V-Charge Summer School - Exercise Image-Based Localization

Torsten Sattler
Computer Vision and Geometry Group, ETH Zurich
torsten.sattler@inf.ethz.ch

July 8, 2014

1 Introduction

In this exercise, you are asked to implement a simple image-based localization method based on kd-tree search. In detail, you need to implement (parts of) a RANSAC-based pose estimation process and you will use the FLANN library to construct a kd-tree and search through it. After implementing the pipeline, you should play around with some of its parameters to get familiar with its behavior. If you are very motivated, you can implement additional improvements. The code for this exercise is in C++. If you have any questions about this language, please let us know.

2 Installing the Code

You will first need to install some libraries that come with it.

Let `source_dir` be the source code directory. First, you will need to compile the FLANN library that comes with it:

- Open a console and got to `source_dir/ext/flann-1.8.4-src`.
- Create a build directory by typing `mkdir build`, then go to the directory (`cd build`)
- Invoke cmake to configure the library and the compiler:
  `cmake -DCMAKE_INSTALL_PREFIX=../ -DCMAKE_BUILD_TYPE=Release ..`
- Compile and install the library: `make install`

3 Getting the Data, Compiling, and Running the Exercise Code

Make sure that you also downloaded the data required for the experiments and extract it in some directory `data_dir`.

Next, you will want to compile the project:

- Go to `source_dir`.

- Create a build directory: `mkdir build`, then go to the directory (`cd build`)
- Invoke cmake to configure the library and the compiler:
  `cmake -DCMAKE_INSTALL_PREFIX=../`
  `-DCMAKE_CXX_FLAGS=-std=gnu++11 -DCMAKE_BUILD_TYPE=Release ..`
- Compile and install the project:`make install`
  This will create a directory `source_dir/bin` containing an executable `localizer`
- In order to run the localization method on the data you downloaded, go to the directory containing the binary and type
  `./localizer data_dir/notredame.info data_dir/query/ 0.7`
  The first two parameters specify a binary file containing the 3D reconstruction and the directory containing the query images, respectively. The last parameter specifies the threshold during Lowe's ratio test to filter out wrong matches. Please don't run the executable just now, because it will cause a Segmentation Fault. You will first need to implement some functions :)

4 Implementing Image-Based Localization

In the following, we will walk you through the functions that you will need to implement in order to get image-based localization to work. All parts of the pipeline that you need to implement are marked with `\\ BEGIN(code)` and `\\ END(CODE)` in the code and also contain a small comment about what you will need to do. We strongly recommend you to check the slides from the lecture for details, but if you have any questions, feel free to ask us!

4.1 RANSAC-based Pose Estimation

We provided a pose solver that computes the camera pose from three 2D-3D matches (see `source_dir/include/image_based_localization/pose_solver.h` for details) and some preliminary code for a RANSAC-loop that calls this solver (`source_dir/include/image_based_localization/P3P_RANSAC.h`). Your task is to complete the RANSAC implementation:

Complete the function `ComputeNumRequiredIterations` that returns the maximal number of iterations that RANSAC needs to take to ensure that we have missed the correct pose with probability of at most `failure_probability` (given in the range $[0, 1]$) under the condition that we already have found a pose with `num_inliers` inliers. Please see the lecture slides for the formula that computes the number of iterations. Inside the function `ComputeCameraPose`, you will need to

- Randomly sample three unique integer numbers (three mutually different numbers) from the range [0, `num_matches`). You can use the random number generator already provided to achieve this goal. Next, use these numbers to select three 2D-3D matches from the given set of matches (the i-th match is given by the i-th columns of the matrices `pos2D` and `pos3D`) and store corresponding 2D and 3D points in the specified matrices.

- Evaluate all poses generated by the `p3p_solver` on all matches. The `PoseSolver` class already provides functionality to check whether a match is an inlier or not. If the number of inliers for the pose that you are currently evaluating exceeds the maximum number of inliers that you have found so far, you need to update the best pose you have found so far (stored in the output parameter `pose`). At the same time, you need to update the number of maximal RANSAC iterations by calling the function that you implemented before.

4.2 Implementing kd-tree Search

In order to obtain a fully-functional localization algorithm, you also have to implement the descriptor matching part. To avoid implementing a kd-tree on your own, you will use the FLANN library. If you are not familiar with this library, please check its documentation (`source_dir/ext/flann-1.8.4-src/doc`). In `source_dir/src/kd_tree_localization.cc`, you need to

- In the function `BuildTree`, compute the mean descriptor for each 3D point and store it in a FLANN matrix. The 3D points and their descriptors are stored in the member variable `point_information_`. In order to safe memory, each descriptor entry of each SIFT descriptor is represented by 1 byte and stored as an unsigned char value. After computing the mean descriptor, you should thus round its entries to the nearest integer value and cast it back to unsigned char so that the mean descriptor requires only 128 bytes of storage (compared to 512 bytes when storing each element as a floating point number). Next, you use the matrix containing the rounded mean descriptors to create a `KDTreeIndex` search index for the member variable `kd_tree_`.

- In the function `TreeBasedMatching`, create a FLANN matrix that holds the descriptors of the query features. Next, use this matrix to perform nearest neighbor search using `kd_tree_`'s `knnSearch` function. Configure the function call such that you obtain the two nearest neighboring 3D points for each query feature. Afterwards, apply Lowe's ratio test on the squared Euclidean distances to the nearest neighbors that are also returned by the function call. Amongst the remaining matches, it still might happen that multiple query features match against the same 3D point. Make sure that you only keep the match with the smallest descriptor distance between the feature and the point and discard the other matches. Return the matches as a set of (feature index, point index) pairs in the output variable `matches`.

- Finally, complete the implementation of the function `LocalizeImage`: After obtaining matches by calling `TreeBasedMatching`, copy the 2D and 3D point positions of the matching features and points into Eigen matrices by using the indices stored in `matches`. Next, use these matrices as an input for RANSAC-based camera pose estimation. You will also need to specify the maximum number of RANSAC iterations.

Congratulations, you have written your (first?) image-based localization pipeline. Make sure that it compiles and run the executable as described above. As an

output, you will see basic information for the matching and pose estimation stages. Most important is the number of localized images, where we consider an image as localized if its pose has at least 12 inliers. Try playing around with the parameters of the pipeline to maximize the number of localized images. You should be able to localize at least 29 of the 30 query images.

5 For the Motivated ...

If you found this exercise too easy or you want to do more advanced things, you can implement advanced methods such as

- adding an additional 3D-to-2D matching step if camera pose estimation fails: Identify matches with a small ratio test value. Next, find all other 3D points visible in at least one database image and match them against the features found in the query image, followed by applying Lowe's ratio test. You might want to construct a kd-tree for these features to accelerate nearest neighbor search.
- implement the advanced sampling scheme for RANSAC introduced in the lecture that exploits co-visibility information.

Please ask Torsten for further details.