Vision Algorithms for Autonomous Mobile Robotics Mini-Project

Katharine Patterson, Heba Hussein,
Tasman de Pury, Ádám Weinhardt-Kovács

Autumn 2023

Contents

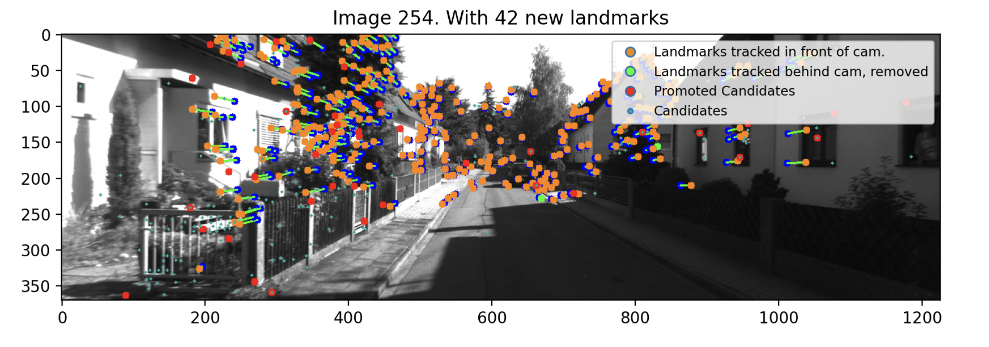

Figure 1: Example Screenshot of our detected landmarks

1 Introduction

Our group implemented a standard VO pipeline without the 3 additional features. Our pipeline has:

- An initialization phase, where we detect and triangulate an initial set of 2D $\rightarrow$ 3D landmark correspondences.
- An ongoing VO phase, where for each new frame we:
  - Track existing 2D keypoints from the prior frame to the new frame, and associate the new keypoints to the existing 3D points.
  - Estimate the new pose of the camera.
  - Identify promising new keypoints and promote good ones to new landmarks.

We followed the suggested implementation structure in the project guidelines and describe the details of our design below.

2 Implementation Details

2.1 Initialisation

We selected our two initialization frames with a parameter `BOOTSTRAP_FRAMES = [0, 3]` for the KITTI and Malaga datasets, which worked well set to the first and fourth image. We used the first and the eighth frames for the Parking dataset, because during the sideways movement the overall angle between consecutive frames are small, thus we need more time to establish keypoint correspondences.

1. *Keypoint Detection:* The pipeline initiates by detecting keypoints in the two initial images using OpenCV's detector: `cv2.goodFeaturesToTrack()` with the Harris Corner setting. The Harris detector parameters, such as max corners, quality level, min distance, and block size, are modifiable through the `params` module.

2. *Keypoint Description and Matching:* A patch of intensity values around the corner is extracted as the descriptor. The descriptors are then matched using a Brute Force Matcher (`cv2.BFMatcher.create`), which compares the descriptors from both sets and finds the best matches, using the `crossCheck=True` flag to make sure best matches are mutual between the images.

3. *Fundamental and Essential Matrix Estimation:* With matched keypoints, the pipeline computes the fundamental matrix using `cv2.findFundamentalMat()` with RANSAC, and then calculates the essential matrix as:

$$E = K^T \times F \times K$$

4. *Pose and Landmarks Initialization:* The `cv2.decomposeEssentialMat()` function decomposes the essential matrix into possible rotation and translation matrices. The correct R and t is found among four possible configurations by returning the one that yields maximum points lying in front of the image plane (with positive depth), and is saved as the homogeneous matrix T. With the relative pose determined, `cv2.triangulatePoints` is used to compute the 3D coordinates of the matched keypoints, thereby initializing the landmarks.

The 3D landmarks are stored in the state variable `state.X` and the 2D keypoints in `state.P` for use by the rest of the pipeline.

2.2 Continous operation

Each section of the Continuous Operation phase takes in our `State` object and updates its contents.

$$S^i = (P^i, X^i, C^i, F^i, T^i)$$

2.2.1 Associating keypoints to existing landmarks

- *KLT Tracking:* The pipeline uses the OpenCV Kanade-Lucas-Tomasi (KLT) function `cv2.calcOpticalFlowPyrLK` to track the keypoints from the previous image `state.P` in the new image. Parameters such as window size, max level, and criteria for termination of the algorithm are customizable via the params module.

- *Outlier Rejection:* Post tracking, it uses `cv2.findFundamentalMat` on the previous and tracked keypoints with the RANSAC flag to get an inlier mask for outlier rejection. Additional logic rejects outliers based on the distance moved in 2D, which refines the set of tracked keypoints further by removing points that have 'jumped' unrealistically:

$$\texttt{dists} = \|P_{new} - P_{old}\|_2$$
$$\texttt{outliers} = |\texttt{dists} - \texttt{mean(dists)}| > \texttt{THRESHOLD} \times \texttt{stddev(dists)}$$

- *State Update:* The function concludes by updating the state with the newly tracked and filtered keypoints, and removing any 3D landmarks from `state.X` that have had their corresponding 2D keypoint filtered.

2.2.2 Estimating the current pose

- *PnP and RANSAC*: The 2D keypoints obtained from previous states and the corresponding 3D landmarks are used as inputs for the OpenCV function `cv2.solvePnPRansac`. This function applies PnP with RANSAC to estimate the rotation R, translation T, and the set of inlier correspondences. As a parameter we can choose between the DLT and P3P solver for the PnP problem, but the P3P made better results.

  To balance accuracy and computation time we used these RANSAC params: *No. of Iterations* = **2000**, *Reprojection Error* = **10**, *Confidence* = **0.999**

- *Optional DLT refinement.* Post RANSAC, the algorithm separates the inliers and outliers among the keypoints and landmarks.

  Optionally, the inliers obtained from the PnP RANSAC process can be further refined using DLT. However, our pipeline works fine without it so we don't activate it by default to save on computation.

- *Outlier rejection.* The function then transforms the 3D points X in the world frame to the newly calculated camera frame P_C:

  $$P_C = K \times [R, T] \times X$$

  and removes any that are behind the camera.

- *State Update.* Finally, the state object is updated with the filtered keypoints (P) and landmarks (X) and the current pose is returned.

2.2.3 Triangulating new landmarks

- *Detecting New Candidate Keypoints:* The pipeline uses a high-contrast version of the new image frame (`cv2.equalizeHist`) and SIFT (`cv2.SIFT`) to identify uniformly distributed potential keypoints. It compares these new keypoints against existing ones in the state (`state.P`) and filters out any that are too close, using Manhattan distance
  (`scipy.spatial.distance.cdist()`).

- *Updating Existing Candidate Keypoints:* We update the set of existing candidate keypoints (`state.C`) using KLT (`cv2.calcOpticalFlowPyrLK()`) for tracking. It then applies RANSAC (`cv2.findFundamentalMat()` with RANSAC) to further refine the tracking, identifying and retaining only those keypoints that are consistent between consecutive frames.

- *Merging Candidates:* New candidate keypoints detected in the current frame are deduplicated with the successfully tracked candidates from prior frames and then merged with the existing tracked candidates.

- *Triangulating New Landmarks:* If more landmarks are needed (We empirically found 500 landmarks was sufficient.) keypoints that are successfully tracked and are determined to have moved sufficiently between frames are triangulated to form new 3D landmarks using `cv2.triangulatePoints()` using the current camera pose and the pose when the keypoint was first observed. Out of the max 1000 candidates we track, keypoints are prioritized for triangulation based on their angular distance moved and proximity to existing landmarks. This ensures a well-distributed set of landmarks across the scene with larger baselines for more accurate depth estimation.

- *Outlier Rejection:* We filter out points triangulated behind the camera, as well as any triangulated to a 3D point more than 3 standard deviations away from the mean location of the tracked 3D points.
- *Updating State:* The state is updated with the newly triangulated landmarks. Candidate keypoints that were not promoted to landmarks are retained for potential triangulation in subsequent frames.

3 Work Process and Task Distribution

3.1 Task Distribution

Each one of us drafted one section, Tasman: Initialization, Katharine: Tracking Keypoints, Ádám: Pose Estimation, Heba: Candidate tracking and promotion. Tasman and Katharine finished earlier and so Tasman switched to also implementing the candidate tracking section (which was more challenging than the other sections), and Katharine switched to visualization and debugging of the overall pipeline.

3.2 Development Approach

We each wrote unit tests for our own sections in isolation using 'proven' datasets from class exercises where possible. We set up centralized testing and visualization helper classes so it was easy to test with different datasets just by modifying parameters.

3.3 Visualization and Debugging

We added options to visualize the global point cloud, colour-coded keypoint and candidate tracking, and the calculated trajectory of the camera. We also maintained a central parameters module where everyone's flags and options were set, making it easier to tweak performance and turn on and off all debug print statements.

4 Problems and Solutions

4.1 Outliers

Outliers were the major problem we faced. All our pipeline components worked fine in isolation when we tested them with clean data from previous class exercises that had no erroneous keypoint matches, but when we ran our pipeline in its entirety on the Kitti, Malaga and Parking datasets, our pose estimation would break entirely within 10 frames.

4.2 Poor keypoint detection/matching

As mentioned above the pipeline would break after 10 frames which we narrowed down to an issue with the newly triangulated landmarks once the initial landmarks had moved out of frame. We found the following issues:

- The detected keypoints were poorly distributed, especially in areas of low light and texture.
- When matching using patch descriptors there were erroneous matches, even after increasing the matching threshold. These poor matches were affecting the camera pose estimation and these errors would then propagate quickly throughout the pipeline.

4.2.1 Solution

The following changes were made:

- The *Initialisation* module was tested independently against known landmark-keypoint correspondences to eliminate this from the issue. The triangulated landmarks had a very high correlation with the ground truth data, confirming our hypothesis about the *Triangulating new landmarks* module.
- In the *Triangulating new landmarks* module: After testing different detectors/descriptor pairs, the best detector (as measured by number of frames of successful running) was found to be AKAZE. Unfortunately this was not available due to library restrictions so SIFT was chosen.
- Changed to KLT tracking instead of descriptor matching.
- Added RANSAC filtering using a `cv2.findFundamentalMat()` model to eliminiate outliers after tracking.

4.3 Corners and Clumping

We then had a pipeline that would work for a few dozen frames, but broke eventually or else broke immediately when the vehicle turned a corner.

- We observed that during turns, new candidates were slow to be promoted and existing ones would, by definition, drift to one side of the frame. At the time we were only prioritizing promoting candidates with a large baseline pose translation, but we observed that in corners there is more rotation than translation, even if the keypoint appears to move far in 2D.
- We also observed that candidates were more likely to be located in clumped areas in the image, such as leafy trees, where there were many high-contrast corner features, which then meant that we were more likely to promote candidates in spatial clumps.

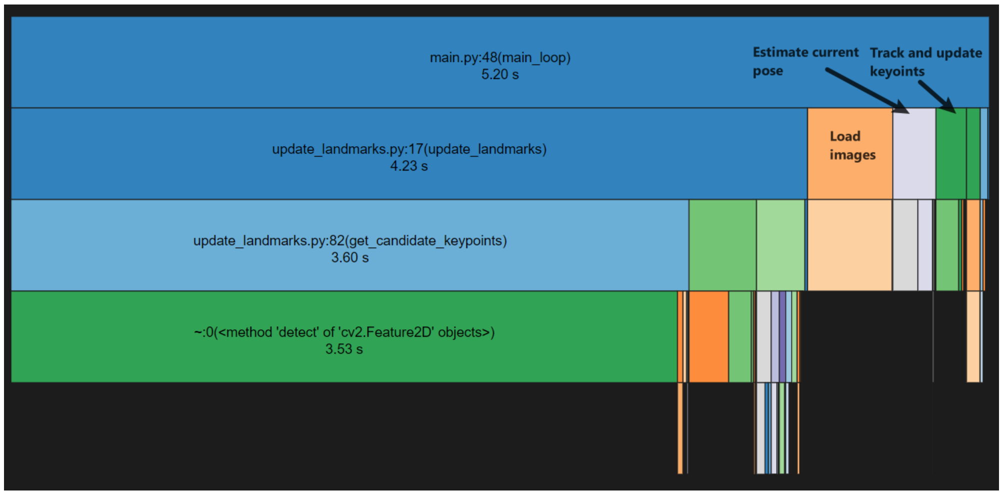

Figure 2: Example screenshot of our final profiler breakdown. Note that the bulk of the runtime is detecting new keypoints using cv2 functions.

4.3.1 Solution

These observations lead us to:

- Equalize the image histogram using `cv2.equalizeHist` for higher contrast to detect candidates more evenly.
- Give higher promotion priority to candidates that were located further away from existing landmarks.
- Implement grid-based filtering, to limit the number of SIFT features in each region of the image, to better distribute new candidates.

4.4 Efficiency

Once the pipeline was running for more frames at once we observed the time to process each frame was far too high, < 1 FPS. Upon further inspection with a profiler (`CProfile`), the *Triangulating new landmarks* module of the code was taking up 95% of the runtime. By vectorising the triangulation and filtering of the candidates we increased the frame rate to 3 FPS in the final screencasts, including the visualisation. Without the visualisation running we achieved a frame rate of 15 FPS. See Figure 2 for an example of the final profiler breakdown.

4.5 Statistical Outliers

Finally, we also observed that sometimes despite our efforts, landmarks would get triangulated in obviously erroneous places. Specifcally behind the camera (clearly impossible) or as extreme outliers from the rest of the 3D point cloud.

4.5.1 Solution

To address this we simply removed any landmarks that were behind the camera, and did a basic filtering on our landmarks to remove any that had a location greater than x standard deviations from the mean point location, where x was a parameter we set and tuned to 3.

4.6 Success!

After these changes, the pipeline started working both around corners and indefinitely in time, as we always kept good candidate quality and coverage.

5 Screencasts and Links

- Github (with README setup instructions)
- Screencasts recorded with Apple Macbook, M1 chip, 64GB RAM. 3.2GHz CPU Freq.
  - KITTI Dataset Screencast
    In this dataset there are a number of other vehicles moving, even when the camera is stationary. We were initially concerned that this would result in artificial movement of the camera being predicted but very few landmarks were tracked on these cars, meaning the pipeline is filtering outliers effectively. This is clearly shown at 12:40 and 13:30.
  - Malaga Dataset Screencast
    Drift is quite an issue with our pipeline and clearly shown in this dataset when the camera returns to the start point. But locally the trajectory seems to be accurate.
  - Parking Dataset Screencast
    In this dataset the pipeline promotes many landmarks on the ceiling of the garage despite there being many clear edges and corners in other parts such as on the cars or the columns. It might be that we used a relatively bigger angle to promote candidate keypoints, thus the candidate keypoints in the cars which were in the scene for a couple of frames, but we did not investigate this further.

Figure 3: Cooling measures

6 Appendix: Code Structure

```
VAMR_VO
├── Reference_material
│   └── main.m
├── visual_odometry
│   ├── bootstrapping
│   │   ├── initialization.py
│   │   └── test_initialization.py
│   ├── continous_vo
│   │   ├── associate_keypoints_to_existing_landmarks.py
│   │   ├── estimating_current_pose.py
│   │   ├── test_associate_keypoints_to_existing_landmarks.py
│   │   ├── test_estimating_current_pose.py
│   │   ├── test_update_landmarks.py
│   │   └── update_landmarks.py
│   ├── params
│   │   └── params.py
│   ├── utils
│   │   ├── image_processing.py
│   │   ├── state.py
│   │   ├── utils.py
│   │   └── visualisation.py
│   └── main.py
└── environment.yaml
```