VISION ALGORITHMS FOR MOBILE ROBOTICS

PROJECT REPORT 2024

Author 1: Clara Seugé

Author 2: Fabio Hübel

Author 3: Anton Pollak

Author 4: Jakob Wolfram

Zurich, 2024

1 Introduction

Estimating a robot's pose, known as the localization problem, is critical for navigation and is often addressed through odometry. Traditional methods using rotary encoders struggle with non-wheeled systems and are prone to errors from wheel slippage. Visual odometry (VO) provides a more reliable alternative, particularly in feature-rich environments.

This report provides a brief overview of a visual odometery pipeline for monocular cameras. The provided algorithm is implemented in Python and publicly available datasets were used to test our code.

2 Visual-Odometry Pipeline

This section provides an overview of the implemented visual odometry pipeline and the underlying theory. First, an initial set of 2D $\leftrightarrow$ 3D correspondences is extracted from the first frames of each dataset. Then, the camera pose is estimated from the subsequent frames. Whenever possible, available `OpenCV` implementations of the algorithms are used for maximum efficiency.

2.1 Bootstrapping

Author Contributions

- Clara Seugé
- Fabio Hübel

2.1.1 Feature Detection

Feature detection is a fundamental process that identifies distinctive points such as corners, edges, or blobs. We opted for the Shi-Tomasi algorithm implemented in `cv2.goodFeaturesToTrack` to identify the most prominent corner points.
This method computes the gradients of the image I using Sobel filters and then calculates the Sum of Squared Differences (SSD) between the original patch $I(x, y)$ and the shifted patch $I(x + \Delta x, y + \Delta y)$ as follows:

$$\begin{aligned}
&\mathrm{SSD}(\Delta x, \Delta y) = \\
&= \sum_{(x,y)\in P} \left(I(x,y) - I(x+\Delta x, y+\Delta y)\right)^2 \approx \\
&\approx \sum_{(x,y)\in P} \left(I_x(x,y)\Delta x + I_y(x,y)\Delta y\right)^2 = \\
&= \begin{bmatrix}\Delta x & \Delta y\end{bmatrix} \mathbf{M} \begin{bmatrix}\Delta x \\ \Delta y\end{bmatrix}
\end{aligned}$$

with

$$\mathbf{M} = \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} = \mathbf{R^{-1}} \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \mathbf{R}$$

A corner is accepted if and only if the following holds:

$$\min(\lambda_1, \lambda_2) > \text{threshold}$$

where λ_1, λ_2 are the eigenvalues of the second moment matrix $\mathbf{M}$.

2.1.2 Descriptor Computation

We use a SIFT descriptor to describe the previously detected keypoints since it is invariant to scale, rotation and partially robust to illumination changes. Hence, we use `cv2.SIFT_create` to initialize the SIFT algorithm object and `sift.compute` to compute feature descriptors for the specified keypoints in an image.

2.1.3 Feature Matching

The descriptors characterize the region around a keypoint and are used for matching features between two images. We use a brute-force approach that computes the Euclidean distance between every descriptor in image 1 and image 2 with `cv2.BFMatcher`. We enforce mutual consistency by setting `crossCheck=True` which means that a match is only valid if the nearest neighbor in descriptor 1 is also the nearest neighbor in descriptor 2.

Finally, we extract the locations of the matched keypoints in the two images and convert them into a numpy array for downstream tasks.

2.1.4 Essential Matrix Formulation

The goal of this step is to compute the initial set of 3D landmark positions. Both the camera matrix $\mathbf{K}$ and the keypoints $[u_i, v_i],\ i \in \{1, 2\}$ are known.

For the first frame, the perspective projection is as follows:

$$\lambda_1 \begin{bmatrix} u_1^i \\ v_1^i \\ 1 \end{bmatrix} = \lambda_1 \vec{p_1}^{\,i} = \mathbf{K} \left[\mathbf{I}|\vec{0}\right] \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

For the second frame, it holds:

$$\lambda_2 \begin{bmatrix} u_2^i \\ v_2^i \\ 1 \end{bmatrix} = \lambda_2 \vec{p_2}^{\,i} = \mathbf{K} \left[\mathbf{R}|\vec{T}\right] \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

By using epipolar geometry, we can derive the epipolar constraint equation:

$$\vec{p_2}^{\,T} \mathbf{E} \vec{p_1} = 0$$

with the essential matrix $\mathbf{E}$ defined as:

$$\mathbf{E} = [\vec{T}]_\times \mathbf{R}$$

This is done with `cv2.findEssentialMat` which computes $\mathbf{E}$ using the matched 2D points and $\mathbf{K}$. The second ouput of this function is a mask array indicating which points are inliers based on RANSAC. We used RANSAC to best estimate and reject outliers with a probability of 0.999 and a threshold of 1.0.

Finally, to recover $\mathbf{R}$ and $\vec{T}$ we can decompose $\mathbf{E}$ using Singular Value Decomposition (SVD) as follows:

$$\mathbf{E} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{T}}$$

We use `cv2.recoverPose` in order to achieve this.

2.1.5 Triangulation of 3D Points

We can now define the camera projection matrices for both views.

$$\lambda_1 \begin{bmatrix} u_1^i \\ v_1^i \\ 1 \end{bmatrix} = \mathbf{M_1} \vec{P} \qquad \lambda_2 \begin{bmatrix} u_2^i \\ v_2^i \\ 1 \end{bmatrix} = \mathbf{M_2} \vec{P}$$

with $\mathbf{M_1} = \mathbf{K} \left[\mathbf{I}|\vec{0}\right]$ and $\mathbf{M_2} = \mathbf{K} \left[\mathbf{R}|\vec{T}\right]$.

By taking the cross-product w.r.t. $p_i,\ i \in \{1, 2\}$ on both the left and right side of the corresponding equation, we get:

$$\vec{p_1} \times \mathbf{M_1} \vec{P} = [\vec{p_1}]_\times \mathbf{M_1} \vec{P} = \vec{0}$$

$$\vec{p_2} \times \mathbf{M_2} \vec{P} = [\vec{p_2}]_\times \mathbf{M_2} \vec{P} = \vec{0}$$

We can then solve for $\vec{P}$ by using SVD. This functionality is implemented in `cv2.triangulatePoints`.

2.1.6 Vizualization

To illustrate the above points and verify that our algorithm works as intended, we plotted the Harris score as well as the corresponding keypoints and their landmarks. To verify that the keypoints are well matched, we also plotted the reprojected 3D points.

Figure 1: Harris Score for parking dataset

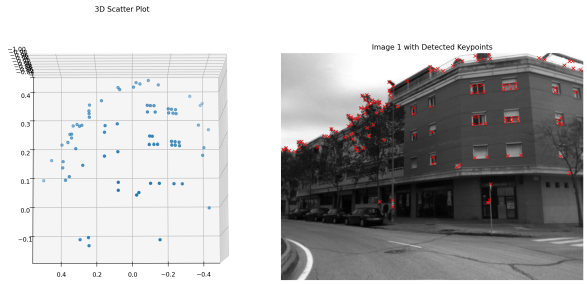

Figure 2: 3D scattering and Feature Detection for Malaga dataset

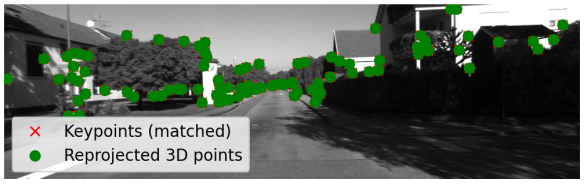

Figure 3: Reprojected 3D points of the matched keypoints for KITTI Dataset

2.2 Continuous Operation

Author Contributions

- Anton Pollak
- Jakob Wolfram

The continuous VO pipeline was implemented in a Markovian way. Hence, processing the current frame only depends on information from the previous frame.
The following data from the previous frame is used as input for processing the consecutive image:

- Keypoints: the currently available keypoints
- Landmarks: set of 3D positions associated to the corresponding keypoints
- Candidate keypoints: set of keypoints found in the current frame that still have the "candidate" status (i.e., they have not been successfully tracked long enough)
- First observations: set of the initial 2D image coordinates retrieved at the first observation of a keypoint (necessary for triangulating new landmarks)
- Poses at first observation: Set of transformation matrices describing the camera poses at the first observations of keypoints (necessary for triangulating new landmarks)
- Keypoint tracker: a counter that indicates how many frames a candidate keypoint has already been successfully tracked

2.2.1 Tracking of current keypoints

First, the currently active keypoints are tracked from the previous frame to the current frame with the Kanade-Lucas-Tomasi (KLT) algorithm implemented in `cv2.calcOpticalFlowPyrLK`. The successfully tracked keypoints are then used to calculate the current camera pose using P3P. Since outliers negatively effect the performance of P3P, RANSAC is first applied to filter inliers. Both algorithms are jointly implemented in the function `cv2.solvePnPRansac`. The resulting rotation and translation vectors are further used to reconstruct the transformation matrix $\mathbf{T_{WC}}$, defining the current camera pose.

2.2.2 Tracking of candidate keypoints

The candidate keypoints are also tracked from the previous frame to the current frame with KLT. The trackable candidate keypoints are split into two groups:

1. If the corresponding *Keypoint Tracker* is above a user-defined threshold L_m (i.e., they were successfully tracked for L_m frames), the keypoints are marked as promotable keypoints.

2. If the corresponding *Keypoint Tracker* is below L_m, the candidate keypoints are stored for the following frame.

2.2.3 Filtering by bearing angle threshold and distance to landmark

The depth error is calculated as follows:

Starting with the formula for depth

$$Z = \frac{bf}{u_l - u_r} = \frac{bf}{D}$$

and approximating ΔZ with a first order approximation, we get:

$$\left|\frac{dZ}{dD}\right| = \left|\frac{bf}{D^2}\right|$$

$$\Delta Z \approx \left|\frac{dZ}{dD}\right| \Delta D = \left|\frac{bf}{D^2}\right| \Delta D$$

$$= \frac{bf}{(bf)^2}\, Z^2 \Delta D = \frac{Z^2}{bf}\, \Delta D$$

Two facts are apparent from this formula:

1. The uncertainty can be improved by increasing the baseline b in the denominator.

2. The uncertainty can be bounded by limiting the distance to the landmark.

Hence, when simply using the obtained promotable keypoints for triangulation, the results are prone to error.
In order to obtain a larger baseline, a minimum bearing angle between the two poses is required. This is implemented as follows:

1. Calculate the rotation matrix from the camera frame C_1 at the first observation of the corresponding keypoint to current camera frame C_2:

$$\mathbf{R_{C2C1}} = \mathbf{R_{C1W}}\, \mathbf{R_{WC2}}$$

2. Calculate the bearing vector for both poses in C_2 frame using the perspective projection equation:

$$\vec{v_1} = \mathbf{R_{C2C1}}\, \mathbf{K^{-1}}\, \begin{bmatrix} u_1 & v_1 \end{bmatrix}^{\mathrm{T}}$$

$$\vec{v_2} = \mathbf{K^{-1}}\, \begin{bmatrix} u_2 & v_2 \end{bmatrix}^{\mathrm{T}}$$

3. Calculate the bearing angle α between the vectors using the dot product formulation:

$$\alpha = \frac{cos^{-1}(\vec{v_1} \cdot \vec{v_2})}{\|\vec{v_1}\| \, \|\vec{v_2}\|}$$

If this angle is larger than a user-defined threshold, the keypoint is considered for triangulation. Otherwise, the keypoint is re-added to the candidate keypoints since the bearing angle might eventually be large enough.

Furthermore, the distance from the camera pose to the landmark was limited to a threshold as well to improve the depth estimation uncertainty and eliminate landmarks that have apparent errors after the triangulation (e.g., the resulting position being behind the camera). This is done by transforming the 3D landmark position into the current camera frame and checking whether the distance from the camera pose to the landmark in z-direction is inbetween the user-defined thresholds.

2.2.4 Extraction of new candidate keypoints

New candidate keypoints have to be extracted continuously, since the current keypoints eventually move out of the field of view. In order to ensure that the new candidate keypoints are unique and not duplicates of the current (candidate) keypoints, the Euclidean distance between the new candidate keypoints and the current (candidate) keypoints (N_{KP}) has to be larger than a user-defined threshold:

$$d_{CKP} = \min_{\forall i \in N_{KP}} \left\| \begin{bmatrix} u_{CKP} \\ v_{CKP} \end{bmatrix} - \begin{bmatrix} u_i \\ v_i \end{bmatrix} \right\|$$

If this is the case, the keypoint is added to the candidate keypoints.

Fig. 4 shows an example plot of extracted keypoints on an image of the KITTI dataset. It shows all groups of keypoints, such as "mature" and candidate keypoints, as well as untrackable ones and those sorted out as duplicates or candidates with too low of a bearing angle for reliable triangulation.

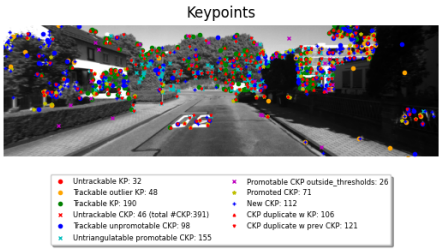

Figure 4: Keypoints overlaid onto the respective image in a KITTI dataset

3 Results and Discussion

This section serves as a short discussion on the quality of the VO pipeline results and provided screencasts.

In general, we achieved acceptable local accuracy on the datasets (see Fig. 5). However, our implementation suffers from scale ambiguity on a global scale. We could mitigate some drift by tuning the parameters but to achieve better results on a global scale, more parameter tuning would be necessary. In addition, further optimization techniques such as Bundle Adjustment or

Pose-Graph Optimization, as well as Place Recognition for loop detection and closure would be required to receive more reliable global results. However, this was outside the scope of this project.

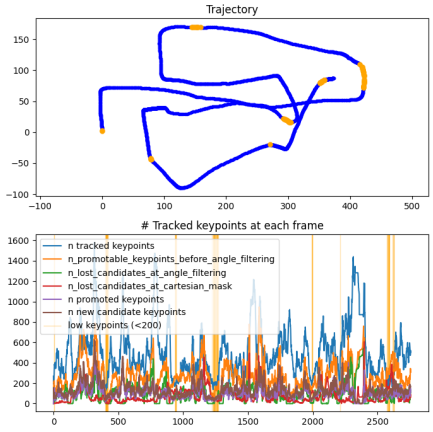

Figure 5: Final trajectory and pipeline metrics on the KITTI dataset

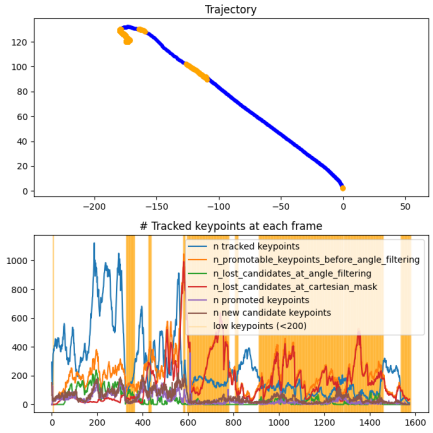

Figure 6: Final trajectory and pipeline metrics on the Malaga dataset

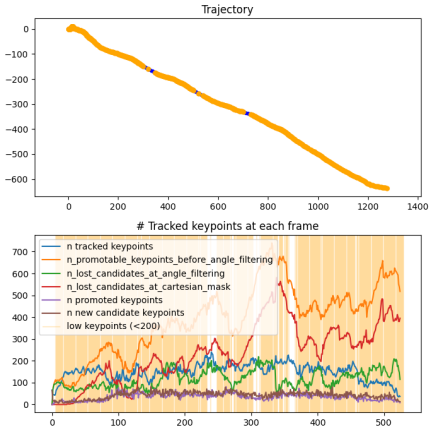

Figure 7: Final trajectory and pipeline metrics on the Parking dataset

3.1 Landmark Contraction

An interesting phenomenon that occurred in some parameter configurations after approximately 1000 frames in the KITTI dataset was the contraction of the 3D positions of the landmarks towards a singular point. The behavior is visible in the trajectory plot of the Malaga dataset, where the car is unable to return to the starting position after the first roundabout.

In a VO pipeline that has no exterior reference (such as fusion with other sensor data (e.g., Visual Inertial Odometry)), this poses a significant problem, since the triangulation of new landmarks relies on the assumption that the previously triangulated landmarks are correct. However, once errors are introduced into the triangulation through outliers, false tracking of keypoints, and other phenomena, the integrity of all following triangulations is compromised.
This occured specifically in situations of

poor hyperparameter tuning and a large frame-to-frame motion in the dataset, which indicates that the direct method used for keypoint tracking (KLT) could be responsible for this behavior. The number of tracked keypoints also drops well below 100 in these cases, which substantiates this conjecture.

We believe that re-initializing the pipeline (i.e., re-performing bootstrapping as soon as the 3D positions of the landmarks contract too strongly to a singular point) could mitigate this effect.

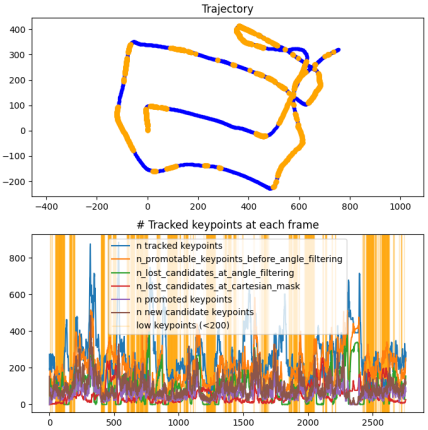

Figure 8: Final trajectory tuned on ARM

3.2 Influence of System Architecture

During testing and finetuning of our implementation, we noticed a strong dependency of the performance on the system architecture, which is likely due to Python dependency issues. We took care to manage our dependencies thoroughly by using virtual environments, but this could not prevent these issues. Our system architectures within the group are ARM (MacOS) and x86 (Linux/Windows). Running the same code with the same hyperparameters on these different machines resulted in different trajectories. Figures 8 and 9 illustrate this: the pipeline tuned on ARM does not perform nearly as well on x86.

To eliminate these issues, we decided to finetune all dataset parameters on an x86 machine, as they are more common.

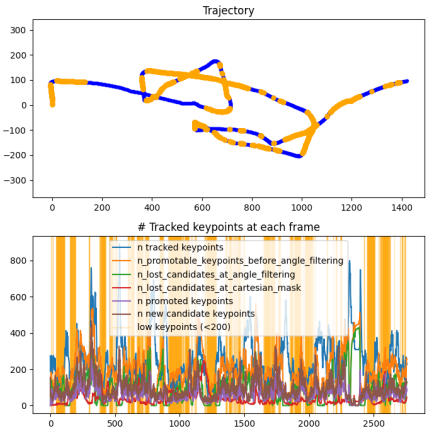

Figure 9: Final trajectory with the exact same code code as on ARM, run on x86