P4 - Visual Inertial Odometry

Uday Sankar
usankar@wpi.edu

Gowri Shankar Sai Manikandan
gmanikandan@wpi.edu

Shaurya Parashar
sparashar@wpi.edu

***Abstract*—In this project, we aim to investigate and implement two approaches for stereo visual-inertial odometry: a classical filter-based method using the MultiState Constraint Kalman Filter (MSCKF), and a deep learning-based method. In the classical approach, we integrate the fundamental mathematical principles of the stereo-MSCKF into a provided base-code framework. In the deep learning-based approach, we implement three models that take visual data, inertial data, and visual-inertial fusion data as input and output the camera pose.**

I. Phase 1 - Classical VIO

A. Introduction

The primary goal of this phase is to determine the scale from an image, which allows for depth evaluation. It is well-known that obtaining depth information from a single camera is challenging without any prior knowledge of the surroundings. A more straightforward alternative involves using a stereo camera with a known pose, where depth can be estimated by matching features. However, there are some limitations to this approach, such as the computational complexity and difficulty of matching, as well as its ineffectiveness in dealing with motion blur commonly found in robotic applications.

To address these challenges, an Inertial Measurement Unit (IMU) can be employed. A typical 6-DoF (Degrees of Freedom) IMU measures both linear and angular acceleration. IMUs excel in handling rapid movements and sudden jolts, where cameras often struggle, but they tend to drift over time, which is a weakness that cameras can compensate for. This complementary nature creates an opportunity for a multi-modal fusion problem, enabling accurate camera pose estimation and subsequent depth determination.

In this phase of the project, we implement [2] and refer to [3] for the mathematical model.

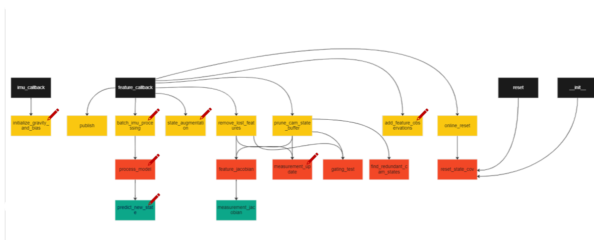

Fig. 1: VIO Pipeline

The pipeline involved in this process is shown in Figure 1. In the given baseline starter code, there were seven functions that had to be written by us using the concepts discussed in the paper[3].

B. Data

The data utilized for testing our implementation is the "Machine Hall 01 easy" (MH 01 easy), a subset of the larger EuRoC dataset. This data was gathered using a 6-DoF sensor mounted on a quadrotor, which followed a specific flight path. In order to obtain the ground truth for the system, a highly accurate Vicon Motion capture system with sub-millimeter precision was employed.

C. Implementation

A starter code has been supplied for the development of the Multi-State Constraint Kalman Filter (MSCKF). To accomplish the implementation of the model, we made modifications to specific functions within the *msckf.py* Python file. From the pipeline in Figure 1, it is clear that there are seven functions to be implemented in the *msckf.py* file. They are as follows:

*1) **initialize_gravity_and_bias**:* In this function, the initial IMU readings are used to establish the bias and initial orientation. The first few readings from the IMU message buffer are averaged to get the angular and linear velocities. The gyro bias is set using the average angular velocity, while gravity is determined by the linear velocity. Subsequently, the normalized gravity vector is incorporated as the IMU state.

By utilizing these two vectors, the initial orientation is set up to ensure consistency with the inertial frame. The quaternions are then incorporated as the IMU's orientation state. The resulting vector is expressed as follows:

$$X_I = \begin{pmatrix} \hat{q}_{IG}^T \\ \hat{b}_g^T \\ \hat{v}_I^T \\ \hat{b}_a^T \\ \hat{p}_I^T \\ \hat{q}_C^T \\ \hat{p}_c^T \end{pmatrix} \tag{1}$$

*2) **batch_imu_processing**:* This function processes the IMU messages within the IMU message buffer based on a specified time constraint. The process model is executed for each IMU input within the given time frame, continuing until the time constraint is reached. Following this, the current IMU ID is updated to the subsequent state IMU ID. All remaining unused IMU messages are then removed from the IMU message buffer.

*3) **process_model**:* This function calculates the camera module's pose based on the latest IMU state update. It takes

the current time, angular velocity (m_gyro), and linear acceleration (m_acc) as input arguments. The error for each IMU state is computed and represented as follows:

$$\tilde{X}_I = \begin{pmatrix} \tilde{\theta}_{IG}^T \\ \tilde{b}_g^T \\ \tilde{v}_I^T \\ \tilde{p}_I^T \\ \tilde{\theta}_C^T \\ \tilde{p}_c^T \\ \tilde{b}_a^T \end{pmatrix} \quad (2)$$

The linearized continuous dynamics for the error IMU state are evaluated as:

$$\dot{\tilde{X}}_I = F_I \tilde{X}_I + G_I n \quad (3)$$

The discrete transition matrices F and Q are calculated as shown:

$$F = \begin{bmatrix} -\left[\hat{\omega}\times\right] & -\mathbf{I}_3 & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ -C\left(\hat{q}_I^T\right)\left[\hat{a}\times\right] & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & -C\left(\hat{q}_I^T\right) & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{I}_3 & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \end{bmatrix} \quad (4)$$

$$G = \begin{bmatrix} -\mathbf{I}_3 & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{I}_3 & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & -C\left(\hat{q}_I^T\right) & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{I}_3 \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \end{bmatrix} \quad (5)$$

The matrix exponential is approximated up to the 3rd order as follows:

$$\phi = \mathbf{I}_{21\times21} + F(\tau)\cdot d\tau + \frac{1}{2}\cdot\left(F(\tau)\cdot d\tau\right)^2 + \frac{1}{6}\cdot\left(F(\tau)\cdot d\tau\right)^3 \quad (6)$$

Lastly, the state is propagated, and the new state is predicted using the 4th-order Runge-Kutta method by calling the "predict_new_state" function.

In simpler terms, this function computes the camera module's pose based on the latest IMU state update. The error for each IMU state and the linearized continuous dynamics for the error IMU state are evaluated. Discrete transition matrices are calculated, and the matrix exponential is approximated. Finally, the state is propagated, and the new state is predicted using the 4th order Runge-Kutta method.

*4) **predict_new_state**:* This function describes the process of propagating the state of an Inertial Measurement Unit (IMU) using a 4th-order Runge-Kutta method. The inputs for this function are the time step $(d\tau)$, gyroscopic data (gyro), and acceleration data for the given state. In simpler terms, the function updates the orientation, velocity, and position of an IMU based on the provided inputs.

First, the error state of the angular velocity (gyro) is normalized. Next, the $(d\omega)$.matrix is computed using the normalized angular velocity:

$$\Omega(\hat{\omega}) = \begin{bmatrix} -\left[\hat{\omega}\times\right] & \omega \\ -\omega^T & 0 \end{bmatrix} \quad (7)$$

The current orientation, velocity, and position are obtained from the IMU state server. With these values, the angular velocity and acceleration are calculated. The Runge-Kutta method is then used to approximate the updated state. The method consists of four steps (k_1, k_2, k_3, k_4):

$$k_1 = f(t_n, y_n) \quad (8)$$

$$k_2 = f\left(t_n + \frac{d\tau}{2}, y_n + k1\cdot\frac{d\tau}{2}\right) \quad (9)$$

$$k_3 = f\left(t_n + \frac{d\tau}{2}, y_n + k2\cdot\frac{d\tau}{2}\right) \quad (10)$$

$$k_4 = f\left(t_n + d\tau, y_n + k_3\cdot d\tau\right) \quad (11)$$

After calculating the approximate orientation, it is converted to quaternions. The velocity and position of the current IMU state are updated based on the new approximation. These updated values are then used as the current state values for evaluating the next state.

*5) **state_augmentation**:* In this function, we compute the state covariance matrix to propagate the uncertainty of the IMU and camera states. First, we obtain the rotation and translation values between the IMU and the camera. Then, we add a new camera state to the state server using the initial IMU and camera states.

Next, we update the state augmentation Jacobian, J_I, as follows:

$$J_I = \begin{bmatrix} C(\hat{q}_{I_G}) & \mathbf{0}_{3\times9} & \mathbf{0}_{3\times3} \\ -\left(C(\hat{q}_{I_G})\right)^T\left[\hat{p}_c\times\right] & \mathbf{0}_{3\times9} & \mathbf{I}_3 \\ \mathbf{0}_{3\times3} & \mathbf{I}_3 & \end{bmatrix} \quad (12)$$

We then resize the state covariance matrix and propagate the covariance of the IMU state. The full propagation of the uncertainty is represented as:

$$P_{k+1|k} = \begin{bmatrix} P_{II_{k+1|k}} & \phi_k P_{IC_{k|k}} \\ \left(\phi_k P_{IC_{k|k}}\right)^T & P_{CC_{k|k}} \end{bmatrix} \quad (13)$$

Finally, the augmented covariance matrix, $P_{k|k}$, is given as:

$$P_{k|k} = \begin{bmatrix} J_{21+6N} \\ J \end{bmatrix}^T \begin{bmatrix} P_{k|k}^{21+6N} \\ J \end{bmatrix} \quad (14)$$

We then update the state covariance in the server.

In simpler terms, this function calculates the uncertainty of the IMU and camera states using the state covariance matrix. The state augmentation Jacobian is updated, and the covariance of the IMU state is propagated. Finally, the augmented covariance matrix is computed and used to update the state covariance in the server.

6) ***add_feature_observations****:* The following points describe

1) Obtain the feature msg as the input for this function.
2) Get the current IMU state ID.
3) Evaluate the number of features.
4) Append each feature one by one in the feature msg to the map server, if it is not already present in the map server.
5) Maintain a count of the number of features tracked.
6) Update the map server for every given state, tracking all the features.
7) Calculate the tracking rate as the ratio of the number of tracked features to the number of current features available.

7) ***measurement_update****:* In order to update the state estimates, a measurement model has been implemented. A residual **r** that depends linearly on the state errors is defined as follows:

$$r = H\tilde{X} + \text{noise}$$

In the aforementioned equation, H represents the measurement Jacobian matrix, while the noise component signifies a zero-mean, white, uncorrelated state error. We have employed an estimated Kalman filter structure in this context. Initially, we verify whether the existing H and r values are zero.

Subsequently, we attempt to minimize the complexity of the Jacobian matrix through the application of QR decomposition, which serves to decrease the computational demands as outlined below:

$$H_x = \begin{pmatrix} Q1 & Q2 \end{pmatrix} \begin{pmatrix} T_h \\ 0 \end{pmatrix}$$

In this case, Q_1 and Q_2 represent unitary matrices, with their columns constituting bases for the range and nullspace of H_x, respectively. Moreover, T_H denotes an upper triangular matrix. Following this, we determine the Kalman gain by employing the subsequent formula:

$$K = PT_H^T (T_H P T_H^T + R_n)^{-1}$$

Here, K denotes the Kalman gain, P symbolizes the state covariance matrix, T_H^T represents the upper triangular matrix, and R_n stands for the covariance matrix of noise. After determining the Kalman gain, we proceed to compute the state error as follows:

$$\Delta X = K r_n$$

Utilizing the calculated state error, the IMU state is updated initially, followed by adjustments to the camera states. Ultimately, the state covariance undergoes an update, and the covariance matrix is adjusted to achieve symmetry.

D. Results

Since we faced difficulties regarding the visualization using Pangolin module, we got the visualization video from Irakli Grigolia's personal computer.

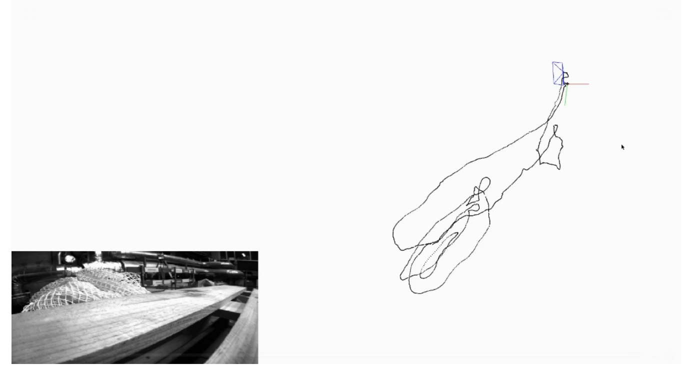

Fig. 2: Trajectory for EuRoC MH_01_easy

II. Phase 2 - Deep VIO

A. ***Introduction***

In the previous phase, we implemented a classical filter-based approach for stereo-visual inertial odometry. Now the question is whether this can be done using deep learning. Therefore, in this phase, we aim to develop a deep-learning architecture that predicts the relative pose between two image frames, along with IMU measurements. The methodology is to implement three models that take in visual data, inertial data, and visual-inertial fusion data, each to return the pose of the drone under study.

B. ***Data***

The approach that we had to take was to train all three models with Machine Hall 02 Easy, Machine Hall 03 Medium, Machine Hall 04 Medium, Machine Hall 04 difficult, and Machine Hall 05 difficult subsets of the EuRoC dataset. The EuRoC dataset involves two forms of data - visual and inertial data. The visual data is collected using two synchronized global shutter grayscale cameras at 20 frames per second at a resolution of 752 x 480. Although the dataset involves two environments, including a large 'Machine Hall' with sparse features and two cluttered 'Vicon Rooms' with rich texture, the one that is being considered for training is the Machine Hall subset. Finally, the ground truth needed for supervised learning is obtained using a Vicon motion capture system which provides 6-DoF pose information at a rate of 200Hz.

However, we faced a major issue when it came to data handling. The data was temporally inconsistent with the ground truth data. One solution that we came up with to address this problem was to subject the data to **interpolation** and the alignment of timestamps. We tried to implement two types of interpolation techniques - **linear interpolation** and **spline**

interpolation. Linear interpolation involves the estimation of values of missing data points by drawing a straight line between two neighboring data points and calculating the value of the missing point along that line.

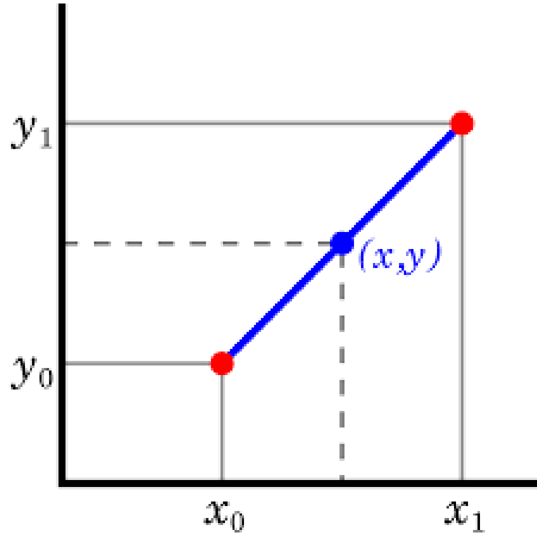

Fig. 3: Linear Interpolation

Similarly, spline interpolation involves the estimation of values of missing data points by fitting a piecewise polynomial function to the available data points and interpolating to estimate the missing points.

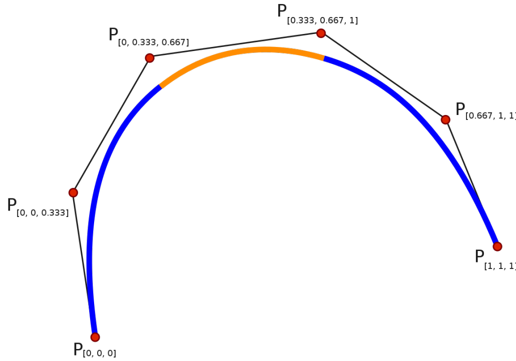

Fig. 4: Spline Interpolation

But unfortunately, we were not able to come up with a reliable implementation of both of these techniques, as the interpolated output was still not matching temporally. Since training with less data seemed better than training with wrongly matched data, we decided to go forward with a simple matching technique where we extract data points that have synchronized matching with the ground truth. Although not a fancy solution, this method worked for us in the models that we implemented.

C. *Visual Odometry*

So the first model that we worked with was the visual odometry model, which takes in stereo camera images and returns the camera pose as output. The first idea for a model to tackle this task we tried to implement was the Feature Pyramid Network (FPN)[1]. The FPN is a neural network architecture commonly used in computer vision tasks such as object detection and semantic segmentation, which we planned to utilize for our purpose. FPN constructs a pyramid of features from the input image, with features at different scales corresponding to different levels of abstraction. The bottom-up pathway extracts feature from the input image using a convolutional neural network (CNN), while the top-down pathway combines these features to generate a set of feature maps at different scales. The resulting feature pyramid can be used for various tasks by applying region proposal networks (RPNs) to each level of the pyramid and aggregating the proposals to generate a final set of object detections. FPN would be preferable because it allows for the effective handling of objects of varying sizes and achieves state-of-the-art performance on a variety of computer vision tasks.

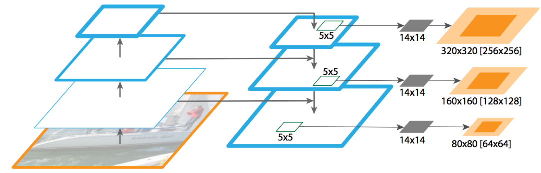

Fig. 5: Feature Pyramid Network

But we ended up not using this network because we had issues while training the data. The losses seemed to increase which is not ideal. So we decided to go for a simpler approach which involved ResNet34 backbones connected with a fully connected layer in the end (Fig 6). The output obtained from the fully connected layer, which is a 7 x 1 vector will be the pose of the camera.

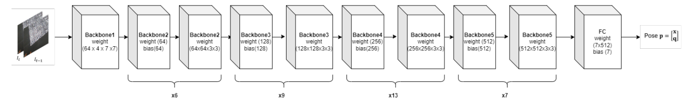

Fig. 6: Visual Odometry Network

Now since the ground truth is the drone pose in the world frame and for training we needed the pose in the body frame, we performed a transformation on the world pose to convert it from the world frame to the body frame.

$$t'_{t+1} = R_t(t_{t+1} - t_t)$$
$$R'_{t+1} = (R_{t+1})^{-1} R_t$$

where R, t are the rotation matrix and translation vector obtained from quaternions with respect to the world frame, and R', t' with respect to the body frame. Once the model is trained, during inference, since we are getting output poses with respect to the body frame, we integrate these poses over time to get the poses in the world frame.

$$t_{t+1} = t_t + R_t t'_{t+1}$$
$$R_{t+1} = R_t R'_{t+1}$$

As discussed in the earlier data section, a major problem that we faced in this section was the temporal inconsistency of the

data with its ground truth. We went with a common timestamp value-based method in order to get the training initiated. Upon training the model using an MSE loss function over 20 epochs, we got the training losses as shown in Figure 7.

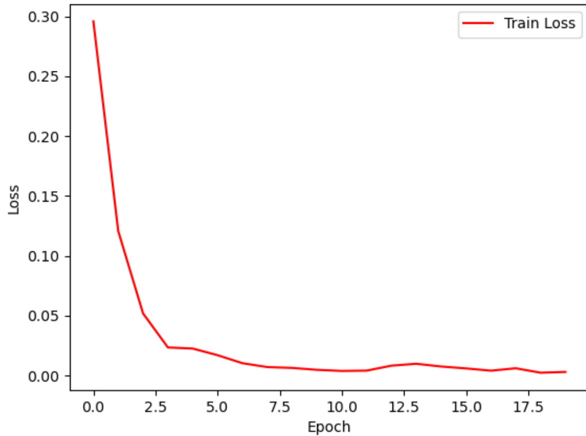

Fig. 7: Training Losses in Visual Odometry Network

For the visualization purpose, we were planning to use the rpg_trajectory_evaluation[4] from the Robotics and Perception Group of the University of Zurich repository. Since the problems we faced earlier and time constraints led to us not being able to properly visualize the test output for the visual odometry network, we have only included the visualization for the inertial odometry network which is the next section.

D. *Inertial Odometry*

For this task, we went with an LSTM (Long Short-Term Memory) based model which is followed by a fully connected layer. Fortunately, we were able to get the training initiated with the first model that we tried.

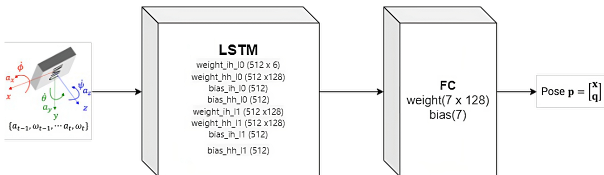

Fig. 8: Inertial Odometry Network

Upon training the model for an MSE loss function over 20 epochs, the losses that we obtained are as follows.

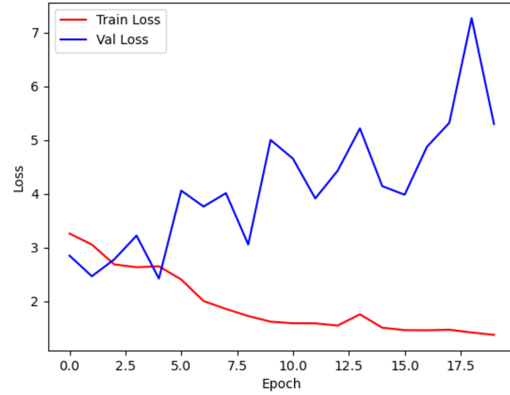

Fig. 9: Training Losses in Inertial Odometry Network

Upon inspection, we can see that although training loss is decreasing over the epochs, the validation loss keeps on increasing in an inconsistent manner. We suspect that the model is most probably overfitting to the data. We also suspect this happening due to data handling mistakes that we might have overlooked in pre-processing part. Regardless, we had to proceed with the visualization due to time constraints.

For visualization, we were able to get the rpg_trajectory_evaluation repository to work for this particular model. The various errors that we obtained are shown in the figures below.

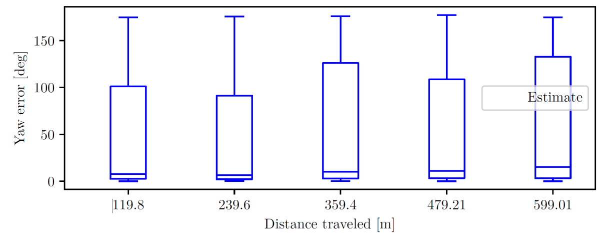

Fig. 10: Relative Yaw Error

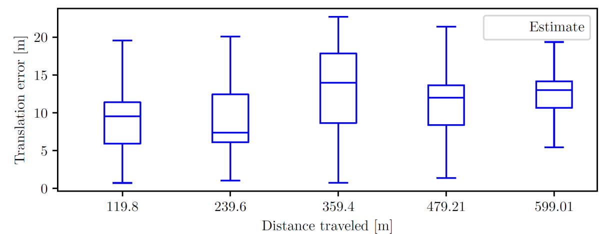

Fig. 11: Relative Translation Error

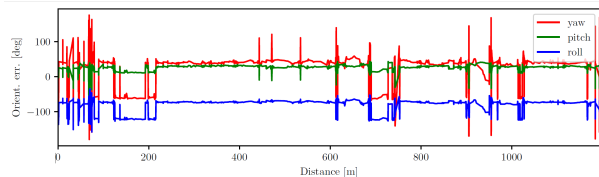

Fig. 12: Rotation Error

Finally, the top and side views of the trajectories obtained using visualization are shown below.

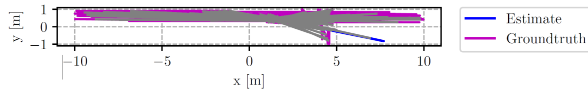

Fig. 13: Top View of Trajectory

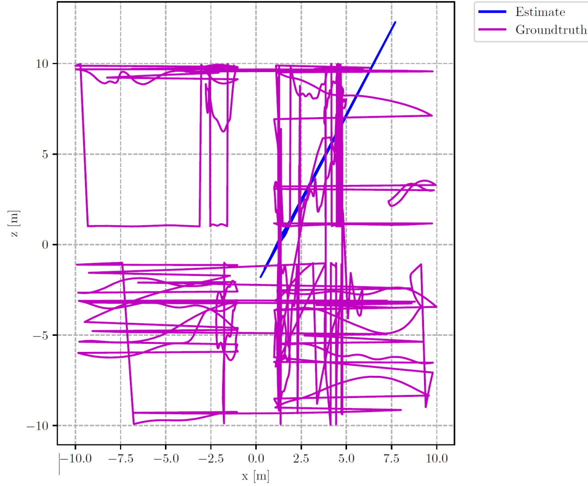

Fig. 14: Side View of Trajectory

Upon inspection, it is obvious that the visualization did not come out as expected. Even the ground truth is being shown to be very different from what it is supposed to be. We suspect that, as discussed earlier, this is happening due to small mistakes that we overlooked in the data handling part of pre-processing. We intend to correct these mistakes and fine-tune the trajectories as per our needs in the future.

E. Visual-Inertial Odometry

Now finally the fusion of both these concepts is the combined Visual-Inertial Odometry network. Unfortunately, we were not able to complete a working model within the given time. But we will talk about how we intended to complete this network in the report. The network we intended to construct is shown below.

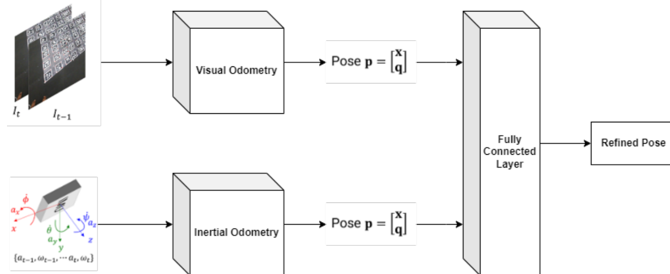

Fig. 15: The Visual-Inertial Odometry Network

So, the visual data will be processed by the Visual Odometry Network and the IMU data will be processed by the Inertial Odometry Network. The 7 x 1 vectors indicating the poses obtained from each model will be passed through a fully connected layer to get the refined pose. The implementation of this combined network is one of the works for the future.

III. Conclusion

From the study, it is clear that the MSCKF implementation for Visual-Inertial Odometry is an excellent method of performing the task at hand. The output obtained is very close to the ground truth. To have a deep learning model that can compete with this implementation, more time and effort need to be put into fine-tuning the construction of the network. This is something our team hopes to accomplish in the future.

References

[1] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection, 2017.

[2] Anastasios I Mourikis and Stergios I Roumeliotis. A multi-state constraint kalman filter for vision-aided inertial navigation. In *Proceedings 2007 IEEE international conference on robotics and automation*, pages 3565–3572. IEEE, 2007.

[3] Ke Sun, Kartik Mohta, Bernd Pfrommer, Michael Watterson, Sikang Liu, Yash Mulgaonkar, Camillo J Taylor, and Vijay Kumar. Robust stereo visual inertial odometry for fast autonomous flight. *IEEE Robotics and Automation Letters*, 3(2):965–972, 2018.

[4] Zichao Zhang and Davide Scaramuzza. A tutorial on quantitative trajectory evaluation for visual(-inertial) odometry. In *IEEE/RSJ Int. Conf. Intell. Robot. Syst. (IROS)*, 2018.