CISC642
Combining data from camera and IMU for odometry with CNN refinements

Abhijeet Mangesh Kulkarni
UDID: 702525838

December 2020

Abstract

This report presents a pipeline used to combine data from monocular camera and IMU readings to do odometry. Furthermore, the feature based approach is extended to use CNN based refinement to reduce the feature detection on moving objects. Almost all algorithms used in this project were taught in the class and their OpenCV and Torchvision implementations are used. The code written in Python and is tested on KITTI dataset.

1 Introduction

In this work, I present a monocular visual-inertial odometry system which leverages feature based methods and deep learning. The need for visual based odometry comes from the requirement for producing low cost and low power navigation systems. The odometry is usually used by an autonomus vehicle to localize itself while simultaneously mapping the surrounding. Vision based solutions are often preferred as it can provide useful complementary information to other sensors such as IMU, GPS, lidar, etc. Odometry itself can be done without using visual data, but the result suffers from drift in case of IMU, intermittent signals in case of GPS, and high cost in case of lidar. Moreover, it is easier to perform loop closure with visual odometry.

2 Pipeline

Input to the algorithm is a sequence of images and stream of IMU data. The images and IMU data needs to be timestamped. The sensor data and images are synchronized. And the images are undistorted.

2.1 Initialization

Initialize the R, orientation, and T, position, to any arbitrary values. If reading from compass and GPS are available, R and t can be initialized to those values. Features are initialized by running the feature detector on the first image. After initialization, a pipeline as shown in Figure 1

2.2 Feature Detection

One of the good features are corners which are ubiquitous in a non smooth image. In this project, Shi-Tomasi's algorithm [3] is used to identify corners in the given image. This algorithm is same as harris corner detector but uses a different scoring function, $R = min(\lambda_1, \lambda_2)$, R is the response and $\lambda_1 \& \lambda_2$ are the eigen values. OpenCV function: `cv2.goodFeaturesToTrack()`

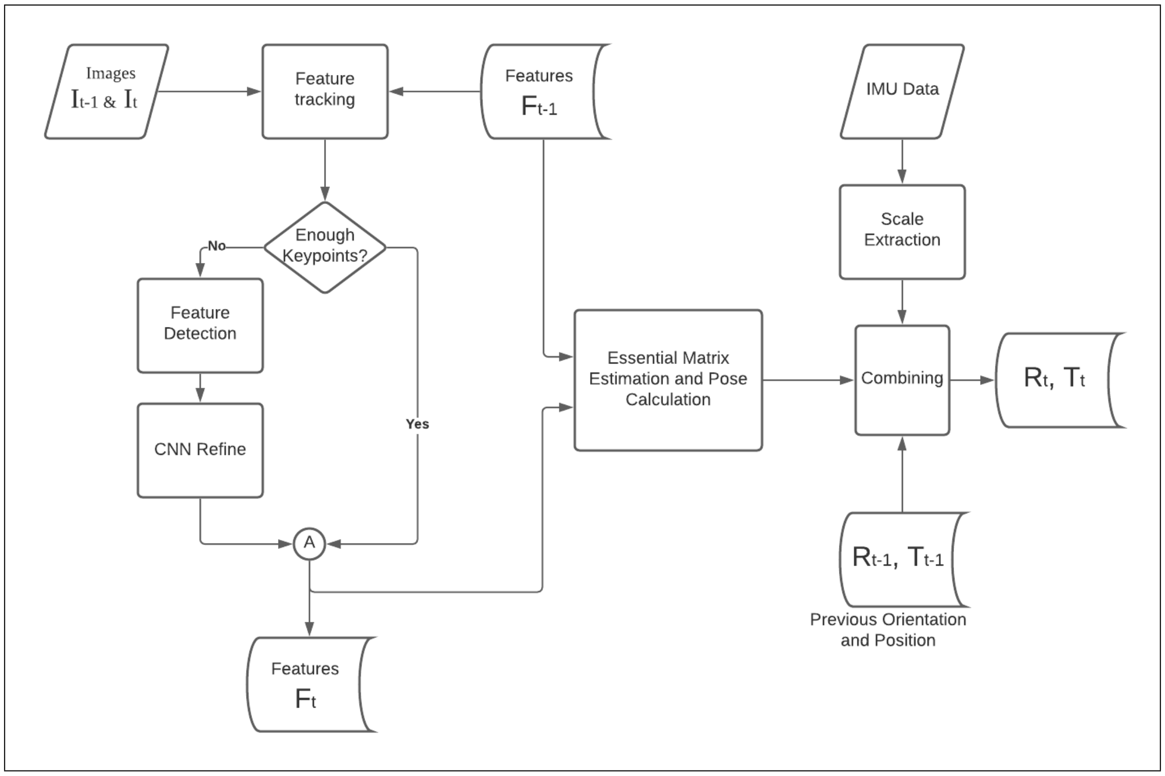


Figure 1: Pipeline for monocular Visual-Inertial odometry followed after initialization.

2.3 CNN based feature refinement

For determining optical flow, it is necessary to track features from first image to second image to estimate the motion of the camera between those two images, when they were captured. For this to work it is assumed that all the features in the image are stationary with respect to a world frame. But in reality, an image will contain moving objects such as cars, people, bikes, etc. Moreover, the moving objects are a good source of corners so the Feature detector will detect those. This can introduced errors in the motion estimation if not enough stationary features are available in a scene and the resulting motion will be a result of the features from these moving objects. To alleviate this problem, CNN can be used to detect these moving objects and exclude them from the features being tracked.
For this project, FCN_ResNet50 [4] convolutional network specialized for semantic segmentation is used. The CNN is pretrained on the COCO train2017 dataset. Using this CNN a bitmask is generated for locating pixels which corresponds to a car in the image. Then the features detected in the location of car are excluded. For example see Figure 2. Model from TorchVision: `torchvision.models.segmentation.fcn_resnet50`

2.4 Feature Tracking

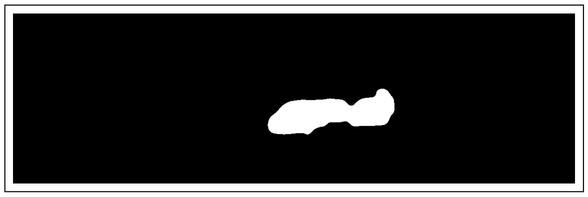
The final features outputted from the sec 2.3 are mostly free of non stationary features. Now, the stationary features are tracked between images using a pyramidal implementation of Lucas-Kanade tracker algorithm [1]. This gives set of correspondences two images which are used to calculate Essential matrix. See Figure 3 for feature tracking. OpenCV function: `cv2.calcOpticalFlowPyrLK()`

2.5 Essential Matrix and Pose Estimation

With available correspondences between two images I_t and I_{t-1}, Essential matrix, E can be estimated. The essential matrix is defined as,

$$F_{t-1}^T E F_t = 0$$

(a) Detected features from 2.2


(b) Car location Bitmask

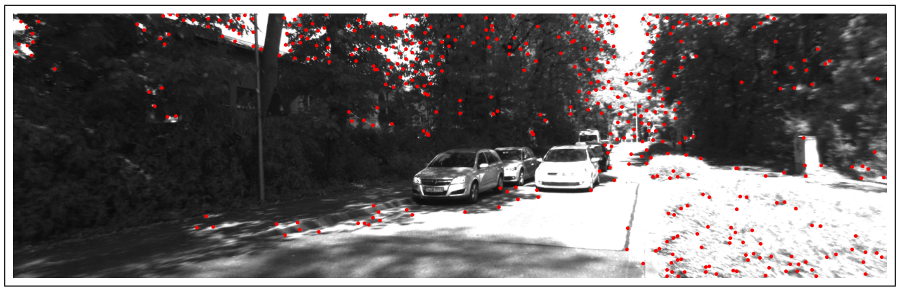

(c) Excluding features on cars

Figure 2: Sub figure (2c) is the output of boolean subtraction between features in (2a) and (2b).

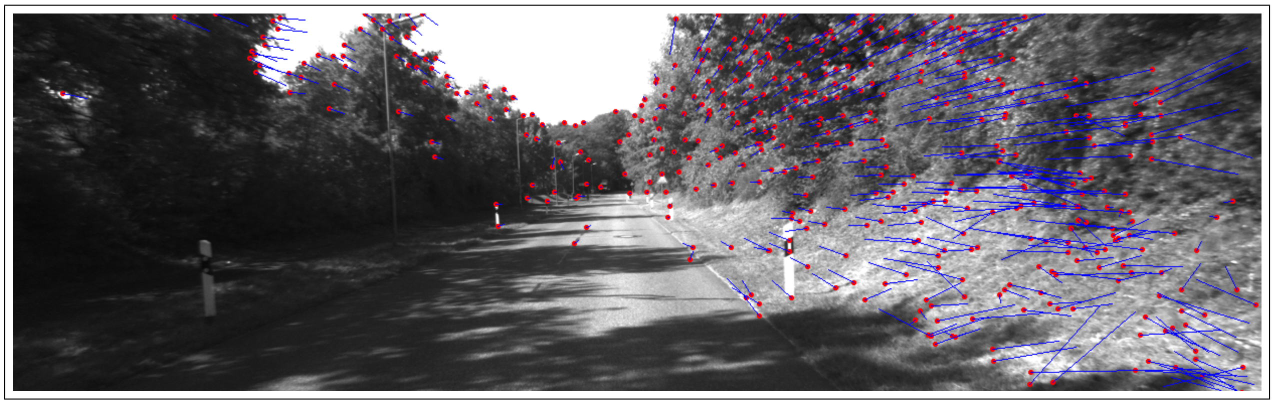


Figure 3: Features tracked from previous image.

where, F_{t-1} and F_t are features at time $t-1$ and t respectively. For this project, Nister's five point algorithm [5] is used to estimate the essential matrix along with RANSAC to use only inliers. After estimating Essential matrix, it is decomposed to find R, relative orientation, using SVD decomposition. The relative position obtained using this decomposition is cannot be used to determine true relative position because the true scale cannot be estimated just by using camera. So we need an external way to get this absolute scale as shown in sec 2.6. OpenCV functions: `cv2.findEssentialMat()` and `cv2.recoverPose()`

2.6 Scale Extraction and Trajectory construction

The data from the IMU is pre-processed to obtain acceleration only in the direction of motion. Then double integrator $d = \int\int a_{fwd} dt$, where a_{fwd} is the forward acceleration, is used to get relative distance, d, between the locations of camera between images I_t and I_{t-1}. Now we have relative rotation, R (from sec 2.5), and relative distance, d, which can be used to calculate true trajectory.

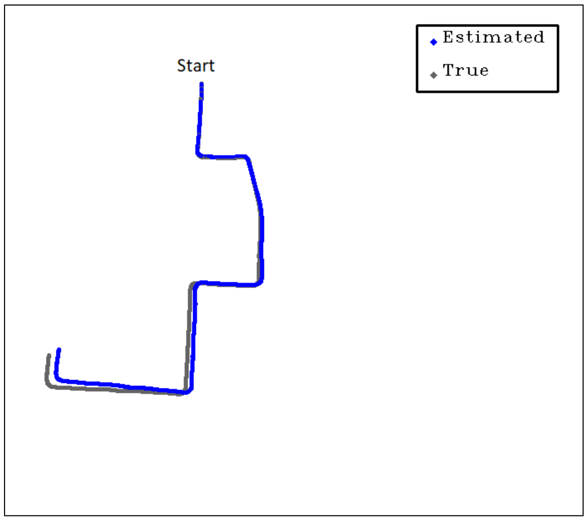
$$R_t = RR_{t-1}$$

$$T_t = T_{t-1} + Rd$$

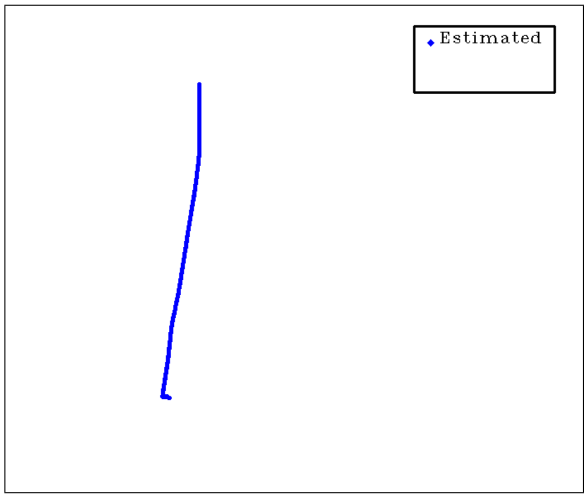
where, $R_t \& R_{t-1}$ and $T_t \& T_{t-1}$ are the true rotation matrices and true positions at time $t \& t-1$ resp.

3 Results

For the demonstration purpose, the code is written in Python 3.8.3 and executed on a computer with CPU:i5-9300h and GPU:Nvidia GTX1660ti. The code is tested on KITTI dataset [2].



(a) KITTI odometry dataset



(b) KITTI raw dataset

Figure 4: Outputs on KITTI dataset

Figure 4a shows the code's output on the 'odometry' subset of KITTI dataset with no IMU reading. Figure 4b shows the code's output on the 'rawdata' subset of KITTI dataset with IMU readings, which has no ground truth but has many moving cars in the images.

4 Conclusion

The code successfully performs monocular visual-inertial odometry on the KITTI dataset. As seen in Figure 4a, the estimate diverges after some time, which might be a result of noisy IMU readings and errors in the rotation estimates because of improper feature tracking on objects such as trees. To improve the results, CNN can be trained to detect trees, people and other common moving objects. Moreover, further filtering, like Kalman filtering, can be done on data to combine data more closely. After feature detection is done, it takes $\approx 0.045sec$ for each frame and for feature detection it takes $0.1sec$(on GPU). This shows the algorithm can be run realtime at 22 frames per second and only slowing down to 10fps when not enough features are present.

References

[1] J.-Y. Bouguet. "Pyramidal implementation of the lucas kanade feature tracker". In: 1999.

[2] Andreas Geiger et al. "Vision meets Robotics: The KITTI Dataset". In: *International Journal of Robotics Research (IJRR)* (2013).

[3] Jianbo Shi and Tomasi. "Good features to track". In: 1994.

[4] Jonathan Long, Evan Shelhamer, and Trevor Darrell. *Fully Convolutional Networks for Semantic Segmentation.* 2015.

[5] D. Nister. "An efficient solution to the five-point relative pose problem". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.6 (2004), pp. 756–770. DOI: `10.1109/TPAMI.2004.17`.