Vision Algorithms for Mobile Robotics mini-project: Monocamera Visual Odometry Pipeline

ANTONIO ARBUES, ANTONIO TERPIN

Compiled January 8, 2021

In this report, we present the monocular visual odometry pipeline developed for the final project of the Vision Algorithms for Mobile Robotics class at ETH Zürich. In particular, the authors claim that the proposed pipeline: (i) achieves a good global trajectory in all of the datasets; (ii) presents a reduced scale drift even in the most challenging datasets; (iii) introduces a continuous bundle adjustment which, even modifying only on the recent poses, ensures a fully consistent global trajectory; and (iv) is a flexible platform that allows further experiments and improvements in a lean way.

1. INTRODUCTION

This project is the assembly of all of the concepts learned during the Vision Algorithms for Mobile Robotics[1] class at ETH Zürich. In particular, the building blocks developed during the exercise sessions have been improved and assembled to build a monocular visual odometry pipeline, with very satisfying results. The source code can be found in the GITHUB repository antonioterpin/visual-odometry-mono, with the detailed instructions on how to get started.

In this report, the proposed pipeline is presented and critically analyzed. In particular, the authors claim that it implements five additional interesting features:

- the pipeline achieves a good global trajectory estimate;
- improvements to combat the scale drift are proposed and implemented;
- the estimated trajectory and the ground truth are quantitatively compared;
- a continuously integrated bundle adjustment is implemented and running on the pipeline; and
- the pipeline is a flexible platform that allows further experiments and improvements, easing the introduction of different blocks

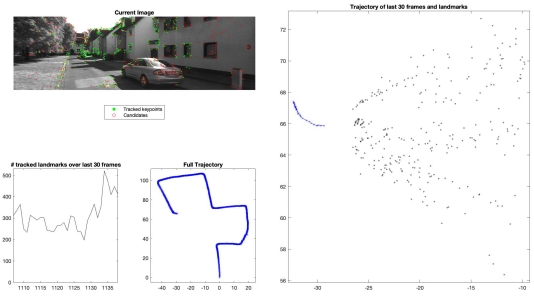

Fig. 1. Screenshot of the output of the pipeline on the KITTI dataset.

A. Workflow

The project conducted by the two authors has followed a progressively increasing improvement with respect to the organizational and job subdivision tasks. In addition, long and short term vision of the project milestones helped keeping the enthusiasm to work for a long period of time.

Despite the pandemic situation, the project development went smoothly due to the lean management approach adopted. The source code has been versioned with `git` and hosted on GitHub. The Gitflow[2] paradigm was used to allow incremental improvements of the pipeline. The main branch was used only for working releases of the pipeline, with features significant enough to receive precious feedback on the pipeline performance. This simplified the integration testing of the different building blocks.

Both C++ and MATLAB have been considered as programming languages for the project. However, the authors considered MATLAB a superior choice for this first stage of the project. Indeed, the long term vision has always been to have a highly configurable pipeline, suitable to easily test different approaches. Therefore, having the ability to rapidly prototype in MATLAB allowed the authors to achieve satisfying results in the given time constraints. Weekly goals have been set and achieved, gradually introducing features until the submitted pipeline has been developed. However, as stated previously, the authors are willing to keep maintaining and improving the pipeline, porting it to C++.

Primary importance has been given to the structure and reusability of the pipeline code, investing a decent amount of

[1] http://rpg.ifi.uzh.ch/teaching.html

[2] https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

time developing a polymorphic-based object-oriented design, to allow replacing of building blocks from the configuration file (`.json`), and through easy extensions of the base, abstract, classes. Hopefully, this pipeline is suitable as a well-grounded structure for a quick-start in Visual Odometry for the community.

B. Datasets

The proposed pipeline has been tested on the following datasets:

- The parking dataset (monocular)
- The KITTI[3] dataset (stereo)
- The Malaga[4] dataset (stereo)

Since the scope of this project was to build a monocular visual odometry pipeline, only the left images of the stereo datasets were used.

Since the authors have primarily tested the pipeline on the KITTI dataset, the images used to provide intuition through the report are taken from this dataset. However, the pipeline has been tested on all of the datasets and the in *Section* 4 the results for all of them are reported.

C. Report structure

The report is structured as follow. First, in *Section* 2, the structure of the proposed pipeline is presented.

Second, in *Section* 3, a brief recount of some interesting problems encountered is presented. In hindsight, the development process would have been much faster knowing these (sometimes simple) considerations, which for lack of experience required many experiments. Hence, the following may help future beginners in producing a working pipeline in less time and thus, they will be able to focus on improving it.

Finally, in *Section* 4 both a quantitative and qualitative evaluation of the pipeline is proposed, and in *Section* 5 a final comment on the results and on the work done can be found.

D. External functions

This project has been developed with the rewarding goal of learning in detail the algorithms studied in class. Therefore, all of the code has been produced by the authors, who had to improve the efficiency to obtain a reasonable speed. The only relevant functions not written by the authors are:

- The `p3p.m` and `solveQuartic.m` implementation by Laurent Kneip[5] [1].
- The `vision.pointTracker` class, i.e., MATLAB implementation of the KLT tracker [2, 3].

This choice was done for mainly two reasons. First, the `p3p` algorithm did not allow many variations in the implementation and thus, it seemed reasonable to use the open-source implementation directly. Second, the custom implementation of the KLT tracker was very inefficient compared to the MATLAB one, and the time constraints on the project did not allow to invest energy on the improvement. Hence, the suggestion in the project statement was followed and the MATLAB implementation was used.

2. PIPELINE STRUCTURE

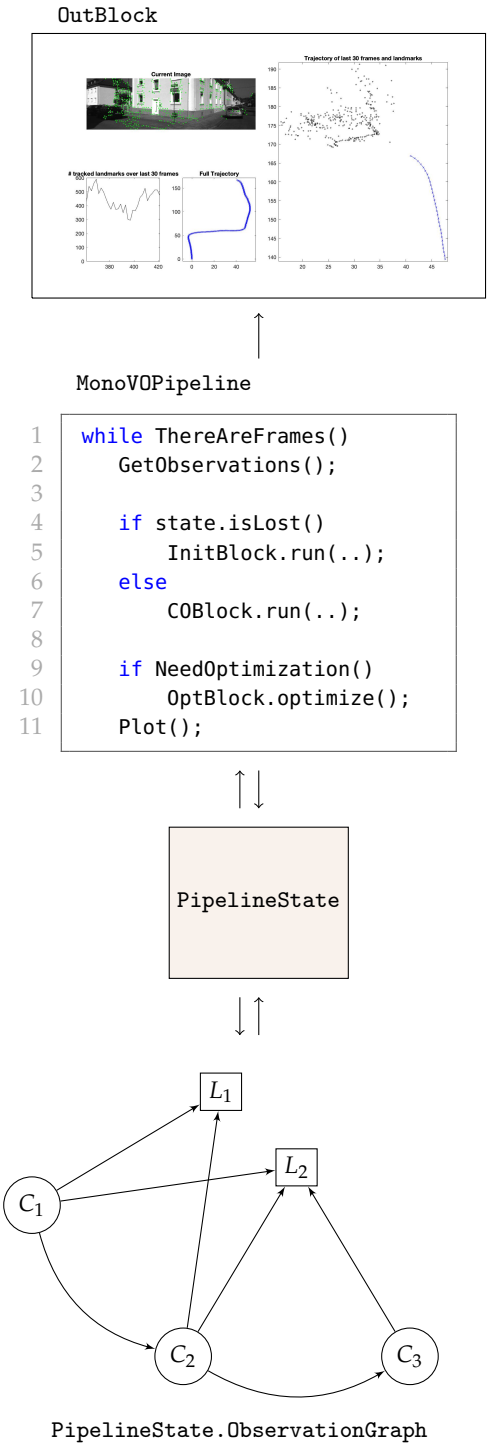

Fig. 2. The working principle of the proposed pipeline.

[3] http://www.cvlibs.net/datasets/kitti/eval_odometry.php
[4] https://www.mrpt.org/MalagaUrbanDataset
[5] http://en.pudn.com/Download/item/id/2969453.html

A. Structure

The proposed pipeline has been designed following two main principles: (i) modularity and (ii) decoupling. The former played a fundamental role to allow the testing and evaluation of the different algorithms that could be used for each function. The latter guaranteed an easy way to debug and find errors. Furthermore, the contribution of each new feature was clearly visible at each step.

Conceptually, the pipeline has the same basic features suggested in the project statement. Indeed, it reads the data from the datasets, it has an initialization step and a continuous operation step. However, an object-oriented structure was employed to meet the above-mentioned principles.

In particular, five modules where designed to assemble the pipeline:

1. `InputBlock`, responsible of abstracting the interface of the pipeline to the datasets (see *Section* A);
2. `InitBlock`, to bootstrap the first set of landmarks and initially localize the pipeline; moreover, it is also responsible to recover the position in case the following block gets lost (see *Section* B);
3. `COBlock`, to continuosly localize the camera with respect to the tracked landmarks (see *Section* C);
4. `OptimizationBlock`, to improve the trajectory estimate and the landmarks position online (see *Section* E); and
5. `OutputBlock`, to provide a runtime user interface.

The authors experienced a non negligible initial overhead in structuring the pipeline, but then it was possible to easily expand the pipeline, add new features and test.

Indeed, all of the basic blocks are developed as abstract classes, that are inherited by the actual blocks. Then, a configuration file was used to both define the parameters of each block and to tell the pipeline which block to mount at run time.

The modularity allowed to develop in a decoupled fashion all of the blocks, test them separately with the exercise sessions, and then integrate them into the pipeline seamlessly.

The working principle of the proposed pipeline is summarized in *Figure* 2. It is analogous to the suggested working principle, with the addition of the online optimization step, which is performed every once in a while, based on the tuning parameters. However, under the hood a `PipelineState` object keeps everything together, allowing the great flexibility of the pipeline, and it drastically differ from the suggested stateless configuration.

The job of `PipelineState` can be summarized as handling the `ObservationGraph`, the graph connecting poses and landmarks. The edges between two cameras are considered implicit, and only the transformation between the initial pose and the current pose is kept in memory. The edges between landmarks and poses, instead, are represented by the keypoints. The flow chart of the `ObservationGraph` maintenance is shown in *Figure* 3.

The MATLAB `graph` structure is used, and the methods `addLandmarks` and `addPose`, respectively, to add a new set of landmarks (used after the initialization to provide the first set of landmarks, and internally for the triangulation) and a new camera pose to the observation graph. The `addLandmarksToPose` is used to add the observation of a tracked landmark at a particular pose, i.e., to add the edges to the observation graph.

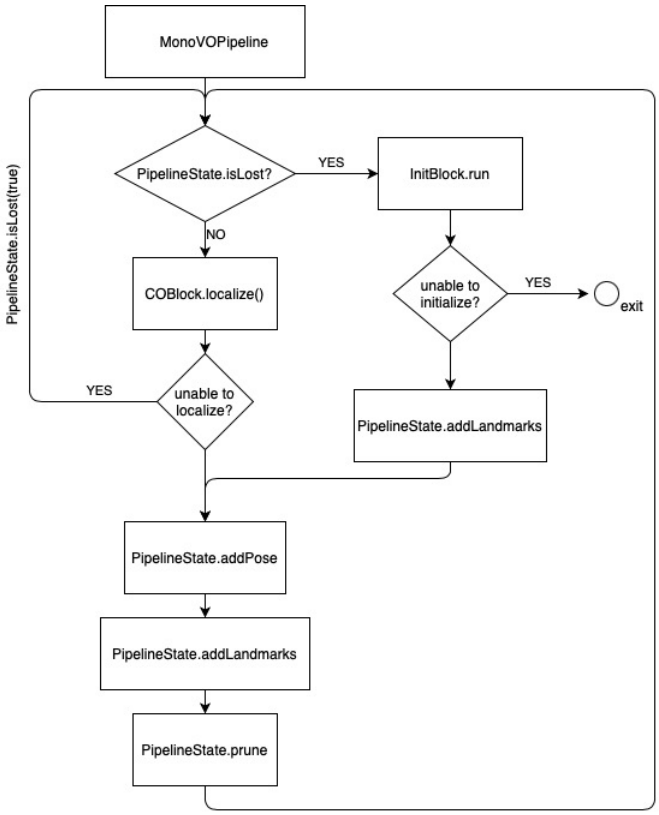

Fig. 3. Simplified flow chart of the pipeline, highlighting the `PipelineState` methods for the `ObservationGraph` maintenance.

The method `getObservations` is used to get the landmarks-keypoints correspondences at a previous pose, to later perform the continuous operation step.

The `isLost` method is used to decide whether to perform either a continuous operation step (`COBlock.localize`) or a initialization step (`InitBlock.run`). The final pipeline, properly tuned, requires only a single initialization step, at the beginning.

Finally, the `prune` method is offered to remove poses older than a selected number of frames, so that also the remaining isolated landmarks could be removed, greatly reducing the memory used and thus, keeping a high efficiency even after a large number of frames.

Moreover, the `PipelineState` is also responsible of storing the keypoints candidate for triangulation, deciding whether to triangulate them or not (see *Section* D), and providing the interfaces to optimize the trajectory and the observations (see *Section* E). Although the initial overhead required to set up such a state setting, the authors believed that it was worth it in the end to have this configuration because it greatly helped in integrating the different features, and in particular the optimization module.

B. The configuration file

A great importance was given to the user interface for the control of the dozens of tuning parameters of the pipeline.

A configuration file in the *.json*[6] format was created for every of the three datasets and updated with all the tuning parameters every time a new feature was added to the pipeline.

In this way, the parameter tuning was eased, and in addition, it makes the pipeline immediately accessible by potential users that were not involved in the development.

In particular, the `config.json` file is read by a class - `Config.m` - and its parameters are imported.
In *Listing* 1 a snippet from the `Config.m` class. Here, among other functions, `loadFromFile(filename)` is defined. It picks every parameter in the `config.json` and loads it in the class of interest.

Listing 1. `Config` - Configuration from `.json` file

```
1 classdef Config
2
3 function obj = loadFromFile(filename)
```

The most relevant control and tuning parameters set in `config.json` are showed in *Table* 5.
In the Repository is possible to find the three config.json files, each one containing the tuning parameters for one of the three datasets.

3. CHALLENGES AND SOLUTIONS

Numerous challenges have been faced during the development of each feature of the pipeline. In this section, a brief summary of the fundamental ones is proposed, with a particular focus on how they have been faced, how the solutions were tested and, eventually, which tricky problems arose and how they have been solved.

[6] https://www.json.org/json-en.html

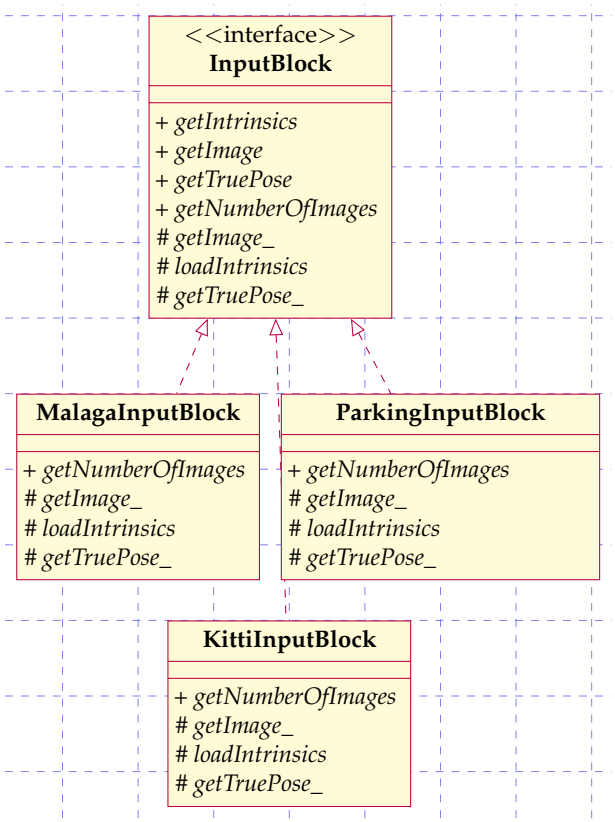

Fig. 4. `InputBlock` (minimal) class diagram.

A. Uniform interface to the datasets

Chronologically, the first thing to do to work with the dataset was actually to extract the data from them. To simplify the addition of other dataset, and handling in the same way all of the provided ones, a common interface was designed through the abstract class `InputBlock`. Then, for each dataset was possible to derive from this abstract class, and implement the methods `getImage_`, `getTruePose_`, `loadIntrinsics` and `getNumberOfImages` to be used seamlessly with the rest of the pipeline.

The information in the dataset can be recovered through the public methods `getImage`, `getTruePose`, `getIntrinsics` and `getNumberOfImages`, that eventually add sanity checks. To run the pipeline on a different dataset, it is enough to change the fields `InputBlock.Handler` and `InputBlock.Path` in the `.json` configuration file.

The sketch of the class diagram of the `InputBlock` is shown in *Figure* 4. The name of the methods is rather self-explanatory, with `getTruePose` returning the ground truth measurements. For a dataset in which there are not these data, or there is not a ground-truth measure per each frame, the ground truth can be interpolated (as it has been done in the case of the Malaga dataset) or can be replaced with dummy data. Indeed, during the run, these are used only for visualization, which can be disabled (e.g., `OutputBlock.plotGroundTruth`) in the configuration file.

There are two main reasons behind this structure for the input block. First, it allows easier debug and test of each input. For instance, it is easy to use the single input object to test the tracking of the features with the `vision.pointTracker` class, in a fully decoupled fashion from the pipeline, without copy-pasting any code, which can be error-prone. Second, it allows

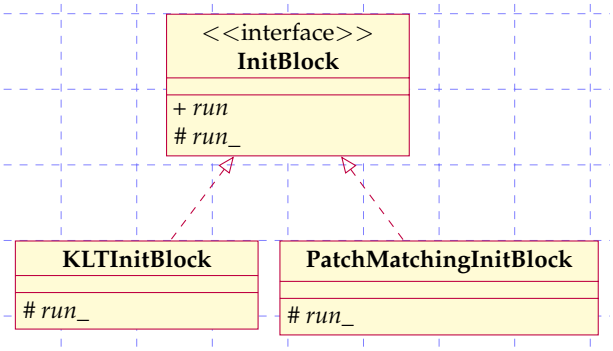

Fig. 5. `InitBlock` (minimal) class diagram.

high scalability and re-usability of the pipeline. Indeed, one main goal of the authors, left as a further work because of the time constraints, is to integrate this pipeline with a toolbox to easily introduce custom recorded datasets.

B. The initialization problem

The first block of the pipeline is in charge of simultaneously locate the camera and bootstrap some landmarks with respect to which to localize in the subsequent frames. This can be done exploiting 2-view geometry, establishing keypoints correspondences between two (sufficiently distant) keyframes, and exploit a Structure From Motion algorithm.

In particular, we use the 8-point algorithm [4] together with `RANSAC` [5] for outlier rejection.

Since in this pipeline calibrated cameras were considered, it was possible to either estimate the essential or the fundamental matrix. The 8-point algorithm with `RANSAC` was preferred over the 5-point algorithm [6] for the initialization because easier to implement (around ten lines of code in MATLAB), efficient enough, and easier to test for correctness. Indeed, it was possible to precisely evaluate the proposed implementation comparing it with the reference solutions provided in the *Exercise 6 - "Eight-point algorithm"*[7], to make sure that they were given the exact result despite the efficiency improvements.

The initialization block to work needs to find a set of keypoints correspondences between two frames. The flexibility discussed in *Section* 2, was exploited to test two different approaches to achieve this result. A minimal class diagram for the InitBlock is shown in *Figure* 5. The classes inheriting from `InitBlock` must overload the `run_` method, which is internally used after sanity checks and before other operations, such as candidates tracking. The `tracker` is a generic object and can be handled freely by the blocks, or ignored. For `KLT` is used to share the tracker between the `InitBlock` and the `COBlock`.

In the first one, a set of keypoints was extracted from both the frames with the `DetectorBlock.extractFeatures` and `DetectorBlock.describeFeatures` methods, and then matched with `DetectorBlock.matchFeatures`.

In the second approach, the KLT algorithm is used to track the features between the two keyframes, considering also the frames in between. Hence, the `DetectorBlock` was used only once, in the first frame.

Actually, the `DetectorBlock` is a configurable block. The authors focused only on the Harris detector (`HarrisDetectorBlock`) because of its high localization properties. However, it is possible to easily extend the base class to implement other feature extraction algorithms. For the feature matching strategies, the Sum of Squared Difference (SSD) is used. The code developed for the `DetectorBlock` was thoroughly tested with *Exercise 03* - "Harris detector + descriptor + matching"[8] before being integrated in the pipeline.

Both methods (KLT and patch matching) turned out to have comparable results in terms of accuracy, but the `KLTInitBlock` had superior performances and allowed to bootstrap a larger amount of high-quality landmarks (450 to 500 compared to the 150 to 200 of the `PatchMatchingInitBlock`). To switch between the just it is enough to change the `InitBlock.Handler` entry in the configuration file, and adjust the parameters in `InitBlock.Params`.

A handy solution, that revealed helpful, effective, and lean during the whole pipeline development, was to centralize commonly used algorithms, providing a generic implementation. For brevity, we describe only the most meaningful one, `RANSAC`.

The intuition here was that `RANSAC` is indeed a generic algorithm for model fitting. Hence, instead of implementing the outlier rejection for each model that had to be fit for the correct functioning of the pipeline, `RANSAC` was implemented as a standalone function that takes two `handles` and returns the selected model and the corresponding inliers mask.

This approach greatly reduced the developing overhead and the debugging process.

Listing 2. `RANSAC`

```
function [model, inliers] = RANSAC(...
    modelFromSample, errorMetric, ...
    nParams, data, nIterations, ...
    otherParams, verbose, confidence)
```

Both the epipolar line distance and the reprojection error was tested as error metrics, but the latter revealed easier and more intuitive to tune and thus, was preferred. The expected consequences were higher accuracy and lower speed. However, introducing the adaptive implementation of `RANSAC` solved the efficiency issue.

The major advantages of having this centralized implementation were:

- Possibility to easily test different metrics;
- Possibility to improve the `RANSAC` algorithm in a single location (e.g., adaptiveness);
- No need to think about the outliers filtering for each model (initialization and continuous operation refer to the same `RANSAC` implementation)
- It is easy to introduce new outliers filtering methods such as Graduated Non-Convexity [7].

Listing 3. `RANSAC` usage example with the 8 point algorithm

```
[pose, inliers] = RANSAC(...
    @fundamentalPoseFromSample, ...
    @reprojectionErrorFromFModel, ...
    8, [p1; p2], nIterations, ...
    [obj.K(:); T_1W(:); errorTh; obj.maxDistance], ...
    verbose, confidence);
```

[7] http://rpg.ifi.uzh.ch/docs/teaching/2020/exercise6.zip

[8] http://rpg.ifi.uzh.ch/docs/teaching/2020/exercise3.zip

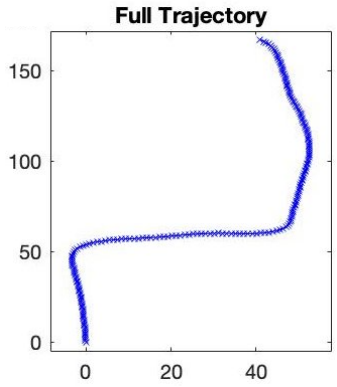

Fig. 6. Trajectory estimate with only the initialization block. As expected, no scale consistency can be ensured, but the shape confirms empirically the correctness of the implementation, already tested with the *Exercise 6*.

All the code relative to `RANSAC` produced for this first block was thoroughly tested exploiting the *Exercise 07 - "P3P algorithm and RANSAC"* . The results were exact compared with the reference implementations and the randomness was faced using the `rng` MATLAB function. In particular, the above referred *Exercise 7* was further used to test the following section.

Finally, we marked as working correctly the initialization block only once it was possible to obtain a qualitatively good full-trajectory estimate shape only using this first block.

In *Figure 6*, the estimate trajectory of part of the KITTI dataset using only the initialization block is shown. As discussed before, no scale consistency can be expected from this simulation settings. However, it is qualitatively visible the quality of the initialization block, which produces the correct trajectory shape (up to the distortion introduced by the missing scale at each step).

This experiment can be reproduced by setting a very high `Pipeline.State.lostBelow` value in the `.json` configuration file. This way, every time the pipeline performs the initialization step because it thinks it is tracking too few landmarks and that it would not be able to reliably estimate the camera pose. No bundle adjustment was introduced at this point of the development.

C. Pose estimation in subsequent frames

The second main block of the pipeline is the `COBlock`, i.e., the block in charge of performing continuously the localization of the camera pose from the previously tracked landmarks. To achieve this goal, the block has to be able to find the $3D-2D$ correspondences between the landmarks tracked at the previous frame and the keypoints of the current frame.

Once these correspondences are given, it is possible to localize the camera with respect to the $3D$ points with one of the many algorithms proposed. In particular, in this work the Direct Linear Transformation (DLT) [8] and the Perspective from 3 points (p3p) [1] were considered.

Whereas the DLT leads to an overdetermined solution, the PnP requires four points for an exact solution. In particular, three points is the minimum amount to have a finite amount of solutions, four, between which then the true solution has to be found. To do this, a fourth point can be considered. In particular, we pick the solution that maximizes the number of landmarks that have a reprojection error lower than a specified threshold

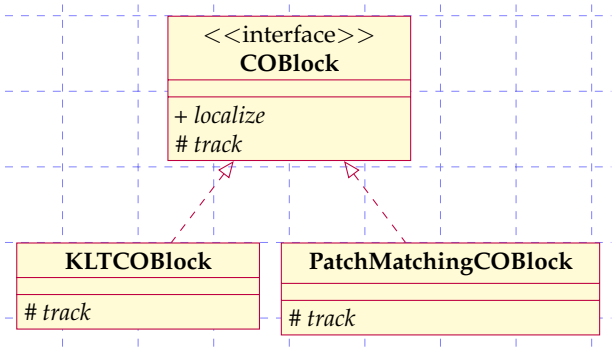

Fig. 7. `COBlock` (minimal) class diagram.

(`COBlock.Params.p3pTolerance`).

Anyway, since the observations from the previous frames were expected to be noisy, `RANSAC` was used to remove the outliers. In *Listing 4* the usage or `RANSAC` with the p3p algorithm is shown. In particular, the considered model function performs also the solution disambiguation discussed before.

Listing 4. `RANSAC` usage example with the p3p algorithm

```
[pose, inliers] = RANSAC(...
    @p3pPoseFromSample, ...
    @reprojectionErrorFromP3PModel, ...
    3, [normalizedBearings; landmarks; keypoints],
        nIterations, [K(:); p3pTolerance^2], ...
    verbose, confidence);
```

In the end, the solution with the p3p and solution disambiguation was preferred over the DLT solution, but they gave nearly exactly the same results in the above-mentioned exercise.

In *Figure 7* the class diagram for the `COBlock` is shown, with the public method `localize` and the protected (to overload) method `track`. The `localize` method additionally introduce the concept of candidates, further discussed in the following section.

The two specialization of the `COBlock` differ only for the way the previously tracked landmarks are associated with the keypoints in the current frame. Using the patch matching revealed really detrimental (see *Section 4*) compared to the KLT-based block. The authors spent a considerable amount of time trying to obtain decent results with the former approach, without success. With the latter, a couple of hours were enough to dramatically improve the pipeline performance. This is actually an expected result, because of the really low repeatability of the Harris detector and the errors introduced by the SSD. Other metrics and approaches could be further investigated, but the experiments totally support the choice of switching to KLT.

In *Figure 8*, a screenshot of the output of the proposed pipeline without new landmarks triangulation is proposed. Every approximatively 20 frames the pipeline needs a new bootstrapping and loses the scale. However, the for around 20 frames is able to correctly localize. When the number of tracked landmarks is low, the uncertainty of the estimate grows, and the position jitters. Hence, keeping a large enough number of high-quality landmarks became the next milestone.

D. Landmarks triangulation and scale drift

With the deployment of the KLT algorithm, the pipeline was able to track the keypoints for around 20 frames. However, a

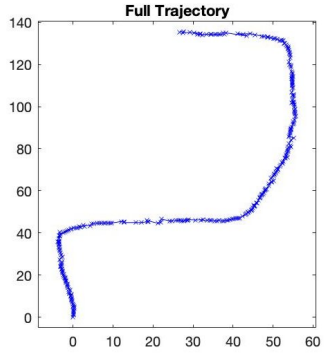

Fig. 8. Trajectory estimate with the initialization block and a rudimental continuous operation block - no triangulation of new landmarks - jitter experienced when number of landmarks is low.

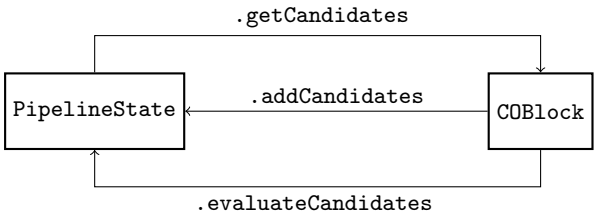

Fig. 9. Scheme of the candidates handling through `PipelineState.getCandidates`, `PipelineState.addCandidates` and `PipelineState.evaluateCandidates`

re-initialization was requested every time the tracked features dropped below a (tuned) reliability threshold. Although this helped to avoid the jitter discussed in the previous section, this also meant to completely lose the scale and thus, the global consistency of the map.

At each frame, new candidate keypoints were extracted from the current frame and tracked in the subsequent frames. In *Figure* 9, the scheme of how the candidates are handled with the main methods to handle them are shown. First, new extracted keypoints are added to the state with `addCandidates`, then, they are tracked with the `COBlock`. To this scope, they are taken from the state with `getCandidates`. Finally, the tracked candidates are evaluated with `evaluateCandidates`. The candidates are stored in the recommended way, keeping the pose of the first frame where they have been seen, the observation in that frame, and the observation in the current frame.

The state handles the quality of the triangulated landmarks, decide whether to triangulate new ones, and update the `PipelineState.ObservationGraph`.

A criteria on whether or not to triangulate a keypoint tracked over multiple frames became quickly necessary, because triangulating every correspondences resulted in poor trajectory estimates due to the low quality of the landmarks.

The first criteria implemented was based on the reprojection error. However, this criteria was very much dependent on the tuning of the threshold the pipeline was not able to keep a high number of landmarks, incurring again in re-initialization.

A second criteria was the one suggested in the project statement on the angle between the hypothetical world point and the two observations. In this way, it was possible to avoid triangulating landmarks with small angle among bearing vectors which means a high level of uncertainty on the actual position of the landmark. A high bearing angle limits the indeterminacy zone of the landmark.

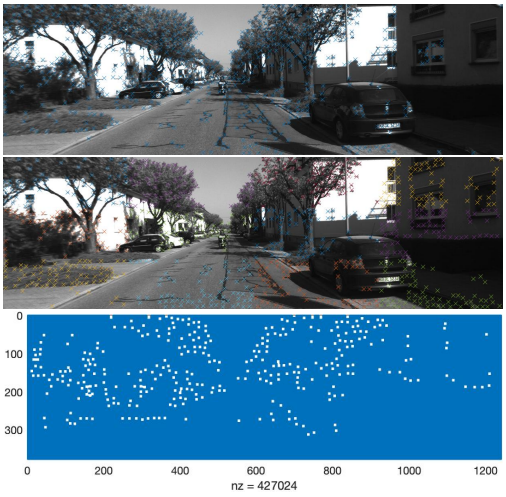

Fig. 10. Uniform (top) vs non uniform (middle) sampling. In the bottom figure the mask used to sample the keypoints is shown.

However, a large portion of the triangulated world points was filtered out by the `p3pRANSAC` at the following iteration. This was solved by introducing a uniform sampling of the proposed candidates. Moreover, the candidates were picked only from regions not too close to the already tracked keypoints. This was achieved through a dynamically built mask. The uniform sampling is achieved by detecting an equal number of corners in every part in which the image can be divided (e.g., top-left, top-right, and so on). In *Figure* 10 a non uniform sampling (top) is juxtaposed to the uniform sampling (middle), with the mask used (bottom). To plot these figures during the simulation is enough to change the parameters `Detector.plotMask` and `Detector.plotKeypoints` to the number of the figure on which to plot. To disable they have to be set to zero.

Although this criterion proved to work very well, a dramatic improvement was obtained with the heuristic of amplifying the distance between the two considered camera poses. In *Figure* 12 this concept is summarized. The angles corresponding to the landmarks lying around the current heading of the camera are amplified much more than the landmarks aligned with the heading of the camera. This method proved to double the number of triangulated landmarks as well as to maintain a sufficient homogeneity in the position of the landmarks in space.

In *Figure* 11 a visualization of the landmark triangulation process in two subsequent frames of the Malaga dataset.

The last triangulation issue relevant for the estimation of a globally consistent trajectory was the scale drift. The observed problem was a very fast decaying of the distance between subsequent poses. It has been concluded that this was due to the fast decaying of the landmarks distance from the camera. This is indeed a positive feedback loop, in which closer landmarks lead to estimated smaller displacement of the camera, which in turn leads to a perceived scene scale reduction and thus, closer landmarks.

This meant that the following camera poses would span a

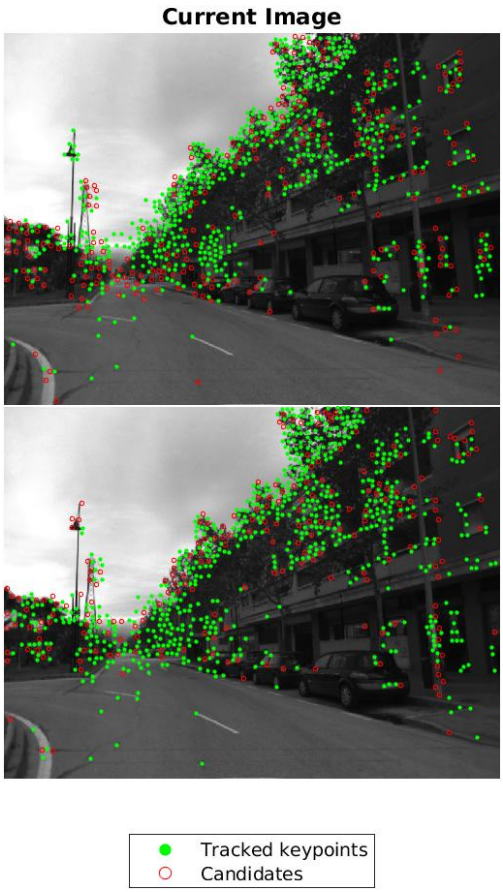

Fig. 11. The landmark triangulation visualized on the image plane. Green dots in the right image replacing the red dots in the left image are keypoints corresponding to triangulated landmarks. The two images were taken from the Malaga dataset.

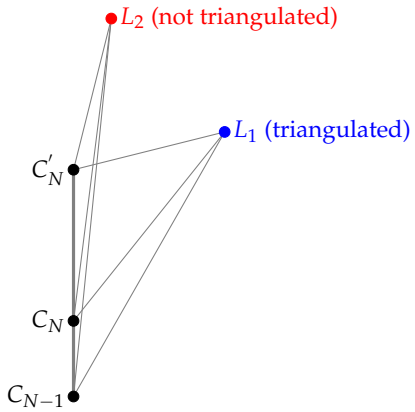

Fig. 12. The heuristic of the pose distance amplification for landmark triangulation filtering. C_i is the i-th camera pose, C'_N is the camera pose after the amplification of the distance. L_j is the j-th hypothetical landmark.

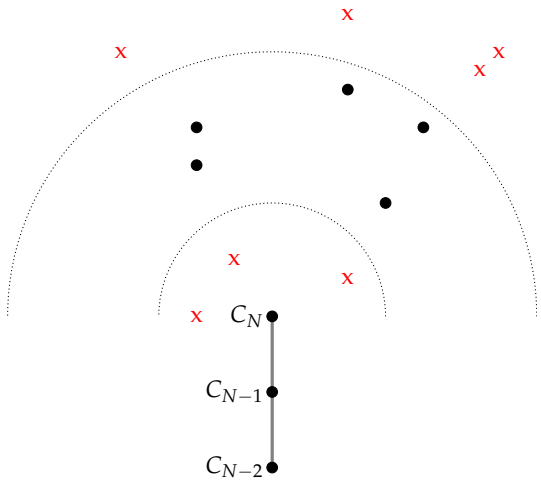

Fig. 13. The heuristic applied in order to avoid scale drift: a minimum and a maximum distance are set and landmarks outside the considered window are discarded (red crosses). C_i represents the i-th camera pose.

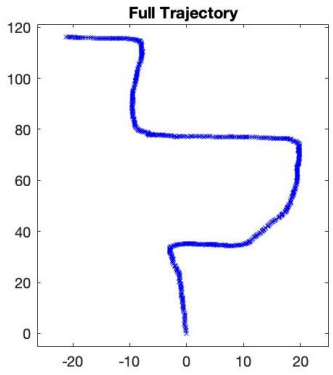

Fig. 14. Trajectory estimate with the triangulation and the described heuristics.

very limited distance compared to the previous ones. This effect is also known as scale drift and to tackle this problem was enough to focus on the triangulation of new landmarks, because they were in the first place the initial source of error, which then lead to the detrimental feedback loop.

It was noted that very distant landmarks not removed by the bearing angle criteria would be anyway deleterious for the sake of maintaining the correct scale. This is because there is virtually no change in their distance from the camera poses as the camera moves, leading to the illusion that the camera itself would stay still. Hence, a maximum distance from the landmarks was implemented and any landmark above the selected distance was removed.

Furthermore, in order to radically avoid the risk of a massive scale drift, also a minimum distance threshold was implemented. The intuition is that by avoiding too close landmarks, the dangerous feedback loop is clamped, allowing the triangulation algorithm to recover and improve the landmarks quality.

A visualization of the adopted heuristics useful to avoid scale drift is provided in *Figure* 13.

The above mentioned heuristics proved to work very well after tuning, achieving the trajectory shown in *Figure* 14.

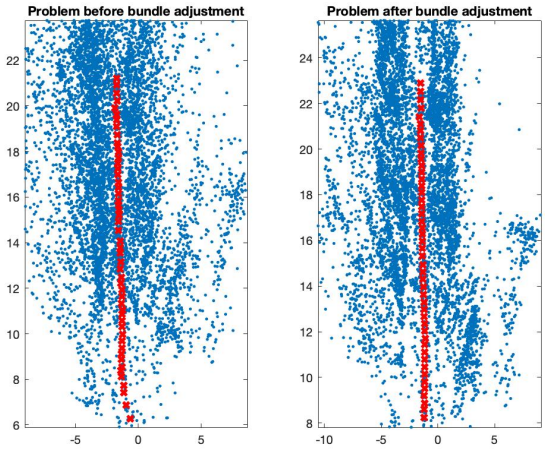

Fig. 15. Example of problem before and after bundle adjustment

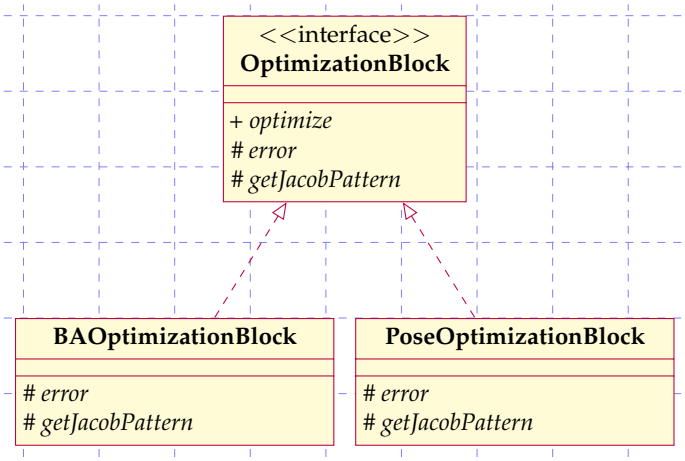

Fig. 16. `OptimizationBlock` (minimal) class diagram.

E. Continuous Bundle Adjustment

To further improve the quality of the trajectory, bundle adjustment was introduced. Given the `PipelineState.ObservationGraph` data structure, it was immediate to integrate the code tested during the *Exercise 09* - "Bundle Adjustment"[9].

Listing 5. Interface tot the pipeline state for `ObservationGraph` optimization.

```
function [hiddenState, observations, bundleIdx] ...
    = getOptimizationDS(state)
function optimizedBundle(state,hiddenState,bundleIdx)
```

The `PipelineState` offers the methods shown in *Listing* 5 to get the bundle data from the `PipelineState.ObservationGraph` in the format suggested in the exercise session and to tell the results after the optimization.

The `bundleIdx` object is an opaque structure used internally to the state to perform the update on the `PipelineState.ObservationGraph`.

[9] http://rpg.ifi.uzh.ch/docs/teaching/2020/exercise9.zip

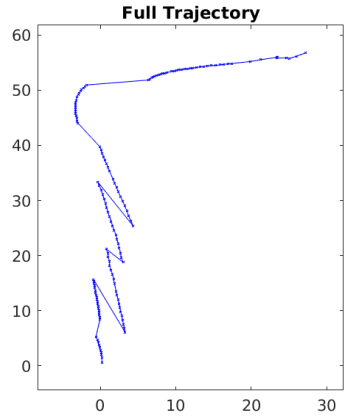

Fig. 17. Bundle adjustment without fixing the first poses leads to an unconsistent trajectory.

The specializations of the `OptimizationBlock` (see *Figure* 16 for a minimal class diagram) define the jacobian pattern and the error function. Eventually, they can add options to the optmizer. The `OptimizationBlock.optimize` methods puts everything together and optimize the given `hiddenState` (see *Exercise 09* for the definition of `hiddenState` and the input of the optimization).

The initial overhead in handling the state as a graph totally paid off during the integration of the optimization within the pipeline, done in a few lines of code. Moreover, it is rather easy to substitute the `BAOptimizationBlock` with other optimization algorithms, such as Pose Graph Optimization.

Regarding the Bundle Adjustment algorithm, implementation details do not differ much from the suggested approach in *Exercise 9*. However, it was not trivial to make it work on successive windows. Indeed, the optimized bundle typically was inconsistent with the poses estimated before and not included in the window (see *Figure* 17). The solution was as simple as changing the Jacobian matrix given to `lsqnonlin`, so that the optimization did not modify the first poses.

The authors also tried to weight the error based on the position of the world points, to prevent landmarks to go behind the camera during the optimization, and based on the smoothness of the trajectory. However, the results were comparable to the solution without these heuristics. Moreover, a `PoseOptimizationBlock` as an alternative to the `BAOptimizationBlock` was implemented, that only optimized on the camera poses. However, the results obtained with the latter were far more satisfying. Remark that `PoseOptimizationBlock` is not an implementation of the Pose Graph Optimization algorithms, but an implementation of the Motion Only Bundle Adjustment optimization algorithm.

In *Figure* 18 the trajectory estimate when including bundle adjustment is shown.

4. RESULTS

The performance of the pipeline has been recorded on the three datasets in their optimized and not optimized versions. In *Table* 2 the links to the screencasts can be found.

In the following sections a systematic analysis of the results is conducted.

A. Ground-Truth comparisons

A quantitative comparison with the Ground Truth has been con-

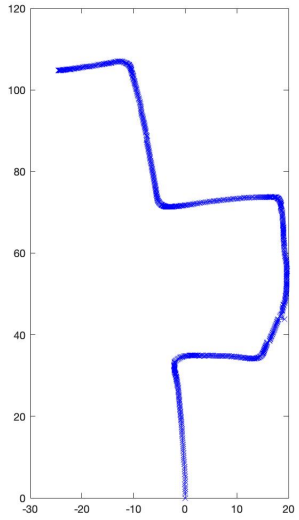

Fig. 18. Trajectory estimate with the optimization block (bundle adjustment).

Error metric	Not optimized	BA
Max median transl	22m	20m
Max median transl %	21%	11%
Max median yaw	7 deg	2 deg

Table 1. Table of comparison with and without Bundle Adjustment in KITTI dataset.

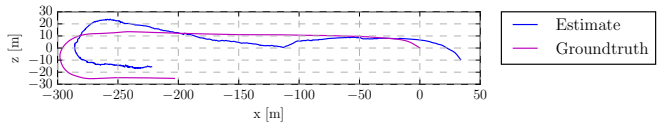

Fig. 19. Trajectory after 1200 frames in Malaga

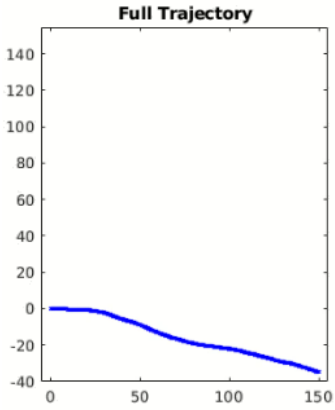

Fig. 20. Trajectory after 570 frames in Parking

Content	Link
KITTI dataset	https://youtu.be/pcvqmT0j50g
KITTI dataset with BA	https://youtu.be/wylit5A9JqQ
Malaga dataset with BA	https://youtu.be/2y6KvV_5wEw
Parking dataset	https://youtu.be/-XjXqDtPnvQ

Table 2. Links to the screencasts.

	Acer Aspire E 15
CPU	2,7 GHz Intel Core i7-7500U
RAM	12 GB DDR4
OS	Ubuntu 18.04
# threads	1

Table 3. Hardware resources.

ducted on the first 700 frames of the KITTI dataset, running both the not-optimized and the optimized version of the pipeline.

The comparison has been eased by the availability of the open-source toolbox of the *Robotics & Perception Group* of the *University of Zurich* [9].

As evident from *Figure* 21, the two trajectories nicely almost overlap.

The estimated trajectory is characterized by a translation error that drops with the distance and whose median is always maintained below the 20%, reaching the best estimate with a translation error of 10%. The yaw error results even more precise: its median always stays below 10%.

The pipeline is shown to suffer of an almost not existent scale drift. In *Figure* 22, it stays really low along the whole considered trajectory.

However, the trajectory did not result truly smooth and the shape was not perfectly overlapping. This was mainly due to high rotation errors, that was solved implementing the Bundle Adjustment optimization.

The analytical results exploiting Bundle Adjustment on the KITTI dataset are illustrated in *Figure* 30.

From *Figure* 26 it is immediate to notice that the quality of the trajectory has been improved by the optimization, both with respect to the angles of the turns, the smoothness of the trajectory and the scale drift.

Figure 28 and *Figure* 29 clearly show the improvement of the estimated trajectory with the `OptimizationBlock`. The maximum median of the yaw error sees a drop to around 2% whereas the worse per cent translation error is around 10%, meaning that the optimization doubles the accuracy of the original pipeline.

5. CONCLUSION

In conclusion, it is important to underline the evolution of the pipeline from the very beginning until the end of the work. A set of independent exercise sessions have been integrated with time in a fully functioning visual odometry monocamera pipeline with a flexible structure and an intuitive configuration file.

The results proved to be satisfactory already in the not optimized version of the pipeline. The bundle adjustment allowed then an even greater accuracy in the estimate, resulting on a

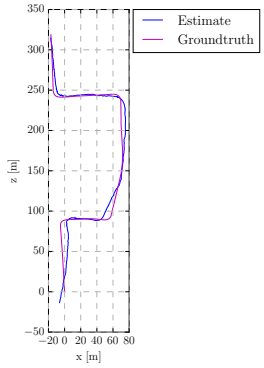

Fig. 21. The estimate of the first 700 frames of the KITTI dataset aligned with the Ground Truth.

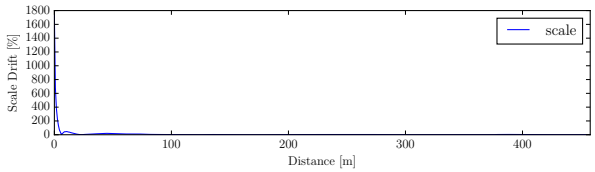

Fig. 22. Scale drift along the trajectory (KITTI) expressed in meters.

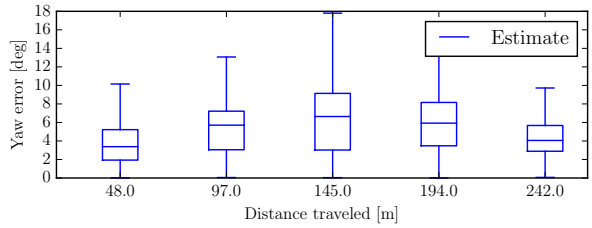

Fig. 23. The yaw error of the estimated trajectory (KITTI) expressed in degrees.

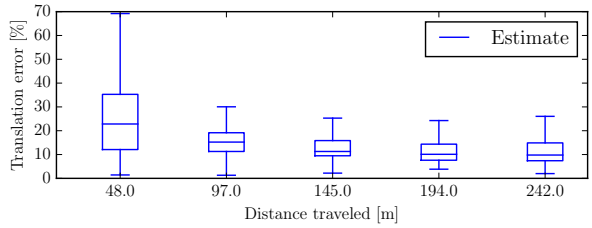

Fig. 24. The per cent translation error of the estimated trajectory (KITTI).

Fig. 25. The results of the analytical comparison between the estimated trajectory, computed without any optimization, and the Ground Truth of the first 700 frames of the KITTI dataset.

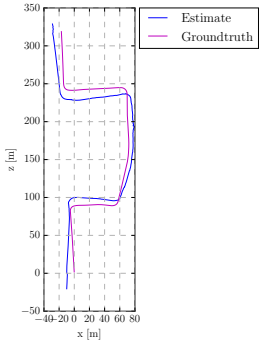

Fig. 26. The estimate trajectory of the first 700 frames of the KITTI dataset aligned with the Ground Truth.

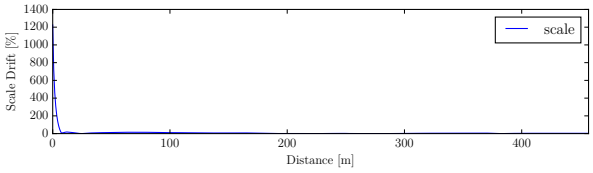

Fig. 27. Scale drift along the trajectory (KITTI) expressed in meters.

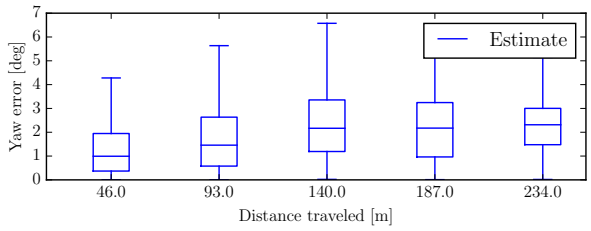

Fig. 28. The yaw error of the estimated trajectory (KITTI) expressed in degrees.

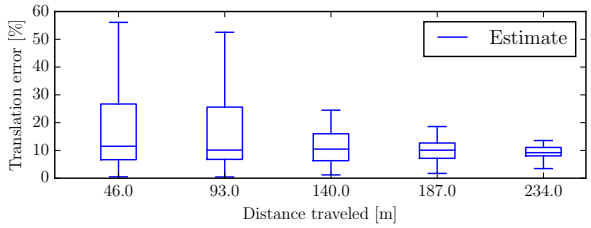

Fig. 29. The per cent translation error of the estimated trajectory (KITTI).

Fig. 30. The results of the analytical comparison between the estimated trajectory, computed with Bundle Adjustment, and the Ground Truth, of the first 700 frames of the KITTI dataset.

globally consistent trajectory even optimizing only on a recent window of frames (last 150 frames every 30 frames in KITTI). In particular, all of the contributions to the algorithms, together with the tuning efforts (especially on the KITTI dataset), resulted in progressively better results.

On the wave of this first successful attempt, the authors want to keep improving the pipeline and port it to `C++`. Numerous improvements are already scheduled as further work:

- try different norms instead of the euclidean in the poses and structure optimization (e.g., Tukey and Huber norms);
- weight different the points based on the depth uncertainty;
- introduce more refined estimation block for triangulating new points, inspired to the SVO pipeline[10];
- try different RANSAC algorithms, introducing the non-holonomic constraints, for faster outlier filtering;
- introduce adaptive minimum and maximum distance for landmarks triangulation, based on the estimated baseline and the relative depth uncertainty;
- try different detectors, and use a scale and rotation invariant Harris detector;
- improve and try different keyframe selection schemes; and
- make it more efficient so that it runs in real time

Hopefully, the proposed pipeline is suitable as a well-grounded structure for a quick-start in Visual Odometry for the community.

REFERENCES

1. L. Kneip, D. Scaramuzza, and R. Siegwart. A novel parametrization of the perspective-three-point problem for a direct computation of absolute camera position and orientation. In *CVPR 2011*, pages 2969–2976, June 2011.
2. Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'81, page 674–679, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc.
3. Simon Baker, Ralph Gross, and Iain Matthews. Lucas-kanade 20 years on: A unifying framework: Part 3. *Int. J. Comput. Vis*, 56, 12 2003.
4. R. I. Hartley. In defense of the eight-point algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(6):580–593, June 1997.
5. Martin A. Fischler and Robert C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, June 1981.
6. D. Nister. An efficient solution to the five-point relative pose problem. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(6):756–770, June 2004.
7. Heng Yang, Pasquale Antonante, Vasileios Tzoumas, and Luca Carlone. Graduated non-convexity for robust spatial perception: From non-minimal solvers to global outlier rejection. *IEEE Robotics and Automation Letters*, 5(2):1127–1134, Apr 2020.
8. Y.I Abdel-Aziz, H.M. Karara, and Michael Hauck. Direct linear transformation from comparator coordinates into object space coordinates in close-range photogrammetry*. *Photogrammetric Engineering Remote Sensing*, 81(2):103 – 107, 2015.
9. Zichao Zhang and Davide Scaramuzza. A tutorial on quantitative trajectory evaluation for visual(-inertial) odometry. In *IEEE/RSJ Int. Conf. Intell. Robot. Syst. (IROS)*, 2018.

[10] https://github.com/uzh-rpg/rpg_svo

Block	Parameter	Explanation
InputBlock	Handler	Name of the selected dataset. Current possible values are *Kitti*, *Malaga*, or *Parking*
	Path	Set the relative path to the datasets, e.g. *data/kitti*, *data/malaga*, or *data/parking*
InitBlock	adaptive	Level of confidence for the adaptive RANSAC algorithm
	samplingSize	Number of [rows, columns] to split up the image for homogeneous detection
	r_T	Radius of the patch considered in the `KLTInitBlock`
	nItKLT	Maximum number of iterations in the `KLTInitBlock`
	lambda	Maximum bidirectional error in the `KLTInitBlock`
	nLevels	Number of layers in the `KLTInitBlock`
COBlock	p3pTolerance	Maximum value for the reprojection error after P3P with RANSAC
	minInliers	Minimum number of landmarks to localize reliably
	nLandmarksReference	Number of desired landmarks to track
	candidateSuppressionRadius	Minimum allowed pixel distance from previously tracked keypoints
OptBlock	isActive	Bool to activate or deactivate the Continuous Bundle Adjustment
	maxBundleSize	Maximum number of recent poses to include in the optimization
	everyNIterations	Number of poses before optimizing the next bundle
	maxIter	Maximum number of iterations allowed for BA convergence
OutputBlock	plotGroundTruth	Bool to activate or deactivate the Ground Truth plot aligned with Full Trajectory
DetectorBlock	Params	Parameters discussed in *Exercise 03*
Pipeline	nSkip	Number of frames skipped for every iteration: $nFrame = 1$ to not skip frames
	startingFrame	Initial frame of the pipeline
	alphaTh	Minumum bearing angle in landmark triangulation
	minDistance	Minimum landmark distance from the camera. Closer landmarks are discarded
	maxDistance	Maximum landmark distance from the camera. Farther landmarks are discarded
	pruneOlderThan	Maximum number of frames any landmark is kept stored in memory
	continuouslyTriangulate	Bool activating or deactivating the landmark triangulation
	lostBelow	Below this number of tracked landmarks the pipeline re-initialize

Table 4. Most relevant parameters in the config.json file