Vision Algorithm for Mobile Robotics

Mini-project: A visual odometry pipeline

Avogaro Niccolò, Bionda Andrea, Burzio Alessandro, Palladin Edoardo

January 7, 2022

Abstract

The goal of this mini-project is to implement a simple, monocular, visual odometry (VO) pipeline with the most essential features: initialization of 3D landmarks, keypoint tracking between two frames, pose estimation using established 2D and 3D correspondences, and triangulation of new landmarks.

1 Introduction

Our project is divided in two parts, the initialization and the continuous operation. The main program *PROJECT.m* act as a glue for the two, loads the selected dataset and makes the pipeline run.

2 Initialization

For the Initialization we used two-view geometry to estimate the relative pose between two frames, and then to triangulate a point cloud of landmarks which can be used later. This is the only method we can use for starting the pipeline, because we don't know neither the starting 3D point cloud nor the baseline distance between two subsequent frames. This is the classical approach to the monocular vision pipeline. The main steps taken to achieve this are:

1. In the history of our project, we started by implementing the initialization with standard feature matching: we manually selected two frames at the beginning of the chosen dataset. We chose frames 1 and 3 to have a baseline large enough to get a good accuracy in computing the initial 3D point cloud. We later tried to skip frames 3 and 4 and chose 1,4 or 1,5 as our bootstrap frames to ensure an even bigger baseline, and got slightly better results.

2. we detected the Harris features with the Matlab function *detectHarrisFeatures.m*, using the standard *FilterSize* and the *MinQuality* equal to 1e-5. This choice was made to ensure a bigger number of points getting detected by Harris, in order to get a better estimate later.

3. We selected the 2000 strongest corners to have a big dataset of corners, and tried to use different feature descriptors with the function *extractFeatures* using ORB and FREAK and also by using directly the patch. We saw that FREAK was the most accurate.

4. We matched the features and estimated the fundamental matrix with the function *estimateFundamentalMatrix.m* with RANSAC to eliminate the outliers. We chose 0.2 as maximum distance threshold for the RANSAC algorithm to consider a point to be an inlier, as it was providing good results.

5. We triangulated the 3D points with the function triangulate and got rid of negative and far away points to have a better estimate in the continuous operation.

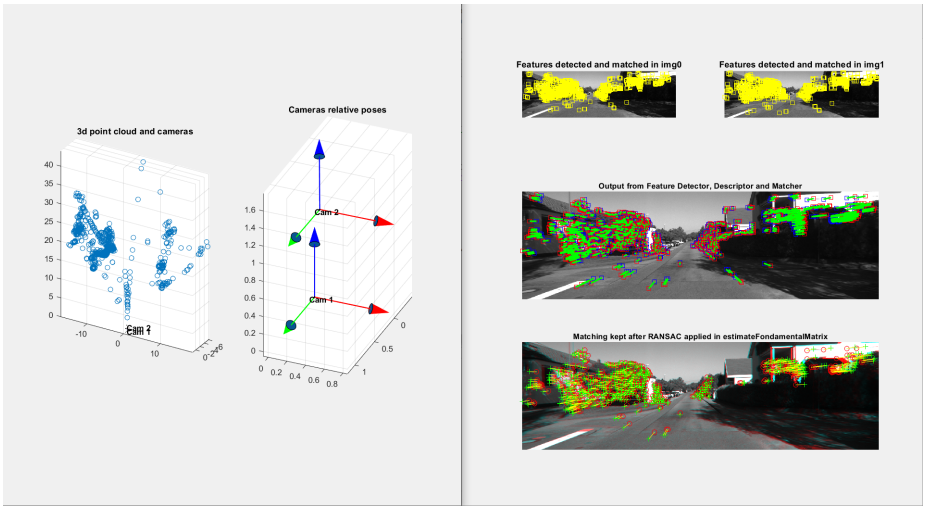

Figure 1: Initialization (KLT) output screenshot

After lots of parameters tuning we saw that our initialization was sporadically providing incorrect results in the camera pose estimation, so we tried to use KLT to track the keypoints instead of matching them. By selecting frames 1,2,3 we respect the temporal consistency assumption for Kanade-Lucas-Tomasi tracker and everything works nicely. Now the pipeline is a little modified:

1. We detect the Harris features with the Matlab function detectHarrisFeatures as before. Now we select just the 1000 strongest corners, and only for the parking dataset we round them to the nearest integer. We saw by plotting the correspondences that the tracking of the features was a bit inaccurate, since the motion in the parking dataset implies matching should always happen on the same scanline while KLT provided sub-pixel matching. This led to a difference in the y-axis pose estimate which made the estimate of the pose drift. By rounding the keypoints locations the problem disappeared.

2. We created a PointTracker object with the *MaxBidirectionalError* set to 1. This is the maximum allowed error after tracking the keypoints forwards and backwards. If the error between the two points is more than 1 pixel, the key point is discarded.

3. Points are tracked from frame 1 to frame 2, and then the remaining ones to frame 3. Then we decided to discard matches with small displacement (less than 3 pixels) to get better accuracy: this was done to ensure a baseline large enough for correct triangulation.

4. We estimate the fundamental matrix, triangulate the 3D points, and cut the far away and backwards ones just like before.

In general this approach provided more reliable landmark triangulations and pose estimates.

3 Continuous operation

The main idea behind continuous operation is to elaborate two consecutive frames and estimate the camera position from the 3D landmarks and 2D correspondences using P3P RANSAC, and

then triangulate new points. Continuous Operation also manages the addition of new keypoints based both on a threshold on the angle between the point vectors of the candidates and their first observation as well as a minimum number of consecutive frames for which a certain candidate keypoint has been seen. The main continuous operation function is run in the *CO_processFrame.m* file. This function has as inputs img_i, img_i_prev, S_i_prev, cameraParams, cfgp which are:

- img_i: Current image of the dataset
- img_i_prev: Previous image
- S_i_prev: State of the previous frame. It is a Matlab structure which contains:
  - Keypoints: 2D points tracked from the previous frame
  - Landmarks: 3D corresponding landmarks from triangulation of keypoints
  - Candidates: 2D points which could become keypoints
  - Candidates count: array where each element contains the number of subsequent frames a certain candidate has been tracked for
  - First observation: 2D pixel coordinates of the first time we observed the candidates
  - Camera Pose of the first observation: pose of the Camera when we first observed the candidates
- cameraParams: Intrinsic camera parameters specific to the dataset
- cfgp: parameter configuration data structure, contains all the parameters for the various parts of the code

3.1 CO_processframe.m

The function performs the following steps:

1. Import the previous frame state.
2. Uses KLT to track the keypoints from the previous to the current frame in the same way done during initialization. Keypoints that are not visible anymore and their corresponding 3D Landmarks are removed from the state variable.
3. We use the function *estimateWorldCameraPose* function to do the P3P with MSAC algorithm (M-estimator Sample and Consensus, a modified RANSAC algorithm). Initially we tried using the one we developed in class in the exercise 7: it took quite a lot of time to figure out that in this case there was no need to flip the 2d points fed to the algorithm, which was the reason even at the first analyzed frame our pose estimates were terrible and the great majority of the points were found to be outliers. In any case, even after fixing this issue the Matlab implementation proved to faster and slightly more reliable. As inputs it receives the 3D landmarks and the keypoints tracked in the current frame. As outputs it gives the pose and the set of inliers.
4. We then apply non-linear refinement: the matlab function *bundleAdjustmentMotion* provides a refined estimate using the Levenberg-Marquardt optimization by moving just the camera pose to minimize the reprojection error. This is not true bundle adjustment as the landmarks are kept fixed instead of being jointly refined with the pose, however it greatly improved the accuracy of our pipeline and decreased jitter between subsequent frames.

Figure 2: example of trajectory with jitter before NL refinement

5. We now track all the candidates from the previous frame onto the current one. These points are features that can become keypoints if some conditions are met -see later. The baseline between two subsequent frames would be too short to triangulate new points with sufficient accuracy. We also delete candidates and their corresponding *First observation* and *Camera Pose of the first observation* when they are not tracked anymore - e.g. viewpoint changed significantly or occlusions are in the way. We set a limit to 500 tracked candidates to not overload the pipeline, keeping always the oldest ones (most likely to get triangulated soon).

6. At this point, we want to find new candidates from features: we first detect new Harris features (*standardFilterSize* and *MinQuality* equal to 1e-5 as before) in the current frame, only keeping the 500 strongest ones. We calculate the euclidean distance between the new features and both previous candidates and keypoints tracked with KLT from the previous frame to the current one. If the distance is above a certain threshold (3 pixels) we select them as new candidates.

7. We select new keypoints from the candidates, only if they are tracked for a sufficient amount of frames (8 frames) or if the angle between the point vector of the first observation and the one of current position exceeds a given threshold ($10 \cdot \frac{\pi}{180} rad$). The angle is calculated as: $\alpha = \arccos((\frac{F \cdot C}{||F|| ||C||})$.

8. These new keypoints are triangulated into 3D landmarks using their first observation, the current 2D pixel coordinates and their respective camera poses. As before, to improve the quality of the function we also check if triangulated point are too far or behind the image plane, but also if their reprojection error is too high. The candidates discarded this way are placed back into the candidates pool.

9. We finally update the state variable.

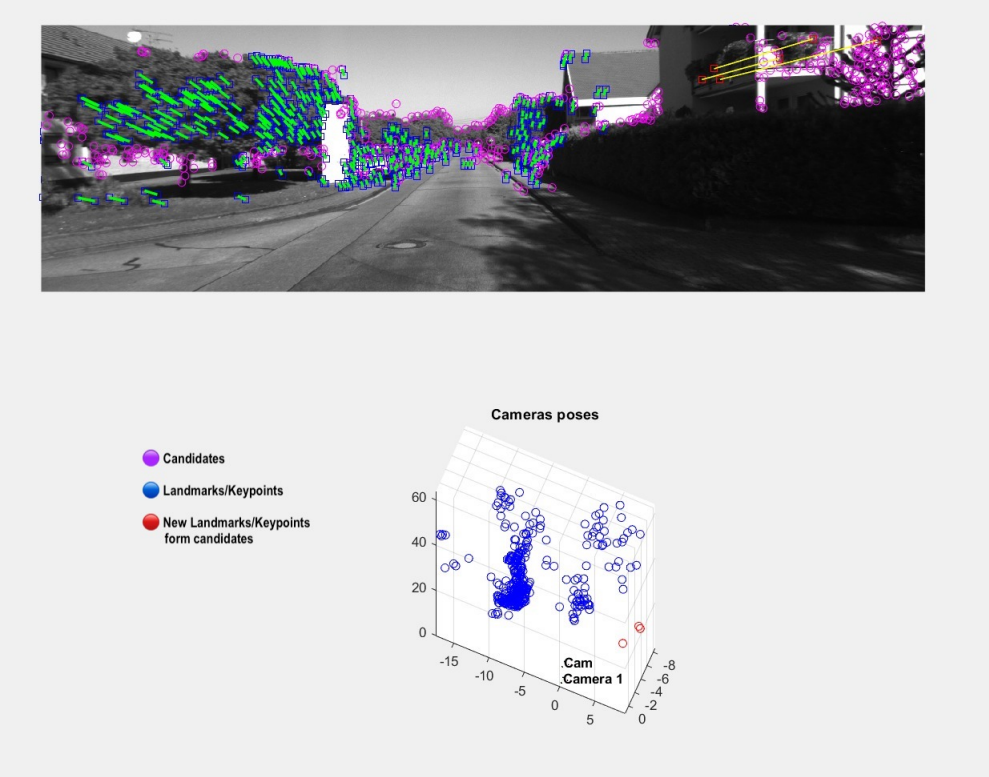

Figure 3: CO_processframe output screenshot

4 Main *PROJECT.m*

In the main file we run the initialization and the continuous operations functions, and we set the plotting of the final results. These are the main parts:

1. Set the dataset path, load the ground truth (if existing), and create the cameraParams object using the intrinsics matrix K.
2. Run the initialization function: it will output the pose of the camera (rotation R_C2_W and translation T_C2_W matrices between world frame and the camera of the last bootstrap frame).
3. Initialize the frame state variables and the pose history used later for the plot.
4. For every frame (excluding the ones used in the initialization):
   (a) If the number of keypoints is less than 100, perform the initialization step again using the 3 previous frames only to find new keypoints and landmarks. Apply rotation and translation to bring the newly found landmarks in the world reference frame, then apply a non-linear refinement step on old plus new landmarks: this is achieved using the Matlab *bundleAdjustmentStructure* function on a single view (view of previous frame), which keeps the camera fixed and minimizes reprojection error by refining just the landmarks. We saw that these operations improved a lot the precision of the algorithm, despite a reduction in its velocity.

(b) Run the continuous operations function, which outputs the state and the pose of the new frame, taking as inputs the two images, the state of the previous frame, an the various parameters.

(c) Add the new pose to the pose history vector.

(d) Plot the found state trajectory and 3D landmarks.

5. Plot the trajectory of all the frames until the last processed one.

5 Config file *getparameters.m*

All mentioned parameters can be modified in the *getparameters* function, which gives as output a data structure containing all the parameters. This was done to keep all parameters in a single place and to avoid forgetting to set a parameter somewhere else in our code while testing different parameter settings. The choice of the dataset is also handled through this function by changing the 'ds' parameter to the desired one.

6 Results

In the screecast during the operation four plots are visible:

1. Top left: local trajectory + triangulated landmarks (balck circles)
2. Top right: tracked keypoints (green), candidates (red), candidates added in the current frame (magenta), candidates added to keypoint set (blue: first detection position, cyan: current position)
3. Bottom left: orientation of previous and current camera, can add 3d Landmarks pointcloud to this plot by setting plot new in config file to true
4. Bottom right: number of keypoints tracked(green) and candidates(red)

At the end of the screencast the full trajectory is displayed alongside the ground truth, if available

6.1 Kitti dataset

The result is locally consistent and quite similar to the ground truth, up to a scale factor. The absence of jitter with respect to the previous image where no non-linear refinement was applied is noticeable.

The pipeline crashed in the Kitti dataset around frame 2330 when the car stops at a crossroad. After a few frames, at a new initialization the estimateRelativePose fails to disambiguate and the pipeline crashes. Further considerations would be needed to handle this edge case.
For this reason, in the screencast we only let the pipeline run until frame 2300: https://www.youtube.com/watch?v=lVYn6uOW2Vs

Figure 4: Kitti dataset result

6.2 Malaga dataset

The result is qualitatively good. Local accuracy is achieved but the overall drift is quite noticeable in this scenario.
The following is a video of our pipeline running this dataset: https://youtu.be/CiI_OeyePWU

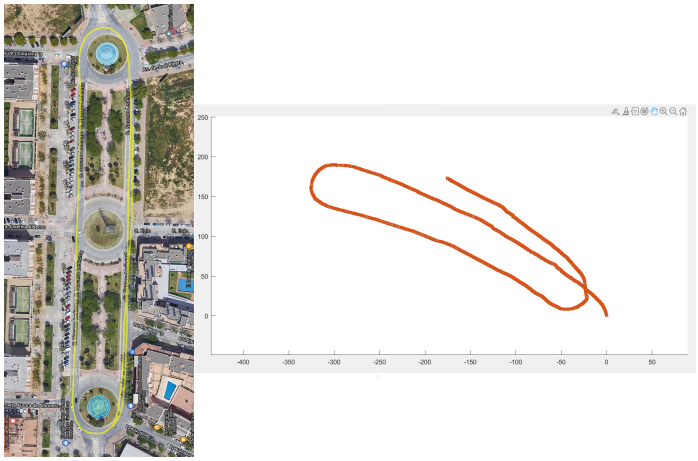

Figure 5: Malaga dataset result

6.3 Parking dataset

The result is qualitatively good. Local accuracy is achieved and the overall drift is low. Drift becomes more noticeable in the first and last sections where no cars are in front of the camera, and features tracked are further away in the scene (higher depth uncertainty).
The following is a video of our pipeline running this dataset: https://youtu.be/EgOp20M0F6E

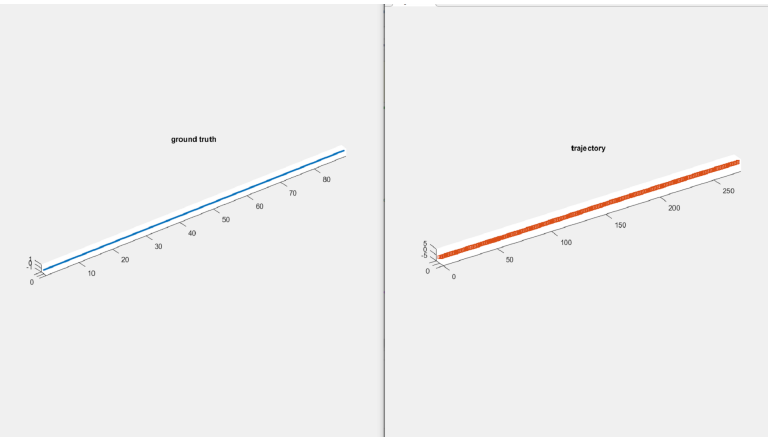

Figure 6: Parking dataset result

7 Custom Datasets

We decided to implement a custom dataset as an additional feature. We thought for quite some time about what environment would be a good choice. We initially wanted to take a recording of a city road, like the one used in the *KITTI* dataset, but being Zurich a very busy city it would have been hard walking around it not to get run over by trams. So, taking as advantage the Christmas break, we imagined instead what a humanoid robot working in a rural area would have to handle in its day-to-day activities and recorded a couple of datasets accordingly. We recorded the videos with an iPhone 11 Pro without any external stabilizers, just by hand, using the main camera at 1080p 30fps with the 26mm f/1.8 lens. We downscaled the recorded videos to 540p resolution (960x540) to have a lighter image to process, and extractedd 1 frame every 10 to reduce the size of the dataset.

7.1 Camera Calibration

Before recording the dataset we had to find the intrinsic parameters of the camera. We used the OpenCV toolbox for Python https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html to calibrate the camera at the right resolution. We printed a checkerboard and recorded a small video for the calibration. We then used 10 frames sampled with a distance of 1s from each other to calibrate the camera, to have a different view for each frame. We then found the intrinsic K matrix:

$$\begin{bmatrix} 845.52896472 & 0. & 468.10562767 \\ 0. & 835.35434918 & 298.01291054 \\ 0. & 0. & 1. \end{bmatrix}$$

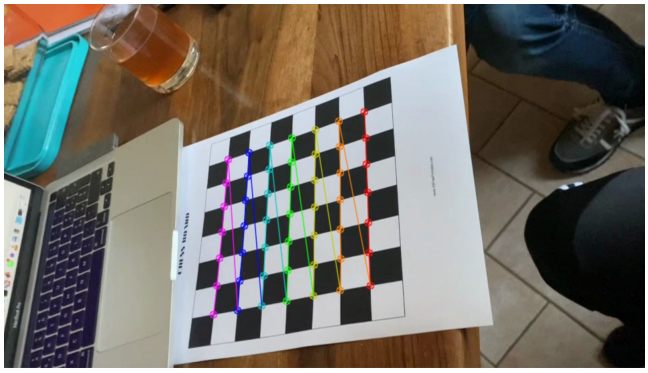

Figure 7: Camera Calibration using Zhang's Method, example on a single image

7.2 Country life Dataset

The sun of winter days in the Italian countryside makes a lot shadows appear on the scene, and the cold temperature makes you want to work inside rather than outside: the first dataset starts in the driveway of a country house, proceeds in a straight line, then takes a sharp turn to the right and downwards into the basement of the house. The difficulty of this dataset consists in the sharp change of illumination between the outdoor and indoor scene, and we also wanted to test our VO pipeline in a non-flat scenario with different levels. We imported the newly recorded dataset into our Matlab environment, started the VO pipeline, and for the first time since the beginning of the project something worked perfectly at the first try. Although admittedly a little difficult to see without knowing the real leayout of the house, it can be inferred from minute 2:50 of the screencast that the pipeline worked well also in the downward section and the two levels of the path indeed lie on separate planes also in the estimated trajectory.
Country Life dataset screencast: https://youtu.be/oIe0Q46ND0M

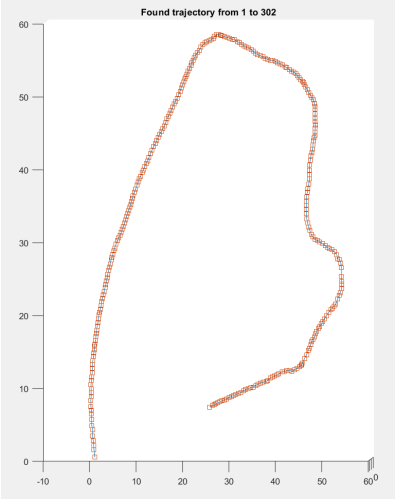

Figure 8: Countrylife dataset result

7.3 Vineyards Dataset

Since the first attempt went much better than expected, we decided to record one more dataset to challenge again the robustness of our VO pipeline in another difficult environment, a vineyard: it is challenging because grass and branches are moved by wind and patterns are very repetitive, and is another classic envoironment a humanoid robot working on a farm would have to navigate. We used the same equipment and setup as in the Country Life Dataset, so there was no need to recalibrate the camera. It is clear from the screencast that it is very challenging for the algorithm to track points, but despite the difficulties, the results are quite good: the estimated camera position returns very close to the starting position and closes a near perfect loop! For reference the length of the longest side is 80 meters.

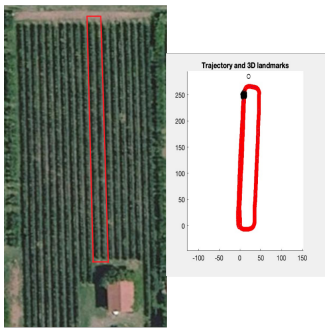

Figure 9: Vineyards dataset result

It should be noted that the recording procedure did not go as smoothly as before (the original video is shaky) and this is indeed reflected in the local pose estimation, which jitters a little bit and seems to travel different distances for different frames (sometimes longer, sometimes shorter). Vineyard dataset screencast: https://youtu.be/etVDBDrH5zo

8 Conclusions

The pipeline implemented has a good overall quality. It is quite robust and works properly with different datasets and scenarios. For example, the two additional datasets are quite challenging for a VO pipeline, yet our solution performs nicely.

The main defect of our program is that it is quite slow: we measured the performance of the pipeline on the parking dataset, and without plotting anything reached a time of 1:41 seconds for 598 frames. The average speed is so 5.92 frames per seconds. The main reason of slowdowns is the re-initialization of the pipeline when the number of keypoints drops below the selected threshold, which happens too often (sometimes once every two or three frames). The most likely reason for this is that there might still be something wrong with the triangulation of newly detected keypoints, or in general with how the rotations and translations are handled: ever since the beginning we had trouble with the triangulation function, which could have easily been solved if only we had spent 5 extra minutes reading the Matlab documentation a bit more carefully.

Additional features to further improve the pipeline would be proper pose-landmark Bundle Adjustment over a sliding window, or loop closure detection and correction to bring the VO pipeline to full SLAM.