Evaluating Visual Odometry for Future Autonomous Vehicle Guidance

hzwr87

Pre-processing

Following an examination of the dataset, the first decision that was made was to remove the lower part of the images, as this contained the bonnet of the car and the shadow of the car and cameras, which would have features that would either not move or move differently to the rest of the features in the image, leading to confusing results.

Feature Extraction and Matching

The approach chosen for this implementation of visual odometry was monocular odometry. Taking inspiration from the examples provided[1, 2], the feature detector used was sparse Optical Flow, using the Lucas-Kanade algorithm. This is backed up by the OpenCV implementation of the FAST feature detector, which is used when the number of matching features that the optical flow algorithm finds is insufficient. When this number of features is deemed insufficient is one of the tunable parameters associated with this implementation. When the FAST feature detector is used, the features are placed into bins in order to prevent features becoming too concentrated in a specific area. The effect of this is shown in Figure 1 below. The feature rich areas shown here would have even more features had binning not been implemented.

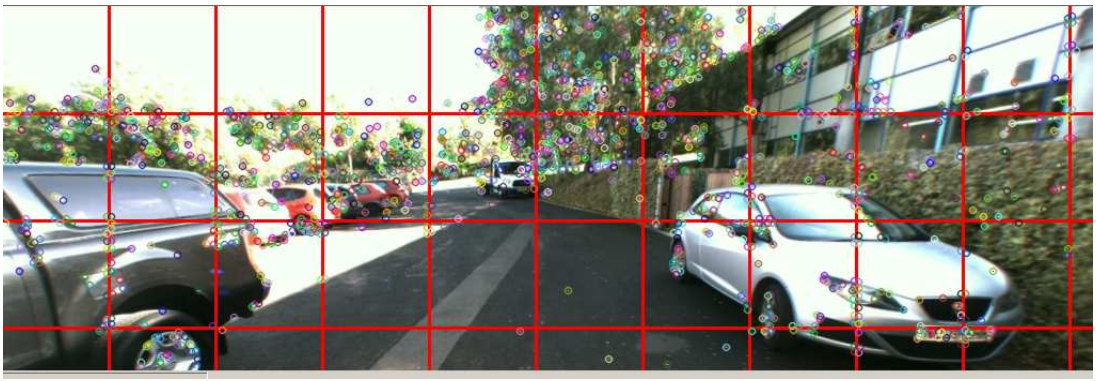


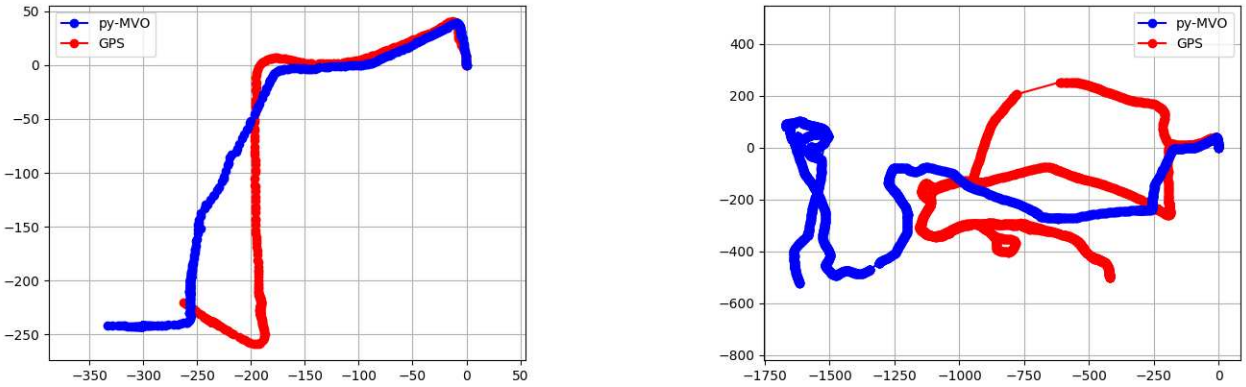
Figure 1: Binning of FAST Features

From these features, we can use included OpenCV functions to calculate the Essential Matric and recover the rotation and translation (R and t). The scale, or approximate number of meters moved between images used to update R and t, is calculated from the ground truth GPS data. To try and cut down on the number of erroneous results and the number of unnecessary computations, we ignore any images with a very small scale value, as this indicates that the car has not moved very far, and is most likely stationary in traffic, following examination of the dataset.

Another important consideration is to only include results where there is a significant forward component to the translation vector. One of the reasons that this has been implemented is possible interpretations of sideways or backwards motion caused by cars travelling in the other direction. The reason that we do not require the forward motion to be dominant, just significant, is that we still want to be able to recognise when the car is going round a corner, but ignore erroneous readings, many of which have negative forward components. The interpretation of "significant" here was determined by experimentation.

Performance

The performance of this system has been evaluated in several different ways. The first of these was using a matplotlib graph to try and compare the shape of this implementation to that of the ground truth GPS data. The purpose of this was to assist early in development and determine if the shape of the trajectory that the odometry was finding was sufficiently close to the shape of the ground truth data, particularly in the first part of the dataset, as this was mainly used for initial testing as it is one of the easier parts, with clearly defined corners, as shown in Figure 2(a).



(a) First part of the dataset, used for testing

(b) Complete Dataset

Figure 2: MatPlotLib plots

The second evaluation method was to convert all of the translations given by this implementation back into GPS points and overlay them on a map of Durham, along with the ground truth data, to determine whether the path that the system had found was accurate in a manner that is more understandable and real-world relevant than the previous plots. This is show in Figure 3.

Figure 3: Map of whole dataset performance

As shown in the figures above, the trajectory of the odometry results goes quite far away from the ground truth, but the general shape of the trajectory is quite close to being correct, it just gets some of the corners wrong. Therefore, when it makes these mistakes, we correct to ground truth data as a post-processing step. The graph in Figure 4 shows the number of corrections required when the euclidean distance between the odometry point and the ground truth point goes about a certain distance threshold. The effect of these corrections is shown in the MatPlotLib (Figure 5) and Google Maps (Figure 6) plots below. As we can see, the results of this improve the performance considerably, even when the distance threshold is large. The full dataset contains 2898 frames, and the GPS ground truth data is only accurate to 5 meters, so the fact that under 10% of the frames need to be corrected (203) when the distance threshold is 5 metres is a sign that the odometry is performing fairly well over the vast majority of the dataset.

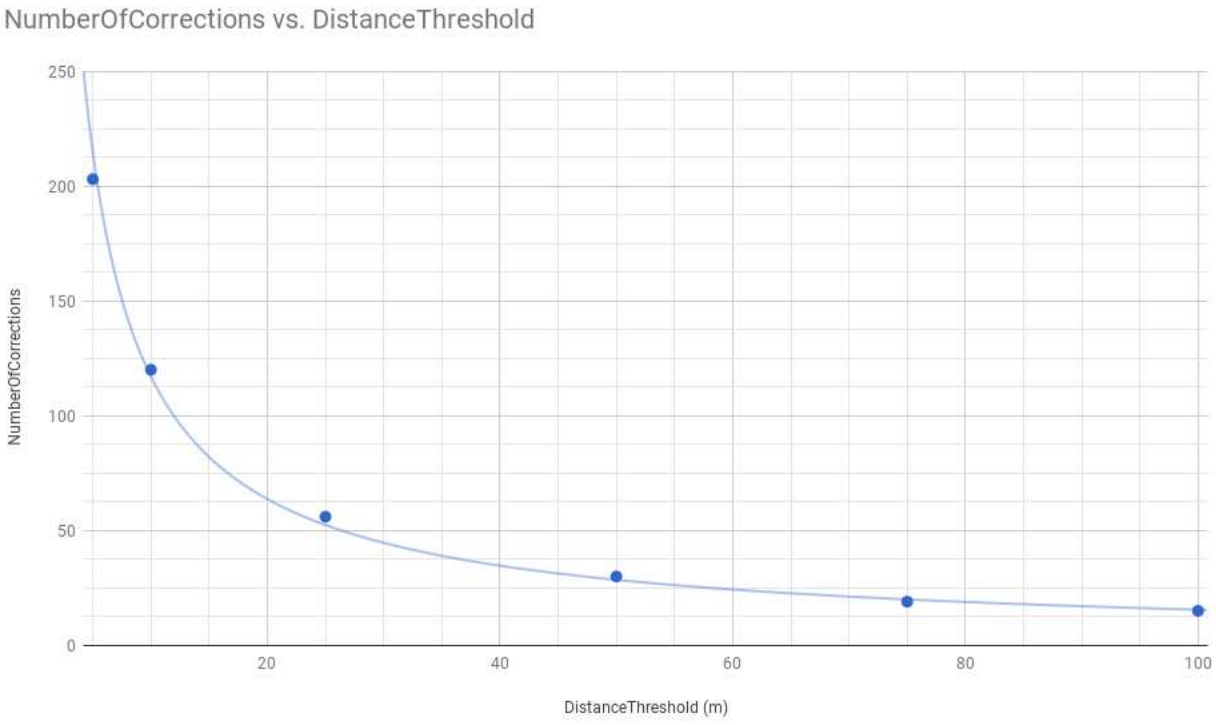
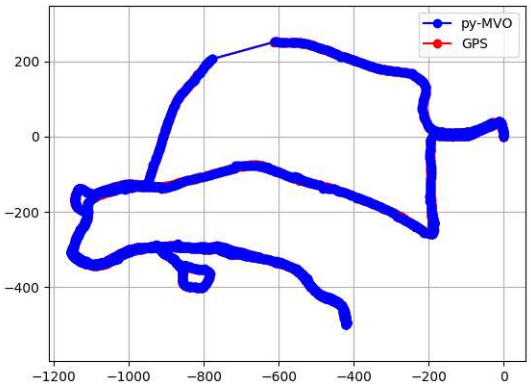
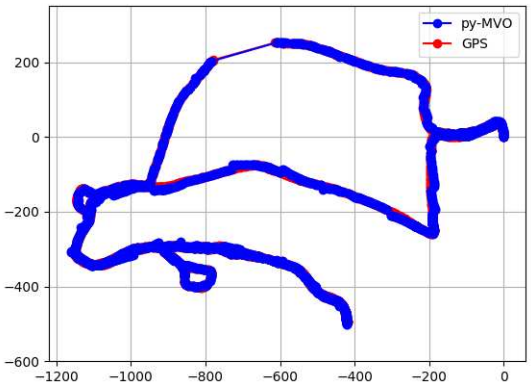


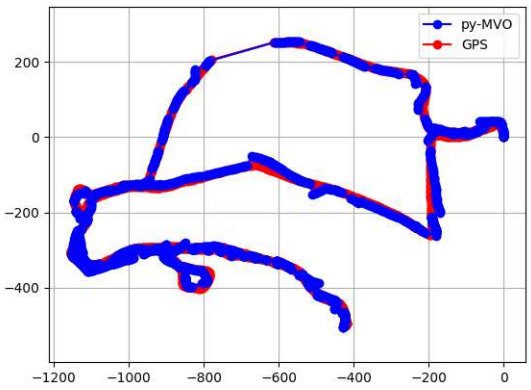
Figure 4: Graph of number of corrections necessary



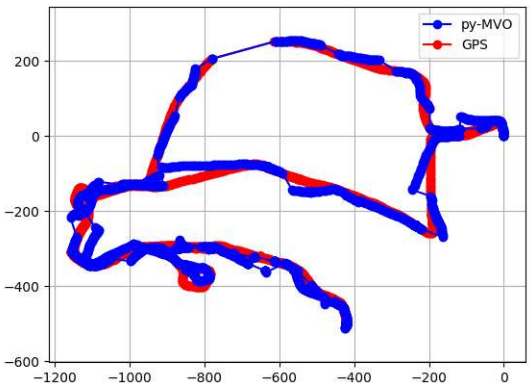
(a) Distance threshold of 5



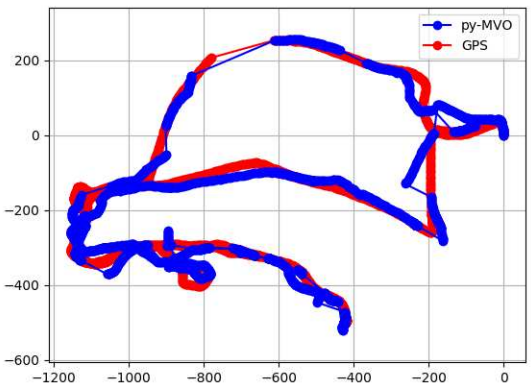
(b) Distance threshold of 10



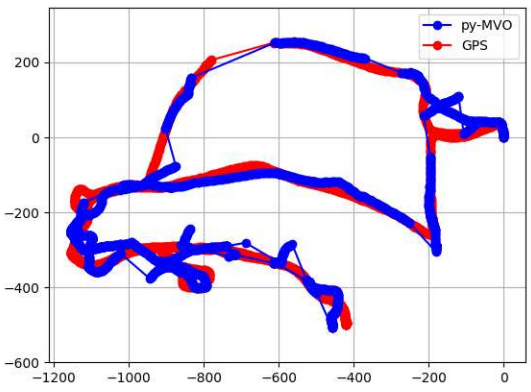
(c) Distance threshold of 25



(d) Distance threshold of 50

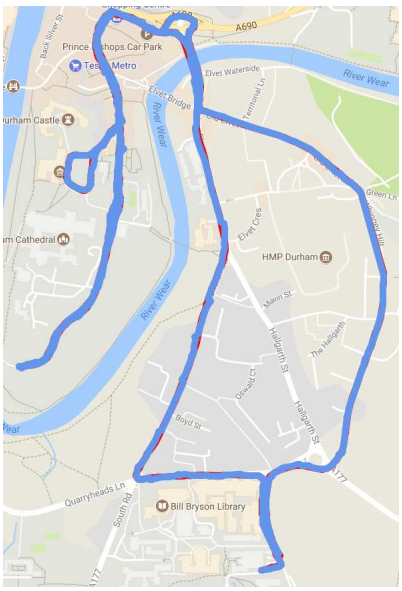


(e) Distance threshold of 75

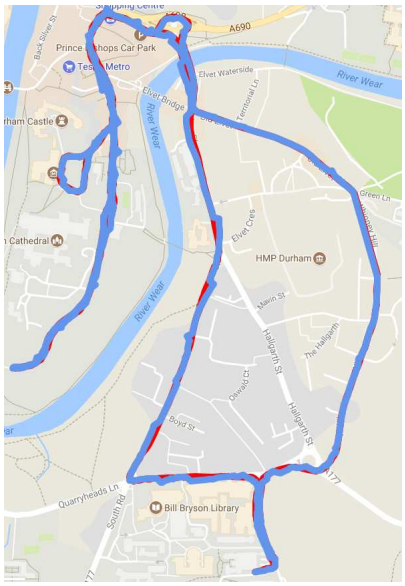


(f) Distance threshold of 100

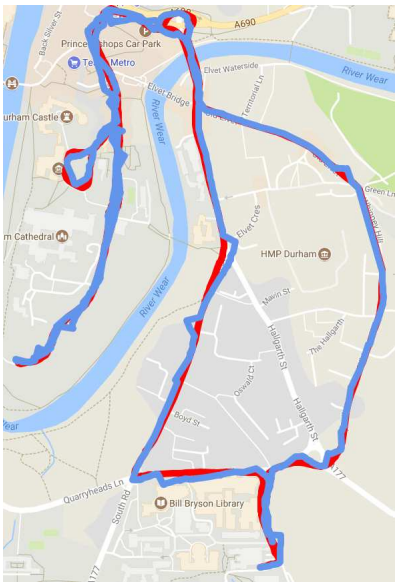
Figure 5: Corrected Plots



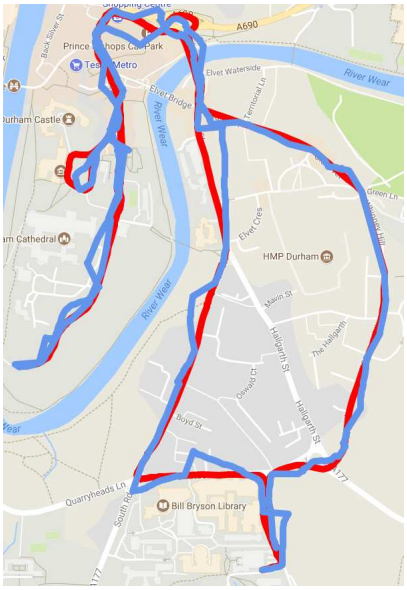
(a) Distance threshold of 5



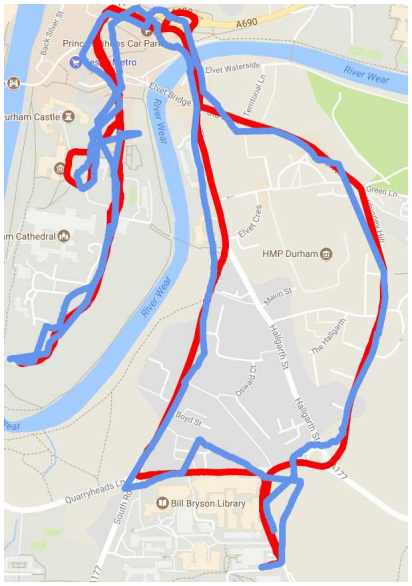
(b) Distance threshold of 10



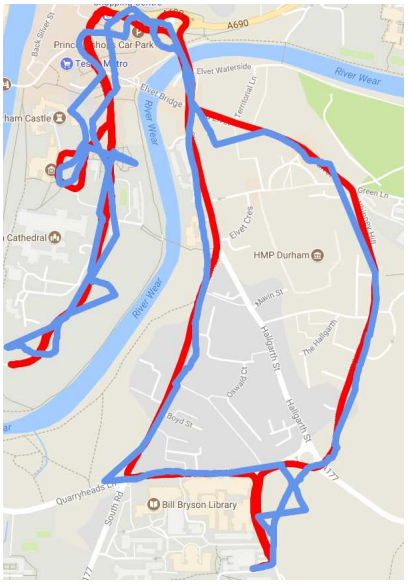
(c) Distance threshold of 25



(d) Distance threshold of 50



(e) Distance threshold of 75



(f) Distance threshold of 100

Figure 6: Corrected Maps

References

[1] A simple monocular visual odometry project in Python - https://github.com/uoip/monoVO-python

[2] Py-MVO: Monocular Visual Odometry using Python - https://github.com/Transportation-Inspection/visual_odometry

[3] Monocular Visual Odometry using OpenCV - https://avisingh599.github.io/vision/monocular-vo/

[4] Visual Odometry from scratch - A tutorial for beginners - https://avisingh599.github.io/vision/visual-odometry-full/