Visual Odometry Miniproject
Large Scale Drone Perception

Karina Bay Bloch (Kablo17) & Nicoline L. Thomsen (nthom18)

12/05-2022

Contents

1 Examination of Dataset **2**
1.1 Plot flight path . . . 2
1.2 Reduce dataset . . . 2
1.3 Discussion for reducing dataset . . . 2

2 Map Initialization **3**
2.1 Extracting features . . . 3
2.2 Essential matrix . . . 3
2.3 Estimating position of points . . . 3
2.4 Estimate relative movement . . . 4

3 3D map Initialization **5**
3.1 Estimate 3D position . . . 5
3.2 Updating map . . . 5
3.3 Reprojection error . . . 5
3.4 Bundle adjustment . . . 6

4 Estimate New Camera Positions **6**
4.1 Match new features with map . . . 6
4.2 Estimate camera position and rotation . . . 6
4.3 Add observation to map . . . 7

5 Conclusion **7**

6 Bibliography **8**

Introduction

The process of determining the placement and orientation of a camera is typically used for what is called visual odometry. It revolves around analysing sequential camera images one by one and with this, the change in position and orientation of the camera carrier over time can be determined. Where it would typically be used, would be as an accurate localisation device of a drone, where its position is determined, and its obstacles can be detected and avoided.

In this miniproject, the purpose is to go through the steps of visual odometry to show, how a sequence of pictures are turned into a camera trajectory. To show how it was done, the paper has been divided into 4 parts:

1. Examination of the acquired dataset
2. Initialization of the map via. features and estimation of camera motion between two images
3. Initialization of the 3D map based on triangulation
4. Tracking the camera motion relative to a generated map

Source code for this project can be found in the GitHub repository: `https://github.com/PositiveBeat/LSDP-Miniprojects-22.git`.

1 Examination of Dataset

The dataset that will be used has been provided beforehand. It consists a logfile, a videofile and a file that contains the parameters of the used camera. The code for this exercise is in the script `dataset_processing.py`.

1.1 Plot flight path

The logfile provided for this project contains information recorded during flight, of a DJI Phantom 4 Pro drone. There are much data included, of which the GPS readings are of interest, to show the flight path. The GPS specific data logging latitude and longitude was extracted, and using a pre-written script (`exportkml.py`) exported to .kml file format, and visualised using Google Earth. The path is show on figure 1.

Figure 1: Flight path of Drone

1.2 Reduce dataset

Using OpenCV's methods for handling videos, a script was written to iterate through all frames in the video file, saving every 50th frame for further processing.

1.3 Discussion for reducing dataset

Saving every 50th frame ensures that there is a significant motion between two cameras, to have more accurate measures of location of features in the scene.

2 Map Initialization

The code for this exercise is in the script `map_initialization.py`, for which much of the code is reused from previous exercises.

2.1 Extracting features

The SIFT feature detector, which is implemented in OpenCV, was used to extract features from the first two frames, shown in figure 2. The SIFT feature detector also comes with a SIFT feature descriptor, which generates a histogram of gradients around a feature, in order to uniquely identify it, even when viewed from different cameras.

(a) Frame 0 (b) Frame 50

Figure 2: SIFT features of the two first frames in the reduced sequence

2.2 Essential matrix

Matching the features is done by brute forcing the descriptors obtained from the SIFT operation. The resulting point pairs are used in the OpenCV function `findEssentialMat()`, to estimate the essential matrix. The essential matrix relates the normalised image coordinates y and y' in two frames, that corresponds to the same 3D point, by equation 1. E is the essential matrix, which describes the rotation and translation between the two frames, as per equation 2.

$$y^T E y' = 0 \tag{1}$$

$$E = R[t]_x \tag{2}$$

The resulting matrix is shown in equation 3. The matches are show in figure 3.

$$\begin{bmatrix} 2.27724071 \cdot 10^{-5} & -5.31934866 \cdot 10^{-1} & -1.20642452 \cdot 10^{-1} \\ 5.31387938 \cdot 10^{-1} & 3.74764023e \cdot 10^{-5} & -4.50596475 \cdot 10^{-1} \\ 1.20651749 \cdot 10^{-1} & 4.50026065 \cdot 10^{-1} & -2.60703435 \cdot 10^{-4} \end{bmatrix} \tag{3}$$

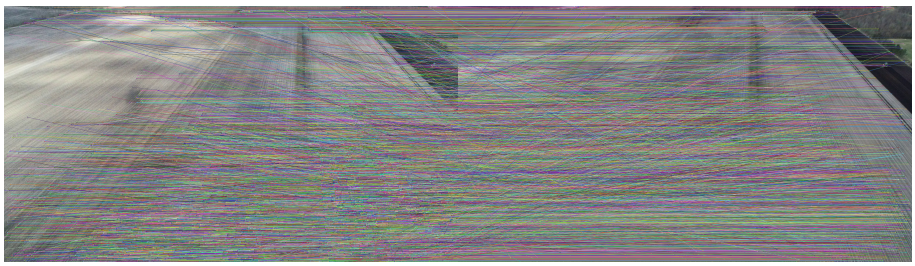


Figure 3: SIFT features matches between the two frames

2.3 Estimating position of points

To get depth information about a point in a frame, at least two frames containing that point is needed. The line projected from the first frame from the camera centre to the point, viewed from

the second frame is the epipolar line for the first frame, and vice versa for the second frame. These lines are drawn in figure 4 [1].

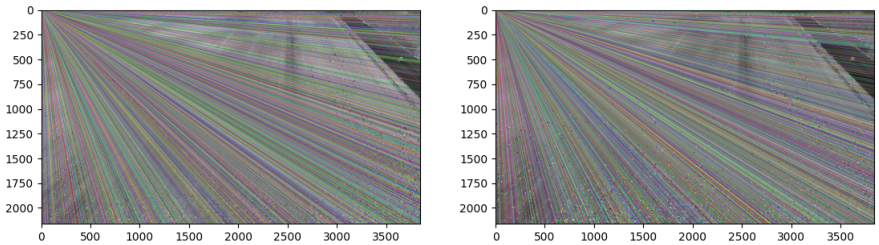
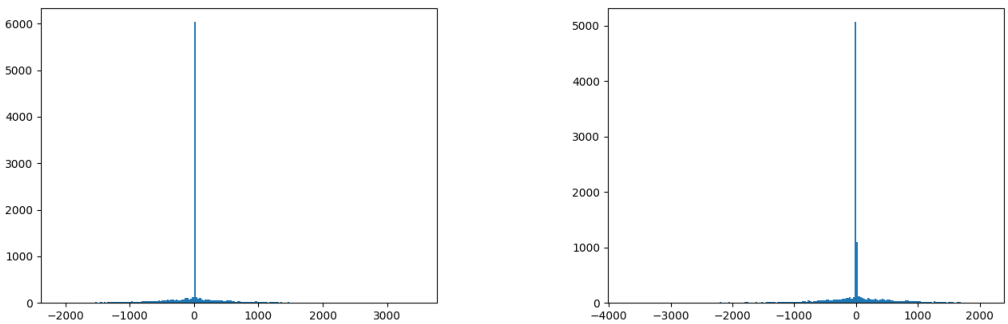


Figure 4: Epipolar lines of the two frames

The epipolar lines are calculated from the essential matrix, and as such it does not represent all points with full accuracy. This creates a difference between the expected position of the point (from the epipolar line), and the actual position. In figure 5 are the distributions of this error for the points in the two frames. As seen in the histograms most observations are close to the value of zero, which indicates a fairly good estimation.



(a) mean: 37.80, std: 586.49

(b) mean: -29.58, std: 582.19

Figure 5: Distribution of the difference between the expected position and the actual position

2.4 Estimate relative movement

To recover the rotation and translation from the essential matrix, it is decomposed to the form in equation 4.

$$E = U \cdot S \cdot V^T \tag{4}$$

The rotation and translation can then be calculated as equation 5.

$$R_e = U \cdot D \cdot V^T \qquad\qquad t = U \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{5}$$

Where the D matrix is:

$$D = \begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Using the provided script and OpenCV's method for calculating this decomposition, the matrix in equation 6 is obtained.

$$\begin{bmatrix} 1 & 0.0001 & -0.001 & -0.7284 \\ -0.0001 & 1 & 0.0003 & -0.1325 \\ 0.001 & -0.0003 & 1 & -0.6722 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6}$$

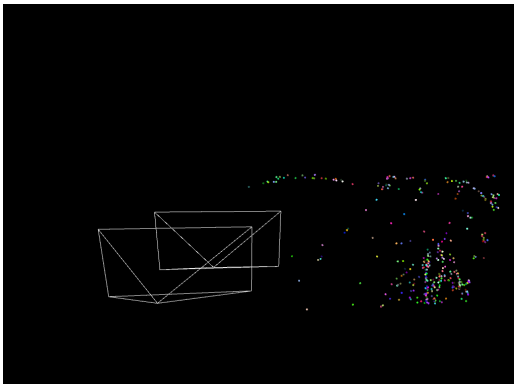
3 3D map Initialization

3.1 Estimate 3D position

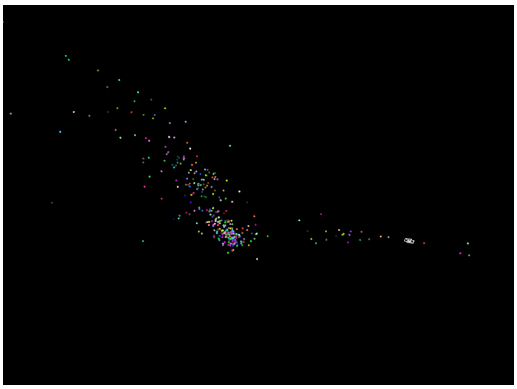
Since there is access to corresponding points and the relative motion between two cameras, the 3D position of these points can be reconstructed with the use of something called 3D point triangulation. in OpenCV there is already an implemented method called `triangulatePoints()`, which will use a projection matrix from the first and second camera together with a list of image coordinates from the same cameras. It will then return the 3D points, which via. the function mentioned above gave an output of 251 points that were projected into 3D between the 2 first frames. A figure of them will be shown in the next exercise.

3.2 Updating map

What is shown below in figure 6 is the estimated 3D points from the corresponding image from last exercise.



(a) Seen from behind the cameras



(b) Seen from the side and turned 90 degrees clockwise

Figure 6: An example of the Map object, with the first corresponding frames. The white lines show the camera and their position, while the coloured dots shows the estimated points

While almost all of the points are on the right side of the cameras (in front), there can be seen several points behind the camera, but it shows clear signs, that the cameras are seeing from above, which is clear to us, because the cameras are on drones.

3.3 Reprojection error

When optimising with something called bundle adjustment, the reprojection error is typically a targeted value to minimise, so that the outcome of the bundle adjustment can be as precise as possible. To calculate the reprojection error of the current state of the map, it is known that the reprojection error is the squared difference in the expected image coordinate, (u_ij, v_ij) and observed image coordinate, $(\hat{u_ij}, \hat{v_ij})$, and by using a premade function the reprojection error can be found:

Listing 1: Reproj. Func.

```
calculate_reprojection_error_for_point(self, point, camera_pose, image_coord)
```

The Total Reprojection error seems to be 23.576586, while the Mean Reprojection error is 0.030304, and these results are what is sought to be reduced in the next exercise with bundle adjustment.

3.4 Bundle adjustment

Bundle adjustment is implemented with the help from g2o, also know as General Graph Optimization, which is a framework, that is typically used for solving optimization problems. First the optimization problem is described for the g2o framework as such:

- The Internal camera properties (focal length and optical centre)
- The External camera properties (location and orientation)
- The Interest point locations
- And the Observations that link cameras and interest point locations

This will fix the internal camera properties, which will then result in lower reprojection errors. To do so with the frames, another function is used, which is called `optimize_map()`. What it does is, that it defines the camera parameters, adds the cameras to the optimization problem, then adds the interest points to the problem and lastly the observations. It then runs the optimizer and extracts the problem parameters. The outcome from the function can be seen in figure 7.

```
Performing full BA:
iteration= 0     chi2= 23.576586         time= 0.00648998        cumTime= 0.00648998     edges= 778      schur= 1        lambda= 1057090435825321.375000         levenbergIter= 8
```

Figure 7: A small cut of the outcome of a Bundle Adjustment. For every BA, an extra line is added

4 Estimate New Camera Positions

4.1 Match new features with map

With the bundle adjustment implemented, the camera position of the new frames has to be estimated. To do so, the image features of the new image has to be used to look for features matches with the points from the currently existing map. This is somewhat the same of what was done in 2.1 with the SIFT feature detector. This time however, the features are also matched against the map, in order to determine if there are enough matches to estimate the cameras position and orientation. For these exercises frame 800, 850 and 900 was used instead of simply the first three frames, because these had more prominent movement between them. Figure 8 shows the matches between the new frame in the map, and the previous frame.

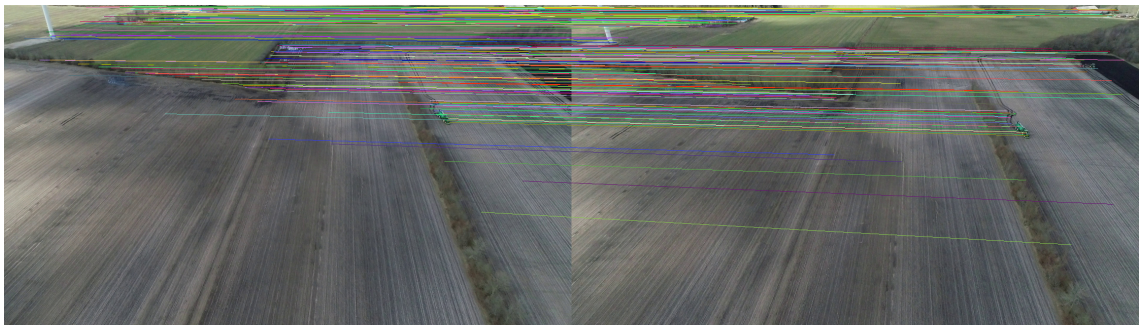


Figure 8: New feature matches with previous frame

4.2 Estimate camera position and rotation

To estimate the current position of the camera, the function `estimate_current_camera_position()` is used, which contains the `cv2.solvePnPRansac()`. This function is a Perspective-n-point pose computation function [2], that computes the object pose when it comes to the camera frame, while using the RANSAC scheme to deal with the outliers. The matrices in equation 7 show the acquired rotation and translation matrix for the new frame (frame 900).

$$R = \begin{bmatrix} 1 & -0.0014 & -0.0025 \\ 0.0014 & 1 & 0.0016 \\ 0.0025 & -0.0016 & 1 \end{bmatrix} \qquad t = \begin{bmatrix} -0.5569 \\ -0.0045 \\ -0.8052 \end{bmatrix} \tag{7}$$

4.3 Add observation to map

Like in 3.2, the map has now been updated with the new camera positions, and can be seen in figure 9. An extra camera has been added to the cluster, which is the addition of the newest estimated camera. It is the camera closest to the points (as the drone flies forwards). Below are the final map statistics.

Number of points in map: 281

Number of cameras in map: 3

Number of observations in map: 389

Total reprojection error: 9.892401

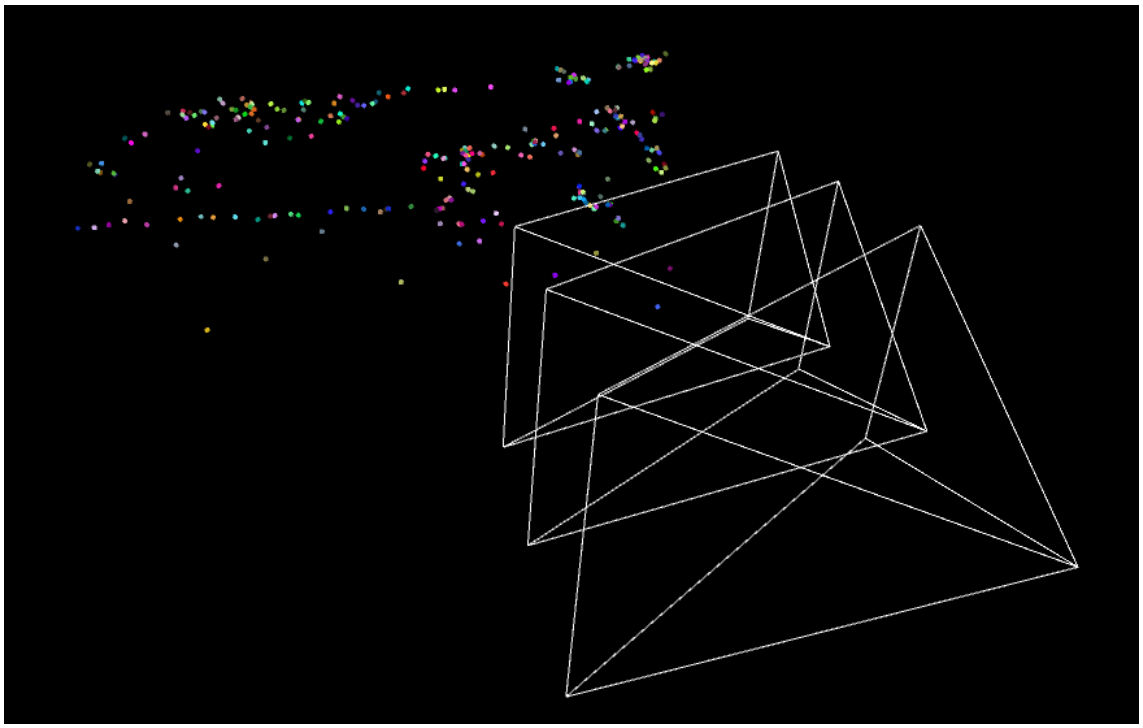


Figure 9: Map viewer showing features and cameras in the updated map

5 Conclusion

The reason for working with every 50th frame instead of every frame is to see a substantial change in the environment and with this, using SIFT will give clear feature detection results, that can be used to further estimate the position of points and the relative movement.

From this, even a 3D map of the points, observations and cameras can be mapped and evaluated to further increase the precision of the cameras and points by looking at reprojection errors and outliers. With bundle adjustment, new camera positions can be acquired by matching newly acquired frames with the existing ones to reach peak performance.

6 Bibliography

[1] Open Source Computer Vision. Opencv: Epipolar geometry. https://docs.opencv.org/4.x/da/de9/tutorial_py_epipolar_geometry.html. Online; accessed 17 May 2022.

[2] Open Source Computer Vision. Perspective-n-point (pnp) pose computation. https://docs.opencv.org/4.x/d5/d1f/calib3d_solvePnP.html. Online; accessed 19 May 2022.

(ﾉ °□°)ﾉ ︵ ┻━┻