

Visual Inertial Fusion

Contents

1 Preliminaries	1
1.1 Outline of the exercise	1
2 Exercise 1: IMU Process Model Implementation	2
2.1 Objective	2
2.2 Mathematical Formulation	2
3 Stereo Vision Measurements	3
4 Tightly-Coupled Optimization Framework	4

In this exercise, we will combine what we have learned in the previous exercises to implement a visual inertial odometry pipeline. We will fuse the information from an Inertial Measurement Unit (IMU) and a stereo camera system to estimate the motion of a moving drone.

1 Preliminaries

1.1 Outline of the exercise

This exercise showcases the development of a Visual-Inertial Odometry (VIO) system that combines data a camera and IMU to estimate the real-time motion of a drone. The system is built using a tightly-coupled optimization framework, which merges high-frequency inertial data with visual features tracked across stereo image pairs to provide accurate and reliable motion estimation. Designed with a multi-threaded architecture, it efficiently manages tasks like sensor data processing, feature tracking, and state estimation simultaneously.

The implementation is tailored to work with the EuRoC MAV dataset and is organized into modular components for image processing, inertial data handling, and filter update.

Implementation Framework

As usual, we provide you with a comprehensive Visual-Inertial Odometry framework, including the complete optimization implementation (`optimization_vio.py`), to run the whole pipeline via (`vio.py`). Your task focuses specifically on implementing the visual-inertial fusion components within this framework.

The provided codebase contains a complete sliding window optimizer implementation for state estimation, thread management, and data handling infrastructure, image processing utilities for feature detection and tracking, visualization tools for debugging and result analysis, and an interface to the EuRoC MAV dataset.

Your implementation will center on three main components in the fusion pipeline. To download the dataset, you will need to visit the EuRoC dataset webpage. For this exercise, we will need the *Machine Hall 01* dataset, so please download the dataset separately using this download link and put the `MH_01_easy` in the `data` folder.

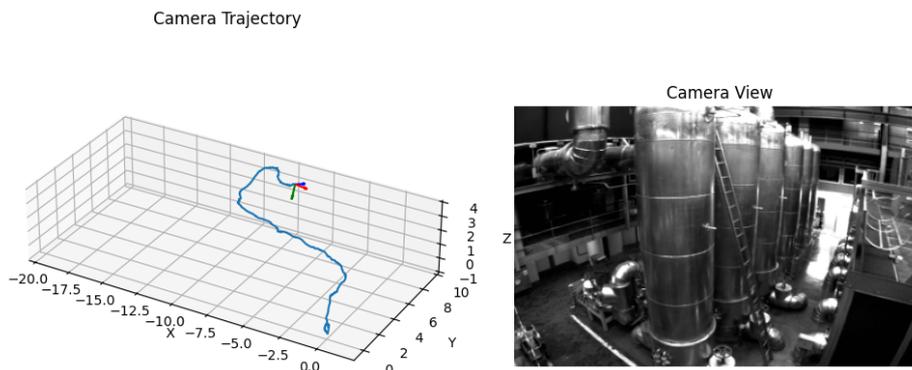


Figure 1: Trajectory estimation using Visual-Inertial Odometry. Left: Estimated trajectory. Right: Image from the EuRoC dataset.

We provide function stubs with detailed documentation about input and output formats in each file. While exact numerical reproduction of reference outputs is not required (due to the nature of sensor fusion and numerical optimization), we encourage you to use the provided random seed functionality (`np.random.seed()`) to ensure reproducible results during development and debugging.

You may utilize code from previous exercises related to computer vision and estimation theory. While reference implementations are available, you're welcome to use your own implementations if you prefer. The framework is designed to be modular, allowing you to focus on the fusion aspects while leveraging the provided visual-inertial fusion and system infrastructure.

Remark: In this exercise, we use FAST corner as the feature points, but since we do not implement it from scratch, we will not allow the usage of the FAST corner in the Mini project.

2 Exercise 1: IMU Process Model Implementation

2.1 Objective

The first exercise focuses on implementing the fundamental IMU process model that propagates the state estimate between visual measurements. This process model forms the backbone of any visual-inertial system by providing high-frequency state updates using inertial measurements from the IMU. The implementation requires careful handling of several critical components to ensure accurate state propagation.

The process model must properly account for sensor biases by applying corrections to both the gyroscope and accelerometer measurements. These biases are time-varying and must be estimated as part of the state. Additionally, the model needs to compensate for the effect of gravity in the acceleration measurements, as the accelerometer measures specific force rather than true acceleration.

A key challenge is maintaining proper quaternion normalization throughout the integration process. Quaternions representing rotation must always maintain the unit norm, but numerical integration can introduce errors that violate this constraint. The implementation must include mechanisms to periodically normalize the quaternion state to prevent these errors from accumulating.

2.2 Mathematical Formulation

The IMU provides raw measurements of angular velocity ω_m and linear acceleration a_m in the body frame. These measurements contain biases and noise that must be accounted for in the state

propagation. The continuous-time system dynamics are described by a set of differential equations for each component of the state vector.

The orientation quaternion q_{WB} evolves according to:

$$\begin{aligned}\dot{q}_{WB} &= \frac{1}{2}\Omega(\omega)q_{WB} \\ \dot{b}_g &= \eta_{bg} \\ \dot{v}_W &= R_{WB}(q_{WB})(a_m - b_a) + g_W \\ \dot{b}_a &= \eta_{ba} \\ \dot{p}_W &= v_W\end{aligned}\tag{1}$$

The quaternion kinematics matrix $\Omega(\omega)$ plays a crucial role in the orientation propagation and is defined as:

$$\Omega(\omega) = \begin{bmatrix} -[\omega \times] & \omega \\ -\omega^T & 0 \end{bmatrix}\tag{2}$$

Before using the raw measurements in these equations, bias correction must be applied. The corrected angular velocity is computed as $\omega = \omega_m - b_g$ where b_g represents the gyroscope bias. Similarly, the corrected acceleration is given by $a = a_m - b_a$ where b_a is the accelerometer bias. The gravity vector g_W must be properly accounted for in the world frame, typically as $g_W = [0, 0, -9.81]^T m/s^2$. The terms η_{bg} and η_{ba} represent random walk processes that model the evolution of the biases.

The rotation matrix $R_{WB}(q_{WB})$ transforms vectors from the body frame to the world frame and is computed from the current quaternion estimate. Throughout the integration process, the quaternion must be periodically normalized to maintain its unit norm property, ensuring valid rotation representations.

To implement the state integration, we must discretize these continuous-time equations. The midpoint integration method provides a good balance between accuracy and computational efficiency. For a time interval $[t_k, t_{k+1}]$, the integration proceeds as follows:

$$\mathbf{R}_{k+1} = \mathbf{R}_k \exp(\boldsymbol{\omega}_{mid} \Delta t)\tag{3}$$

$$\mathbf{v}_{k+1} = \mathbf{v}_k + (\mathbf{a}_{mid} - \mathbf{g}) \Delta t\tag{4}$$

$$\mathbf{p}_{k+1} = \mathbf{p}_k + \mathbf{v}_k \Delta t + \frac{1}{2}(\mathbf{a}_{mid} - \mathbf{g}) \Delta t^2\tag{5}$$

where $\boldsymbol{\omega}_{mid}$ and \mathbf{a}_{mid} are computed using bias-corrected measurements at the interval midpoint.

3 Stereo Vision Measurements

Geometric Framework

Stereo vision provides rich 3D information through the observation of features from two cameras with known relative pose. The geometric relationship between a 3D point and its projections forms the basis for our measurement model.

Consider a 3D point \mathbf{p}_W in the world frame. This point is transformed into the camera frame through:

$${}_C\mathbf{p} = \mathbf{R}_{CW} {}_W\mathbf{p} + {}_C\mathbf{t}_{CW}\tag{6}$$

where \mathbf{R}_{WC} and \mathbf{t}_{WC} represent the camera pose. For a calibrated stereo camera system with baseline b , the point projects onto the left and right image planes according to:

$$\pi_L({}_C\mathbf{p}) = \begin{bmatrix} f_x \frac{x_C}{z_C} + c_x \\ f_y \frac{y_C}{z_C} + c_y \end{bmatrix}\tag{7}$$

$$\pi_R({}_C\mathbf{p}) = \begin{bmatrix} f_x \frac{x_C - b}{z_C} + c_x \\ f_y \frac{y_C}{z_C} + c_y \end{bmatrix}\tag{8}$$

The camera intrinsics (f_x, f_y, c_x, c_y) define the projection properties of each camera. The baseline b introduces a horizontal disparity between the left and right images, enabling depth perception.

Stereo Reprojection Error

For a stereo observation $\mathbf{z} = [u_L, v_L, u_R, v_R]^T$, the reprojection error is:

$$\mathbf{r}_{proj} = \begin{bmatrix} \pi_L(C\mathbf{p}) - [u_L, v_L]^T \\ \pi_R(C\mathbf{p}) - [u_R, v_R]^T \end{bmatrix} \quad (9)$$

4 Tightly-Coupled Optimization Framework

The tightly-coupled optimization framework integrates information from both the IMU and vision modalities into a unified estimation process. This approach ensures that all sensor measurements contribute collaboratively to refine the estimated state.

State Vector Structure

At the core of the optimization lies the state vector, which encapsulates both the motion state and the observed features within the environment. Specifically, the full state vector in the optimization window is structured as:

$$\mathbf{x} = \left[\underbrace{\mathbf{p}_I, \mathbf{v}_I, \mathbf{q}_{WI}, \mathbf{b}_a, \mathbf{b}_g}_{\text{IMU state}}, \underbrace{\mathbf{p}_{f_1}, \dots, \mathbf{p}_{f_N}}_{\text{Feature positions}} \right] \quad (10)$$

Combined Cost Function

The optimization minimizes a combined cost function:

$$J(\mathbf{x}) = \underbrace{J_{\text{IMU}}(\mathbf{x})}_{\text{IMU term}} + \underbrace{J_{\text{vision}}(\mathbf{x})}_{\text{Vision term}} \quad (11)$$

The IMU term comes from Part 1:

$$J_{\text{IMU}}(\mathbf{x}) = \sum_{k=1}^K \|\mathbf{r}_{\text{IMU},k}(\mathbf{x})\|_{\Sigma_{\text{IMU},k}}^2 \quad (12)$$

where $\mathbf{r}_{\text{IMU},k}$ represents the error between predicted and measured IMU states.

The vision term from Sec. 3 is:

$$J_{\text{vision}}(\mathbf{x}) = \sum_{i=1}^N \sum_{j \in \mathcal{O}_i} \|\mathbf{r}_{proj,ij}(\mathbf{x})\|_{\Sigma_{proj}}^2 \quad (13)$$

Optimization Problem

The complete optimization problem becomes:

$$\mathbf{x}^* = \underset{\mathbf{x}}{\operatorname{argmin}} J(\mathbf{x}) \quad (14)$$

This is solved using iterative methods like Levenberg-Marquardt:

1. Linearize residuals around current estimate:

$$\mathbf{r}(\mathbf{x} + \delta\mathbf{x}) \approx \mathbf{r}(\mathbf{x}) + \mathbf{J}\delta\mathbf{x} \quad (15)$$

2. Solve the normal equations:

$$(\mathbf{J}^T \mathbf{W} \mathbf{J} + \lambda \mathbf{I}) \delta\mathbf{x} = -\mathbf{J}^T \mathbf{W} \mathbf{r} \quad (16)$$

3. Update state: $\mathbf{x} \leftarrow \mathbf{x} + \delta\mathbf{x}$

For implementation, you can utilize SciPy's `least_squares` solver which provides an efficient implementation of the Levenberg-Marquardt algorithm:

The `residual_function` should return the concatenated IMU and vision residuals, properly weighted by their respective covariances. The solver will automatically handle the iterative optimization process and return the optimal state estimate.