Vision-Based Navigation
V. Usenko, N. Demmel, D. Cremers
Winter Semester 2019/2020

Computer Vision Group
Department of Informatics
Technical University of Munich

Exercise Sheet 4

Topic: SfM, Triangulation, PnP, Bundle Adjustment

Submission deadline: Sunday, 01.12.2019, 23:59

Hand-in via merge request

General Notice

The exercises should be done by yourself. We will use Ubuntu 18.04 in this lab course. It is already installed on the lab computers. If you want to use your own laptop, you will need to install Ubuntu yourself. Ubuntu 16.04 and macOS *may* also work, but might require some more manual tweaking.

Structure from Motion Pipeline

In the previous week we have extracted features and computed pairwise matches. This week we will use these matches to create a map of keyframes and 3D landmarks in a Structure from Motion (SfM) pipeline. Unlike for a SLAM system, we assume that all images are available right away and the focus is not on real-time performance. Our SfM pipeline has the following steps, some of which you will implement in this exercise. For convenience during development, each step can be triggered manually in the `sfm` executable with the corresponding button.

1. **Feature detection and matching:** This is what we did in exercise sheet 3. After feature matching we have inlier matches between image pairs. Note that the computed features and matches are cached in the files `corners.cereal` and `matches.cereal`, which means that after restarting the application you don't have to recompute them.
2. **Building feature tracks:** Here we combine the pairwise matches to longer feature tracks by computing the transitive closure over all inlier matches. We discard wrong matches (feature track contains more than one observation in the same camera) and also require a minimum length (3). Each feature track consists of a list of the observing images and the corresponding feature indices. Feature tracks are the basis for landmarks. We provide a implementation for you.
3. **Initializing the map:** To initialize the map we take a simple approach and select a stereo pair as the first two cameras. We use the known relative pose from the extrinsic calibration and triangulate all shared feature tracks as the initial landmarks. You will implement this in Exercise 1.

4. **Selecting new camera candidates:** For each of the remaining images we check how many feature tracks it shares with the already mapped landmarks. In each round round we may add a certain maximum number of images (15) to the map, if they observe a sufficient number of landmarks (40). If no more images can be added that way, then we allow to add a smaller number of images (2) with a smaller (minimal) number of landmark observations (10). If that also fails, the mapping process is complete, as no more images can be added.
5. **Adding new cameras:** For each of the candidate cameras we first try to localize it using the 3D-2D correspondences from the feature tracks shared with existing landmarks. For this we employ the perspective-n-point (PnP) algorithm in a RANSAC scheme. If the number of inliers from RANSAC is sufficient, we add the new camera to the map with the initial pose refine based on all inliers. You will implement the PnP RANSAC with final pose refinement in Exercise 2.
6. **Adding new landmarks:** For each of the new cameras, we check if there are any feature tracks that are not yet part of the map but are observed by at least 2 cameras. Each new landmark is triangulated based on a pair of images, using the estimated camera poses in the map. This uses the same triangulation that is also needed for map initialization, and which you will implement as part of Exercise 1.
7. **Optimizing the map:** After each round of adding cameras and landmarks (and also after initialization), we perform map optimization to refine both the camera poses as well as the landmark positions. This is known as Bundle Adjustment (BA) and you will implement this in Exercise 3.
8. **Removing outliers:** Since there may be some outlier observations left, or some of the landmark positions may have converged to a local minima, it is important to remove outliers from the map after bundle adjustment and then optimize again. You don't have to implement this, but you will investigate the implemented criteria in Exercise 4.

As mentioned, you can use the GUI to perform this individual steps of the pipeline manually. There is also a `next_step` button that automatically chooses the appropriate next step. If you enable the `next_step_hint` option, the next step in the pipeline that should be performed is printed on the command line. By enabling `continue_next` you can run the steps as fast as possible one after another. Note that there is no multi-threading and the GUI might freeze up while a step is running (for example bundle adjustment with a larger map may take several seconds). You can also save and load your progress to and from the `map.cereal` file. `show_extra_options` reveals a set of additional parameters you may want to experiment with. Please consult `src/sfm.cpp` for details on the pipeline implementation, the colors in the GUI, as well as the exposed parameters. Finally, you may find the `--max-frames` command line option useful to speed up testing during development.

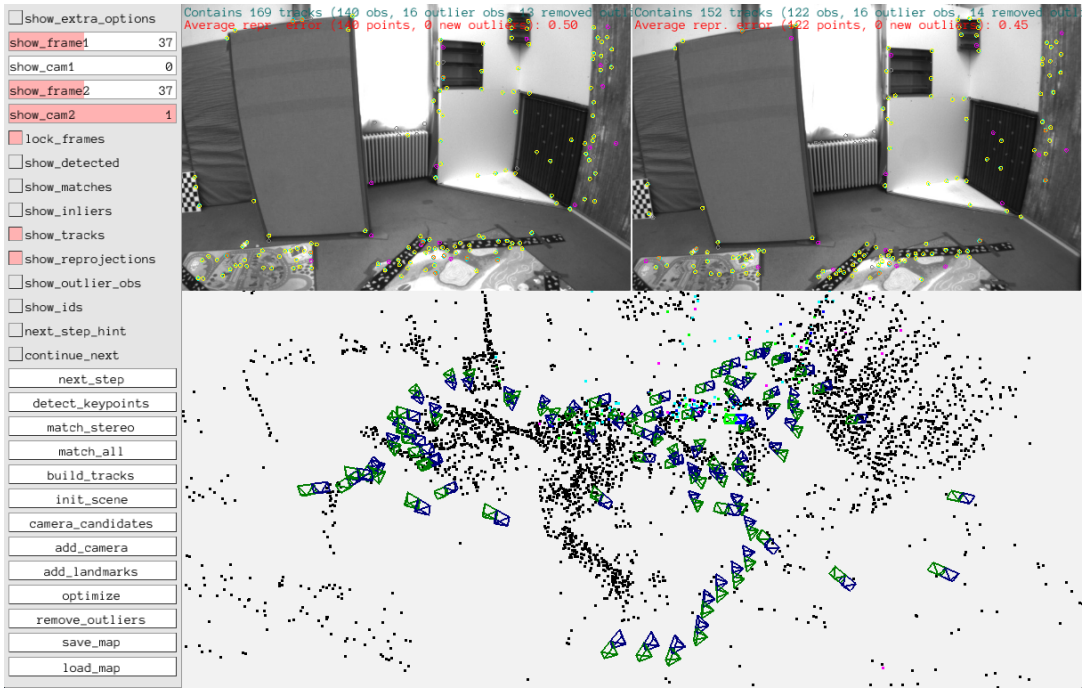

Figure 1: View of a map built with this SfM pipeline. After completing all the exercises on this sheet you should be able to create a similar map.

Note: Except for the map initialization, in this exercise we do not explicitly model the stereo camera, but instead we add the left and right images independently to the map. It would also be possible to add images in stereo pairs and make use of the extrinsic calibration to only estimate a single pose in the map for each stereo pair, which is something you could experiment with yourself.

You should add the implementation in the marked sections in `include/visnav/map_utils.h` and `include/visnav/reprojection.h`.

Exercise 1: Map Initialization and Landmark Triangulation

(i) **Initialize Cameras:** To create the initial two cameras in the map, we take a stereo pair, and set the left camera (id 0) to the identity pose and the right camera to the relative pose according to the extrinisic camera calibration. Add this implementation to `initialize_scene_from_stereo_pair()` by appropriately filling the `cameras` variable. We will in the next step also add the initialization of the first set of landmarks in that function.

Note: If we didn't have stereo calibration, we could also initialize the first two cameras from the relative pose of the 5-point algorithm from feature matching. However, using the stereo calibration has two advantages. Firstly, it provides a metric initialization, i.e. our map will have a true metric scale, whereas the pose from the 5-point algorithm may be arbitrarily scaled. Secondly, the initial pose from stereo calibration is very precise, allowing us to fix the scale gauge

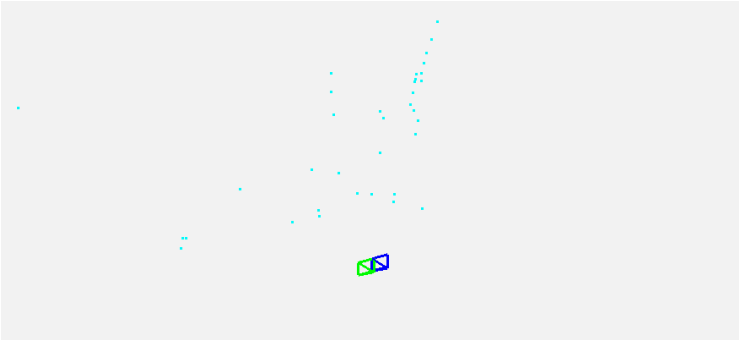

Figure 2: Map initialization: Initial couple of cameras and triangulated landmarks.

during bundle adjustment by fixing the two camera's poses (more on that in Exercise 3).

(ii) **Triangulate Landmarks:** To complete the map initialization and also later after adding new cameras, we need to add new landmarks to the map. We do this by triangulating feature tracks shared between image pairs that have already been added to the map (and thus have estimated camera poses). Complete the implementation in `add_new_landmarks_between_cams()`, where for a given pair of cameras in the map we first compute the set of shared feature tracks, and then add all tracks that are not yet part of the map. For triangulation you can use OpenGV's `CentralRelativeAdapter` (which should be familiar from the 5-point algorithm) and `triangulate()` function. Newly created landmarks should be inserted in the `landmarks` data structure with the same key as the corresponding feature track, using the triangulated position (transformed into world frame). Make sure to fill the set of observations `obs` with all observations of existing cameras in the map, not just for the two cameras used for triangulation. After implementing landmark triangulation, complete `initialize_scene_from_stereo_pair()` with an appropriate call to `add_new_landmarks_between_cams()`. The result should look similar to Figure 2 (you may have a few outlier landmarks, which is fine at this stage).

Exercise 2: Localizing cameras with PnP

After initializing the map and computing the first set of camera candidates, we need to localize them in the map using the known 3D-2D correspondences from the feature tracks. We use PnP in a RANSAC scheme to be robust against remaining outlier matches. It is important to do a refinement of the camera pose using all inliers from RANSAC, since this initial camera pose is used in subsequent steps for landmark triangulation. The result should look like Figure 3.

Complete the implementation `localize_camera`, where you are given an image to localize as well as a list of feature tracks this image shares with the given set of landmarks. You need to output the camera pose as well as a list of inlier track ids.

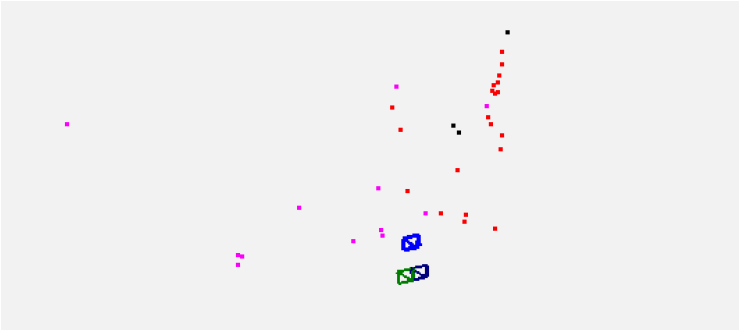

Figure 3: Camera localization: Localizing the first camera after map initialization.

You may use OpenGV's `CentralAbsoluteAdapter` together with the corresponding RANSAC implementation `AbsolutePoseSacProblem` that uses a minimal variant of PnP taking exactly 3 points.

Note that the inlier threshold for RANSAC is given in pixels. Please refer to `http://laurentkneip.github.io/opengv/page_how_to_use.html#sec_threshold` on how to convert this to the appropriate threshold for bearing vectors. You may make the approximation of a constant focal length to compute the threshold, for example 500.

Exercise 3: Bundle Adjustment

After adding new cameras and landmarks, we need to periodically refine the camera poses and landmark positions in order to avoid build-up of errors. We do this with Bundle Adjustment (BA), which minimizes the reprojection error. This optimization is quite similar to the optimization we used in sheet 2 for camera calibration. The differences include: a) we now also optimize the 3D positions, b) we now don't explicitly model the stereo camera such that every image gets its own pose, and c) we typically keep the intrinsics fixed.

Compute the reprojection error in `BundleAdjustmentReprojectionCostFunctor` and setup an appropriate optimization problem in `bundle_adjustment()` to implement BA. You should optimize over all camera poses and landmark positions, using all observations stored in the landmarks (without `outlier_obs`). The parameter blocks corresponding to `fixed_cameras` should be kept constant to constrain the gauge freedom (global pose and scale).

When `use_huber` is set to `true` in the options, you should use `ceres::HuberLoss` with the provided parameter. Discuss in your answer PDF what a robust loss function such as Huber does and why we should use it here, but not in the calibration from sheet 2.

Exercise 4: Outlier Filtering

After optimizing the map with bundle adjustment, we need to remove outliers which might originate from wrong matches that weren't yet filtered, or from wrongly tri-

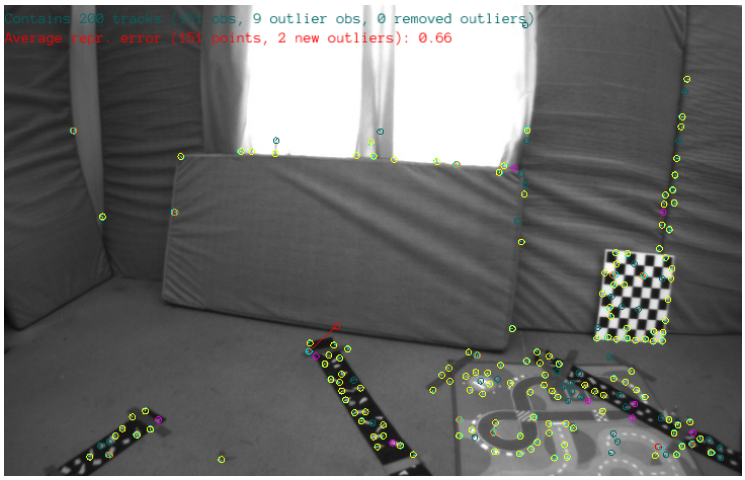

Figure 4: Outlier observation with large reprojection error after optimization indicated by large red line.

angulated landmarks that converged to some local minimum (see Figure 4). Have a look at `remove_outlier_landmarks` and describe in your answer PDF the different implemented criteria to detect outliers. For each criterion, what do you think might be the cause of such an outlier and why do we need to remove it?

Exercise 5: Building a Map

After implementing all steps in Exercises 1 – 3, try to create a map using all images. For the completed map, provide a screenshot similar to Figure 1 in you answer PDF. How many cameras can be added to the map? How long does it take? Which parts of the pipeline are taking the most time? Do you have any suggestions on how to maybe speed up the map building process? Note that you can also automate the SfM pipeline by running the `sfm` executable without GUI and then load the saved map int he GUI if wanted.

Unit Tests

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

```
gtest_discover_tests(test_ex4 DISCOVERY_TIMEOUT 60)
```

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.