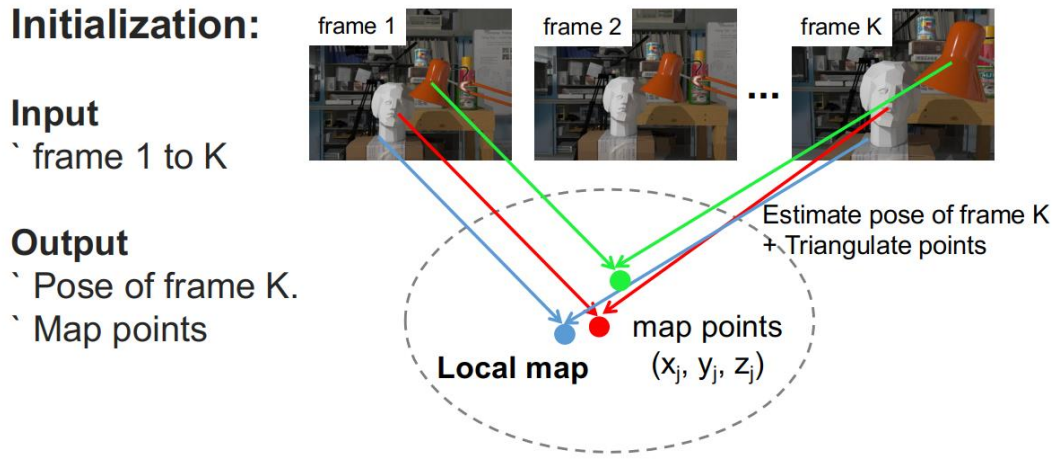


Fig. 3.2. Overview of the initialization stage.

3.2.2 Compute pose of frame 2

We've set frame 1 as the world frame, and now we want to estimate the relative pose between frame 1 and frame 2. The steps are as follows:

Step 1: Uniformly extract ORB keypoints of the two frames, and find the matched keypoints.

Step 2: Compute the Essential (E_{21}) and Homography (H_{21}) Matrix between the two frames by using the matched keypoints and RANSAC algorithm.

Step 3: Compute Symmetric Transfer Error of both E and H. Choose E or H who has a smaller error[7].

Step 4: Decompose E or H to get the relative pose between frame 2 and frame 1, which is the Rotation (R_{21}) and Translation (t_{21}).

Details of some the basic algorithms were introduced in Chapter 2.

3.2.3 Triangulate points and repeat until frame K

By estimating the Essential matrix or the Homography matrix, we can remove some incorrectly matched points, and get the inlier matched points. Then we do the following triangulation steps:

Step 5: Do triangulation for the inlier points, and obtain their 3d positions.

Step 6: Remove the points whose triangulation angle is smaller than 1°. If the number of remaining points are less than 40, or the median triangulation angle of the rest points is smaller than 2°, I will discard the frame 2, because the relative motion is too small and the triangulation results are inaccurate.

Repeat all the above steps on frame 3, frame 4, etc., until the triangulation is good at some frame. Let's say it's finally good at frame K.

Step 7: Push the points triangulated from frame 1 and frame K to the local map.

[7] To be more specific, I use the error to compute a score proposed in ORB-SLAM, and choose E or H based on this score. If score(H)/[score(H)+score(E)]>threshold, I will choose E for decomposition.

3.2.4 Scale the depth

We know that the depth of the scene cannot be recovered from a monocular camera without extra knowledge. Here, the obtained motion $\mathbf{R}_{21}$ and $\mathbf{t}_{21}$ are also subject to a scale factor that we don't know. In real-world implementations, this scale is estimated using extra sensors, such as laser, inertia sensor, or encoder of the car's wheel. Here, I simply set the Translation $\mathbf{t}_{21}$ to have the same length as the ground truth, so that I can make comparison between my estimated trajectory and the ground truth trajectory.

Besides scaling the relative pose, the map points' positions are also scaled correspondingly. Until now, we finally get the pose of frame K, as well as an initial local map with map points.

3.2.5 OpenCV functions

The main OpenCV library functions used in the iniliazation stage are listed as follows:

- cv::ORB
- cv::FlannBasedMatcher
- cv::findEssentialMat
- cv::findHomography
- cv::recoverPose // decompose E to get relative camera pose
- cv::decomposeHomographyMat // decompose H
- cv::filterHomographyDecompByVisibleRefpoints

3.3 Tracking

In the previous initialization stage, we've computed the camera poses of frame K. In the tracking stage, we will keep on estimating the camera pose of frame K+1, K+2, etc.

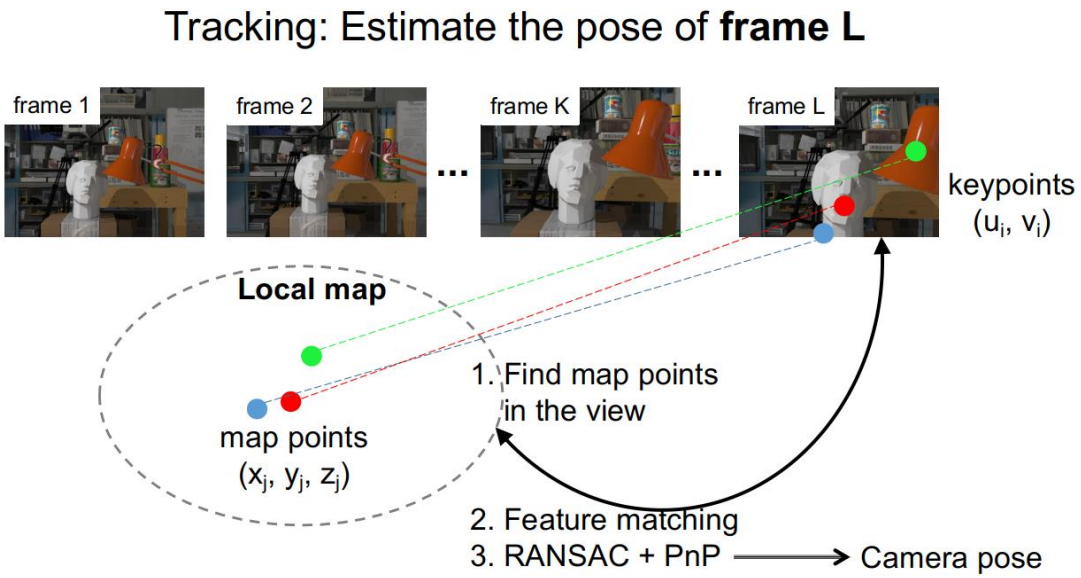


Fig. 3.3. Workflow of tracking stage: estimate camera poses after frame K (after the initialization stage) by using RANSAC and PnP algorithms.

Suppose the current frame is L. The three steps in tracking are shown in **Fig. 3.3**, and illustrated as follows:

Step 1: Find the map points that can be seen in the previous frame, i.e. frame L-1.

In the current frame L, we only want to do keypoints matching with the map points we could see. However, we don't know the pose of frame L. How to tackle with this issue? Since frame L-1 and frame L are near each other, we could use the pose of frame L-1 for an approximation, and then find the map points in the view.

Step 2: Do keypoints matching between these map points and the points in the current frame L. This step is also called "finding 3D-2D correspondence", where 3D

indicates the map points with (x, y, z) position, and 2D indicates the keypoints in image coordinate with (u, v) position values.

Step 3: Estimate camera pose from the matched points by using RANSAC and PnP (Perspective-n-Points) algorithm. I used the OpenCV function "cv::solvePnPRansac", which uses the Direct Linear Transformation method to compute camera pose from 6 pairs of matched points sampled by RANSAC.

3.4 Local Map

Local map stores the map points that are near the current frame.

Just a recall, in the initialization stage, we've computed the pose of frame K and pushed some points to the map to generate the initial local map. Also, frame 1 and frame K are stored as the keyframes.

After tracking stage, since we know the pose of the current frame, we can try adding more points to the map if the current frame is considered as a keyframe. The procedures are as follows:

Step 1: Is current frame a keyframe? If the relative pose between current frame and previous keyframe is large enough, where the relative translation or rotation is larger than certain threshold, the current frame is marked as a keyframe and used for triangulating more map points.

Step 2: If the current frame is a keyframe, do keypoints matching with the previous keyframe. Get inliers by Epipolar constraint. If an inlier keypoint hasn't been triangulated before, then triangulate it to obtain its 3D pos, and push it to the local map.

Step 3: Clean up local map: Remove map points that are (1) not in current view, (2) with a view angle larger than the threshold, (3) rarely be matched as inlier point. This clean-cup step makes the map "local" to the current camera pose.

Step 4: Meanwhile, a graph is built. Every inlier point is connected to its matched point in the (1) local map or/and (2) previous keyframe. The inlier points come from the PnP algorithm or/and triangulation algorithms.

3.5 Optimization

This is the final step of visual odometry. In this part, I implemented the Bundle Adjustment to optimize the camera poses of the last N frames and the current map points. The purpose is to improve the accuracy of the whole visual odometry system. The workflow is shown in **Fig. 3.4**.

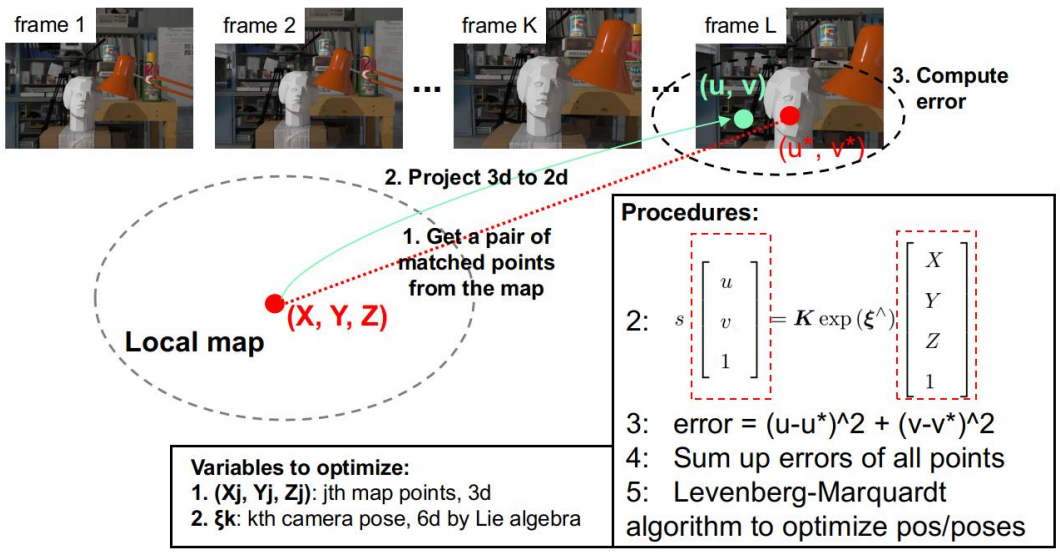


Fig. 3.4. Workflow of the optimization stage, where the accuracy of the estimated camera poses and map points are improved.

The procedures are as follows:

Step 1: Find the 3D-2D corresponding point pairs in the previous N frames by utilizing the graph we've just built in Chapter 3.4.

Step 2: Project the 3D point onto 2D, by using camera extrinsics and intrinsics.

Step 3: Compute distance (error) between the observation (observed 2D point pos) and the estimation result (2D pos projected from 3D).

Step 4: Sum up errors over all pairs of points.

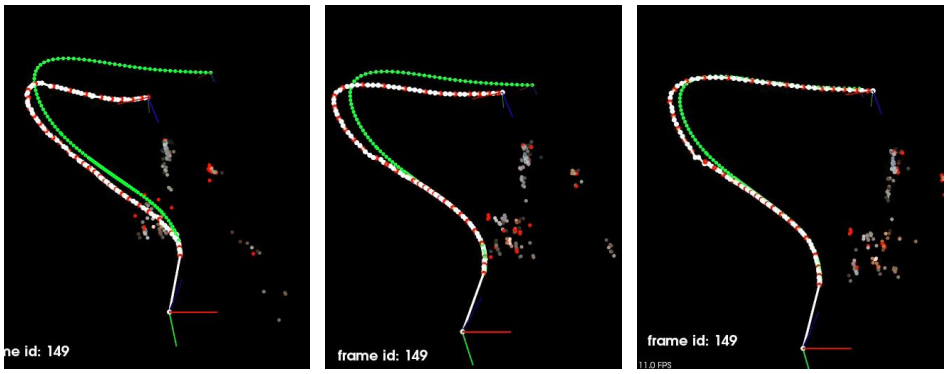
Step 5: Take derivative and do optimization by using the Levenberg-Marquardt algorithm, which is a variant of Gaussian-Newton method. Thanks to g2o library, this optimization problem can be easily set up by using g2o's built-in datatypes of *VertexSBAPointXYZ*, *VertexSE3Expmap*, and *EdgeProjectXYZ2UV*. The library of "g2o", i.e. "graph to optimization", is a general framework for graph optimization.

4. Result

A sequence of snapshots of the estimated camera trajectory has been shown in Chapter 1.3.

Here I made another study of comparing the effectiveness of the optimization under different settings. The result is shown in **Fig. 4.1**. In (a), the optimization is not used. In (b), the optimization is done on the current camera pose as well as the map points. In (c), the optimization is done on the camera poses of the last five frames.

The result shows that the optimization improves the accuracy of the estimated trajectory, as seen from figure (b) and (c) that the green line and white line are closer to each other than figure (a).



(a) (b) (c)

Fig. 4.1. Estimated camera trajectory with different settings of optimization. Green line is the ground truth trajectory, and the white is the result of visual iodometry. Three different settings are: (a) No optimization (b) Optimize on one camera poses and map points. (c) Optimize on five camera poses.

5. Software

5.1 Language and libraries

This project is implemented using C++, and is mainly based on following libraries:

- OpenCV: for image processing.
- g2o: for optimization.
- PCL: for visualization.
- Eigen/Sophus: for data type conversion between OpenCV and g2o.

A rough introduction of how to install these libraries are put on my Github.

5.2 Software structure

The C++ programs are stored in following structure in the include/ folder:

```
include
├── my_basics
│   ├── basics.h
│   ├── config.h
│   ├── eigen_funcs.h
│   ├── io.h
│   ├── opencv_funcs.h
│   └── README.md
├── my_display
│   ├── pcl_display.h
│   └── pcl_display_lib.h
├── my_geometry
│   ├── camera.h
│   ├── common_include.h
│   ├── epipolar_geometry.h
│   ├── feature_match.h
│   └── motion_estimation.h
├── my_optimization
│   └── g2o_ba.h
└── my_slam
    ├── common_include.h
    ├── commons.h
    ├── frame.h
    ├── map.h
    ├── mappoint.h
    ├── README.md
    └── vo.h
```

The main program is src/run_vo.cpp. After compilation, run it by the following command: *$ bin/run_vo config/config.yaml*

The parameters of algorithms are configured in config/config.yaml.

6. Conclusion

In this project, I implemented a monocular visual odometry system in C++. The performance of the system was tested on a public video. The result of the estimated trajectory is close to the ground truth, indicating that this visual odometry system works.

7. Reference

I referenced the course lecture notes as well as the following documents for completing my project:

1. Xiang Gao, Tao Zhang, Yi Liu, Qinrui Yan (2017). 14 Lectures on Visual SLAM: From Theory to Practice. Publishing House of Electronics Industry.
2. Mur-Artal, R., Montiel, J. M. M., & Tardos, J. D. (2015). ORB-SLAM: a versatile and accurate monocular SLAM system. IEEE transactions on robotics, 31(5), 1147-1163.
3. Mur-Artal, R., & Tardós, J. D. (2017). Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. IEEE Transactions on Robotics, 33(5), 1255-1262.
4. New Tsukuba Stereo Database for the testing video.

Though I wrote the majority of the code, I also borrowed some codes from the above references, which are:

1. From "14 Lectures on Visual SLAM": (1) Software structure of visual odometry. (2) Some example usage of each library's functions.
2. From "ORB-SLAM2": The programs for computing Symmetric Transfer Error and the criteria for choosing either Essential or Homography matrix (for calculating relative camera pose).