Vision-Based Navigation
J. Chui, S. Klenk, D. Cremers
Summer Semester 2023

Computer Vision Group
Department of Informatics
Technical University of Munich

Exercise Sheet 3

Topic: Feature Detectors, Descriptors, Epipolar Geometry, RANSAC

Submission deadline: Sunday, 21.05.2023, 23:59

Hand-in via merge request

General Notice

The exercises should be done by yourself. We use Ubuntu (20.04, 18.04). It should work on other Linux distributions, as well as on macOS, but we do not guarantee full support, some more manual tweaking might be required.

Part 1: ORB Descriptors

In the lecture we explained you how to detect keypoints in images and compute their descriptors. In this exercise you will implement keypoint detection, descriptor computation and matching. We provide you the skeleton code that loads and visualizes the dataset. The code framework is provided in `src/sfm.cpp`, and you can start the application with:

```
./build/sfm --dataset-path data/euroc_V1 --voc-path data/ORBvoc.cereal
```

Make sure a suitable `opt_calib.json` file created in the previous exercise sheet is present in the working directory (or manually pass `--cam-calib`). In the next exercise sheet we will then build on these results to create a 3D map of camera poses and matched feature points.

Note: Computed features and matches are cached in the files `corners.cereal` and `matches.cereal`, which after restarting the application are loaded automatically. To recompute features, you need to manually press `detect_corners`, which also clears matches. The different matching variants are cumulative, i.e. if you press `match_stereo` and then `match_all`, the result from both is shown.

1. Before computing the ORB descriptors [4] we first need to select keypoints and estimate their orientations. In this exercise we already provide you a list of the keypoints detected with the Shi-Tomasi algorithm [5] implemented in OpenCV. Inspect the function `detectKeypoints` in `include/visnav/keypoints.h` for the details. In the same file you should implement the angle computation in the `computeAngles` function. The angle θ of a keypoint can be computed as (see [4] for details):

$$m_{pq} = \sum_{(x,y)\in N} x^p y^q I(x+u, y+v), \tag{1}$$

$$\theta = \text{atan2}(m_{01}, m_{10}), \tag{2}$$

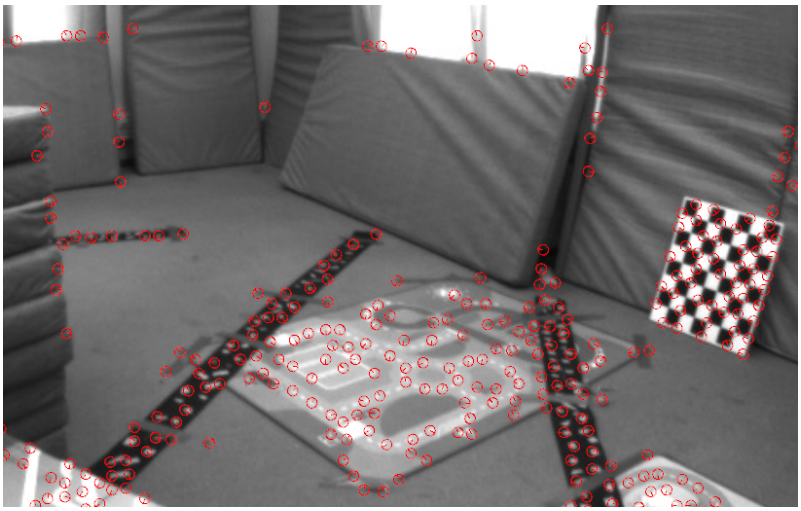

Figure 1: The keypoint orientation is indicated by a line within the circles.

where (u, v) is the location of the detected keypoint and m_{01} and m_{10} are the intensity moments. Here, we choose a circular patch N around a keypoint with radius 15 (`HALF_PATCH_SIZE`). It means that for the keypoint at (u, v), we take all points $(x+u, y+v)$ where $x^2+y^2 <= 15^2$. After implementing the angle computation you should be able to see the orientation of the features as shown in Figure 1.

2. ORB uses a BRIEF descriptor, which is a binary descriptor with 256 bits containing 0 and 1. Each bit is the result of a simple binary test that compares image intensities around the detected keypoint. The algorithm is described in the following:

   - Given an image I, keypoint $\mathbf{c} = (c_x, c_y)^\top$ and its angle θ, we compute a 256-bit descriptor. The descriptor is stored as a `std::bitset`

   $$\mathbf{d} = [d_1, d_2, \ldots, d_{256}], d_i \in \{0, 1\}.$$

   - For each $i = 1, \ldots, 256$, d_i is computed as follows. We take two offsets around $\mathbf{c}$, say, $\mathbf{p}_a = (x_a, y_a)^\top$ and $\mathbf{p}_b = (x_b, y_b)^\top$, and rotate them with the angle θ

   $$\begin{bmatrix} x_a{}' \\ y_a{}' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_a \\ y_a \end{bmatrix}, \tag{3}$$

   such that $\mathbf{p}'_a = (x'_a, y'_a)^\top$ are the rotated coordinates of $\mathbf{p}_a$. The same rotation procedure is applied to $\mathbf{p}_b$ to get $\mathbf{p}'_b$. We compare the image intensity of $I(\mathbf{c}+\mathbf{p}'_a)$ and $I(\mathbf{c}+\mathbf{p}'_b)$. If $I(\mathbf{c}+\mathbf{p}'_a) < I(\mathbf{c}+\mathbf{p}'_b)$, then $d_i = 1$, otherwise $d_i = 0$. To get integer coordinates of the rotated points please use the `round` function.

   In `include/visnav/keypoints.h`, please implement the `computeDescriptors` function. Note that all 256 offset coordinates $\mathbf{p_a}$ and $\mathbf{p_b}$ (ORB pattern) are

given in the same file. If you are interested how the pattern was selected, see [4].

3. Let us next look at brute force matching of ORB features. After computing the descriptors, we need to match them according to the descriptors. Brute force matching is a simple and commonly used approach for feature matching, especially when the number of features is not large. Given two sets of descriptors, say, $\mathbf{P} = [p_1, \ldots, p_M]$ and $\mathbf{Q} = [q_1, \ldots, q_N]$, for each descriptor in $\mathbf{P}$, we find a descriptor in $\mathbf{Q}$ that has the minimum (Hamming) distance. Check the documentation for `std::bitset` to efficiently implement the Hamming distance computation. To filter the wrong matches you should use several checks:

   - Discard matches with distance larger or equal to the `threshold`.
   - Discard matches if the distance to the second best match is smaller than the smallest distance multiplied by `dist_2_best`.
   - Match P to Q and Q to P and consider the match valid only if it agrees between these two matchings.

   In `include/visnav/keypoints.h`, please implement the `matchDescriptors` function according to these instructions. The matching results should be similar to Fig. 2. You should press `show_ids` to visualize the matched indices.

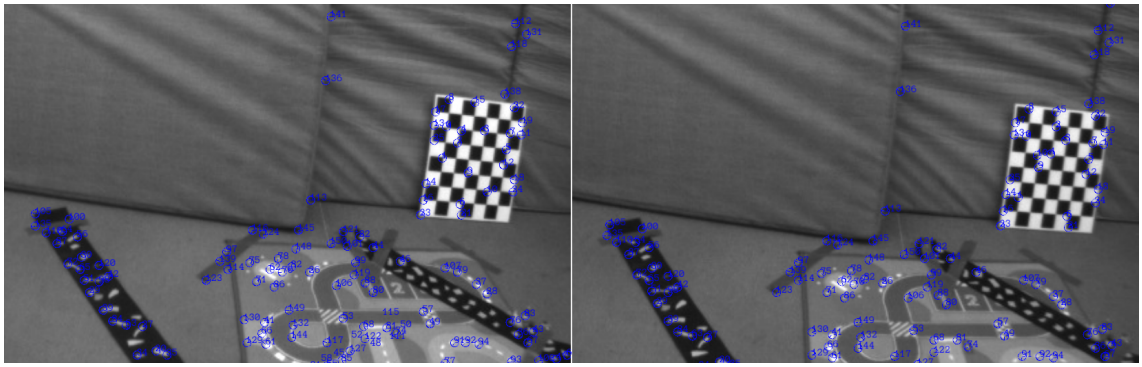

Figure 2: Matched key points.

Part 2: Epipolar constraint

Despite the filtering stage our matches still may contain outliers. In this exercise we will use the epipolar constraint to filter them out when matching images from left and right cameras of the stereo setup.

The idea of epipolar geometry is visualized in Figure 3. If point X is observed by two cameras, this point and two focal points O_L and O_R of the cameras form a plane. The epipolar constraint states that the vectors $O_L X$, $O_R X$, $O_L O_R$ should lie in the same plane. Let $\mathbf{x}_L$ and $\mathbf{x}_L$ be the bearing vectors we obtain by unprojecting the detected 2D image points. $\mathbf{x}_L$ and $\mathbf{x}_L$ correspond to $O_L X$ and $O_R X$, but are given

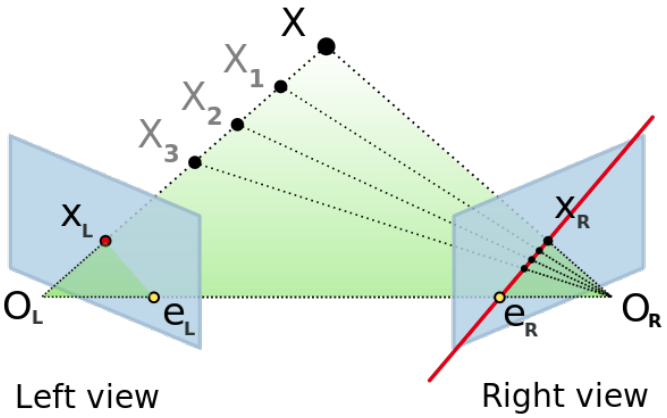

Figure 3: Epipolar constraint.

in the coordinates of the respective camera frame. Moreover, note that $O_L O_R$ and R_{LR} (the rotation between cameras) for the case of stereo camera is known from the calibration.

The epipolar constraint can be formulated using the essential matrix as follows

$$\mathbf{x}_L^\top E \mathbf{x}_R = 0. \tag{4}$$

First derive the computation of the essential matrix from the transformation between two cameras in the PDF file. Then implement the `computeEssential` and `findInliersEssential` functions in `include/visnav/matching_utils.h`. When computing the essential matrix please **normalize the translation vector**. For inliers the constraint should be fulfilled up to a given threshold. After implementing the epipolar constraint you should be able to see (Figure 4) that it helps to reject the wrong matches even when the descriptors are very similar.

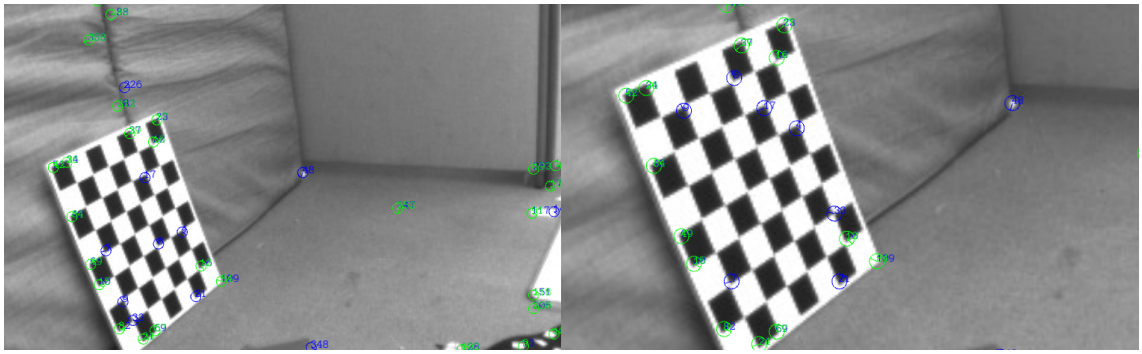

Figure 4: Inliers shown in green. Rejected matches shown in blue.

Part 3: Five-point algorithm and RANSAC

For the previous exercise we assumed that we know the transformation between the cameras. It turns out that it is possible to compute it using 5 matching points

between the images [2] and use the same math to find inliers. Since the algorithm is a bit tedious to implement you should use the OpenGV library. Read the documentation at `https://laurentkneip.github.io/opengv/page_how_to_use.html` and implement the central relative pose problem with RANSAC in the `findInliersRansac` function in `include/visnav/matching_utils.h`.

Hints:

- If RANSAC is successful, you should refine the model parameters (relative pose) using ALL inliers (for example using `opengv::relative_pose::optimize_nonlinear()`). Once you have refined the model parameters, you should update the set of inliners using the refined relative pose. This is analogous to the last step in RANSAC. You can have a look at the implementation of `opengv::sac::Ransac::computeModel()` and use the same method to determine the set of inliers.
- You should store both the final set of inliers, as well as the final (refined) relative pose. Since the scale of translation is not determined, you should normalize the translation vector.
- By default `lock_frames` is active in the GUI such that you always see a stereo pair. To investigate the result of `match_all` (and later `match_bow`) visually, you need to uncheck `lock_frames` and then select two frames with different frame-ids for the left and right image.

Part 4: Bag-of-Words for Place Recognition

In order to establish point correspondences we need to match images. For image pairs that form stereo pairs we can use `findInliersEssential` function for others we can use `findInliersRansac`. One option is to make a bute-force matching that considers all possible pairs as candidates. This approach is implemented in `match_all` function in `src/sfm.cpp`. In the `match_bow` you will implement matching with a place recogition approach that allows us to find candidate pairs using the bag-of-words descriptors. You should proceed as follows:

- Read the papers [3] and [1] to get familiar with the state-of-the-art place recognition methods.
- Look at the `match_all` and `match_bow` functions in `src/sfm.cpp`. What is the main difference between them? What does the `num_bow_candidates` parameter control?
- Inspect the `BowVocabulary` class in `include/visnav/bow_voc.h`. Implement the `transformFeatureToWord` method that propagates a given feature through the vocabulary tree and saves the corresponding word and its weighting. After that, implement the `transform` method that builds a BoW descriptor from an array of features for a certain frame. You should use L1 normalization for the BoW descriptor.

- In the `BowDatabase` class in `include/visnav/bow_db.h` implement an inverted index that would enable fast search of the frames with closest BoW vectors. You should use the L1 difference for comparison. Implement the `insert` and `query` methods.
- After implementing the BoW matching compare the number of candidate pairs and inliers when using the `match_all` and `match_bow` functions (`clear_matches` may be helpful). In our case we have 2×82 images. What would be the number of candidate pairs for 2×1000 images for these two functions? Please include the answers in the submission PDF file.

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

```
gtest_discover_tests(test_ex3 ...
```

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.

References

[1] D. Galvez-Lpez and J. D. Tardos. "Bags of Binary Words for Fast Place Recognition in Image Sequences". In: *IEEE Transactions on Robotics* 28.5 (2012), pp. 1188–1197.

[2] D. Nister. "An efficient solution to the five-point relative pose problem". In: *IEEE transactions on pattern analysis and machine intelligence* 26.6 (2004), pp. 756–770.

[3] D. Nister and H. Stewenius. "Scalable Recognition with a Vocabulary Tree". In: *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*. Vol. 2. 2006, pp. 2161–2168.

[4] E. Rublee et al. "ORB: An efficient alternative to SIFT or SURF". In: *2011 International Conference on Computer Vision*. 2011, pp. 2564–2571. DOI: 10.1109/ICCV.2011.6126544.

[5] Jianbo Shi and Carlo Tomasi. "Good features to track". In: *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR'94., 1994 IEEE Computer Society Conference on*. IEEE. 1994, pp. 593–600.