Vision-Based Navigation
V. Usenko, N. Demmel, D. Schubert, D. Cremers
Summer Semester 2020

Computer Vision Group
Department of Informatics
Technical University of Munich

Exercise Sheet 5

Topic: Real-Time Visual Odometry

Submission deadline: Sunday, 07.06.2020 23:59

Hand-in via merge request

General Notice

The exercises should be done by yourself. We use Ubuntu 18.04 in this lab course. Other Linux distributions and macOS *may* also work, but might require some more manual tweaking.

Real-Time Visual Odometry

In this exercise we will implement a real-time capable visual odometry. It is very similar to SfM from the previous exercise but with some optimizations to allow real-time execution for a sequential list of images. Firstly, we only select a subset of frames (keyframes) to run the optimization, and for all other frames we just use the current map to localize them. Secondly, we use a projection-based search to match the descriptors between the map and the current frame.

Exercise 1: Skeleton Code

Download the odometry dataset by executing the following commands:

```
cd data
./download_dataset.sh
```

It might take some time to download the dataset.

We provide the skeleton code in `src/odometry.cpp`. Inspect the `next_step` function and describe in the PDF file a workflow for the implemented odometry method. After compiling the `odometry` application, you can then run it with

```
./build/odometry --dataset-path data/V1_01_easy/mav0/
```

Exercise 2: Setting Up the Problem

Implement the following functions in the `include/visnav/vo_utils.h`:

- In function `project_landmarks` implement the projection of landmarks to the image plane using the current camera location. Ignore all points that are behind the camera (Z coordinate in the camera coordinate frame is less than

cam_z_threshold) or project outside of the image. For the projections save their 2D locations and IDs of the landmarks that they originate from.

- In function find_matches_landmarks implement finding the matches between projected landmarks and detected keypoints in the current frame. For every detected keypoint search for matches inside a circle with radius match_max_dist_2d around the point location. For every landmark the distance is the minimal distance between the descriptor of the current point and descriptors of all observations of the checked landmark. Use the absolute descriptor distance threshold and the distance to the second best match similar to Exercise 3 to filter the outliers. *Hint:* You should use the checks to find the best match to a *landmark*, where the *landmark distance* in turn is the best match between the current descriptor of the point and all observations of the landmark.

- In function localize_camera find the camera pose of the provided camera and the list of inliers using the landmark to keypoints matches and PnP. This should be similar to the localize_camera in Exercise 4 but here we do not explicitly model feature tracks.

- In function add_new_landmarks implement adding of new landmarks and observations. For all inlier landmark matches add observations for the left camera to the existing landmarks. If the keypoint appears also in md_stereo.inliers, add an observation for the right camera as well. For inlier stereo matches that were not used in the previous stage, triangulate and add new landmarks from stereo. Here next_landmark_id is a running index of the landmarks, so whenever you have added a new landmark you should increase next_landmark_id by 1.

- In function remove_old_keyframes remove old cameras and corresponding observations if the number of keyframe pairs (left and right image is a pair) is larger than max_num_kfs. The IDs of all the keyframes that are currently in the optimization should be stored in kf_frames. Removed keyframes should be removed from cameras and landmarks with no observations left should be moved to old_landmarks. *Hint:* The frame IDs can be seen as timestamps for the images, i.e. they induce a temporal order on keyframes.

If everything is implemented correctly you should see the visualization of the visual odometry as in Figure 1.

Exercise 3: Optimization

The optimization is done in the optimize function in src/odometry.cpp. Internally it relies on bundle_adjustment function that you implemented in Exercise 4. What is the difference to the optimize function in src/sfm.cpp from the previous exercise? What is the functionality of the variables opt_finished and opt_running? What will happen if we remove them? Please write the answers in the PDF file.

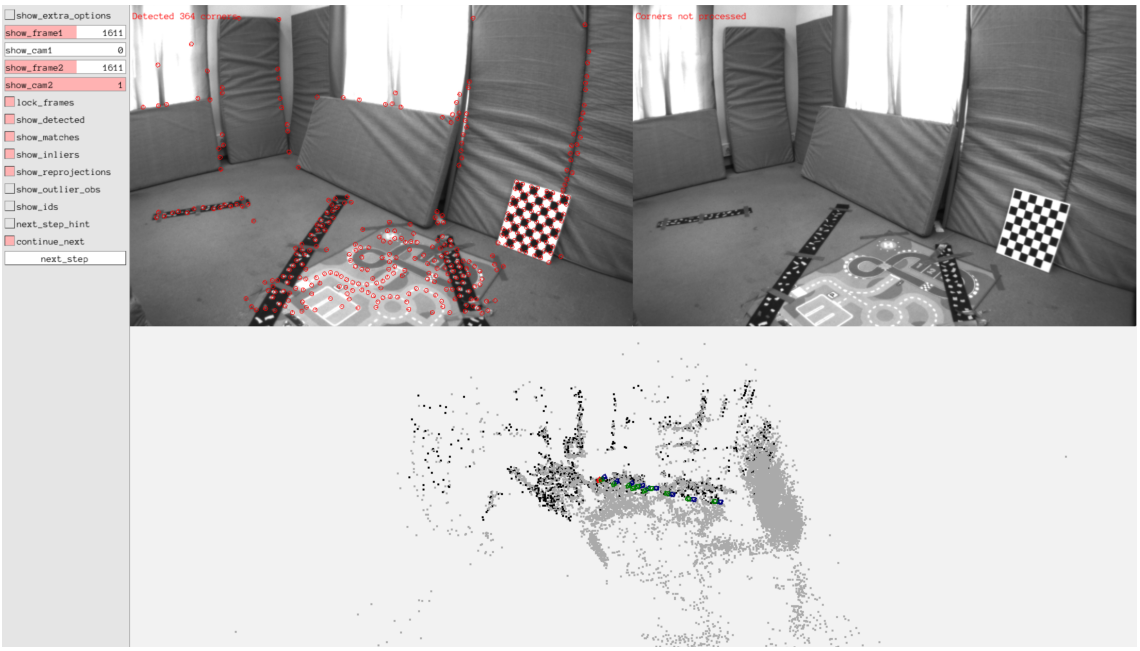

Figure 1: Visualization of the visual odometry method implemented in this exercise. Black landmarks are active. Gray landmarks are out of optimization window and are not optimized anymore.

Unit Tests

Before submitting the exercise uncomment the following in `test/CMakeLists.txt`:

```
gtest_discover_tests(test_ex5 ...
```

If everything is implemented correctly the system should pass all tests.

Submission Instructions

A complete submission consists both of a PDF file with the solutions/answers to the questions on the exercise sheet and a merge request against the `master` branch with the source code that you used to solve the given problems. Please note your name in the PDF file and submit it as part of the merge request by placing it in the `submission` folder.