[[1]](#footnote-1)

Monocular Visual Odometry Using Adaptive Motion Trackers

Kevin Zhao and Owen Ren

*Abstract*—Tracking the position of vehicles over time has commonly been accomplished by using GPS and Inertial Measurement Units (IMU). But in situations where the GPS signal is unreliable, or the absolute error is too great for the scale of the application, other methods must be used. Visual Odometry is the reconstruction of position over time using a first-person video feed. Currently, cameras are a cost-effective sensor that has the potential to accurately capture key movement information. Our project involves reconstructing a motion trajectory map based on the pre-recorded monocular video footage of driving. Combining motion tracking and feature detection with novel visual speedometry, our algorithm can reconstruct the driven path accurately up to a global scaling factor.

*Keywords*— Feature Detection, Motion Tracking, Monocular Visual Odometry, Visual Speedometry

1. Introduction

Odometry is defined as the measurement of position over time, which can be easily determined for vehicles. One can take the tire circumference and multiply by the number of wheel rotations to determine the distance travelled. However, for applications such as legged robots, drones, or underwater vehicles, it is much more difficult to apply the traditional odometry techniques. Current techniques used to determine position over time include GPS and Inertial Measurement Unit (IMU). While GPS provides absolute positioning, establishing a connection with the satellites is not always possible, and the error can be too large for small-scale operations. IMU sensors, or accelerometers, have position drift due to the double integration required. Thus, there is a definite niche for visual odometry.

Furthermore, determining the absolute position is crucial for safe operation of autonomous vehicles in dense urban environments. Thus, Visual Odometry presents a new method of determining position and orientation over time by estimating the motion through a sequential set of images. By using a camera as its primary sensor, Visual Odometry provides a cost-effective alternative to expensive sensors such as LiDAR that is also capable of producing accurate results. Though processing images is computationally expensive, advances in computer hardware will result in this technique being used in more real-time applications. Currently, it has already been used in different robotic applications, such as the Mars Exploration Rovers [1].

In this project, we aim to implement a Monocular Visual Odometry algorithm that combines feature detection with feature tracking to estimate the motion of a video captured by a camera on a vehicle driving on residential roads. We also present a methodology to estimate the relative speed of the motion, constraint to a scale factor, by implementing visual speedometry.

# Related Work

A large majority of Monocular Visual Odometry implementations utilize feature detection and feature matching algorithms, such as Lucas Kanade Optical Flow to track features from one frame to the next [2]. One of the main limitations of Monocular Visual Odometry is its inability to determine the distance travelled from a sequential set of images. This is due to the scale ambiguity resulting from the monocular camera setup.

Different implementations of visual odometry that solve the scale ambiguity problem include a stereo camera setup [3] and fusing visual odometry data with IMU and/or GPS sensors, otherwise known as Visual Inertial Odometry [4].  Stereo Visual Odometry solves the scale ambiguity problem by having two cameras pointing in the direction of motion that are coplanar and parallel to each other. Similar to how humans have two eyes, this stereo setup allows for depth perception while estimating motion. By fusing camera data with IMU and/ or GPS sensors, more information about the incremental motion is known and quantified and thus, a more accurate map can be produced that highlights the distance travelled in each direction. These algorithms are extremely popular in the simultaneous localization and mapping (SLAM) problem for constructing maps for robots in unknown environments.

# System Overview

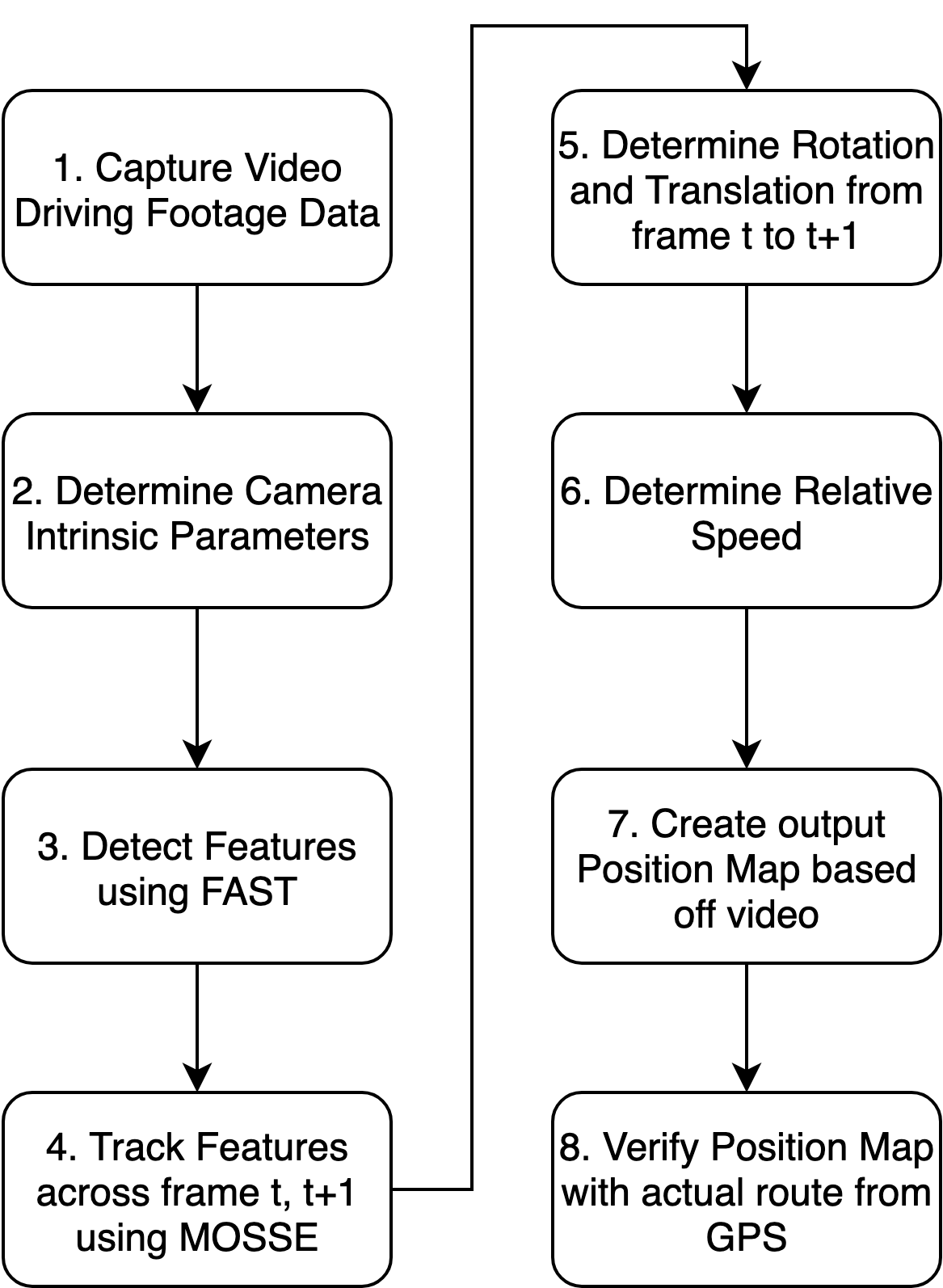


Figure 1: Overview of Proposed System

The above diagram outlines the steps of our project, beginning with data collection. Each step is detailed in the following sections.

1. Data Collection

The camera we used in this system was the GoPro Hero 8. This camera allowed us to shoot high resolution 1080p 30 Frames Per Second, which is one of the requirements for feature tracking in our algorithm.

In addition, this camera includes a GPS sensor in which we could retrieve telemetry data and verify our calculated trajectory with the actual trajectory of the driven route. Below are the camera settings used during data collection.

|  |  |
| --- | --- |
| Resolution | 1080p |
| Frames per second (FPS) | 30 |
| FOV | Linear |
| ISO Min | 100 |
| ISO Max | 400 |
| White Balance | Auto |

Table 1: GoPro Hero 8 Camera Settings

1. Camera Calibration

Before processing the video data, the intrinsic parameters of the GoPro Camera, modelled as a pinhole camera, must be determined through camera calibration. This allows an accurate representation of the points in the real world to the captured images and removes any distortion that may appear along edges of an object.

To find the camera matrix, which consists of the intrinsic parameters of a camera, we perform Zhang’s Camera Calibration Algorithm [5] using OpenCV libraries [6]. This process is documented in [6] but essentially involves taking 10 images of a checkerboard pattern at different angles and orientations using the GoPro 8. Using the OpenCV function cv2.findChessboardCorners(), the function locates the position of all the corners of the checkerboard and appends the location of the corners to two lists, one for 3D points and one for image points. The assumption for the location of the 3D points is that all images were taken with z = 0 and thus taken in the XY plane. Iterating through the set of images, we can determine the camera matrix through the OpenCV function cv2.calibrateCamera().

Camera Matrix

The parameters fx , fy refers to the focal length from the image plane to the optical center of the camera and should be equal to each other with a tolerance of 0.3. The parameters cx, cy refers to the optical axis which is at the center of the frame. In the proposed algorithm, frames were resized to (800,600) to keep the GoPro’s original 4:3 frame ratio.

1. Pseudo Code of Algorithm + Implementation Details

There are two modules to the implementation: the main module, which processes data, and the features module, which encapsulates data and manipulates features. The pseudocode of the main module is as follows:

Main:

foreach frame in footage{

foreach quadrant in frame{

if featureCount < threshold{

FAST.captureFeatures()

}

}

captureRoadFeatures()

roadFeatures.update()

roadPoints1 = roadFeatures.position

roadPoints2 = roadFeatures.history[n]

speed = calculateRelSpeed(roadPoints1, roadPoints2)

features.update()

every n frames{

points1 = features.position

points2 = features.history[n]

E = LMedS(points1, points2)

translation, rotation = SVD(E)

displacement += translation\*direction\*speed

direction \*= rotation

plot(displacement)

}

}

And the pseudocode for the feature manager is as follows:

Update:

foreach feature{

newpos = MOSSE.trackFeature()

push(this.history, newpos)

if movement < min\_movement or trackingFailed{

strike++

this.setInactive()

}

else {

this.setActive()

}

if strike > threshold or self.outOfBounds{

this.delete()

}

}

1. Feature Detection

The goal of Feature Detection is to capture features within the frame that can be tracked to the next frame and the change in location of the feature from one frame to the next determines the relative motion. A comparison of the most common feature detectors is shown in the table below.

|  |  |  |
| --- | --- | --- |
| Feature Detector | Advantages | Disadvantages |
| FAST | Extremely Fast | Less Robust in cluttered environments |
| SIFT | Robust to clutter | Slow, low accuracy in changing lighting conditions |
| SURF | Robust | Slow |
| ORB | Robust to Image Compression | Slow |

Table 2: Comparison of common Feature Detectors

From the experiments in [7], the FAST Feature detector algorithm is nearly 2.8 times faster than the next fastest algorithm in this list, ORB. For this reason alone, FAST is generally the most used feature detector in SLAM applications and is the chosen feature detector in this proposed algorithm.

Features from Accelerated Segmentation Test, developed by Rosten, Porter, and Drummond [8], is a corner detection algorithm capable of detecting interest points in a computationally fast manner. The term corner and feature is used synonymously to denote a feature within a frame.  The algorithm works by selecting interest points in a frame and comparing the intensity of the surrounding pixels to the interest point. In the figure below, we compare 16 pixels surrounding interest point P and for a corner to exist, at least n contiguous pixels in the circle must exist that are either brighter or darker than pixel P, + / - a threshold. To expedite this process, this algorithm first examines the pixels at position 1,5,9, and 13 and for a corner to exist, at least 3 out of the 4 pixels must pass the above condition. If this initial condition is passed, it then exams the surrounding pixels.

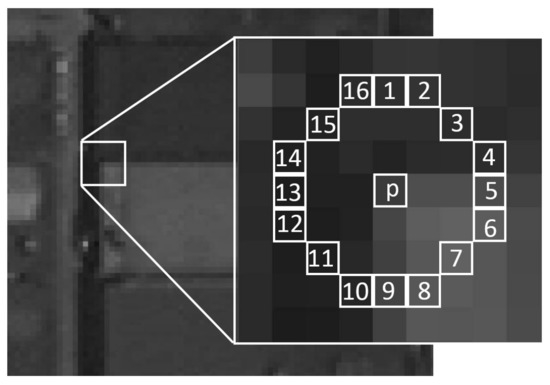


Figure 2: Image to show the Interest Point P in the FAST algorithm. Image taken from [9].

To implement this feature detector, the OpenCV Function cv2.FastFeatureDetector\_create() was utilized. For our algorithm, the threshold value was set such that the number of features detected would not exceed 200. A larger number of features would be redundant in our case and would slow down the algorithm,

Furthermore, our algorithm segments the frame into 4 quadrants, and it was found through experimentation that having a lower threshold in the bottom two quadrants led to more accurate feature tracking. This can be attributed to the fact that in the video, the top two quadrants mainly consisted of the sky and less important features while the lower two frames had more prominent features in various objects on the road.

The feature detector is not called every frame but only when the number of features falls below a calibration threshold for the specified quadrant. To summarize, the FAST Feature Detector is capable of detecting features in a computationally efficient manner that is nearly 3 times faster than the next fastest feature detector and is the algorithm used in this project.

1. Motion Tracking

The OpenCV library implements several motion tracking algorithms, listed in table X.

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm | Core Mechanism | Advantages | Disadvantages |
| BOOSTING | Updated classifier network | None | Legacy tracker |
| MIL | Updated classifier network | None | Legacy tracker |
| KCF | Updated classifier network | Fast | Unrecoverable |
| TLD | Adaptive error avoidance | Recoverable | Highly inaccurate |
| CSRT | Spatial reliability map | High accuracy | Slow |
| MedianFlow | Linear predictive | Failure detection | Erratic motion |
| MOSSE | **Initialized classifier network** | **Extremely fast** | **Lower accuracy** |
| GOTURN | Deep convolutional NN | Persistent | Unrecoverable |

Most modern motion trackers rely on a shallow neural network that is trained at runtime to generate a filter to be used for correlation matching; however, they differ in how the network is trained, the type of filter that is generated and have unique error detection methods.

Out of the listed trackers, BOOSTING and MIL were not considered due to being legacy trackers. For our purposes, failure detection and speed are important characteristics while occlusion recovery and mutation invariance are of low priority. This leaves MedianFlow, KCF and MOSSE as the remaining options. Upong testing, MedianFlow and KCF ran significantly slower than MOSSE, without noticeable accuracy benefits. Thus, MOSSE was chosen as the primary motion tracker with MedianFlow as a backup.

The MOSSE tracker does not sequentially train its neural network; instead, it is trained only once at the initialization of the tracker. This makes the algorithm much faster than the alternatives. The resulting filter is optimized in a way that its cross-correlation with the neighbourhood produces a Gaussian-shaped squared error profile.

In OpenCV, MOSSE can be initialized using cv2.trackerMOSSE\_create()

MOSSE functions better at high frame rates where the change is subtle, yet essential matrix estimation gives better approximations of the matrix the bigger each step is due to numerical stability. Therefore, to optimize both the motion tracking and the essential matrix approximation, the motion trackers and essential matrix calculations are done at different intervals. While the motion trackers are updated every frame, the essential matrix calculation happens once every n frames, dictated by the smoothness of the particular footage. For footage with high FPS or gradual motion, a large n can be used. For footage with low FPS or erratic motion, a small n can be used.

1. Calculating Translation + Rotation

The essential matrix is defined using the following null-space relation:

Where and are screen-space points that correspond to the same world-space point, only put through different projections. The essential matrix thus contains information about the difference between the projection matrices. In this application, are the screen-space coordinates of features in step , yn are the screen-space coordinates of features in step , and E contains the camera’s positional difference between step and .

To calculate the matrix, there are two established methods: RanSaC and LMedS. RanSaC, or random sample consensus, is a Monte Carlo fitting method that uses a large number of proto-models to fit random subsets of samples. The proto-model with the largest number of agreements thus becomes the final result.

LMedS, or least median squared, is a least-squares based regression method that minimizes the median of squared residues using gradient descent:

LMedS is included in the OpenCV function recoverPose().

As each feature encapsulates its own position history, the correspondence between xn and yn are effectively guaranteed, unlike feature matching-based approaches using KLT. Due to the strong reliance of RanSaC on having outliers in the data set, the method performed poorly for this application.

To recover the translation and rotation between each step, singular value decomposition is applied:

Where **Σ** contains the singular value, **U** and **V** are semi-unitary rotation matrices, is the cross-product form of the translation and **R** is the rotation. Since **U** and **V** contain the rotation data, the possible rotation matrices can be extracted using the unitary operator **W**:

Thus, using the definition of SVD, the translation matrices must be:

Where refers to the regular and inverse of W.

The OpenCV function that implements this is decomposeEssentialMat().

Because there are two possible results for each the translation and rotation matrices, the following heuristics are used for selection:

* Prefer the smaller angle of the possible rotations
* Prefer the positive translation
* If the translation is sideways, ignore it and use 0

These heuristics are optimized for driving footage; different heuristics can be made for other vehicles. For example, in the case of aerial drones, a possible set of heuristics might be:

* Prefer the rotation with the smallest roll
* Prefer the translation that aligns best with accelerometer data
* Higher motor current implies ascension vs. descension

1. Initial Results

An incremental approach was taken in the design of this algorithm. Our preliminary results were tested from Minecraft video footage, as it provided a set number of objects and features and allowed for debugging. Upon confirmation of rotation and translation motion using Minecraft footage, we proceeded with video driving footage.

Below is an early reconstruction of the trajectory using a camera matrix that was generated with an incorrect frame size without speedometry data. This revealed errors in our camera matrix that needed to be fixed to correctly output the trajectory. In addition, we used a constant scale factor, which did not account for increased velocity during parts of the path that were longer. To account for this, visual speedometry was implemented.

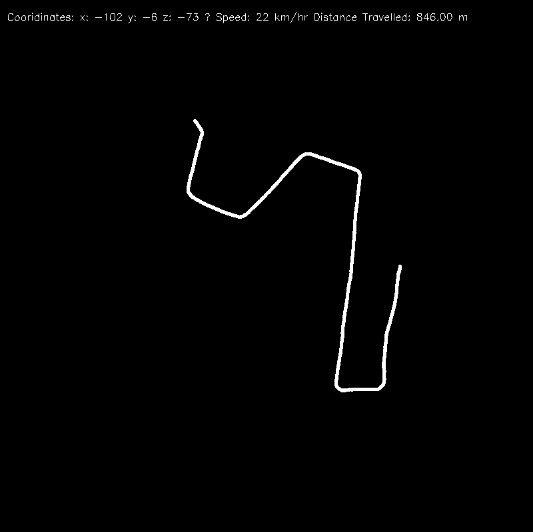


Figure 3: Initial results from algorithm with no speedometry data or calibration

1. Visual Speedometry

Due to the lack of magnitude data from essential matrix decomposition, some form of speedometry is needed. This could either come from real sensor data or from analyzing the footage with certain assumptions. One way of estimating depth is by analyzing a set of points with fixed lateral displacement.

The simplified and normalized perspective projection matrix for a square screen is:

Where is the screen-space position vector, is depth, is focal length and , are world-space coordinates.

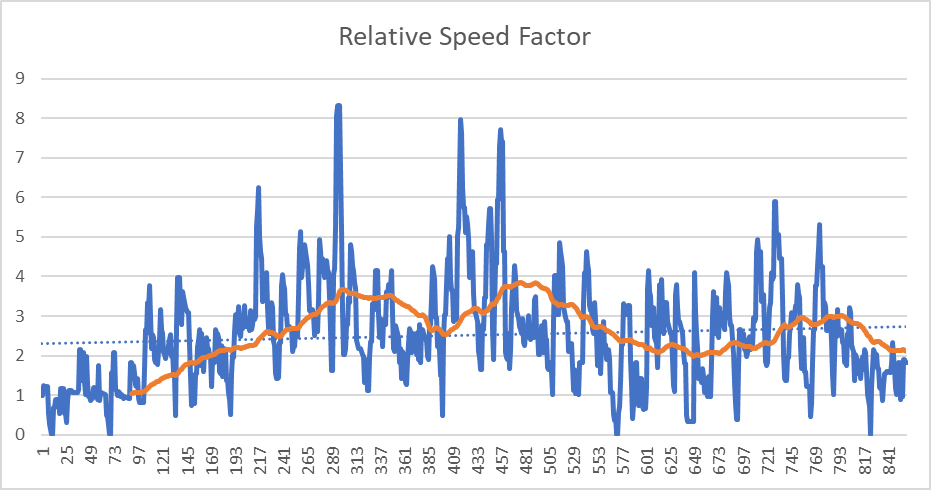
Only the forward speed is needed, so it can be estimated using:

Where y' is the screen-space vertical coordinate of the road feature.

Since the exact distance to the road surface () in pixels is not known, it can be simply absorbed into the focal length as a constant. This constant can be calibrated for later using landmarks in the final computation.

Due to the uniform nature of road texture, road feature tracking must be done differently from general feature tracking. The rough position of the road on screen-space is known, thus feature detection is not necessary; instead, a Monte Carlo based sampler is used. Every frame, a random spread of points are generated on the road. Motion tracking is then attempted on each point. After a few frames, most features will have lost traction; however, only one actively tracked feature is needed to estimate relative speed. If multiple features are tractable, the maximum computed speed is used.

Whereas other features are tracked using MOSSE, since the tracker functions poorly with uniform textures, the road is instead tracked using KCF, or kernelized correlation filter. The KCF is an improvement over BOOSTING, by utilizing the DFT and faster regression methods to increase speed. While KCF still runs significantly slower than MOSSE, due to the minimum number of features needed being small, the overall performance cost is not large.



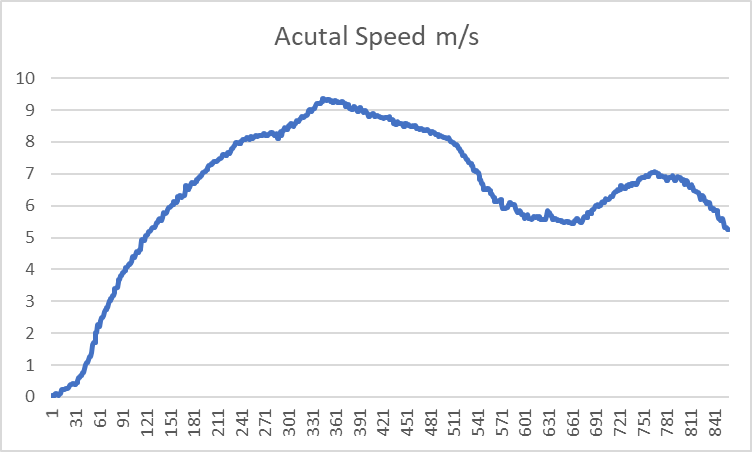
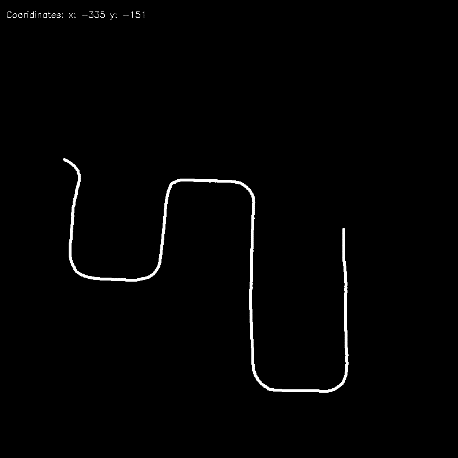
Figure 4: Relative Speed Factor calculated from algorithm

Figure 5: Actual Speed from GoPro GPS Sensor

Above is a comparison with our calculated relative speed factor compared with the GPS telemetric speed for the video footage from the start to the end. Using a moving average filter on our calculated speed factor, the results (orange line) is similar to the GPS telemetric speed and the noise can be attributed to the chaotic nature of weak-feature tracking in visual speedometry. Note: The Relative Speed Factor is unitless.

1. Final Results and Verification

The following figure shows a reconstruction of the trajectory generated from the video driving footage with visual speedometry.



Two verification was separated into two categories. The first verified the relative speed factor that was generated from visual speedometry and the second verified the reconstructed trajectory with the actual GPS coordinates from the GoPro 8.

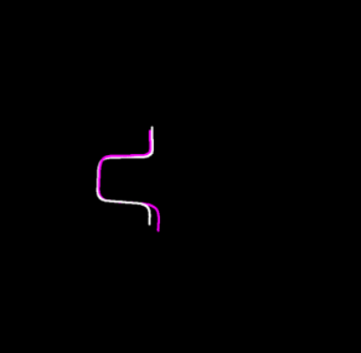
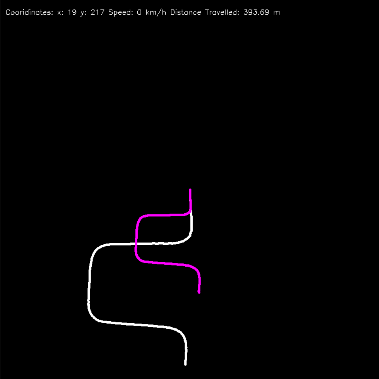


Figure 6: White Path: Trajectory using relative scale factor. Pink Path: Trajectory using actual speed.

Left: Unscaled comparison

Right: Scaled Comparison

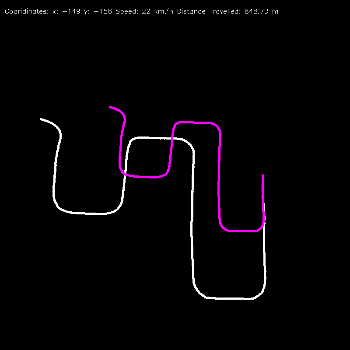
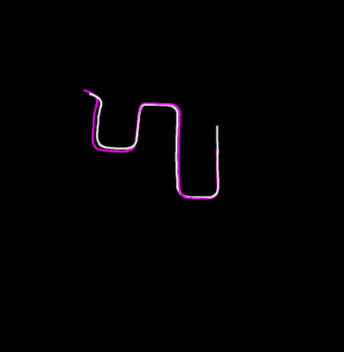
 

Figure 7: White Path: Trajectory using relative scale factor. Pink Path: Trajectory using actual speed.

Left: Unscaled comparison

Right: Scaled Comparison

GPS

After applying some scaling to the images on the left and overlaying the two trajectories, the two paths overlap at every segment except for the final turn. Nevertheless, this verifies that our visual speedometry algorithm of estimating the relative speed factor is accurate.

To verify the results of the trajectory, we extracted the GPS data from the GoPro 8 and converted the longitude and latitude coordinates to relative X and Y coordinates that can be used to plot in Python. After applying some scaling to the trajectory, the final results are shown below:

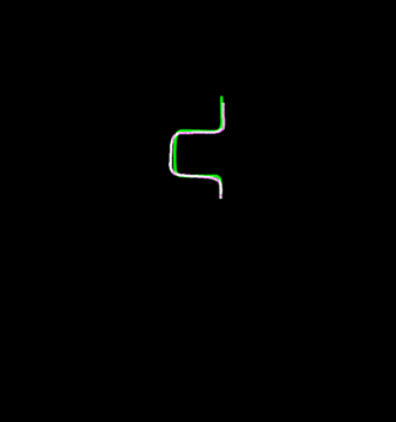
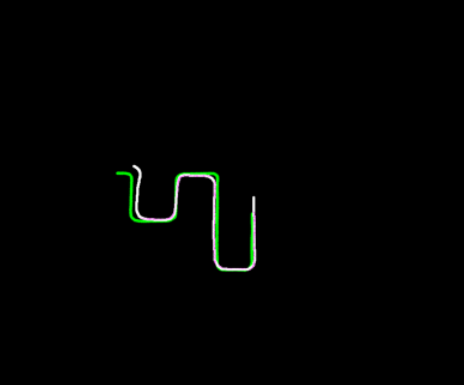


Figure 8: Verification of Trajectory with GPS Coordinates

Green: Actual GPS Trajectory

White: Reconstructed Trajectory from our algorithm

Apart from some drift during the straight segments, the reconstructed trajectory is fairly accurate. The drift is likely caused by unbalanced feature densities, where one half of the frame has more features than the other, which causes the LMedS algorithm to slightly bias towards one direction. This effect is worsened during turns, where all features move more rapidly across the screen and are quickly deleted. Though our algorithm factors this imbalance by segmenting the frame into four quadrants, there is room for improvement. For example, the screen could be further divided into more blocks; however, the increasing probability of a block containing no capturable details also poses a challenge.

1. Conclusion

To conclude, our project is an implementation of Monocular Visual Odometry using Feature Tracking instead of the common approach of using Feature Matching. Our algorithm was able to accurately reconstruct a trajectory map based off a pre-recorded video mounted on the dashboard of a vehicle driving around a neighbourhood and we accomplished the expected results and anticipated deliverables as set out by the project proposal. To determine the relative speed, visual speedometry was implemented to estimate the relative speed of the vehicle. Upon verifying our results with the actual trajectory from the GPS sensor of the GoPro 8, our conclusion is that this method provides an accurate reconstruction.

This project has provided a wealth of new learnings in the computer vision space, specifically involving the estimation of motion and position over time using only a camera sensor. This technique is an area with rapid development for robotics and autonomous vehicles to tackle the issues of simultaneous localization and mapping (SLAM).

*Summary of Learnings*:

|  |  |
| --- | --- |
| Perspective Projection | Camera projection  Camera Matrix |
| Image/vector Processing | Feature Detection  Feature Tracking  Block processing  Least Median of Squared Error |
| Epipolar Geometry | Essential Matrix |
| Visual Speedometry | Estimating relative speed through ground plane tracking |

1. Future Work

While this algorithm proved to accurately reconstruct the path generated from a pre-recorded input video, future work can be dedicated to optimizing this algorithm for computation speed for real-time applications in robots or autonomous vehicles. Fusing this algorithm with an Inertial Measurement Unit or GPS Sensor will improve the accuracy of the output trajectory and resolve the scale ambiguity issued.

However, if integration with other sensors is not desired, there is room for improvement in relative speed estimation through calibrating the estimated motion with specific objects of interest in the video.

1. Summary of Project Work Split

|  |  |
| --- | --- |
| Kevin | Owen |
| * Implementation of feature tracking on detected features * Output translation and rotation generation from tracked features * Refinement of Visual Speedometry | * Data collection and camera calibration * Integration of feature detector * Visual Speedometry * Verification of trajectory |

References

1. [↑](#footnote-ref-1)