VAMR Mini Project: VO Pipeline

Autumn 2024

Landson Guo 24-941-767
Cedric Landsmeer 24-953-390
Luca Zwahlen 20-719-894
Seyyid Ökkes Palta 20-742-524

Aim statement and approach

This project aimed at designing and implementing a monocular visual odometry (VO) pipeline with a specific focus on the camera motion of wheeled platforms (in this case cars). The camera motion is reconstructed for several datasets and the achieved trajectory of camera poses is compared to the ground truth. The primary aim has been to achieve local accuracy of the trajectories over large distances without implementing place recognition and full visual SLAM. To the extent possible, we try to achieve globally consistent trajectories as one of the pipeline's features. The computational efficiency is considered a secondary objective in this project.

Bonus Features

Given these goals, the following bonus-features were implemented:

- *Quantitative error analysis*
  To evaluate the accuracy, two different, complementary error measures are computed: absolute trajectory error (ATE) and relative trajectory error (RTE). Where ground truth pose information is available at sufficient quality (particularly for *Kitti* and *Parking*), the RTE can be used to gauge local accuracy. It allows the visualisation of representative summary statistics and the trajectory scale drift. The ATE, on the other hand, can be used to evaluate the global accuracy of the trajectory. Both quantities correct for the quantities that are unobservable during monocular VO: absolute scale, absolute rotation and absolute position. [1] was used as a source, but the algorithms were implemented from scratch.

- *Additional dataset*
  To achieve better accuracy and speed, the use of a non-holonomic constraint on the camera motion was evaluated [2]. However, this constraint does not hold precisely unless the camera is positioned above the rear axis of the vehicle. In all three provided datasets (*Kitti, Malaga*, and *Parking*), the camera is positioned at some distance from the rear wheels. To evaluate how this constraint can be used to improve the performance of the pipeline, a new dataset was recorded where this condition is fulfilled. The camera was placed in the boot, precisely above the rear axle, and faces backwards while the car is moving.

- *Pose-graph optimization*
  Seeing that the aim was set to provide decent estimation quality at an acceptable speed, the use of sliding-window pose-graph optimization was evaluated. The feature can be activated and deactivated at will.

Development

The pipeline was developed unit by unit, starting from the bootstrapping module all the way to the additional features working on the output, such as the error measures and pose-graph optimisation. Dividing both the code and the development of the pipeline into functional units as described below helped in tracking problems in the code, visualising intermediate results and experimenting with different approaches.

Structure of the code

The core VO pipeline is split across two classes: **`Bootstrap`** (located in `intialise_vo.py`) and **`VO`** (located in `cont_vo.py`), which handle the bootstrapping phase and continuous VO phase individually. We have a **`Dataloader`** object which handles the loading of each dataset.

Bootstrap

This class takes the initialization frames for each dataset and performs *Structure from Motion* to get an initial estimate of the transformation between those frames. Then it performs triangulation to initialize the 3D landmarks, and catalogs the data necessary to initialize the state in the continuous VO step.

VO

This class handles all processing after the bootstrap step. The primary function of interest is `process_frame`, which takes in the next frame in a sequence of images and outputs the pose for that frame (along with possibly some optimized poses for previous frames), as well as members of the internal state for visualisation purposes.

Following the suggestion from the mini-project outline, the VO object houses an internal markovian state as follows:

- `P_i`: 2D pixel locations of each landmark that we are using for localisation
- `X_i`: 3D world locations of each landmark that we are using for localisation
- `C_i`: The set of candidate keypoints we are tracking that may be added as landmarks later
- `F_i`: The 2D pixel location of the candidate keypoints we're tracking, in the first frame in which they were detected
- `T_i`: The set of transforms from world to the camera in which they were first detected of the candidate keypoints we are tracking

process_frame

The `process_frame` function performs 3D-2D correspondence in order to extract pose estimates. As laid out in the code it follows a 9 step process:

1. Perform KLT on the newly added frame to get keypoint-landmark correspondences

2. Perform PnPRansac to obtain a pose estimate for the new frame
3. Run KLT on possible candidate keypoints to determine if they are still trackable
4. For each tracked keypoint, calculate an angle between the initial sighting and the most recent sighting
5. Convert all candidate keypoints to landmarks if they reach the angle threshold
6. Update all markovian state elements with the new values
7. Run feature detection if we don't have enough candidate keypoints
8. Run pose-graph optimisation
9. Return pose and state values to be visualised

The overall flow remains as described in the initial problem statement, but many optimisations and additional features were added to each of these steps, as is discussed in the next section.

Challenges and Solutions

Angle Constraint (Step 4 of process_frame)

One of the big challenges we were running into was an inability to get enough points to fulfill the angle constraint.

Generally speaking, as mentioned in lecture, a good rule of thumb is to perform triangulation when the baseline is 10% of the depth to the keypoint, which would correspond to an angle threshold of about 0.1 radians. However, in testing we were only able to achieve about a tenth of that, or 0.01 radians, as otherwise we simply wouldn't be able to get enough landmarks to perform proper tracking.

We ultimately were unable to determine the root cause of this issue. We implemented 4 different angle calculation strategies and compared their results to confirm correctness in the angle calculation formula but the results were always the same.

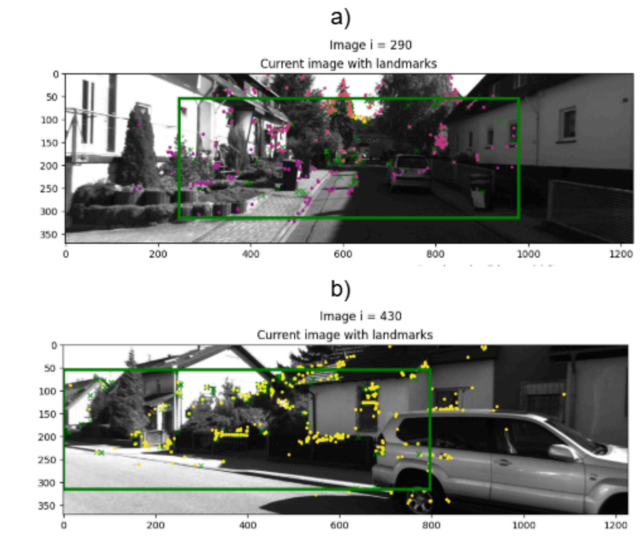

Figure 1: "Sampling rectangle" strategy for preventing feature loss. a) Sampling candidate keypoints from the centre while going straight. Candidates are shown as green X'es b) "Turning mode"

We tried addressing this by implementing a feature that would detect turns and proactively sample keypoints towards the focus of expansion of the image, such that the candidate points would remain in frame for as long as possible. This was implemented by imposing that new candidate features can only be detected in a "*Sampling rectangle*" that shifts from the centre of the image to either of the edges when a turn is detected. Turn detection turned out to work very reliably for the sharp corners of the

Kitti dataset: when a threshold on the estimated frame-to-frame rotation vector was passed for two consecutive frames, “turning mode” was activated.
However, in the end, we discovered that it was simply more effective to greatly increase the number of keypoints universally across the image. The added computation cost was not detrimental and scale drift was significantly reduced. Apparently, using more features increased the chances that a sufficient number of landmarks remained available for PnP, while the lower quality of depth triangulation was compensated by the sheer number of features.

Adding New Candidate Keypoints (Step 5 of process frame)

Another issue we were running into was insufficient keypoint detection for corners. Generally speaking the algorithm would perform well on the straightaways, but perform poorly as soon as a corner was reached, as we would quickly lose too many keypoints to continue proper tracking.

We noticed that a major issue for these turns was that the keypoints would end up being concentrated on certain high feature areas such as trees, so that while the total number of keypoints would seem high from the algorithm's point of view, as soon as that tree disappeared the algorithms performance would severely degrade.

To combat this we implemented block feature detection, splitting the frame into multiple “blocks” and running feature detection on each block separately, to ensure that there was a sufficient number of keypoints across the entirety of the image instead.

In addition, we allowed for a combination of both SIFT and Shi-Tomasi features when doing keypoint detection. Since KLT and Shi-Tomasi base their detection on the same concept (directions of high image gradient) we found Shi-Tomasi features to be generally better at being tracked for long durations. However in certain cases such as the Malaga dataset, the Shi-Tomasi would pick up far more invalid keypoints in the sky and so SIFT would result in far more accurate results.

Optimisation for Scale Drift (Step 8 of process frame)

Scale drift is another challenge we encountered much of, where the scale of the transformations and the corresponding landmarks would gradually shift throughout the trajectory of the camera.

Among the other enhancements above, we decided to implement Sliding Window Pose-Graph Optimization as a slightly simpler and more performant version of bundle adjustment. Our base pipeline is single-threaded and the Markovian structure means we don’t keep information about which landmarks were visible in past frames, so implementing full bundle adjustment would have required a significant refactoring. In this way, we were still able to perform some non-linear refinement with minimal modifications to the front end structure, and even though the optimization step was performed synchronuously, the performance is still reasonable.

Even without full bundle adjustment, Pose Graph Optimization proved to be effective at combatting scale drift in the output trajectory, able to reduce the Absolute Trajectory Error by over 50% in some cases.

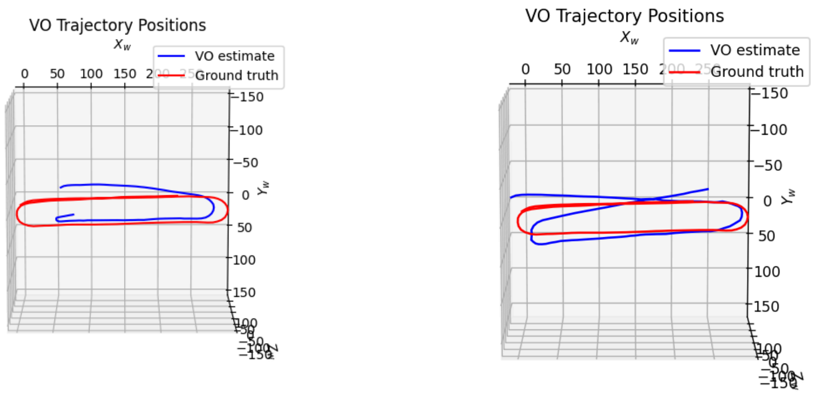

Figure 2: Before (left) and After (right) applying Pose Graph Optimization
Absolute Trajectory Error (RMSE): 50.064m on left, 22.379m on right

The version we implemented used a three frame sliding window, and would use keypoint-based DLT to compute additional transformations for each possible pair of image in the sliding window, and then optimize those transformations with the result from the main pipeline.

Optimisation for Performance (One Point RANSAC)

We attempted to optimize the runtime of the algorithm by implementing One Point RANSAC, in order to reduce the number of iterations required to perform PnP. This was motivated by the fact that RANSAC requires substantially fewer iterations to complete if you reduce the number of points it needs to select. For example, given a desired probability of success of 99% and an inlier ratio of 50%, RANSAC will require about 1177 iterations for the 8-point algorithm, 145 iterations for the 5-point algorithm, and just 7 for the 1-point solution.

The one major constraint with One-Point RANSAC is that it requires non-holonomic planar motion. Luckily, the datasets we designed this algorithm for (Parking, Kitti, Malaga) all involve non-holonomic planar movement and so we can exploit the motion constraints to optimize the RANSAC algorithm. In addition, for the datasets we recorded (“Own_1” and “Own_2”), we explicitly decided to place the camera directly on top of the rear axle in order to meet the assumptions for one point RANSAC (this is why the car is driving backwards in these datasets).

We decided to select the inliers according to the maximum bin, not the reprojection error. This left the implementation too imprecise to do a single PNP step. We resorted to still doing PNP with RANSAC on the points that were not filtered out by 1-Point RANSAC. The computation time of 1-Point RANSAC is roughly equalized by the lower computation time of a PNP RANSAC iteration, due to having fewer points (note that the iteration count does not change). Turning on 1-Point RANSAC can result in sharper, smoother corners but it tends to suddenly fail when parameters are not tuned enough.

Results and Visualization

We ran our VO pipeline over five datasets. *Parking*, *Kitti*, as well as *Malaga* are the unmodified versions of the respective datasets linked to this project.
We extended those with “Own_1” and “Own_2”, a dataset that was recorded out of the boot of a car with a CANON EOS R7 mounted above the rear axis. The video was shot in 4k with no software correction or stabilization on an early winter morning and includes two sharp turns at approximately 15 km/h. The dataset was recorded at Alpenstrasse, Kriens - Luzern. The path recorded can be seen in figure 3. The video was downsampled to five frames per second and a resolution of 1600 by 900. The camera was calibrated with the opencv provided checkerboard calibration, the images were then undistorted accordingly.

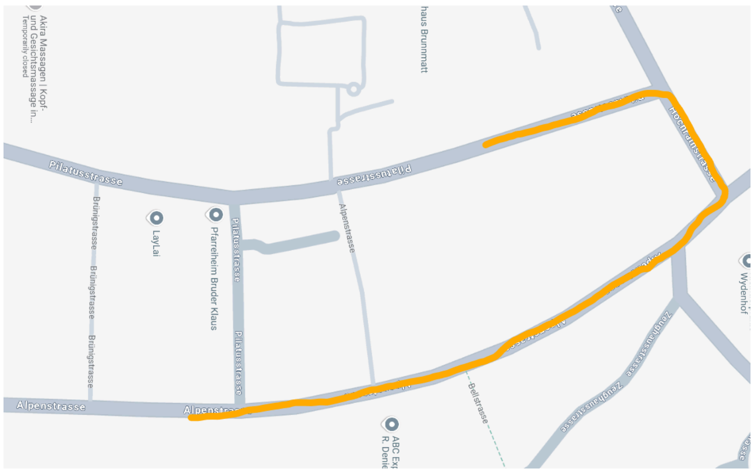

Figure 3: The Geographic location of the self-recorded dataset.

Different datasets work best with different configurations. While the default works fine for all, certain parameters help some datasets and hurt others. We include a high default feature count for preventing significant scale drift, note that the rotation and position tracking (up to consistent scale) require only a fraction of the feature count. For more details on features and hyperparameters, consult the code submitted with this report.

We provide eight runs of our VO pipeline, shown in the visualization video (https://www.youtube.com/watch?v=hEA7zJPG3Gs).

- Run 1: *Parking*. Except for some rotation inaccuracy at the very start, the method stabilizes well and runs in a straight line. The relative trajectory error is low and the same holds for the absolute trajectory error of RMSE = 2.58m.
- Run 2: *Kitti*: Here we can see some scale drift adding up, but the relative error of small segments is still good except for a few outliers. The absolute error suffers through the scale drift and is RMSE = 22.2m
- Run 3: *Malaga*: Due to the use of SIFT features as explained earlier in the report, the run takes longer and only manages about ten frames per second. The scale drift is marginal and the absolute error is RMSE = 21.24m. No relative error could be provided because of missing ground truth rotation matrices.
- Run 4: *Own 1*: The trajectory overlaps well with the road structure, though no ground truth data can be provided.

- Run 5: *Own 2:* The trajectory overlaps well with the road structure. Corners seem a bit sharper, which can be partially attributed to using 1-Point Ransac to filter out extreme outliers before continuing with the pipeline. No ground truth data can be provided.
- Run 6: *Kitti* with less points: this is to demonstrate that the feature count being high helps scale drift. The corners get still tracked well, though due to the drift, the absolute error is high at RMSE = 67.9m.
- Run 7: *Kitti* with optimization: This is to demonstrate that pose graph optimization helps combat the scale drift, though our implementation is slow and introduces artifacts at the corners. The RMSE is lower at 25.38m, still with less features than the default.
- Run 8: *Malaga* with optimization: pose graph optimization again helps a little with scale drift, but introduces corner artifacts in the positions. The absolute error is at RMSE = 21.03m.

The evaluation and visualization was run on a Ryzen 9 7950x 16/32 cores/threads @ 5.2Ghz. Only the python threading tool was used to create a single thread for running. Since this python thread can span over multiple cores, we cannot definitively say how many CPU threads were used, though the overall usage was around 30%. The Computer used had 64GB of 6000MHz dual channel memory, about 1GB of which was used by our VO pipeline.

Error Measures

The main challenge during the implementation of the error measures was the inhomogeneous ground truth data for the different datasets or the lack thereof. For *Kitti* and *Parking*, direct ground truth poses are provided. However, for the *Malaga* dataset, they have to be extracted by combining the provided sensor data.

The ground truth positions can be found using the GPS signal, be it at a much lower temporal resolution than for the other datasets. A big challenge, however, is that the GPS file does not provide useful orientation information and that also the inertial estimates in the IMU data are much more noisy than the orientation ground truth for *Kitti* and *Parking*. An attempt at recovering an approximate ground truth orientation was made by using multiple GPS positions and assuming that the orientation of the z-axis is in the direction of motion. The results are still not as informative as for the *Kitti* and *Parking* datasets. Since the orientation ground truth is only relevant in the implementation of the relative error, the RMSE of the ATE can be computed without any problems.

The self-produced dataset does not, unfortunately, have a ground-truth GPS track. This is mainly due to the difficulty of accurate temporal alignment between the frames and the GPS signal.

Despite these restrictions, the error provided useful indications for what pipeline parameters and configurations are the most promising. Particularly the ability of the relative error measure and the associated visualisation to correct for scale drift proved to be very informative. This correction of scale drift is possible because each sub-trajectory is aligned to the corresponding sub-trajectory of the ground truth independently of the others; so the varying alignment transform parameters allow an observation of how drift evolves throughout the trajectory.

External Code

ChatGPT was used mainly for help with matplotlib. The visualization code was often inspired by a ChatGPT solution, though manual changes needed to be applied often. There is a function in the utils.py file called ‘linear_LS_triangulation’ which does Least Squares approximation for point triangulation. The code is copied from a GitHub repository (https://github.com/Eliasvan/Multiple-Quadrotor-SLAM/blob/master/Work/python_libs/triangulation.py). The function takes the same input and returns the same output as the cv2 equivalent, though the computation is more precise.

Use of the OpenCV tutorials [3, 4], was used for incorporating the cv2 implementation of SIFT and KLT within our pipeline. Code from the Kitti dataset page [5] was used for extraction of the K matrix.

Conclusions

The project has allowed us to put together elements from various parts of the course. The pipeline is the result of an iterative design process informed by theoretical considerations on the one hand and trial and error on the other. In this scope, we could not hope to design a truly optimal pipeline and hence many ideas for possible improvements remain:

With regard to achieving global accuracy of the trajectory, implementing place recognition and loop closure are very promising directions. Also, a further investigation of the behaviour of the angle constraint may be very informative. The constraint proved to be effective at rejecting candidate features without motion parallax (on clouds for example), but failed to adhere to our quantitative predictions. Finding efficient ways of filtering for high-quality landmarks may have allowed for an alternative solution relying fewer features and thus with a slightly lower computational cost. Relatedly: parallelising mapping and localisation in separate threads, as is done by many real-time VO solutions, may provide a further boost in performance. Finally, implementing full sliding window bundle adjustment in addition to the pose-graph optimisation should provide further boosts to global consistency in the output trajectory.

Author Contributions

Landson	Base implementation of bootstrapping and continuous vo (running SIFT, getting bootstrap estimate, running KLT, initial angle calculation, updating state, initial new feature extraction), pose-graph optimisation
Luca	1-Point RANSAC, Own Dataset, Parallel angle calculation, threading, new feature extraction, parameter tuning, miscellaneous changes and bug fixes.
Seyyid	One of the angle calculation techniques, parameter tuning, miscellaneous changes and bug fixes.
Cedric	Error measures, data loader, parts of bootstrapping, parts of visualisation

References

[1] Zhang, Scaramuzza, *A Tutorial on Quantitative Trajectory Evaluation for Visual(-Inertial) Odometry*, IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2018.

[2] Scaramuzza, *1-Point-RANSAC Structure from Motion for Vehicle-Mounted Cameras by Exploiting Non-Holonomic Constraints*, International Journal of Computer Vision, 2011

[3] “OpenCV: Feature Matching,” docs.opencv.org. https://docs.opencv.org/4.x/dc/dc3/tutorial_py_matcher.html

[4] “OpenCV: Optical Flow,” docs.opencv.org. https://docs.opencv.org/3.4/d4/dee/tutorial_optical_flow.html

[5] Arman Asgharpoor Golroudbari, “KITTI Dataset,” Apr. 2023. https://armanasq.github.io/datsets/kitti/#intrinsic-matrix