POLITECNICO
MILANO 1863

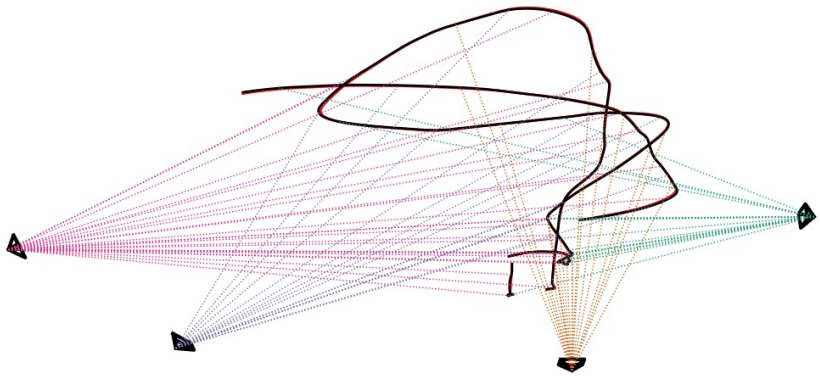
Image Analysis and Computer Vision
Project report

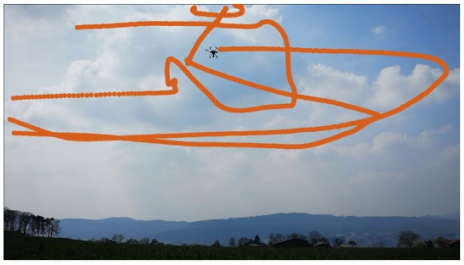
Elia Pontiggia

247274 - 10716792

Proposal F02: Reconstructing drone trajectories and cameras from multiple images

June 5, 2025





Contents

1 Introduction

1.1 Problem Formulation

With the growing ubiquity of unmanned aerial vehicles (UAVs) across industries—ranging from cinematography to environmental monitoring—accurately tracking their motion in three-dimensional space has become increasingly vital. While onboard navigation systems using GNSS, IMUs, or visual-inertial odometry are widely employed, they may be unavailable or unreliable in certain scenarios (e.g., GNSS-denied environments or during external monitoring for regulatory compliance). In such cases, outside-in tracking—inferring the UAV's position using external sensors—is the only viable approach.

Conventional external tracking systems, such as motion capture setups or theodolite-based solutions, are expensive, require specialized equipment, and are limited in scalability and deployment flexibility.

Therefore, there is a clear demand for a more accessible, scalable, and low-cost tracking solution that works with consumer-grade hardware and minimal calibration requirements.

This project aims to address this challenge by implementing a pipeline for reconstructing the 3D trajectory of a UAV inspired by the work of Li et al.

1.2 State of the Art

This project is inspired by the work by Li et al., that addresses this challenge by introducing a novel method for reconstructing the 3D trajectory of a flying object—such as a UAV—using only videos recorded from an ad-hoc network of unsynchronized, uncalibrated cameras. These cameras, which may have unknown poses, frame rates, and rolling shutter distortion, are placed independently around the flight area. The system relies solely on known intrinsic parameters of each camera (e.g., focal length, distortion) and automatically estimates all other variables—including camera poses, temporal offsets, and rolling shutter effects—during processing.

The pipeline begins by detecting the UAV in each video and generating approximate 2D trajectories, which are then used to derive correspondences across cameras. An incremental structure-from-motion strategy is used to compute initial camera poses and triangulate 3D points. The resulting geometry is refined using an extended bundle adjustment that incorporates rolling shutter modeling and motion regularization (e.g., minimizing kinetic energy or force). This allows for highly accurate reconstruction (errors below 40 cm) even when using low-cost consumer cameras like smartphones or action cams.

This approach represents a significant advancement in practical UAV tracking by eliminating the need for prior camera synchronization or rigid installation constraints, making it suitable for scalable and cost-effective deployments in real-world outdoor environments.

2 Comparison with the original work

Our project closely follows the methodology outlined in the paper, but it is not as complete as the original work. Table 1 shows a comparison of the main differences between our implementation and the original work:

3 Implementation

Our implementation of the project closely follows the methodology outlined in the paper, even though it is not as complete as the original work.

The dataset used for this project is the same used and described in the paper, that is accessible at `https://github.com/CenekAlbl/drone-tracking-datasets`

We decided to focus on three main aspects of the pipeline:

- **Camera intrinsic calibration:** A method to calibrate the cameras and obtain their intrinsic parameters
- **Drone detection:** An algorithm to detect the drone in each frame of the videos, which is crucial for tracking its trajectory.
- **Offline trajectory reconstruction:** A method to reconstruct the drone's trajectory a posteriori, using the detected drone positions from multiple cameras.
- **Online trajectory reconstruction:** A real-time implementation that reconstructs the drone's trajectory as it flies, using the same detection and reconstruction methods.

Paper aspect	Implemented	Notes
Camera intrinsic calibration	Yes	The paper assumes that the cameras are precalibrated
Drone detection	Yes	In the paper it is not explained how the drone is detected, they focus only on the trajectory reconstruction.
Time shift estimation	Yes	We implemented a simple brute-force search algorithm, while the paper uses a solver implemented by another author.
Camera registration	Yes	
Correspondences between unsynchronized cameras	Yes	We didn't perform the linear approximation of the drone motion, but it was still quite effective.
Rolling shutter correction	No	
3D trajectory reconstruction	Yes	
Bundle adjustment: spline trajectory refinement	Almost	We optimized only camera poses, not the 3D splines nor α, β parameters.
Bundle adjustment: least kinetic energy	No	
Bundle adjustment: least force	No	
Offline trajectory reconstruction	Yes	
Online trajectory reconstruction	Yes	

Table 1: Comparison between our implementation and the original work

All the aspects are implemented in Python, heavily relying on the OpenCV library for computer vision tasks. The code is structured in a modular way, allowing for easy integration and testing of each component.

Below is a detailed description of each aspect of the implementation. We also included some results and plots to illustrate the performance of the algorithms. The full plots and results can be found in the `plots/` folder of the archive attached to this report.

3.1 Camera Intrinsic Calibration

The first step in the pipeline is to calibrate the cameras and obtain their intrinsic parameters. This is crucial for the subsequent steps, as the intrinsic parameters are used to undistort the drone detections and to compute the projection matrices for each camera.

The paper assumes that the cameras are precalibrated, since it is a quite easy task to perform with a chessboard pattern. However, for the completeness of the project, we decided to implement a simple yet effective camera intrinsic calibration method, based on the OpenCV library.

The calibration process follows Zhang et al.'s method, which uses a chessboard pattern to estimate the intrinsic parameters of the camera. All the necessary steps are already implemented in the OpenCV library, so we only had to write a simple script to load the images, detect the chessboard corners, compute the intrinsic parameters, and output a couple of files containing the comparison between the computed intrinsic parameters and the ground truth values provided in the dataset.

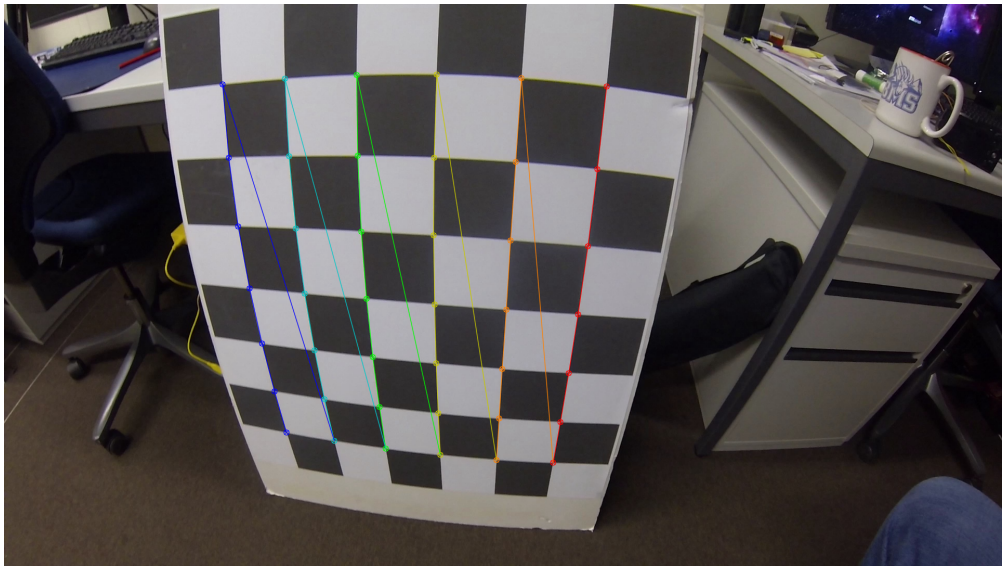


Figure 1: Example of the chessboard corners detected in the gopro3 camera

3.2 Drone Detection

The first step for the 3D trajectory reconstruction is to detect the drone in each frame of the videos.

Initially, the drone was detected using a classical background subtraction method. We applied a Gaussian Mixture-based background subtractor (MOG2) to isolate moving objects against a mostly static sky background. After noise filtering with morphological operations, we analyzed contours in the foreground mask and filtered drone candidates based on area, aspect ratio, and vertical position constraints. Among these, the best candidate was selected either by spatial continuity (if a previous position was known) or by maximum area. To reduce jitter and improve stability, the detected positions were smoothed over time using a weighted average with the recent tracking history. This approach worked well when the drone was clearly visible and motion was distinct, but it became unreliable in the presence of occlusions, intermittent visibility, or when the drone was stationary.

To improve robustness, especially during frames where the drone was not clearly visible or briefly occluded, we integrated optical flow tracking using the Lucas-Kanade method. After an initial detection via background subtraction, we initialized tracking by identifying good features within the drone's bounding box. Between consecutive frames, these features were tracked to estimate the drone's displacement, allowing us to update its position even in the absence of fresh detections. We incorporated validation checks to assess confidence in the optical flow results and fall back to background subtraction when necessary. This hybrid approach significantly enhanced the continuity and accuracy of the trajectory, particularly during challenging segments of the video sequence, and reduced the frequency of detection failures.

Figure 2: Drone detection results using background subtraction (top), optical flow tracking (middle), and YOLOv8 (bottom)

To achieve this result, it is also possible to use a deep learning-based object detection algorithm, such as YOLO or SSD, that could give better results in terms of accuracy and robustness. We decided to use YOLOv8, which is a state-of-the-art object detection algorithm. We used the pre-trained weights provided by the `yolov8n`, which is a small and fast version of YOLOv8, without fine tuning.

Results

The drone detection algorithm was evaluated on some videos from the dataset, and the results were imperfect but promising. The main limitations were:

- In the majority of the frames, the algorithm identified the partially visible grass as the drone, because it was moved by the wind. This problem was mitigated by applying a threshold on the area of the detected object, to be set for each video (light blue line in Figure 2)
- The algorithm often mistook the drone for other moving objects in the scene, such as birds. Unfortunately, this problem could not be solved, as the drone was often too small in the frame to be distinguished from other objects. To partially mitigate this issue, we saved also the velocity of the detected object for each frame (as shown in 3), and we think that this information could be used to filter out false positives
- The deep learning-based detection algorithm completely removed the false positives, but it was not able to detect the drone in some frames, especially when the drone was too small in the frame, as shown in Figure 4

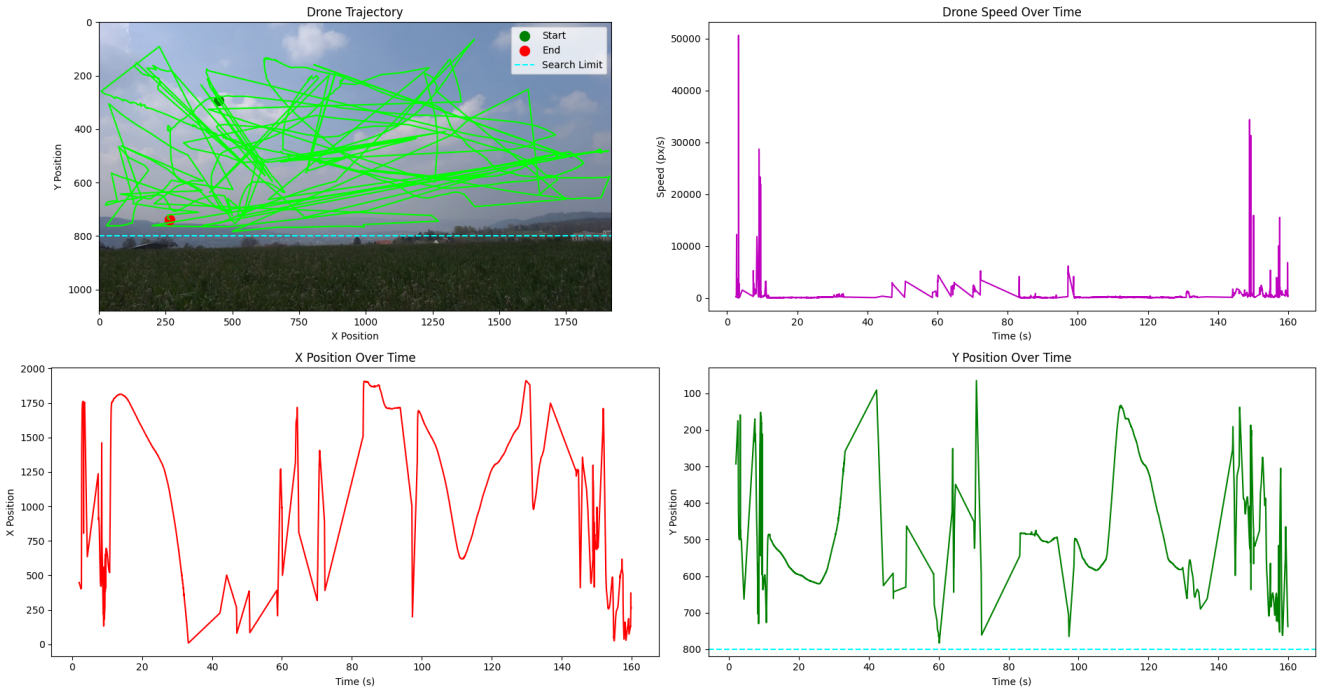


Figure 3: Analysis of the drone detection algorithm on a video from the dataset (Dataset 1, cam 2). It is clear the heavy presence of false positives, as the algorithm often detects birds and other moving objects as the drone. The velocity of the detected object is also shown, which could be used to filter out false positives.

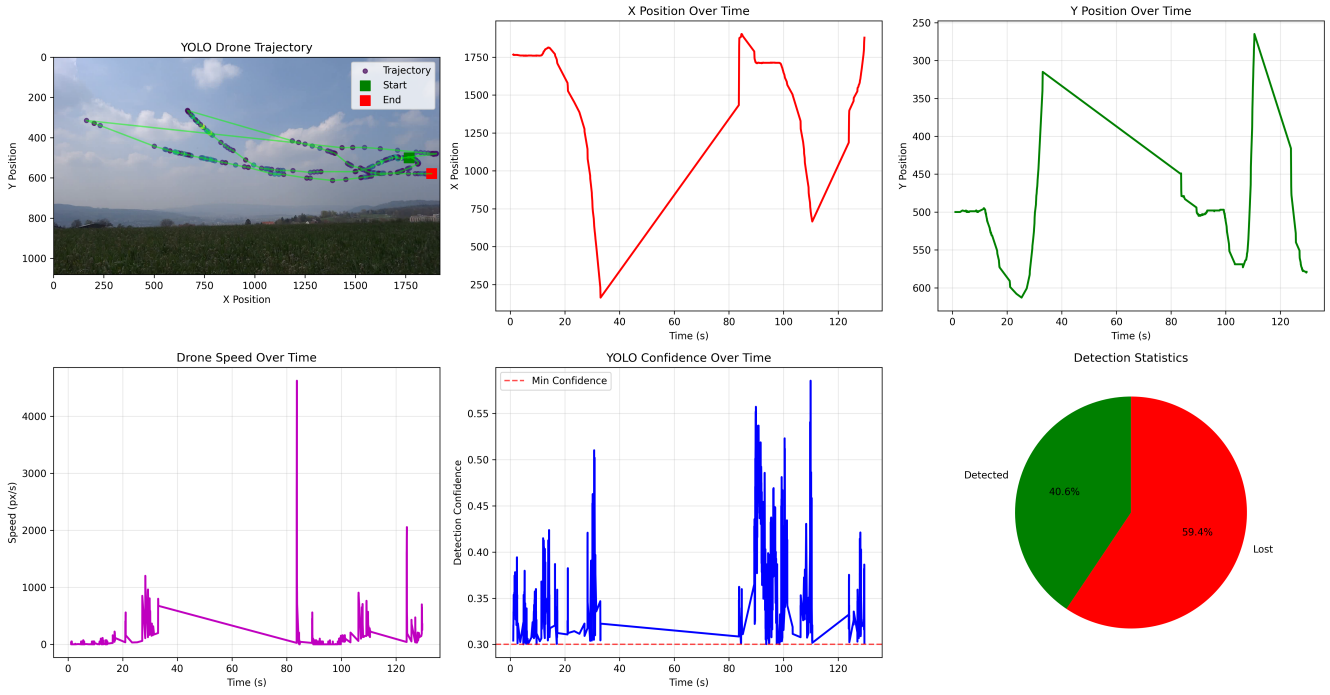


Figure 4: Analysis of the drone detection algorithm using YOLOv8 on a video from the dataset (Dataset 1, cam 2)

Due to the limitations of the detection algorithm, we decided to use the drone detections already given in the dataset. This allowed us to focus on the trajectory reconstruction aspect of the project.

3.3 Offline Trajectory Reconstruction

The next step was to reconstruct the drone's trajectory by loading all the detections in memory and then applying the reconstruction algorithm.

3.3.1 Data loading

The first step was to load the drone detections from the dataset. The dataset provides the drone detections in various `txt` files, divided in the repo by dataset and camera. Each `txt` file was automatically fetched and parsed, extracting the drone detections and frame indexes. In order to correctly handle the data and perform all the necessary operations, we decided to undistort the drone detections using the intrinsic parameters provided in the dataset at this stage. The intrinsic parameters were loaded from the calibration files provided in the dataset, and the drone detections were undistorted using OpenCV's `undistortPoints` function.

The undistorted drone detections were then saved in a `.csv` file, unique for each dataset, with the same columns as the original `txt` files (frame index, x coordinate, y coordinate) with the addition of the camera index. This allowed us to easily access the drone detections for each camera and perform the necessary operations on them, both in the offline and online trajectory reconstruction algorithms.

3.3.2 Temporal shift estimation

One of the main goals in the paper is to have a system that does not require any synchronization between the cameras. This means that the timestamps of the frames are not aligned, and we needed to find the best temporal shift and scale factor between the cameras to align the drone detections.

following the paper notation, we can label each detection j in each camera i with a timestamp

$$t_i^j = \alpha_i j + \beta_i \tag{1}$$

Where β_i is an offset and α_i is a scale factor which toghether map the frame index j to a global time (or, equivalently, to the frame index of a reference camera k such that $t_k^j = j$).

To estimate the temporal offsets (β) and scale factors (α) between cameras, we implemented a straightforward algorithm that performs a brute-force search for the optimal temporal shift for each camera pair. The relative frame rate between cameras is imposed by setting the scale factor as the ratio of their respective frame rates.

For each candidate value of β within a specified range, the algorithm shifts the detections of one camera accordingly and computes the fundamental matrix F between the two cameras using the available drone detections as point correspondences. F is estimated via RANSAC, and the number of inliers serves as a score to assess the quality of the temporal alignment. To refine the estimate, the search is repeated iteratively: after each iteration, the best β is used to define a narrower and finer search range, and this process is repeated for four iterations.

An example of the results of this algorithm is shown in Figure 5

Figure 5: Example of the results of the temporal shift estimation algorithm (dataset 4, cameras 0-5). The x-axis shows the temporal shift β in seconds, while the y-axis shows the number of inliers found for each candidate value of β. It is clear that the algorithm finds a peak in the number of inliers for a specific value of β, which is the best temporal shift between the two cameras.

To establish correspondences between cameras, we adopted the approach described in the paper: for each detection in camera i, we search for a temporally overlapping set of consecutive detections in camera j (after applying the current temporal shift). If such an overlap exists, we fit a spline to the detections in camera j and evaluate it at the timestamp of the detection in camera i. This provides a set of matched points that can be used to compute F[1].

[1]In order to decrease the computational effort, we decided to pre-compute and save all 2D splines only once, and then shift the timestamps when needed

This method, while simple, proved effective for aligning the detections across unsynchronized cameras and enabled robust estimation of the temporal shifts required for subsequent trajectory reconstruction.

From then on, each camera pair was processed in order of decreasing number of inliers, so that the cameras with the most reliable detections were processed first. This was done to ensure that the trajectory reconstruction was as accurate as possible.

3.3.3 First camera pose estimation

We started the trajectory reconstruction by estimating the relative pose of the best camera pair, which was the one with the most inliers in the estimation of F. We referred to the camera whose detections had been interpolated with the spline as the `main_camera` (mc), and the other one as the `secondary_camera` (sc).

The following step was to compute the essential matrix E. Since all the cameras were precalibrated, this could be done with the simple formula

$$E = K_{mc}^T \cdot F \cdot K_{sc} \tag{2}$$

Once we obtained `E`, it was possible to recover the reciprocal pose of the two cameras thanks to OpenCV's function `cv.recoverPose`.

Since the relative positions could only be determined up to an unknown scale, we set the `main_camera` pose at the origin with an identity rotation. All subsequent positions were then expressed relative to the baseline distance between these two cameras.

Once the 3D poses of the two cameras had been established, we could compute the projection matrices `P` for both cameras using the definition

$$P = K \cdot [R|t] \tag{3}$$

And, consequently, we used them to triangulate the 2D-2D correspondences, finding an initial set of 3D points.

3.3.4 Other camera registration

The process of registering additional cameras proceeded iteratively, selecting at each step the unregistered camera (`nth_camera`, nc) with the highest inlier ratio with respect to another in the set of already localized cameras (new `main_camera`).

When a new couple was selected for localization, the same registration procedure was used: all 3D point and spline timestamps were mapped into the time frame of mc, which then temporarily served as the global time reference for this step. For each detection in nc, we searched for a temporally overlapping 3D spline and evaluated it at the timestamp of the detection in the camera. This provided a set of 3D-2D correspondences that could be used to localize nc into the global frame using the PnP algorithm, with RANSAC employed to handle outliers.

3.3.5 3D Splines extension

Once each new camera was registered, we could extend the 3D splines to include the new detections. This was done by triangulating the new 2D-2D correspondences in the pair mc - nc, and then checking if the new 3D points extended the existing splines' time range. If they did, we extended the splines to include the new points, otherwise we simply added the new points to the existing splines. If a segment of new 3D points were able to bridge the gap between two existing splines, we merged the two splines into one. This was done to ensure that the 3D splines were as continuous as possible, and to avoid having too many small segments. This process is illustrated in Figure 6

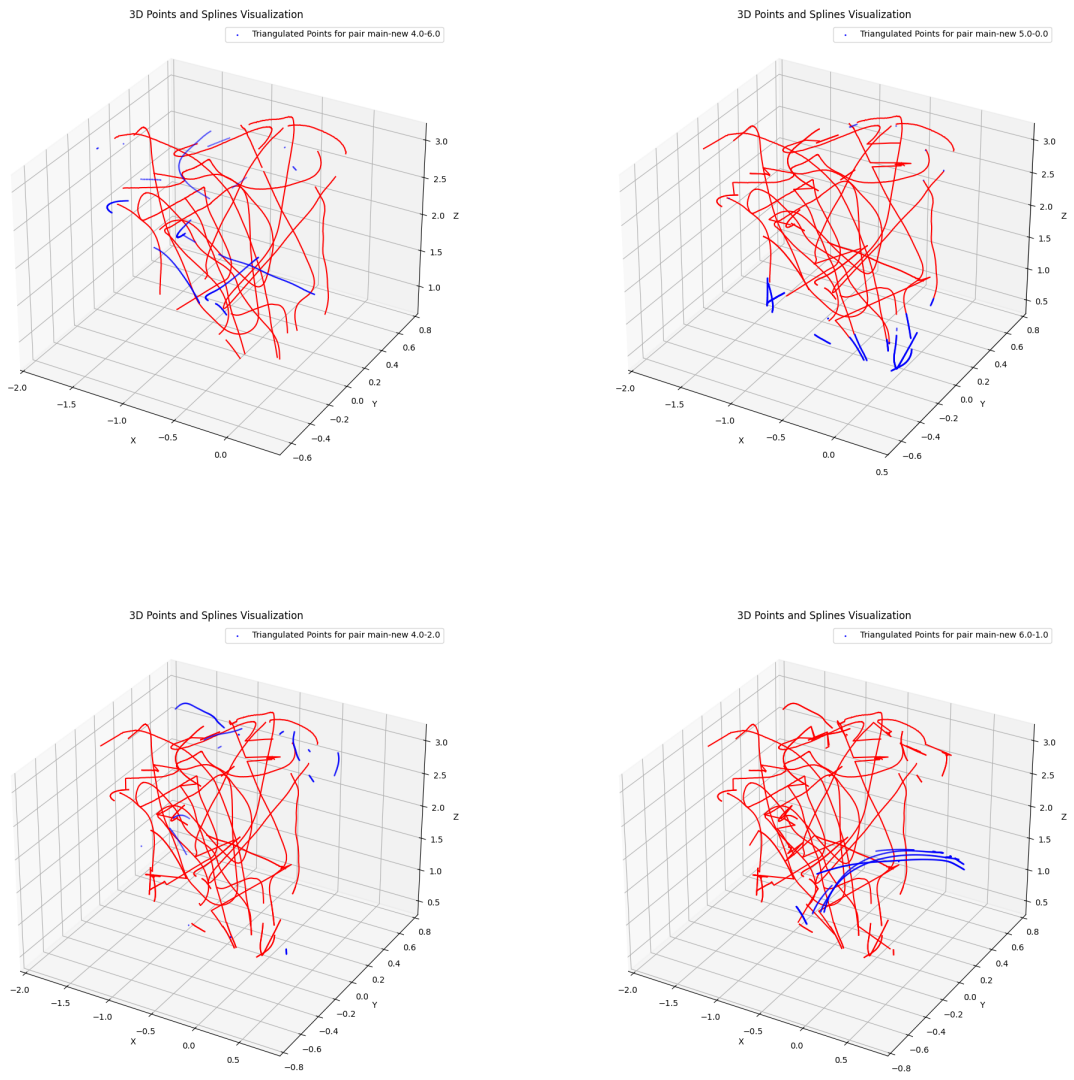


Figure 6: Example of how the 3D splines are extended with the addition of new cameras (dataset 4). When a new camera is registered, the already existing 3D splines (red) are extended to include the new 3D points (blue)

3.3.6 Bundle Adjustment

Since the procedure described above was prone to accumulate errors, we applied a bundle adjustment step to refine the 3D points and camera poses. This was done by interpolating the 3D splines at the timestamps of the detections in each camera, and then using the 2D-3D correspondences to minimize the reprojection error. The optimization was performed by Python's `scipy.optimize.least_squares` function, which allowed us to minimize the error in a robust and efficient way. The effects of the bundle adjustment can be seen in Figure 12.

The bundle adjustment was performed for each registered camera every time a new camera was added.

After all cameras were registered, we should have had a complete 3D trajectory of the drone, with all cameras registered in the same global frame and all 3D points visible from at least two cameras. However,

the implemented approach suffered from some limitations, that we are going to discuss in section 4.

3.4 Online Trajectory Reconstruction

The paper also suggest the implementation of an online trajectory reconstruction algorithm, that is able to reconstruct the drone's trajectory in real-time as it flies.

3.4.1 Environment and data loading

In this case, we needed to simulate the stream of data from the cameras, as if it were coming from a real-time video feed. To do this, we choose to use ROS2 as the framework to handle the data stream, and we implemented a ROS2 node that receives the drone detections from every camera and reconstructs the trajectory in real-time. To simulate a real-time data stream, we performed the following data preprocessing steps:

- We loaded the drone detections from the `.csv` file described in Section 3.3.1.
- For each camera and each detection, we computed the global timestamp by applying the α and β parameters provided in the dataset, ensuring robustness to timing inconsistencies.
- We sorted all detections by their global timestamps and created a list of messages to be stored in a ROS2 bag file.

Note: since in this approach we simulated the real-time data stream, we did not need to estimate the temporal shifts between cameras, as the timestamps were already aligned.

During the bag replay, all the logic for the trajectory reconstruction was implemented in a single ROS2 node, which subscribed to the drone detections.

3.4.2 Data handling

Since the data was received in real time, it was essential to handle the data stream efficiently. To achieve this, we implemented a subscriber that collected drone detections from each camera and stored them in a buffer. The buffer was organized as an array, where each element corresponded to a specific camera and contained a list of detections grouped by contiguous timestamps. This structure enabled fast access to the detections from each camera and made it easy to identify timestamp segments that were ready for spline interpolation without the need for repeatedly searching through the entire buffer.

3.4.3 Cameras localization

In order not to lose any incoming message during the trajectory reconstruction, all the logic for the camera localization and trajectory reconstruction was implemented in a separate thread. The tasks to be performed in this thread were the following:

After a sufficient number of detections had been collected in the buffer, the algorithm would find the correspondences between cameras and, if the number of correspondences between two cameras was sufficient, it would compute the fundamental matrix F.

If F brought to a sufficient number of inliers, the algorithm would compute the essential matrix E and recover the relative pose of the two cameras in the same way as in the offline trajectory reconstruction.

After the relative pose of the two cameras was computed, the algorithm would generate the 3D positions of the drone by triangulating the 2D-2D correspondences between the two cameras, and then generate the initial 3D splines for the drone trajectory.

Periodically, the algorithm would check if cameras not yet registered had enough correspondences with the 3D splines. If so, it would compute the relative pose of the new camera with respect to the global frame using the PnP algorithm

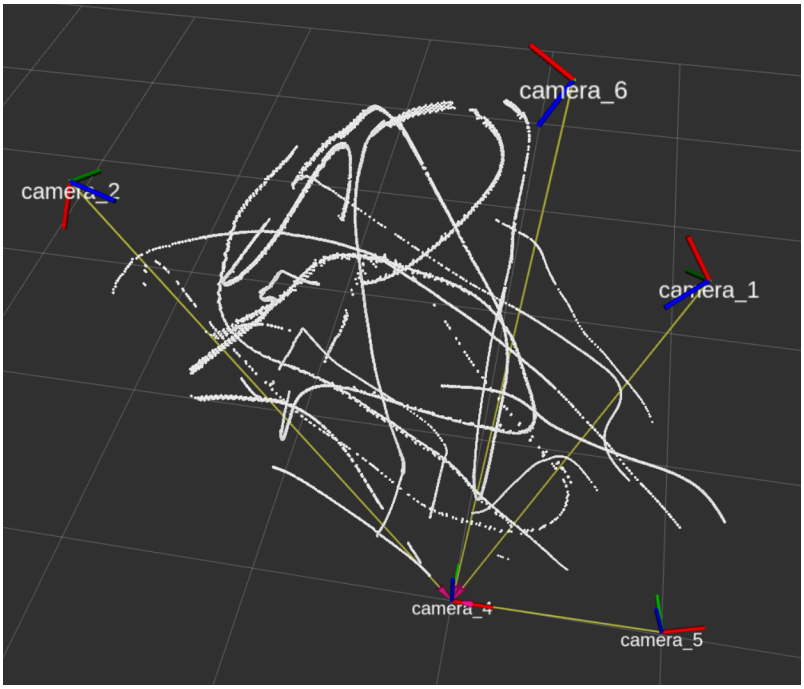


Figure 7: Example of the 3D trajectory reconstructed by the online algorithm (dataset 4). The drone positions are shown as white dots, while the cameras are shown as ROS tf frames

3.4.4 3D Splines extension

In parallel with the camera localization, the algorithm would also keep up to date the 3D trajectory of the drone in the following way:

- For each new detection received from a camera pc, the algorithm would interpolate with a 2D spline the last 5 detections of the drone in pc (if the detections were not too far apart in time)
- For every other camera i, if their last detection was within the 2D spline time range, the algorithm would evaluate the spline at the timestamp of the detection and generate a 2D-2D correspondence.
- Every correspondence $pc - i$ would be used to triangulate a 3D point, then these points would be averaged and added to the 3D splines of the drone trajectory.
- If the new 3D point was sufficiently close to last spline segment in time, the algorithm would extend the spline to include the new point.

This approach allowed the algorithm to keep the 3D trajectory of the drone up to date in real-time, while also registering new cameras as they became available.

3.4.5 Bundle Adjustment

As the paper suggested, the algorithm would periodically perform a bundle adjustment to refine the 3D trajectory of the drone and the camera poses. This was done in the same way as in the offline trajectory reconstruction (same code), and it was performed periodically (every 1000 detections) and in a separate thread to avoid blocking the main thread that was receiving the drone detections.

4 Limitations of the implementation

The implemented pipeline, while functional, has several limitations that should be mentioned:

4.1 Drone Detection

The limitations of the drone detection algorithm were already discussed in Section 3.2

4.2 Offline Trajectory Reconstruction

during the β search phase, some of the smartphone cameras were recording at a variable frame rate. In some cases the fluctuation in FPS was very high, e.g. Mate 7 and Mate 10 in this dataset. For this reason, the algorithm was not able to find a peak in the number of inliers for these cameras, as shown in Figure 8. This is because our implementation relies on the assumption that the frame rate is constant, and therefore it cannot handle cameras with variable frame rates.

The authors of the paper suggest to remap the detected points to a fixed frame rate with a linear interpolation, but we did not implement this step in our code. Instead, we simply ignored these cameras, as they were not essential for the trajectory reconstruction. An effect of this is shown in Figure 8, where the algorithm fails to find a peak in the number of inliers for each camera pair including the above mentioned cameras.

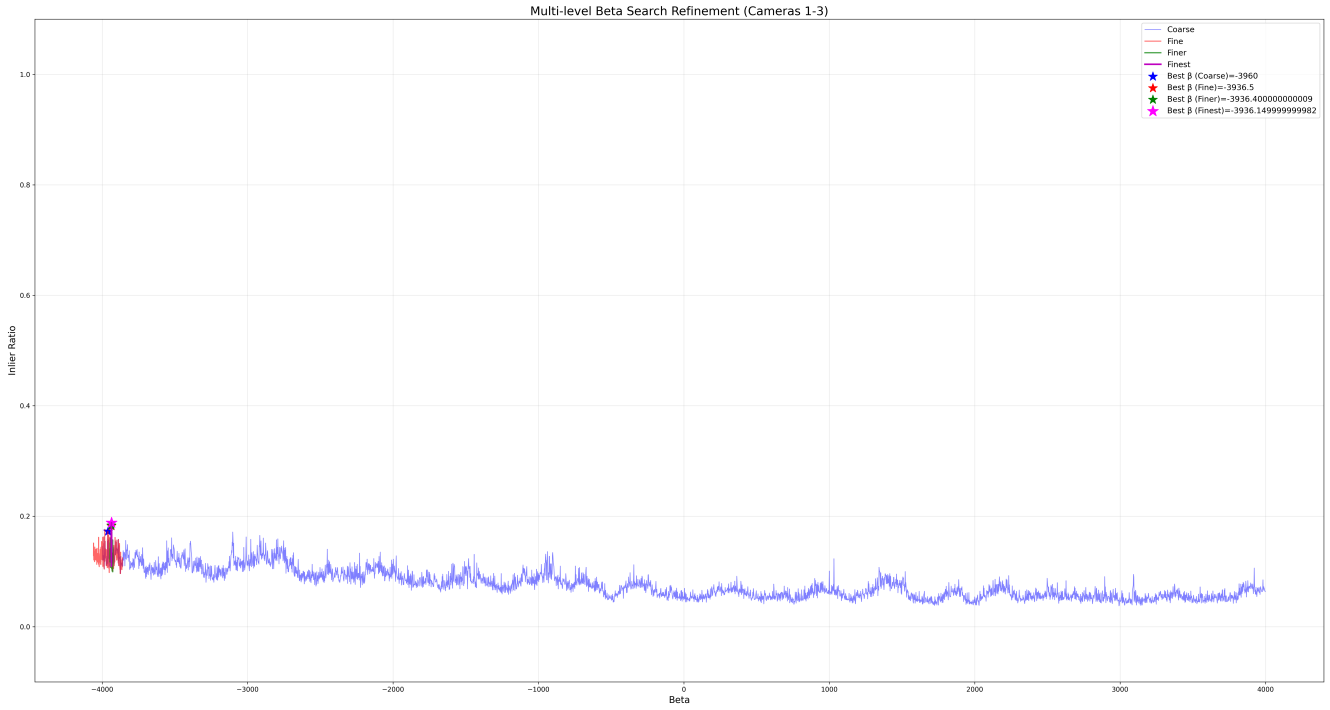


Figure 8: Example of the results of the temporal shift estimation algorithm (dataset 4, cameras 1-3). Due to the variable frame rate of the camera, the algorithm fails to find a clear peak in the number of inliers

Another limitation of the offline trajectory reconstruction is that the process of camera localization was prone to accumulating errors. As a result, when a new camera was registered, the 3D splines were not always continuous, and the reconstructed trajectory could be inaccurate. This led to temporal discontinuities in the splines, which in turn reduced the effectiveness of the bundle adjustment step. These issues are partially visible in Figure 12, where the bundle adjustment does not fully correct the reprojection errors. They are also evident in Figure 9, which shows discontinuous 3D splines and an imprecise trajectory, and in Figure 14, where the bundle adjustment fails to refine the camera position.

This problem triggered a cascade effect: the inaccuracies in the 3D splines led to poor triangulation of new points, which in turn caused further inaccuracies in the camera poses.

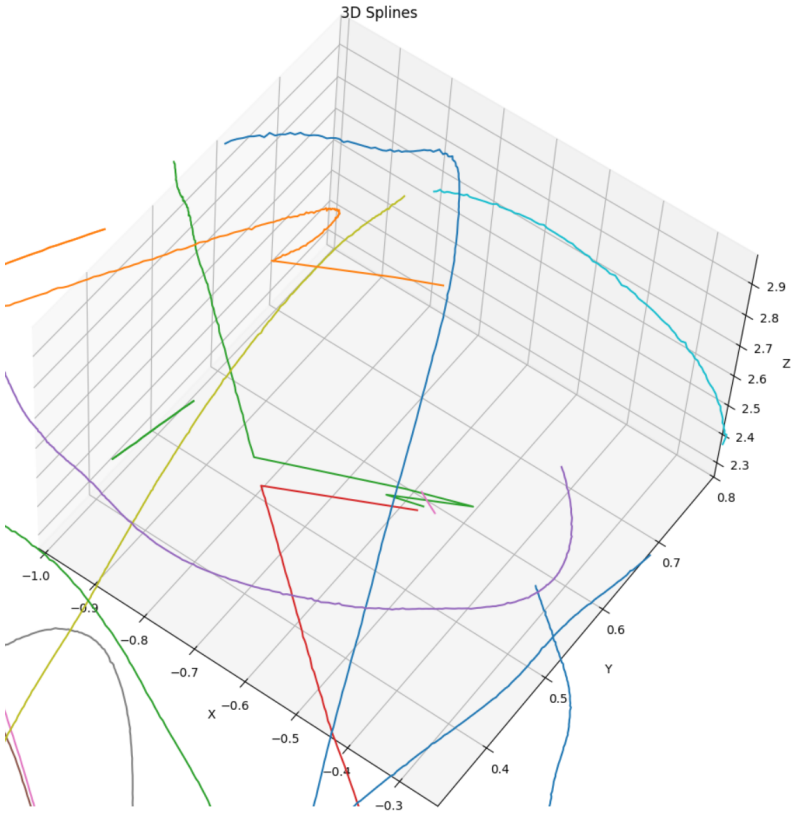


Figure 9: Example of the discontinuity in the 3D splines (dataset 4)

4.3 Online Trajectory Reconstruction

The online trajectory reconstruction algorithm suffered from similar limitations as the offline one, and they were even more pronounced due to the real-time nature of the implementation.

Since the dataset provided ground truth for α, β parameters only for datasets 3 and 4, we could test the online trajectory reconstruction only on these datasets.

The lower number of correspondences available for each camera pair made the computed F less reliable, and the algorithm was more prone to accumulating errors. This led to a more pronounced drift in the 3D trajectory (Figure 10), which was not corrected by the bundle adjustment step.

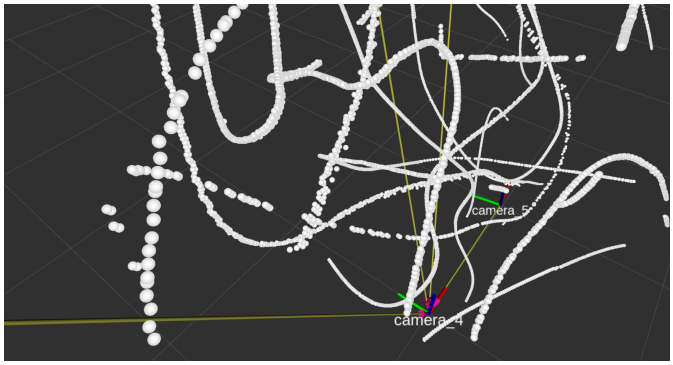


Figure 10: Example of the noisy drone detections in the online trajectory reconstruction (dataset 4)

The 3D splines were so noisy that the bundle adjustment step actually worsened the results, intro-

ducing more errors instead of correcting them. This effect is illustrated in Figure 13. This cycle of errors often led to the inability to produce a trajectory at all (Figure 11).

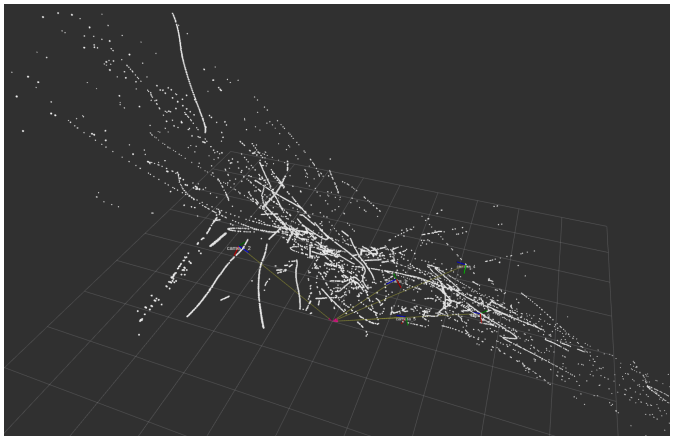


Figure 11: Example of the algorithm failing to produce a trajectory due to bundle adjustment introducing too many errors.

A new issue was also introduced by the real-time nature of the implementation: the algorithm was not able to correctly handle the fisheye camera (gopro3); we think that this was due to the fact that the fisheye camera was calibrated as a pinhole camera, and therefore the computation of its distortion coefficients in the intrinsic calibration step was not accurate enough. To mitigate this issue, we decided to ignore the fisheye camera in the online trajectory reconstruction, as it was not essential for the trajectory reconstruction. In future work, we could try to distinct between pinhole and fisheye cameras, and apply a different rectification method for each type of camera.

The dataset also provided the ground truth trajectory for the drone in some case. Unfortunately, due to the limitations above mentioned for both methods, the reconstructed trajectory was not accurate enough to be compared with the ground truth, so we had to skip the comparison step. However, we can still qualitatively assess the performance of the algorithm by visualizing the reconstructed trajectory in 3D space, as shown in Figures 7, 6.

5 Instructions to run the code

Since we implemented both the offline and online trajectory reconstruction algorithms, we provided two different scripts to run them.

5.1 Environment setup

Since the project relies on the dataset provided by the authors of the paper that is publicly available on GitHub (and, moreover, quite large), we decided to not include it in the repository. We directly cloned the dataset repository in the root directory of our project, and made some little modifications:

- We extracted each video from the original dataset and saved each video in the same folder as their compressed version
- Some cameras in the dataset are provided with two versions of calibration files (e.g. mate10): for a easier automatic loading of the calibration files, we kept only one of the two versions, removing the suffix `_1` or `_2` from the file name. This way, the code can load the calibration files without having to specify which version to use.

Said that, in order to run the code, you need to have all the Python dependencies installed, this can be done by running the following command in the root directory of the project:

```
pip install -r requirements.txt
```

5.2 Camera Intrinsic Calibration

The camera intrinsic calibration can be run by executing the following command:

```
python camera_calib.py
```

This script will automatically load the checkerboard images from the dataset, detect the corners, compute the intrinsic parameters, and save both the intrinsic parameters (in a `json` file) and the plots showing the comparison between the computed intrinsic parameters and the ground truth values provided in the dataset. The plots will be saved in the `plots/intrinsic_calibration/` folder.

5.3 Drone Detection

The drone detection algorithm can be run by executing a single Python script, with appropriate arguments:

```
python moving_video.py [-h] [--y_limit Y_LIMIT] dataset camera

Drone tracking with background subtraction and optical flow.

positional arguments:
  dataset            Dataset number
  camera             Camera number

options:
  --y_limit Y_LIMIT  Y limit for search area (default: 1080)
```

This script will save in the `plots/` folder a `.mp4` video with the drone detections overlaid on the original video (image 2 is a frame of this video), a `json` file with the drone detections, and an image with the analysis of the results of the background subtraction algorithm (image 3), and the same set of files for the optical flow algorithm.

5.4 Offline Trajectory Reconstruction

The offline trajectory reconstruction can be run in a similar way, by executing the following command:

```
python main.py [-h] dataset_no

3D Spline Reconstruction

positional arguments:
  dataset_no  Dataset number to use
```

This script will execute all the steps described in above, saving all the results in the `plots/` folder. To see the results, in a more interactive way, you can uncomment the very last line of the script, which will display all the intermediate results (Note: this will open a lot of windows, so it is recommended to comment it back after you are done).

In the first step, the script will check the existence of a `.csv` file with the results of the β search, and if it does not exist, it will run the temporal shift estimation algorithm, that is quite time consuming, even if it is heavily parallelized. For this reason, we suggest not to delete the `.csv` files named `beta_results_dataset_X.csv`.

5.5 Online Trajectory Reconstruction

The online trajectory reconstruction is a bit more complex, as it requires ROS2 to be installed and running.

The code is structured in a ROS2 package, that has to be put in the `src/` folder of a ROS2 workspace. To run the online trajectory reconstruction, you need to follow these steps (assuming you are in the ROS2 workspace root directory):

```
colcon build
source install/setup.bash
ros2 run drones drones
ros2 bag play <path_to_the_bag_file> -r 1.5
```

This will start the ROS2 node that reconstructs the drone trajectory in real-time, and it will replay the bag file containing the drone detections (the bag file is already provided in the repository). The rate of the replay can be adjusted with the `-r` option, but we suggest to keep it at 1.5x the original speed, as this is the speed that most closely resembles the real-time time flow of the dataset.

To keep the node implementation as simple as possible, instead of following the ROS2 conventions and having a launch file and a configuration file, we decided to hardcode the parameters in the node itself. This means that you will have to modify the first lines of the `drones.py` file to change the parameters of the algorithm.

6 Conclusions

This project presents a robust and modular implementation of a drone trajectory reconstruction pipeline, inspired by state-of-the-art methods in outside-in tracking with unsynchronized cameras. The implementation covers key components, with careful integration of computer vision tools and frameworks like OpenCV and ROS2. Although the pipeline suffers from several limitations, it demonstrates strong potential for scalable, low-cost UAV tracking in real-world conditions.

Future improvements could focus on refining the bundle adjustment process to incorporate motion regularization, improving robustness to camera variability, and enhancing the detection component for better performance in occluded or low-visibility scenarios. Despite the challenges, this work serves as a strong foundation for further research and development in drone tracking systems using consumer-grade equipment.

Bulky Plots

In order to keep the report clean and readable, we decided to move the bulkiest plots to the end of the document

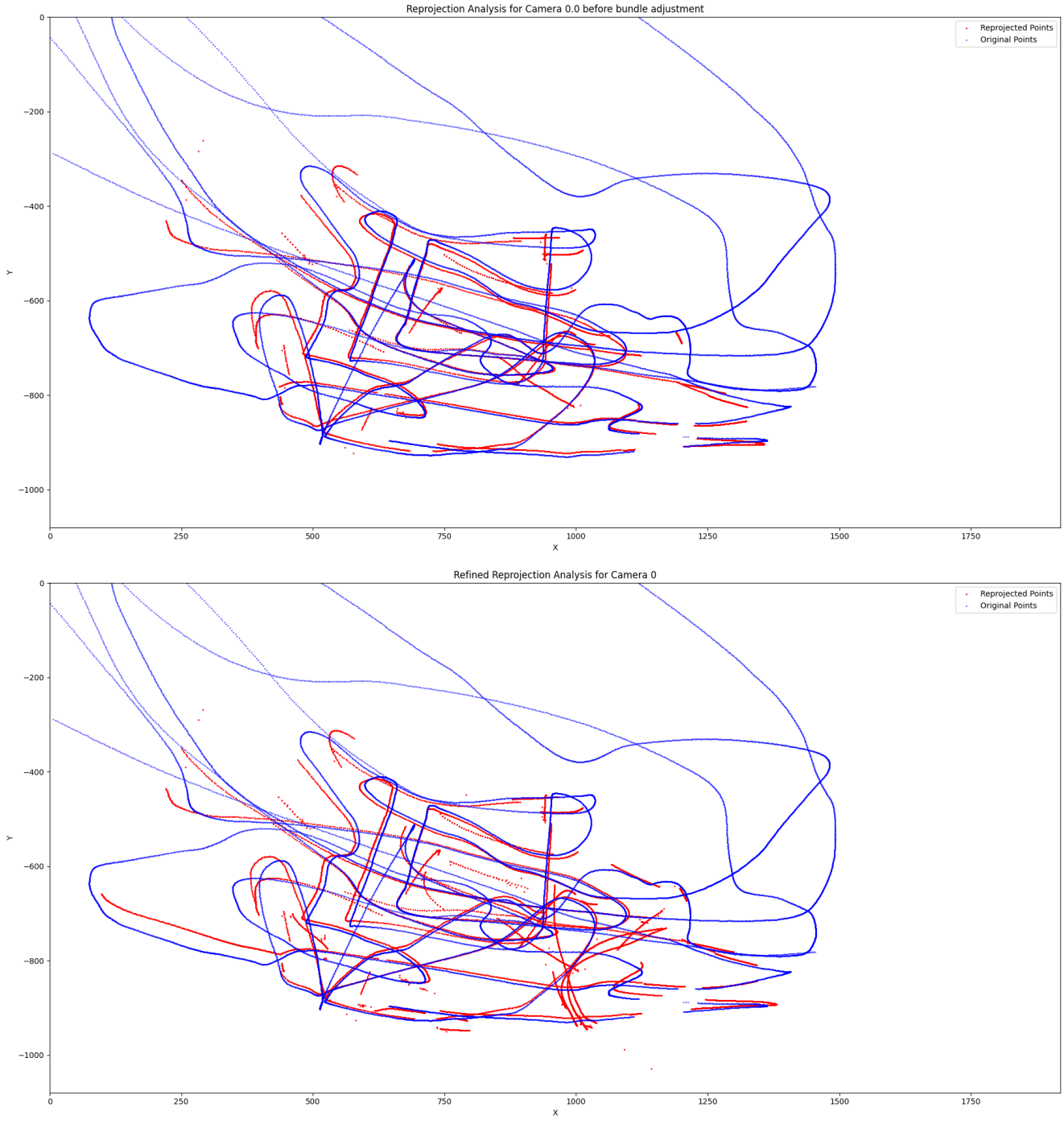


Figure 12: Offline approach - Reprojection error analysis for camera 0 (dataset 4). The top plot shows the reprojection error before the bundle adjustment, while the bottom plot shows the reprojection error after the bundle adjustment

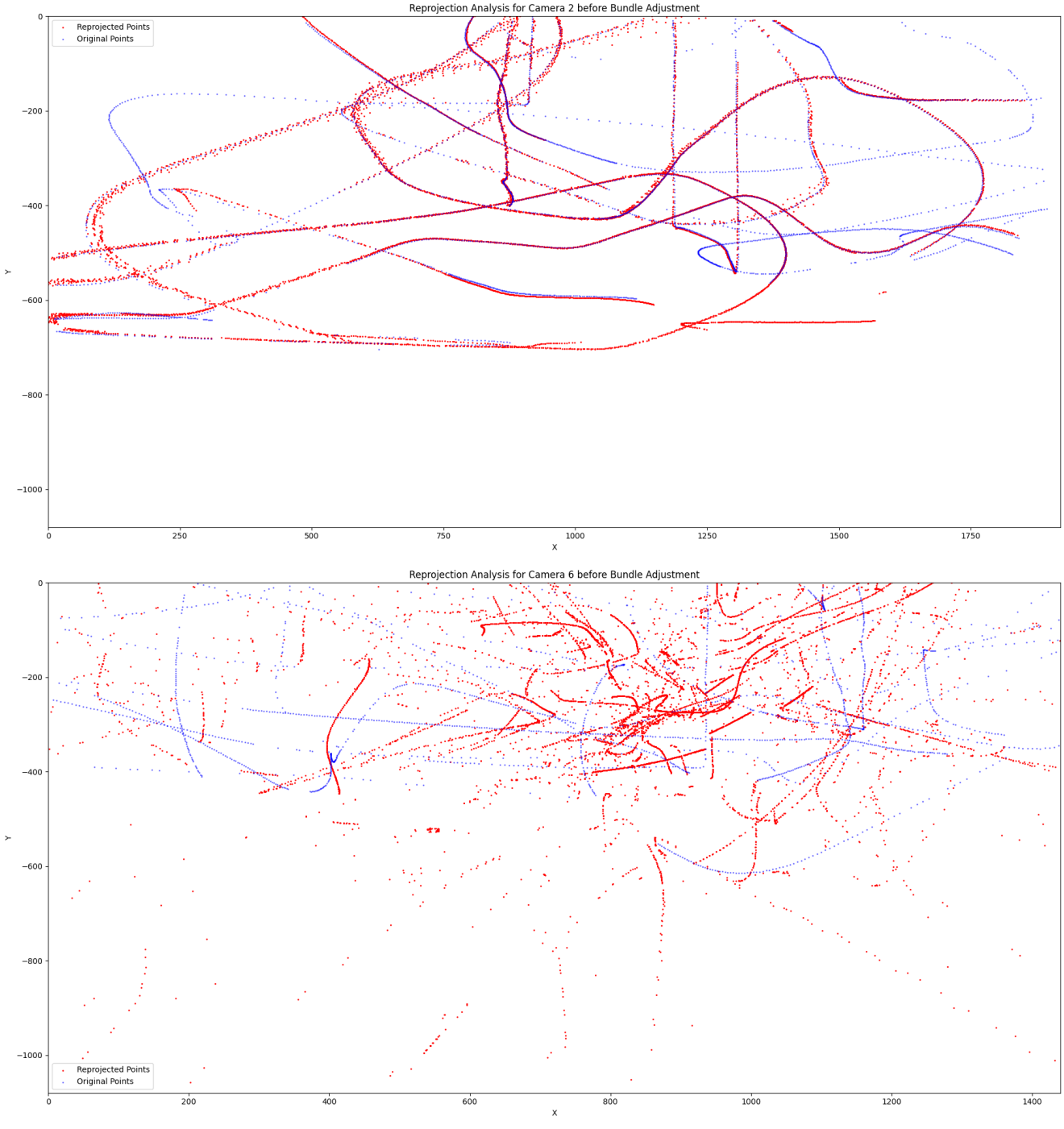


Figure 13: Online approach - Reprojection error for camera 2 (dataset 4): before (top) and after (bottom) bundle adjustment. The bundle adjustment worsens the results, most likely due to the noisy (although plausible) 3D splines

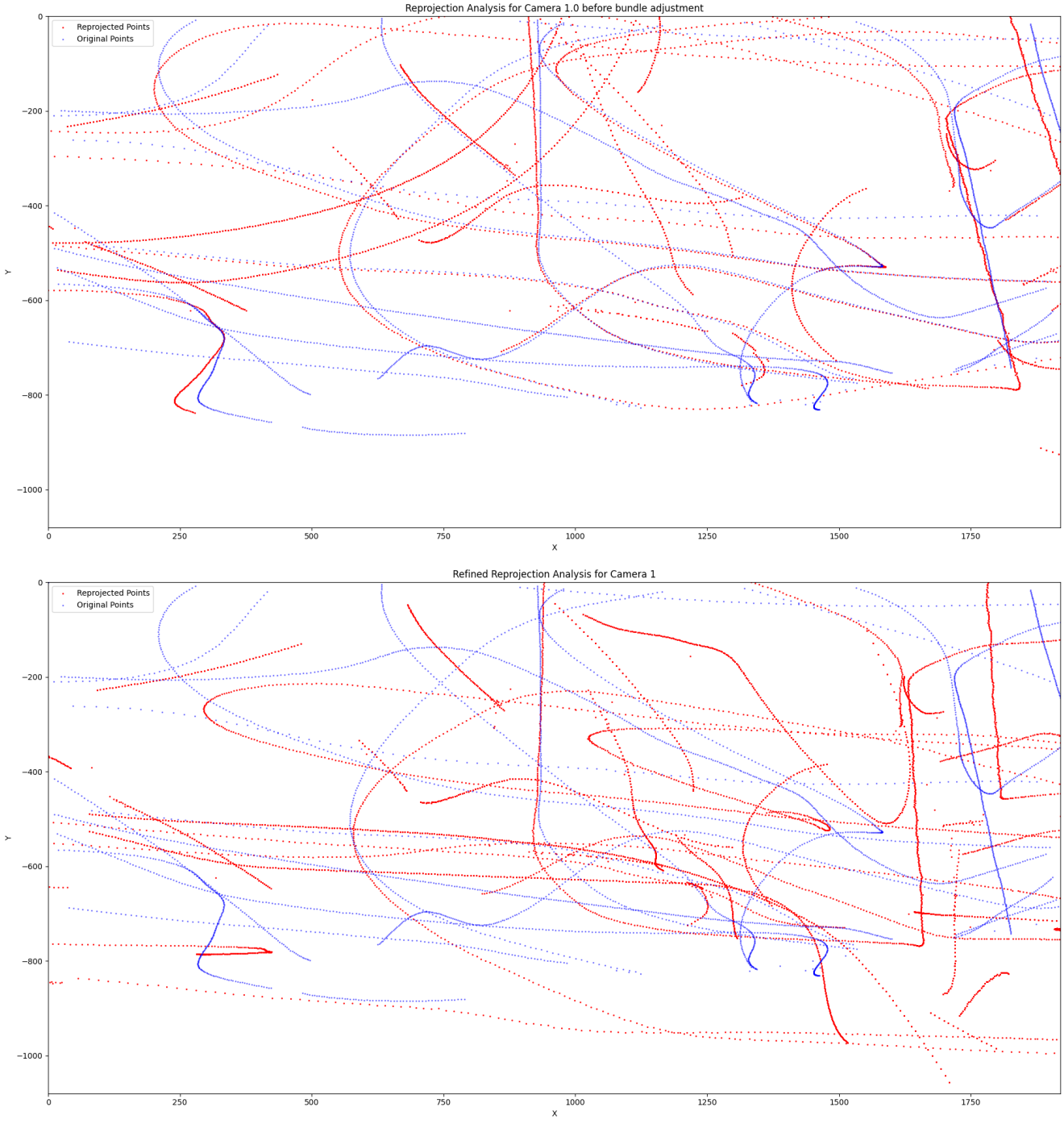


Figure 14: Offline approach - Reprojection error analysis for camera 1 (dataset 4). The top plot shows the reprojection error before the bundle adjustment, while the bottom plot shows the reprojection error after the bundle adjustment. It is clear that the bundle adjustment completely fails to refine the trajectory, as the 3D splines are not consistent