6. Implementation

The solvers, RANSAC, sampling methods, and an evaluation framework are written in C++ using libraries Eigen [31] and PoseLib [15]. The code also uses library RansacLib [37], extending it with methods developed in this thesis. The experiments are managed by Python scripts.

The project was tested only with Linux. Please, run it in Linux or in WSL. To run the code for the experiments, only C++ compiler, CMake and Python with pip is needed. First, open the *HybridRANSAC* directory and install the Python requirements:

```
> pip install -r requirements.txt
```

Then, open the *SemiGeneralizedHomography* directory, clone the Eigen [31] library and build all the solvers, RANSAC, speed test and the evaluation experiment using CMake:

```
> cd SemiGeneralizedHomography
> git clone https://gitlab.com/libeigen/eigen.git
> cmake . -B build
> cmake --build build --config RelWithDebInfo -j
```

To run the Python scripts that manage RANSAC experiments, first create the datasets (this needs to be done only the first time) and then run the desired experiment. In the example bellow, we move to the *experiments* directory, create datasets and run an experiment:

```
> cd ../experiments
> python3 create_datasets.py
> python3 experiment_uniform_real_SF.py
```

Following is the documentation for the code, where section names denote the names of directories, see Section A.1 describing attachments.

6.1 experiments/

6.1.1 create_datasets.py

This file contains one function:

- **create_datasets(...)** - this function loads the COLMAP [34, 35] model containing 3D points, images an cameras, takes the 3D points and projects them using the cameras in function **get_inlier_images(...)** to get images with only perfect inlier points. After that it loads the selected pairs of images to create matches between them using **get_pair_matches(...)** followed by creating the list of query images that will be used in RANSAC.

The script then uses this function to create planar KingsCollege dataset together with all-inlier KingsCollege and ShopFacade real datasets.

6.1.2 exp_utils.py

Besides utilities for creating plots, the file contains one useful function:

- **run_experiments(...)** - creates a simple pool to run more RANSAC experiments in parallel (since they are all single-thread) by calling its executable with specified parameters. It uses this pool to run more of the given experiment configurations at once.

6.1.3 matches_from_model.py

This file contains functionality to add noise and outliers to the data. However, it turned out that the dataset have a quite large memory footprint, so it is better to store only datased containg all inliers and no noise, and add them during the experiments. The other main functions include:

- **get_inlier_images(...)** - for all given cameras projects 3D points associated with them and creates images with no noise and no outliers.
- **get_pair_matches(...)** - prepares matches for given pairs of query and database images, which consist of pairs of coordinates of points both in query and in database images, to be saved into text files for later use in RANSAC.

6.1.4 experiment_{*}.py

Simple scripts which set the parameters for each experiment. Then it executes and plots them. The name of the file suggest which experiment it executes (see Section 5.3). The last part of the file name denotes what dataset the experiment uses - planar KingsCollege (_planar) or one of the modified real datasets - KingsCollege (_KC) or ShopFacade (_SF). After creating the datasets using **create_datasets.py**, the experiments can be run like this:

```
> python3 experiment_good_bad_real_SF.py
```

When the the experiment script finishes, the results can be found in directories *plots* and *results_and_plots* inside the HybridRANSAC directory. If necessary, the script will create the directories. Directory *plots* contain pdf figures, while directory *results_and_plots* contains output of the RANSAC and figures as images.

6.2 SemiGeneralizedHomography/

The code inside this directory is adapted and extended from the experiments used for evaluating the solvers for our recent paper [6], extending the RansacLib library [37] with our proposed Hybrid RANSAC, sampling strategies we developed and other utilities.

6.2.1 hybrid_ransac_eval/eval_localization_calibrated.cpp

This file provides a code implementation for evaluating different RANSACs with calibrated solvers in a localization application. The user interface is designed for the intended use inside Python scripts decribed in Section 6.1. The code loads previously created input datasets and sets specified inlier ratios and noise for the cameras (by adding gaussian noise or randomizing points), together with settings specific for each type of experiment. During the execution of the experiments, progress reports are generated. Finally, the code generates statistics which can be further analyzed.

6.2.2 RansacLib/proposed_hybrid_ransac.h

The implementation of our **ProposedHybridRansac** is inspired by the standard RANSAC from RansacLib [37]. The current state of the RANSAC is represented by structure **PHybridRansacStatistics** with main members:

- **num_iterations** - holds the value of the current iteration by counting them
- **best_num_inliers** - number of inlier data poins with the best model found so far.
- **best_model_score** - score of the best model found so far.
- **inlier_indices** - vector containing indices of inlier data points (with current best model).
- **inlier_ratios** - vector of estimated inlier ratios for each pinhole camera from the generalized camera.
- **stats_after_new_best_model** - vector containing more extensive statistics about the state of the RANSAC after finding each new best model (score, orientation and position errors, selected cameras for the minimal sample leading to the best model and the iteration at which the best model was found).

The **ProposedHybridRansac** itself has no members and requiers an instance of **PHybridRansacStatistics** to track its state. The main methods implemented are:

- **EstimateModel(...)** - implemention of the Proposed Hybrid RANSAC from Algorithm 3.
- **ScoreModel(...)** - sums squared errors for each data point.
- **GetInliers(...)** - computes inlier ratio of the whole dataset, or inlier ratios for each pinhole camera and optionally saves them.

6.2.3 RansacLib/inlier_sampling.h

Implementing the Algorithm 4 and Algorithm 6 directly without the Probability sampling Algorithm 5, which was introduced only to make the explanation of the algorithms more clear. The important members of the class **InlierSampling** are:

- **Sample(...)** - chooses the cameras based on their given probabilities and draws the minimal sample.
- **ChooseCam(...)** - selects one camera based on the quasi probability distribution, which doesn't have to sum to one (because it is implicitly normalized).
- **DrawHybridSample(...)** - uniformly at random selects given number of distinct indices of matches from a given camera.

6.2.4 RansacLib/proposed_hybrid_sampling.h

A template prepared for future work, where sampling is guided also by prior probabilities, as in Hybrid RANSAC by Camposeco et al. [7]

6.2.5 RansacLib/{*}_sampling_two_cameras.h

A version of **inlier_sampling.h** and random **sampling.h**, that are modified to choose matches only from two cameras for experimental purposes.

6.2.6 solvers/

Directory of each solver contains cpp directory containing the C++ implementations of the solvers developed in this thesis, together with directory original_matlab, which contains Matlab implementations of the solvers proposed in [6]. The C++ code was developed based on the Matlab implementation. Details on the implementation of the solvers, which take coordinates of query poits, database points and camera center as input and return estimated homographies as output can be found in Section 2.3.

6.2.7 ransac_solvers/hybrid_H50_ransac_solver.{*}

This file implements RANSAC wrapper for the implemented solvers. It is inspired by the Solver class functionality required by RansacLib [37]. In order to design and test novel hybrid sampling approach, we needed to modify and extend the interface. The main members of the **HybridH50RansacSolver** class are:

- **matches_** - RansacLib [37] made a deliberate design choice to make the Solver classes responsible for encapsulating the input data. This is because different solvers might require additional data in some very specific form. The solver implemented in this thesis organizes the input matches in a vector, where matches from each camera are grouped together in a continuous block. Multiple blocks from each camera are concatenated to form the complete vector, effectively encapsulating the input matches.

- **cameras_** - a vector of Camera objects, denoting each pinhole camera inside the generalized camera.
- **match_ranges_** - a set of pairs $(\text{start}_i, \text{end}_i)$ denoting the starting and ending indices of matches from i-th camera.

The main methods implemented are:

- **min_sample_size()** - returns the number of points required by the minimal solver
- **num_data()** - returns the number of matches stored in the solver
- **get_num_of_cams()** - returns the number of pinhole cameras forming the generalized camera.
- **get_match_ranges()** - returns a const pointer to match_ranges_
- **MinimalSolver(...)** - calls the suitable minimal solver on a set of given indices, based on the sources of the correspondences.
- **EvaluateModelOnPoint(...)** - evaluates a given pose on a given data point and returns its Triangulation error described in 4.

6.2.8 matlab_stability_tests/

A directory containing Matlab scripts that measure the numerical stability of the solvers. They were used in our paper [6] and were only slightly modified to test the new implementations. The code in this directory is not part of the contributions of this thesis, and is included only for the purpose of reproducing the stability experiments if needed. It contains third party work, including scene generator introduced in [32].

6.2.9 solver_tests/test_sh{*}.h

An adapter to run the selected implementation of a given solver.

6.2.10 solver_tests/tests.cpp

One of the factors that affect the measured speed is the input data management. In order to do deal with that, a dataset of N random inputs for the solvers was created. Then it runs for each solver W warm-up iterations with each input in order to maximize the probability of loading them into a low level cache. After this, each input was tested for I iterations. The tests can be performed inside the *SemiGeneralizedHomography* directory by running :

```
> build/solver_tests/tests  N  W  I  S  O
```

where N is the desired size of the random dataset, W is the number of warm-up iterations to perform, I is number of iterations and S determines the interval (-S, S) from which the data is randomly generated. The brief results will be saved to file O, more extensive result are printed using standard output. All the parameters have default values a can be omitted.