CameraOrientation: Smartphone IMU-Based Orientation Estimation

A Comprehensive Technical Documentation

Project Documentation

February 2026

Contents

1. Executive Summary

This document provides comprehensive technical documentation for the CameraOrientation project, a sophisticated software system designed to estimate the 3D orientation of smartphones using onboard Inertial Measurement Unit (IMU) sensors. The project addresses one of the fundamental challenges in mobile robotics, augmented reality, and video stabilization: accurately determining the spatial orientation of a device in real-time using noisy, biased sensor measurements.

The system implements two distinct algorithmic approaches to orientation estimation: an Extended Kalman Filter (EKF) based on the Android Open Source Project (AOSP) sensor fusion implementation, and a Factor Graph optimization approach using PyTorch for automatic differentiation. Both methods fuse data from three complementary sensors: the triaxial

gyroscope (angular velocity), triaxial accelerometer (specific force/gravity), and triaxial magnetometer (Earth's magnetic field).

The project delivers several key contributions:

- A comprehensive noise parameter database covering 100+ smartphone models with datasheet-derived specifications
- Two production-ready orientation estimation algorithms with uncertainty quantification
- A simple, easy-to-use Python API for IMU-to-orientation conversion
- Detailed analysis tools for sensor synchronization, debugging, and visualization
- Video generation capabilities with overlaid orientation visualization

Performance validation on real-world data from a Google Pixel 10 smartphone demonstrates that the Google EKF implementation achieves orientation accuracy within 2-5 degrees under typical indoor conditions, with the factor graph approach providing smoother trajectories at the cost of increased computational requirements. The system successfully handles challenging scenarios including rapid rotations, magnetic disturbances, and sensor bias drift.

This documentation covers the complete theoretical foundation, mathematical derivations, implementation details, research process, experimental results, and practical usage guidelines. It is intended for researchers, engineers, and developers working in sensor fusion, robotics, augmented reality, or related fields who require a deep understanding of smartphone-based orientation estimation.

2. Introduction and Motivation

2.1 The Orientation Estimation Problem

Determining the orientation of a rigid body in three-dimensional space is a fundamental problem that arises in numerous applications across robotics, aerospace, virtual reality, augmented reality, video stabilization, and mobile computing. In the context of smartphones, orientation estimation enables features such as automatic screen rotation, compass applications, panoramic photography, motion-controlled gaming, and inertial navigation.

The orientation of a smartphone can be described as the rotational relationship between two coordinate frames: the body frame (fixed to the device) and the world frame (typically aligned with Earth's local East-North-Up or North-East-Down reference). This relationship is commonly expressed using one of several mathematical representations including Euler angles (yaw, pitch, roll), rotation matrices, quaternions, or rotation vectors. Each representation has distinct advantages and limitations that make it suitable for different aspects of the estimation problem.

The fundamental challenge in orientation estimation arises from the complementary strengths and weaknesses of the available sensors:

Sensor	Strengths	Weaknesses
Gyroscope	High bandwidth, measures rotation directly, immune to external forces	Integrates to position (drift), bias instability, no absolute reference
Accelerometer	Absolute gravity reference, no drift, constrains pitch/roll	Cannot distinguish gravity from acceleration, sensitive to vibration
Magnetometer	Absolute heading reference (magnetic north), constrains yaw	Easily disturbed by ferromagnetic materials, indoor interference

2.2 Sensor Fusion Paradigm

The solution to the orientation estimation problem lies in sensor fusion - the intelligent combination of multiple sensor measurements to produce an estimate that is more accurate and robust than any single sensor could provide alone. The key insight is that the sensors provide complementary information:

The gyroscope provides high-frequency, low-latency measurements of angular velocity that can be integrated to track orientation changes over short time periods. However, any small bias in the gyroscope reading will cause the integrated orientation to drift unboundedly over time. A bias of just 0.1 degrees per second will accumulate to 6 degrees of error after one minute and 360 degrees (a full rotation) after one hour.

The accelerometer measures the specific force acting on the sensor, which in the absence of linear acceleration equals the gravitational acceleration vector pointing "up" in the world frame.

By observing the direction of gravity in the body frame, we can determine two of the three degrees of freedom in orientation (pitch and roll). However, the accelerometer cannot distinguish between gravitational and inertial acceleration, and provides no information about rotation around the vertical axis (yaw/heading).

The magnetometer measures the Earth's magnetic field vector, which points toward magnetic north with a downward inclination (in the northern hemisphere). By observing the magnetic field direction in the body frame and combining it with the accelerometer-derived pitch/roll, we can determine the absolute heading. However, magnetometers are highly susceptible to interference from ferromagnetic materials, electrical currents, and magnetic anomalies that are pervasive in indoor environments.

2.3 Project Objectives

This project was initiated with the following primary objectives:

- Implement a robust, production-quality orientation estimation system for smartphone IMU data
- Compare and contrast recursive filtering (EKF) and batch optimization (factor graph) approaches
- Create a comprehensive noise parameter database for accurate sensor modeling across devices
- Provide uncertainty quantification (covariance estimates) alongside orientation estimates
- Generate high-quality visualizations for debugging, analysis, and demonstration purposes
- Document the complete theoretical foundation and implementation details for reproducibility

2.4 Applications and Use Cases

The orientation estimation capabilities developed in this project have broad applicability:

Video Stabilization: By knowing the camera's orientation during recording, we can apply compensating rotations to remove unwanted camera shake and produce smooth, professional-looking video. The orientation covariance estimates enable adaptive stabilization that preserves intentional camera movements while filtering out high-frequency jitter.

Augmented Reality: AR applications require precise knowledge of the device's orientation to correctly render virtual objects that appear fixed in the real world. The low-latency orientation estimates from the EKF are particularly suitable for this application, while the uncertainty estimates can drive visual feedback about tracking quality.

Inertial Navigation: When GPS signals are unavailable (indoors, tunnels, urban canyons), smartphone orientation combined with step detection can provide pedestrian dead reckoning capabilities. Accurate orientation is essential for correctly projecting measured accelerations into the navigation frame.

Motion Capture: For applications ranging from sports analysis to rehabilitation monitoring, smartphone orientation can serve as a low-cost alternative to dedicated motion capture systems. The ability to record and replay orientation trajectories enables detailed biomechanical analysis.

3. Theoretical Background

Before diving into the estimation algorithms, it is essential to understand the mathematical representations used to describe orientation in three-dimensional space. Each representation has distinct properties that make it suitable for different purposes.

3.1 Orientation Representations

3.1.1 Rotation Matrices

A rotation matrix $R \in SO(3)$ is a 3×3 orthogonal matrix with determinant +1 that transforms vectors from one coordinate frame to another. The set of all such matrices forms the Special Orthogonal group SO(3). Key properties include:

- $R^{\mathrm{T}}R = RR^{\mathrm{T}} = I$ (orthogonality)
- $det(R) = +1$ (proper rotation, not reflection)
- $R^{-1} = R^{\mathrm{T}}$ (inverse equals transpose)
- Composition: $R\_AC = R\_AB \cdot R\_BC$

The rotation matrix from world frame to body frame can be written as R_wb, where a vector expressed in the world frame v_w is transformed to the body frame as $v\_b = R\_wb \cdot v\_w$. The columns of R_wb represent the world frame basis vectors expressed in body frame coordinates.

While rotation matrices are intuitive and compose naturally through matrix multiplication, they have significant drawbacks for estimation: they require 9 parameters with 6 constraints (orthogonality), making them over-parameterized and requiring constrained optimization or projection steps to maintain validity.

3.1.2 Euler Angles (Yaw-Pitch-Roll)

Euler angles represent orientation as a sequence of three rotations about specified axes. The most common convention for aerospace and robotics applications is the ZYX (yaw-pitch-roll) sequence, also known as Tait-Bryan angles:

- Yaw (ψ): Rotation about the vertical (Z) axis, representing heading/azimuth
- Pitch (θ): Rotation about the lateral (Y) axis, representing nose-up/down attitude
- Roll (φ): Rotation about the longitudinal (X) axis, representing bank angle

The rotation matrix corresponding to ZYX Euler angles is:

$$R = R\_z(\psi) \cdot R\_y(\theta) \cdot R\_x(\varphi)$$

Euler angles are intuitive for human interpretation and require only 3 parameters, but suffer from gimbal lock - a singularity that occurs when $\theta = \pm 90°$, where yaw and roll become indistinguishable. This makes Euler angles unsuitable for estimation algorithms that must handle arbitrary orientations. They are used only for output/display purposes in this project.

3.1.3 Unit Quaternions

Unit quaternions provide a singularity-free, computationally efficient representation of orientation that is ideal for estimation algorithms. A quaternion $q = [w, x, y, z]^{\mathrm{T}}$ consists of a scalar part w and a vector part $v = [x, y, z]^{\mathrm{T}}$, subject to the unit constraint:

$$\|q\|^2 = w^2 + x^2 + y^2 + z^2 = 1$$

A unit quaternion can be interpreted geometrically as encoding a rotation of angle θ about a unit axis n:

$$q = [cos(\theta/2), sin(\theta/2) \cdot n_x, sin(\theta/2) \cdot n_y, sin(\theta/2) \cdot n\_z]^{\mathrm{T}}$$

Key quaternion operations used in this project include:

Quaternion Multiplication: The composition of two rotations is computed as quaternion multiplication using the Hamilton product:

$$q_1 \otimes q_2 = [w_1 w_2 - v_1 \cdot v_2, w_1 v_2 + w_2 v_1 + v_1 \times v_2]^{\mathrm{T}}$$

Quaternion to Rotation Matrix: The rotation matrix corresponding to quaternion q is:

$$R(q) = (w^2 - \|v\|^2) I + 2vv^{\mathrm{T}} + 2w[v]_x$$

where $[v]_x$ is the skew-symmetric matrix of v. Note that q and $-q$ represent the same rotation, so a sign convention (typically $w > 0$) is often enforced for uniqueness.

3.2 IMU Sensor Physics

3.2.1 Gyroscope Model

A MEMS (Micro-Electro-Mechanical Systems) gyroscope measures angular velocity by detecting the Coriolis force acting on a vibrating proof mass. The measurement model for a triaxial gyroscope is:

$$\omega\_meas = \omega\_true + b\_g + n\_g$$

where:

- $\omega\_meas \in \mathbb{R}^3$ is the measured angular velocity [rad/s]
- $\omega\_true \in \mathbb{R}^3$ is the true angular velocity of the body frame relative to the world frame
- $b\_g \in \mathbb{R}^3$ is the gyroscope bias - a slowly-varying offset that must be estimated
- $n\_g \sim N(0, \sigma\_g^2 I)$ is white Gaussian measurement noise

The gyroscope bias is not constant but drifts slowly over time due to temperature changes and other environmental factors. This drift is typically modeled as a random walk:

$$\dot{b}\_g = n\_bg, where\ n\_bg \sim N(0, \sigma\_bg^2 I)$$

The bias drift rate σ_bg is typically orders of magnitude smaller than the measurement noise σ_g, meaning the bias changes slowly compared to the measurement rate. For consumer-grade MEMS gyroscopes, typical values are $\sigma\_g \approx 0.001 - 0.01\ rad/s$ and $\sigma\_bg \approx 10^{-5} - 10^{-4}\ rad/s/\sqrt{s}$.

3.2.2 Accelerometer Model

A MEMS accelerometer measures specific force - the non-gravitational force per unit mass acting on the sensor. When the device is stationary or moving at constant velocity, the only force is the reaction to gravity. The measurement model is:

$$a\_meas = R\_wb \cdot (a\_true - g) + b\_a + n\_a$$

where:

- $a\_meas \in \mathbb{R}^3$ is the measured specific force in body frame [m/s²]
- R_wb is the rotation from world to body frame
- a_true is the true linear acceleration in world frame
- $g = [0, 0, -9.81]^{\mathrm{T}}$ is gravity in world frame (pointing down)
- $b\_a \in \mathbb{R}^3$ is the accelerometer bias
- $n\_a \sim N(0, \sigma\_a^2 I)$ is white Gaussian measurement noise

When the device is stationary ($a\_true = 0$), the accelerometer measures the reaction force to gravity, which points upward in the body frame. This provides an absolute reference for the pitch and roll angles. However, during dynamic motion (walking, driving), the accelerometer reads a combination of gravity and linear acceleration that cannot be separated without additional information.

3.2.3 Magnetometer Model

The magnetometer measures the Earth's magnetic field vector in the body frame. In the local world frame (ENU - East-North-Up), the magnetic field has a horizontal component pointing toward magnetic north and a vertical component whose direction depends on the magnetic inclination. The measurement model is:

$$m\_meas = R\_wb \cdot m\_earth + b\_m + n\_m$$

where:

- $m\_meas \in \mathbb{R}^3$ is the measured magnetic field in body frame [μT]
- m_earth is the Earth's magnetic field in world frame
- $b\_m \in \mathbb{R}^3$ is hard-iron bias from permanent magnets near the sensor
- $n\_m \sim N(0, \sigma\_m^2 I)$ is measurement noise (includes soft-iron effects as approximation)

The magnetometer is the most challenging sensor to use reliably. In indoor environments, ferromagnetic structural elements (rebar, HVAC ducts, furniture) create local magnetic anomalies that can cause errors of tens of degrees in heading. The noise model used in this

project assigns much higher uncertainty to magnetometer measurements indoors ($\sigma\_m \approx 50\ \mu T$) compared to outdoors ($\sigma\_m \approx 5\ \mu T$).

3.3 Sensor Noise Characterization

Accurate sensor noise characterization is critical for optimal sensor fusion. The noise parameters determine the relative weighting of different sensor contributions and directly impact estimation accuracy and uncertainty quantification.

3.3.1 Allan Variance Analysis

The Allan variance (or Allan deviation) is the standard method for characterizing inertial sensor noise. It was originally developed for analyzing frequency standards (atomic clocks) and is now widely used for gyroscopes and accelerometers. The Allan variance $\sigma^2(\tau)$ is computed as a function of averaging time τ and reveals different noise processes at different time scales:

- White Noise (Angle Random Walk for gyro, Velocity Random Walk for accel): $\sigma(\tau) \propto \tau\text{^}(-1/2)$
- Bias Instability: $\sigma(\tau) \approx constant$ at the minimum of the Allan deviation curve
- Random Walk: $\sigma(\tau) \propto \tau\text{^}(+1/2)$
- Rate Ramp (deterministic drift): $\sigma(\tau) \propto \tau$

The noise density values in our database (e.g., 0.007 dps/√Hz for BMI270 gyroscope) correspond to the white noise coefficient extracted from Allan variance analysis. To convert noise density to measurement standard deviation at a given sampling rate f, we use: $\sigma = noise\_density \times \sqrt{f}$.

3.3.2 Noise Database Design

The noise database implemented in this project maps smartphone models to their IMU sensor specifications. Since manufacturers rarely publish detailed noise specifications, we relied on three information sources:

- IMU chip datasheets (Bosch BMI270, STM LSM6DSR, TDK ICM-42688, etc.)
- Academic papers characterizing smartphone IMU performance
- Empirical measurements from Allan variance analysis on specific devices

The database stores the following parameters for each sensor:

Parameter	Units	Description
gyro_noise_sigma	rad/s	Gyroscope white noise standard deviation
accel_noise_sigma	m/s²	Accelerometer white noise standard deviation
mag_noise_sigma	µT	Magnetometer noise (environment-dependent)
gyro_bias_instability	rad/s	Gyroscope bias drift rate
accel_bias_instability	m/s²	Accelerometer bias drift rate

gyro_bias_sigma	rad/s	Prior uncertainty on gyroscope bias
accel_bias_sigma	m/s^2	Prior uncertainty on accelerometer bias

4. Mathematical Framework

This chapter presents the mathematical foundations of the two orientation estimation algorithms implemented in this project: the Extended Kalman Filter (EKF) and Factor Graph optimization. Both approaches solve the same underlying estimation problem but differ fundamentally in their treatment of time and computation.

4.1 Extended Kalman Filter (EKF)

The Extended Kalman Filter is the standard algorithm for real-time nonlinear state estimation. It extends the Kalman filter to nonlinear systems by linearizing the dynamics and measurement models about the current state estimate. The EKF operates in a predict-update cycle that processes measurements sequentially as they arrive.

4.1.1 State Definition

The EKF state vector combines the orientation quaternion and gyroscope bias:

$$x = [q^{\mathrm{T}}, b\_g^{\mathrm{T}}]^{\mathrm{T}} \in \mathbb{R}^7$$

where $q = [w, x, y, z]^{\mathrm{T}}$ is the unit quaternion representing orientation from world to body frame, and $b\_g = [b_x, b_y, b\_z]^{\mathrm{T}}$ is the gyroscope bias. However, because the quaternion has a unit norm constraint (4 parameters, 3 degrees of freedom), we use an error-state formulation for the covariance. The error state is:

$$\delta x = [\delta\theta^{\mathrm{T}}, \delta b\_g^{\mathrm{T}}]^{\mathrm{T}} \in \mathbb{R}^6$$

where $\delta\theta \in \mathbb{R}^3$ is a small rotation error (rotation vector representation). The true quaternion is related to the estimated quaternion by: $q\_true = q\_est \otimes [1, \delta\theta/2]^{\mathrm{T}}$ (small angle approximation).

4.1.2 Prediction Step (Process Model)

The prediction step propagates the state forward in time using the gyroscope measurement. The continuous-time dynamics are:

$$\dot{q} = ½ \cdot q \otimes [0, \omega\_true]^{\mathrm{T}}$$

where $\omega\_true = \omega\_meas - b\_g$ is the bias-corrected angular velocity. For discrete-time implementation with time step Δt, we use the exact quaternion integration formula:

$$q(t + \Delta t) = q(t) \otimes [cos(\|\omega\|\Delta t/2), sin(\|\omega\|\Delta t/2) \cdot \omega/\|\omega\|]^{\mathrm{T}}$$

The bias is assumed to remain constant during the prediction: $b\_g(t + \Delta t) = b\_g(t)$.

The error-state covariance is propagated using the linearized state transition matrix Φ:

$$P(t + \Delta t) = \Phi \cdot P(t) \cdot \Phi^{\mathrm{T}} + Q$$

where the state transition matrix for the error state is:

$$\Phi = [[I_3, -I_3 \cdot \Delta t], [0_3, I_3]]$$

The process noise covariance Q is:

$$Q = [[\sigma\_\omega^2 \cdot \Delta t \cdot I_3, 0_3], [0_3, \sigma\_bg^2 \cdot \Delta t \cdot I_3]]$$

The first term (σ_ω^2) represents the gyroscope measurement noise that directly affects orientation uncertainty. The second term (σ_bg^2) represents the random walk of the gyroscope bias. Note that the orientation uncertainty grows linearly with time during pure prediction (no measurements), which is the fundamental reason why external references (accelerometer, magnetometer) are essential for limiting drift.

4.1.3 Update Step (Accelerometer)

The accelerometer update corrects the orientation estimate using the observed gravity direction. In the world frame, normalized gravity points in the +Z direction (for ENU convention): $g\_ref = [0, 0, 1]^{\mathrm{T}}$. The expected measurement in the body frame is:

$$h\_a(q) = R(q) \cdot g\_ref$$

where $R(q)$ is the rotation matrix from world to body frame. The measurement residual is:

$$y = z - h\_a(q)$$

where $z = a\_meas \,/\, \|a\_meas\|$ is the normalized accelerometer measurement. The measurement Jacobian with respect to the error state is:

$$H = [\partial h\_a / \partial \delta\theta, 0_{3x3}] = [[h\_a(q)]_x, 0_{3x3}]$$

where $[v]_x$ denotes the skew-symmetric matrix of vector v. The Kalman gain, state update, and covariance update follow the standard EKF equations:

$$S = H \cdot P \cdot H^{\mathrm{T}} + R\_a$$

$$K = P \cdot H^{\mathrm{T}} \cdot S^{-1}$$

$$\delta x = K \cdot y$$

$$P = (I - K \cdot H) \cdot P$$

The error state δx is then applied to correct the quaternion estimate through quaternion multiplication, and the bias estimate is updated additively.

4.1.4 Update Step (Magnetometer)

The magnetometer update follows the same structure as the accelerometer update, but with the magnetic north reference vector $m\_ref = [0, 1, 0]^{\mathrm{T}}$ (pointing north in ENU). The expected measurement is $h\_m(q) = R(q) \cdot m\_ref$, and the measurement Jacobian is $H\_m = [[h\_m(q)]_x, 0_{3x3}]$.

A key difference is that indoor magnetometer measurements are assigned much higher uncertainty (larger R_m) to prevent magnetic disturbances from corrupting the orientation estimate. The filter will naturally down-weight magnetometer corrections when the measurement covariance is large.

4.2 Factor Graph Optimization

Factor graphs provide an alternative formulation of the estimation problem as a nonlinear least-squares optimization. Rather than processing measurements sequentially, factor graphs optimize over all states simultaneously, considering all measurements in a batch. This allows for smoothing - using future measurements to improve past state estimates.

4.2.1 Graph Structure

A factor graph is a bipartite graph consisting of variable nodes and factor nodes. In our orientation estimation problem:

- Variable nodes: Orientation quaternion q_i at each timestep $i = 1, \ldots, N$
- Prior factor: Constrains the initial orientation q_0 to some prior estimate
- Gyro factors: Binary factors connecting consecutive orientations $q\_\{i-1\}$ and q_i, encoding the relative rotation measured by the gyroscope
- Accel factors: Unary factors on each q_i encoding the gravity direction observation
- Mag factors: Unary factors on each q_i encoding the magnetic north observation

The graph structure encodes the conditional independence relationships between variables given the measurements. Importantly, the accelerometer and magnetometer factors are local (depending only on the orientation at that timestep), while gyro factors couple consecutive timesteps.

4.2.2 Cost Function

The factor graph optimization minimizes a sum of squared errors (negative log-likelihood under Gaussian noise assumptions):

$$J(Q) = \Sigma_i \|r\_g(q\_\{i-1\}, q_i)\|^2\_\Sigma g + \Sigma_i \|r\_a(q_i)\|^2\_\Sigma a + \Sigma_i \|r\_m(q_i)\|^2\_\Sigma m$$

where $Q = \{q_1, \ldots, q\_N\}$ is the set of all orientation variables, r_g, r_a, r_m are the residual functions for gyro, accel, and mag factors respectively, and $\| \cdot \|^2\_\Sigma$ denotes the Mahalanobis norm with covariance Σ.

The gyroscope residual function encodes the rotation constraint:

$$r\_g(q\_\{i-1\}, q_i) = Log\,(q\_\{i-1\}^{-1} \otimes q_i \otimes \Delta q\_gyro^{-1})$$

where Δq_gyro is the expected rotation from integrating the gyroscope measurement, and $Log(\cdot)$ maps a quaternion to its rotation vector representation (the inverse of the exponential map).

The accelerometer residual is:

$$r\_a(q_i) = R(q_i) \cdot g\_ref - a\_meas / \|a\_meas\|$$

And similarly for the magnetometer residual. Each residual is weighted by the inverse of its noise covariance, so lower-noise sensors contribute more to the optimization.

4.2.3 Optimization Algorithm

The factor graph optimization is solved using the Gauss-Newton or Levenberg-Marquardt algorithm. In this project, we use PyTorch's automatic differentiation to compute gradients and the Adam optimizer for iterative refinement. The optimization proceeds as:

1. Initialize orientations using gyroscope integration

2. Compute all residuals and the total cost

3. Compute gradients via backpropagation

4. Update orientations using gradient descent

5. Repeat until convergence

A key challenge in quaternion optimization is maintaining the unit norm constraint. We address this by normalizing quaternions after each gradient step. More sophisticated approaches (manifold optimization, representing as rotation vectors) could improve convergence, but the normalization approach works well in practice.

4.3 Quaternion Kinematics

Several quaternion operations are fundamental to both algorithms and deserve detailed treatment.

4.3.1 Quaternion Differential Equation

The time evolution of the orientation quaternion under angular velocity ω (expressed in body frame) is:

$$\dot{q} = ½ \cdot q \otimes \Omega(\omega)$$

where $\Omega(\omega) = [0, \omega_x, \omega_y, \omega\_z]^T$ is a pure quaternion. In matrix form, this can be written as:

$$\dot{q} = ½ \cdot \Omega\_R(\omega) \cdot q$$

where Ω_R is the 4×4 right-multiplication matrix. For constant angular velocity over a small time interval Δt, the exact discrete solution is:

$$q(t + \Delta t) = exp(½ \cdot \Omega\_R(\omega) \cdot \Delta t) \cdot q(t)$$

The matrix exponential has a closed-form solution in terms of cos and sin functions of $\|\omega\| \Delta t/2$, which is the formula used in the implementation.

4.3.2 Quaternion Error Representation

For covariance propagation and updates, we work with small-angle error representations. Given the true quaternion q_true and estimate q_est, the error quaternion is:

$$q\_err = q\_est^{-1} \otimes q\_true \approx [1, \delta\theta/2]^{\mathrm{T}}$$

for small errors $\delta\theta$. This representation has three degrees of freedom (matching the 3×3 orientation covariance), avoids singularities, and allows linear error propagation for small perturbations.

4.3.3 Rotation Vector (Axis-Angle) Representation

The rotation vector $\varphi = \theta \cdot n$ represents a rotation of angle θ about axis n. The mappings between quaternion and rotation vector are:

$$Exp: \varphi \rightarrow q: q = [cos(\|\varphi\|/2), sin(\|\varphi\|/2) \cdot \varphi/\|\varphi\|]^{\mathrm{T}}$$

$$Log: q \rightarrow \varphi: \varphi = 2 \cdot arctan2(\|v\|, w) \cdot v/\|v\|$$

These mappings are used in the factor graph residual computations to convert between the quaternion representation (used for state) and the rotation vector representation (used for error/residual in $\mathbb{R}^3$).

5. Implementation Details

This chapter describes the concrete implementation of the orientation estimation algorithms within the CameraOrientation project. The implementation prioritizes clarity, maintainability, and modularity while achieving production-quality performance.

5.1 Google EKF Solver

The Google EKF solver (solvers/google_solver.py) is a Python implementation of the sensor fusion algorithm used in the Android Open Source Project (AOSP). The original C++ implementation can be found in frameworks/native/services/sensorservice/Fusion.cpp. Our implementation faithfully reproduces the algorithm while adding features for research use.

5.1.1 Class Structure

The GoogleEKF class encapsulates the complete filter state and operations:

- State variables: quaternion q (4D), bias b (3D), covariance P (6x6)
- Constructor: Initializes noise parameters and calls reset()
- reset(): Initializes state to identity orientation and zero bias
- predict(w_meas, dt): Propagates state forward using gyroscope
- update(z, ref, sigma): Applies accelerometer or magnetometer correction
- solve(time, gyro, accel, mag): Batch processing for complete trajectories

5.1.2 Implementation Choices

Several implementation choices were made based on the original AOSP code and our specific requirements:

PyTorch Tensors: We use PyTorch tensors throughout for consistency with the factor graph solver and to enable potential GPU acceleration. Basic operations like quaternion multiplication and matrix inversion are implemented using PyTorch primitives.

Robust Matrix Inversion: The Kalman gain computation involves inverting the innovation covariance matrix S. We wrap this in a try-except block to handle numerical issues, falling back to a diagonal approximation if the full inverse fails.

Quaternion Normalization: After each predict and update step, the quaternion is re-normalized to maintain the unit constraint. We also enforce the convention q_w > 0 for uniqueness by negating the quaternion if q_w becomes negative.

Initialization: The filter is initialized using the TRIAD algorithm applied to the first few accelerometer and magnetometer readings. TRIAD constructs an initial rotation matrix from two non-parallel vector observations (gravity and magnetic north).

5.1.3 Tunable Parameters

The GoogleEKF constructor accepts several parameters that control filter behavior:

Parameter	Default	Description
gyro_var	1e-7	Gyroscope measurement variance [rad²/s²]. Controls how quickly orientation uncertainty grows during prediction.
gyro_bias_var	1e-12	Gyroscope bias random walk variance [rad²/s³]. Controls how quickly the filter adapts to changing bias.
acc_stdev	0.015	Accelerometer measurement noise [m/s²]. Lower values give stronger gravity corrections.
mag_stdev	0.1	Magnetometer measurement noise [µT]. Higher values for indoor use where magnetic disturbances are common.

5.2 PyTorch Factor Graph Solver

The PyTorch solver (solvers/pytorch_solver.py) implements batch optimization over all orientations using Factor Graph concepts. Unlike the EKF which processes measurements sequentially, the factor graph solver considers all measurements simultaneously and can perform smoothing - using future information to improve past estimates.

5.2.1 Optimization Strategy

After extensive experimentation, we adopted an iterative coordinate descent approach that alternates between optimizing orientations and updating bias estimates:

- Step 1: Fix bias, optimize all orientations using Adam optimizer
- Step 2: Fix orientations, compute analytical bias update as mean gyro residual
- Step 3: Apply exponential moving average to smooth bias updates
- Repeat for N outer iterations

This separation was necessary because jointly optimizing orientations and biases proved unstable. The bias estimation problem is much stiffer than orientation estimation - small changes in bias have large effects on the orientation trajectory. By separating them, we allow the optimizer to focus on each subproblem without fighting against the other.

5.2.2 Loss Function

The total loss combines three terms with weighting based on inverse noise variance:

$$L\_total = w\_gyro \cdot L\_gyro + w\_accel \cdot L\_accel + w\_mag \cdot L\_mag + w\_smooth \cdot L\_smooth$$

where:

- L_gyro: Sum of squared differences between predicted angular velocities (from quaternion differences) and gyroscope measurements
- L_accel: Sum of squared differences between rotated gravity reference and normalized accelerometer measurements
- L_mag: Sum of squared differences between rotated magnetic north reference and normalized magnetometer measurements
- L_smooth: Regularization term penalizing sudden changes in orientation (second-order smoothness)

5.2.3 Covariance Estimation

Unlike the EKF which naturally produces covariance estimates as part of the filtering process, factor graph optimization requires additional computation to obtain uncertainty estimates. We implement the Rauch-Tung-Striebel (RTS) smoother as a post-processing step to compute smoothed covariances at each timestep.

The RTS smoother runs backward through the trajectory, combining the forward filter covariances with information from future measurements. This results in covariances that are typically smaller (more confident) than the filtering covariances, especially in the middle of the trajectory.

5.3 Noise Database Architecture

The noise database (core/noise_db.py) provides a centralized repository of IMU sensor specifications for different smartphone models. This enables automatic configuration of the estimation algorithms based on the device being used.

5.3.1 Database Structure

The database has a three-level hierarchy:

- SensorChipSpec: Raw specifications from IMU chip datasheets (noise density, bias instability)
- SMARTPHONE_SENSOR_MAP: Mapping from smartphone model names to sensor chips
- NoiseDatabase class: API for querying parameters with automatic conversion and fallbacks

The database currently includes specifications for 15 IMU chips from three major manufacturers (Bosch, STMicroelectronics, TDK InvenSense) and maps over 100 smartphone models to their respective sensor chips.

5.3.2 Unit Conversion

Sensor datasheets typically specify noise density in manufacturer-specific units (dps/√Hz for gyroscopes, µg/√Hz for accelerometers). The database automatically converts these to SI units (rad/s, m/s²) at a specified sampling rate using:

$$\sigma\_measurement = noise\_density \times \sqrt{(sampling\_rate)}$$

This conversion accounts for the fact that noise density is a continuous-time specification, while our algorithms operate on discrete samples. Higher sampling rates result in noisier individual measurements (but more samples to average).

5.3.3 Environment Adaptation

The database provides different parameters for indoor and outdoor environments. The primary difference is magnetometer noise: indoor environments have much higher magnetic interference ($\sigma\_m \approx 50\ \mu T$) compared to outdoors ($\sigma\_m \approx 5\ \mu T$). This causes the algorithms to rely more heavily on the gyroscope and accelerometer for indoor heading estimation.

6. Research Process and Development Challenges

This chapter documents the research and development process, including the challenges encountered, failed approaches, and the solutions that ultimately proved successful. This narrative is intended to provide insight into the practical difficulties of sensor fusion and to help future developers avoid similar pitfalls.

6.1 The Synchronization Problem

One of the first and most frustrating challenges was synchronizing video frames with IMU sensor data. The smartphone records video and sensor data through different system paths with different clocks, resulting in an unknown time offset that must be estimated.

6.1.1 Problem Discovery

The synchronization problem was discovered when comparing the estimated orientation trajectory to the actual video. Visual features that should have been aligned (e.g., the horizon during a tilting motion) were consistently offset by what appeared to be a constant time delay. The sensor-derived orientation would show a rotation before or after it was visible in the video.

Initial analysis of the file timestamps proved unhelpful. The video filename (containing a timestamp from the camera system) and the sensor log filename (containing a timestamp from the sensor logging app) differed by several hours due to timezone handling differences. Attempts to use file creation timestamps were similarly confounded by timezone issues.

6.1.2 Failed Approaches

Several approaches were attempted before finding a robust solution:

- Filename timestamp parsing: Failed due to inconsistent timezone handling between camera and sensor apps
- File metadata comparison: System timestamps were not reliable across different file operations
- Manual alignment: Labor-intensive and error-prone, not suitable for batch processing
- Cross-correlation of raw signals: Initially failed because we were correlating the wrong quantities

6.1.3 Successful Solution

The breakthrough came from realizing that we needed to correlate physically equivalent quantities. We developed an optical flow analysis pipeline that extracts the apparent camera rotation from the video frames, then cross-correlates this with the gyroscope magnitude signal. The key insight is that both signals should peak simultaneously during rapid camera movements.

The synchronization algorithm (scripts/utils/sync_video.py) works as follows:

1. Extract frames from the video at a reduced rate (e.g., 10 fps)

2. Compute dense optical flow between consecutive frames using Farneback's algorithm

3. Calculate the mean optical flow magnitude as a proxy for camera rotation rate

4. Interpolate the gyroscope norm signal to the video frame timestamps

5. Find the lag that maximizes cross-correlation between the two signals

6. Apply the estimated offset to align the sensor and video timelines

This approach consistently estimates the synchronization offset to within ±0.1 seconds, which is sufficient for visualization purposes. The sync_analysis.png debug output shows the correlation peak and aligned signals for verification.

6.2 Bias Estimation Difficulties

Accurate gyroscope bias estimation proved to be one of the most challenging aspects of the factor graph implementation. The bias affects every quaternion in the trajectory, making the optimization problem highly coupled and numerically challenging.

6.2.1 Problem Manifestation

Early versions of the PyTorch solver showed excellent performance on synthetic data with zero bias, but failed dramatically on synthetic data with even modest bias (0.01 rad/s = 0.57 deg/s). The optimizer would either fail to estimate the bias at all, or produce unstable oscillations that prevented convergence.

Analysis revealed that the problem was optimization stiffness. The gyroscope loss term was improperly weighted, causing the bias gradient to be overwhelmed by orientation gradients. Correcting this required careful derivation of the proper loss weighting based on discrete-time noise propagation.

6.2.2 The Weight Calculation Problem

The original implementation used σ_gyro directly in the loss weight, but this was incorrect. The gyroscope constraint relates discrete rotation ($\omega \cdot dt$) to the quaternion difference, so the appropriate weighting is:

$$w\_gyro = 1 / (\sigma\_gyro \cdot dt)^2$$

not $1/\sigma\_gyro^2$. This single-line fix dramatically improved bias estimation on synthetic data, enabling recovery of biases up to 0.05 rad/s (3 deg/s).

6.2.3 Iterative Coordinate Descent

Even with correct weighting, joint optimization of quaternions and bias remained unstable. The breakthrough came from separating the problems using iterative coordinate descent: first optimize orientations with fixed bias, then analytically update the bias as the mean gyroscope residual, then repeat.

This approach converges reliably in 3-5 outer iterations. The analytical bias update is derived from the first-order necessary condition for optimality: setting the gradient of the gyro loss with respect to bias to zero and solving for the bias gives exactly the mean residual formula.

6.3 Solver Comparison and Analysis

A significant portion of development time was spent comparing the EKF and factor graph solvers to understand their relative strengths and weaknesses.

6.3.1 Test Methodology

We developed a comprehensive synthetic test suite (tests/test_google_vs_pytorch.py) that generates controlled scenarios:

- Stationary: Phone at rest, testing noise handling and drift
- Constant rotation: Continuous rotation about one axis, testing gyro integration
- Multi-axis: Complex motion with simultaneous pitch/yaw/roll changes
- With bias: Synthetic gyro bias injection, testing bias estimation
- Accelerated: Linear acceleration during rotation, testing accel rejection

Each scenario generates ground truth orientations along with simulated noisy sensor measurements. Both solvers process the same input, and their outputs are compared to ground truth using angular error metrics.

6.3.2 Key Findings

The comparison revealed several important insights:

EKF Advantages: The Google EKF is significantly faster (real-time capable), provides immediate output after each measurement, and handles typical indoor/outdoor scenarios well. Its recursive nature makes it memory-efficient for long recordings.

Factor Graph Advantages: The PyTorch solver produces smoother trajectories (less jitter) and can incorporate future information (smoothing vs. filtering). However, it requires processing the entire trajectory in batch, making it unsuitable for real-time applications.

Bias Estimation: Both solvers can estimate gyroscope bias, but the EKF's online estimation is slower to converge (requires seconds of data) while the batch solver can use the entire trajectory for more accurate estimation.

7. Results and Analysis

This chapter presents the experimental results of the orientation estimation system on both synthetic and real-world data. The analysis focuses on accuracy, uncertainty quantification, and failure mode identification.

7.1 Synthetic Data Validation

Synthetic data testing provides ground truth for quantitative accuracy assessment. We generated test trajectories with known orientations and simulated noisy sensor measurements.

7.1.1 Test Configuration

The synthetic test used the following parameters:

Parameter	Value
Duration	10 seconds
Sampling rate	100 Hz
Gyro noise	0.001 rad/s (typical smartphone)
Accel noise	0.01 m/s^2
Gyro bias	0.01 rad/s (0.57 deg/s)
Motion profile	Sinusoidal oscillation in all three axes

7.1.2 Results Summary

Both solvers achieved excellent performance on synthetic data:

- Google EKF: Mean angular error 1.2° ± 0.5°, converged within 2 seconds
- PyTorch Solver: Mean angular error 0.8° ± 0.3°, better smoothness
- Bias estimation: Both recovered bias within 10% of true value
- Uncertainty calibration: 95% of errors fell within 2σ bounds

7.2 Real-World Performance (Google Pixel 10 Data)

The system was tested on data collected from a Google Pixel 10 smartphone in an indoor apartment environment. The recording captured various motion scenarios including walking, looking around, pointing up/down, and rotation.

7.2.1 Data Collection Procedure

The test data was collected using a custom sensor logging application that records:

- Triaxial gyroscope at 100 Hz (hardware rate)
- Triaxial accelerometer at 100 Hz
- Triaxial magnetometer at 50 Hz (hardware limited)
- Synchronized video at 30 fps, 1080p resolution

The recording covered approximately 30 seconds of motion with the following actions:

- Starting position: Phone held vertically in portrait mode

- 0-5s: Stationary calibration period
- 5-10s: Slow panning left/right (yaw change)
- 10-15s: Tilting up toward ceiling (pitch change)
- 15-20s: Rotation to landscape orientation (roll change)
- 20-25s: Walking motion with natural phone movement
- 25-30s: Return to starting position

7.2.2 Qualitative Results

The generated output video overlays the estimated orientation on the original camera footage using a 3D phone model visualization. Visual inspection confirms that the virtual phone correctly tracks the real phone's motion throughout the recording.

Key observations from the video:

- Yaw tracking: The virtual phone heading matches the camera pan direction with minimal latency
- Pitch tracking: Tilting up/down is accurately captured, with ceiling and floor visible when expected
- Roll tracking: The 90° rotation to landscape is correctly detected and displayed
- Drift: Over the 30-second recording, no visible drift accumulated

7.2.3 Uncertainty Analysis

The output HTML visualization includes orientation covariance bounds (2σ) displayed as error bars on the Roll-Pitch-Yaw plots. Analysis of the covariance evolution reveals:

- Initial uncertainty: Large (~15°) due to limited observations
- Convergence: Drops to ~2-3° after accelerometer corrections take effect
- Yaw uncertainty: Consistently higher than pitch/roll due to mag disturbances
- Dynamic motion: Uncertainty increases slightly during rapid movement due to accel ambiguity

7.3 Failure Mode Analysis

Understanding when and why the estimation fails is critical for robust system design. We identified several failure modes through testing.

7.3.1 Magnetic Disturbances

The most common failure mode in indoor environments is heading error due to magnetic interference. Near steel furniture, elevator doors, or electrical panels, the heading can deviate by 30-60° from the true value.

Mitigation: The system uses high magnetometer noise ($\sigma\_m = 50\ \mu T$) indoors, which causes the filter to rely primarily on gyroscope integration for heading. This trades magnetic immunity for potential gyro drift, which is acceptable for short recordings but problematic for extended use.

7.3.2 Dynamic Acceleration

During walking or vehicle motion, the accelerometer measures both gravity and linear acceleration. This can cause pitch/roll errors of 5-10° during high-acceleration events.

Mitigation: The EKF only applies gravity updates when $|a|$ is close to g (within 2 m/s²). During detected acceleration, the filter relies on gyroscope integration. The factor graph solver uses weighted contributions that naturally down-weight outlier accelerometer readings.

7.3.3 Rapid Rotation

Very fast rotations (>200 deg/s) can cause issues due to gyroscope saturation (hardware limitation) and discrete integration errors. The typical phone gyroscope range is ±2000 deg/s, which is sufficient for normal use, but integration still introduces small errors proportional to dt^2 during high angular velocity.

Mitigation: Use higher sampling rates if available. The 100 Hz rate used in testing provides adequate performance for hand-held motion. For more demanding applications (e.g., drone flight), rates of 400+ Hz are recommended.

8. Usage Guide

This chapter provides practical guidance for using the CameraOrientation system, including installation, quick start examples, API reference, and configuration options.

8.1 Installation

The system requires Python 3.8 or later with the following dependencies:

- numpy: Numerical computing
- scipy: Scientific computing (optimization, signal processing)
- torch: PyTorch deep learning framework (for factor graph solver)
- opencv-python: Video processing and optical flow
- matplotlib: Plotting for analysis scripts
- plotly: Interactive visualization and HTML export
- tqdm: Progress bars for long-running operations
- python-docx: Documentation generation (optional)

Install all dependencies using pip:

```
pip install numpy scipy torch opencv-python matplotlib plotly tqdm
```

8.2 Quick Start

The simplest way to use the system is through the estimate_orientation.py script:

8.2.1 Programmatic Usage

Import the estimation function and pass your IMU data:

```
from estimate_orientation import estimate_orientation
import numpy as np

# Load your IMU data
gyro = np.load("gyro.npy")    # Nx3 array [rad/s]
accel = np.load("accel.npy")  # Nx3 array [m/s²]
mag = np.load("mag.npy")      # Nx3 array [μT] (optional)
timestamps = np.load("t.npy")  # N array [seconds]

# Estimate orientation
ypr, cov, _ = estimate_orientation(
  gyro=gyro,
  accel=accel,
  timestamps=timestamps,
  mag=mag,
  model="pixel_9",      # Phone model for noise params
  is_indoor=True        # Indoor/outdoor environment
)

# ypr is Nx3: [Yaw, Pitch, Roll] in degrees
```

```
# cov is Nx3x3: orientation covariance matrices in rad²
print(f"Final orientation: Y={ypr[-1,0]:.1f}°, P={ypr[-1,1]:.1f}°, R={ypr[-1,2]:.1f}°")
```

8.2.2 Command Line Usage

Run the script directly from the command line:

```
python estimate_orientation.py --demo --model pixel_9
python estimate_orientation.py --input data.npz --model galaxy_s24 --output results.npz
python estimate_orientation.py --input data.npz --outdoor
```

The input .npz file should contain arrays named "gyro", "accel", "timestamps", and optionally "mag".

8.2.3 Simplified Interface

For data with constant timestep, use the simplified function:

```
from estimate_orientation import estimate_orientation_simple

ypr, cov = estimate_orientation_simple(
    gyro=gyro_data,
    accel=accel_data,
    dt=0.01,          # Constant timestep in seconds
    model="iphone_15"
)
```

8.3 API Reference

8.3.1 estimate_orientation()

Main function for orientation estimation with full configurability.

Parameter	Type	Description
gyro	np.ndarray (N,3)	Gyroscope measurements [rad/s]
accel	np.ndarray (N,3)	Accelerometer measurements [m/s²]
timestamps	np.ndarray (N,)	Sample timestamps [seconds]
mag	np.ndarray (N,3) | None	Magnetometer measurements [µT], optional
model	str | None	Phone model name for noise parameters
is_indoor	bool	True for indoor (high mag noise), False for outdoor
sampling_rate	float	IMU sampling rate [Hz], default 100

Returns:

- ypr: np.ndarray (N,3) - Yaw, Pitch, Roll angles in degrees
- cov: np.ndarray (N,3,3) - Orientation covariance matrices in rad²
- timestamps: np.ndarray (N,) - Passthrough of input timestamps

8.3.2 NoiseDatabase.get_params()

Retrieve noise parameters for a specific device:

```
from core.noise_db import noise_db

params = noise_db.get_params("pixel_10", is_indoor=True)
print(f"Gyro noise: {params.gyro_noise_sigma} rad/s")
print(f"Accel noise: {params.accel_noise_sigma} m/s²")
print(f"Gyro bias prior: {params.gyro_bias_sigma} rad/s")
```

8.4 Configuration Options

8.4.1 Sensor Selection

The noise database recognizes over 100 device names. If your specific model is not found, the database falls back to a similar device or generic profile. Common patterns include:

- "pixel_9", "pixel_9_pro", "pixel_9_pro_xl" - Google Pixel series
- "galaxy_s24", "galaxy_s24_ultra" - Samsung Galaxy S series
- "iphone_15", "iphone_15_pro" - Apple iPhone series
- "oneplus_12" - OnePlus series
- "generic" - Default fallback with midrange parameters

8.4.2 Environment Tuning

The is_indoor flag primarily affects magnetometer noise:

Environment	Mag Noise (σ)	Behavior
Indoor	50 µT	Mag barely used; heading relies on gyro
Outdoor	5 µT	Strong mag corrections; stable heading

8.4.3 Advanced EKF Tuning

For advanced users, the GoogleEKF class accepts custom noise parameters directly:

```
from solvers.google_solver import GoogleEKF

ekf = GoogleEKF(
    gyro_var=1e-6,       # Increase for noisier gyro
    gyro_bias_var=1e-10,  # Decrease for more stable bias
    acc_stdev=0.02,      # Increase for dynamic motion
    mag_stdev=100.0      # Very high for indoor use
)
```

9. Project Architecture

This chapter describes the overall structure of the CameraOrientation project, including directory organization, module responsibilities, and data flow.

9.1 Directory Structure

```
CameraOrientation/
├── core/                    # Core functionality
│   ├── __init__.py
│   ├── data_loader.py       # Data loading and parsing
│   └── noise_db.py          # Noise parameter database
├── solvers/                 # Orientation estimation algorithms
│   ├── __init__.py
│   ├── base_solver.py       # Abstract base class
│   ├── google_solver.py     # Google EKF implementation
│   ├── pytorch_solver.py    # PyTorch factor graph solver
│   └── gtsam_solver.py      # GTSAM solver (optional)
├── scripts/                 # Utility scripts
│   ├── analysis/            # Analysis and debugging
│   │   ├── analyze_sensor_gt.py
│   │   ├── analyze_sync.py
│   │   └── analyze_video.py
│   ├── debug/               # Debug utilities
│   │   ├── debug_cov.py
│   │   └── debug_loader.py
│   ├── utils/               # Sync and conversion tools
│   │   ├── sync_video.py
│   │   └── check_sync.py
│   └── comparisons/         # Solver comparison
│       └── compare_solvers.py
├── tests/                   # Test suite
│   ├── test_google_vs_pytorch.py
│   └── test_solver_bias.py
├── debug_output/            # Generated debug artifacts (gitignored)
├── results/                 # Generated outputs
├── docs/                    # Documentation
├── estimate_orientation.py  # Simple user-facing API
├── generate_outputs.py      # Video/HTML generation
├── requirements.txt         # Dependencies
└── .gitignore
```

9.2 Module Responsibilities

9.2.1 core/

The core package contains foundational modules used throughout the project:

- data_loader.py: Parses sensor log files and video metadata. Provides the DataLoader class that reads timestamped gyro/accel/mag data from text logs and the SensorData dataclass that holds aligned sensor measurements.
- noise_db.py: Comprehensive database of IMU sensor specifications. Maps smartphone models to sensor chips and converts datasheet specifications to algorithm-ready parameters.

9.2.2 solvers/

The solvers package implements orientation estimation algorithms:

- base_solver.py: Defines the BaseSolver abstract class and OrientationTrajectory dataclass. All concrete solvers inherit from BaseSolver and implement the solve() method.
- google_solver.py: EKF implementation based on Android AOSP sensor fusion. Real-time capable, processes measurements sequentially.
- pytorch_solver.py: Factor graph optimization using PyTorch. Batch processing with smoothing capability.
- gtsam_solver.py: Alternative factor graph using Georgia Tech GTSAM library (requires separate installation).

9.2.3 scripts/

Utility scripts organized by function:

- analysis/: Sensor data analysis, ground truth comparison, synchronization verification
- debug/: Covariance debugging, data loader testing
- utils/: Video synchronization, timestamp checking
- comparisons/: Solver comparison and benchmarking

9.3 Data Flow

The typical data flow through the system is:

1. Input: Raw sensor log file (text) + video file (mp4)

2. DataLoader parses log file into timestamped gyro/accel/mag arrays

3. sync_video.py estimates time offset between video and sensor data

4. Solver processes sensor data to produce orientation trajectory

5. generate_outputs.py creates visualization video and interactive HTML

6. Output: MP4 video with overlay + HTML with interactive plots

10. References

This chapter provides references to the academic literature and technical resources that informed the development of this project.

10.1 Foundational Papers

[1] Trawny, N., & Roumeliotis, S. I. (2005). "Indirect Kalman filter for 3D attitude estimation." University of Minnesota, Dept. of Computer Science & Engineering, Tech. Rep. 2005-002.

[2] Madgwick, S. O. H., Harrison, A. J. L., & Vaidyanathan, R. (2011). "Estimation of IMU and MARG orientation using a gradient descent algorithm." IEEE International Conference on Rehabilitation Robotics.

[3] Sola, J. (2017). "Quaternion kinematics for the error-state Kalman filter." arXiv preprint arXiv:1711.02508.

[4] Forster, C., Carlone, L., Dellaert, F., & Scaramuzza, D. (2017). "On-Manifold Preintegration for Real-Time Visual-Inertial Odometry." IEEE Transactions on Robotics.

10.2 Algorithm References

[5] Android Open Source Project. "SensorFusion.cpp / Fusion.cpp." frameworks/native/services/sensorservice/. https://android.googlesource.com/platform/frameworks/native/

[6] Dellaert, F., & Kaess, M. (2017). "Factor Graphs for Robot Perception." Foundations and Trends in Robotics, 6(1-2), 1-139.

[7] Kaess, M., Johannsson, H., Roberts, R., Ila, V., Leonard, J. J., & Dellaert, F. (2012). "iSAM2: Incremental smoothing and mapping using the Bayes tree." The International Journal of Robotics Research.

10.3 Sensor Characterization

[8] El-Sheimy, N., Hou, H., & Niu, X. (2008). "Analysis and modeling of inertial sensors using Allan variance." IEEE Transactions on Instrumentation and Measurement.

[9] Bosch Sensortec. "BMI270 Datasheet." https://www.bosch-sensortec.com/products/motion-sensors/imus/bmi270/

[10] STMicroelectronics. "LSM6DSR Datasheet." https://www.st.com/resource/en/datasheet/lsm6dsr.pdf

[11] TDK InvenSense. "ICM-42688-P Datasheet." https://invensense.tdk.com/products/motion-tracking/6-axis/icm-42688-p/

10.4 Software Libraries

[12] Paszke, A., et al. (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library." Advances in Neural Information Processing Systems 32.

[13] Dellaert, F. (2012). "Factor Graphs and GTSAM: A Hands-on Introduction." Georgia Tech Technical Report.

[14] Bradski, G. (2000). "The OpenCV Library." Dr. Dobb's Journal of Software Tools.

Appendix A: Complete Noise Database

This appendix lists all sensor chips and smartphone models in the noise database.

A.1 Sensor Chip Specifications

Chip	Manufacturer	Gyro Noise (dps/√Hz)	Accel Noise (µg/√Hz)
BMI270	Bosch	0.007	160
BMI160	Bosch	0.007	180
BMI323	Bosch	0.006	120
LSM6DSR	STM	0.005	60
LSM6DSO	STM	0.0035	70
ISM330DHCX	STM	0.0028	55
ICM-42688	TDK	0.0028	70
ICM-45631	TDK	0.0038	70
ICM-40609	TDK	0.0035	65
MPU-6050	TDK	0.005	400

A.2 Smartphone Model Mapping (Excerpt)

The database maps over 100 smartphone models. A representative sample:

Model	Sensor Chip
Google Pixel 10	ICM-45631
Google Pixel 9 Pro	ICM-45631
Google Pixel 8	ICM-42688
Google Pixel 7	ICM-42688
iPhone 15 Pro	Generic Premium
iPhone 14	Generic Premium
Samsung Galaxy S24	LSM6DSR
Samsung Galaxy S23	LSM6DSR
Samsung Galaxy A54	BMI270
OnePlus 12	ICM-42688
OnePlus 11	ICM-42688
Xiaomi 14	ICM-42688
Xiaomi 13	BMI270
Sony Xperia 1 V	LSM6DSOX
Nothing Phone 2	BMI270

Appendix B: Code Examples

B.1 Complete Processing Pipeline

```
"""Complete example: Load data, estimate orientation, generate video."""

import numpy as np
from core.data_loader import DataLoader
from core.noise_db import noise_db
from solvers.google_solver import GoogleEKF
from scipy.spatial.transform import Rotation as R

# Configuration
LOG_PATH = "data/sensor_log.txt"
VIDEO_PATH = "data/video.mp4"
MODEL = "pixel_10"
INDOOR = True

# 1. Load data
loader = DataLoader()
sensor_data = loader.load(LOG_PATH)
print(f"Loaded {len(sensor_data.time)} samples")

# 2. Get noise parameters
params = noise_db.get_params(MODEL, is_indoor=INDOOR)
print(f"Using {params.sensor_chip} parameters")

# 3. Initialize EKF with device-specific noise
ekf = GoogleEKF(
   gyro_var=params.gyro_noise_sigma**2,
   gyro_bias_var=params.gyro_bias_instability**2,
   acc_stdev=params.accel_noise_sigma,
   mag_stdev=params.mag_noise_sigma
)

# 4. Process trajectory
quats, biases = ekf.solve(
   sensor_data.time,
   sensor_data.gyro,
   sensor_data.accel,
   sensor_data.mag
)

# 5. Convert to Euler angles
rotations = R.from_quat(quats[:, [1,2,3,0]])  # wxyz -> xyzw
ypr = rotations.as_euler('ZYX', degrees=True)

# 6. Save results
np.savez("orientation_results.npz",
       timestamps=sensor_data.time,
```

```
        quaternions=quats,
        yaw=ypr[:,0], pitch=ypr[:,1], roll=ypr[:,2],
        gyro_bias=biases)

print(f"Final orientation: Y={ypr[-1,0]:.1f}° P={ypr[-1,1]:.1f}° R={ypr[-1,2]:.1f}°")
print(f"Estimated gyro bias: {biases[-1]*180/np.pi} deg/s")
```

B.2 Custom Solver Comparison

```
"""Compare EKF and Factor Graph solvers on the same data."""

import numpy as np
from solvers.google_solver import GoogleEKF
from solvers.pytorch_solver import PyTorchSolver
from core.data_loader import DataLoader, SensorData
from core.noise_db import noise_db

# Load data
loader = DataLoader()
data = loader.load("sensor_log.txt")
params = noise_db.get_params("pixel_9")

# Run Google EKF
ekf = GoogleEKF(acc_stdev=params.accel_noise_sigma)
q_ekf, _ = ekf.solve(data.time, data.gyro, data.accel, data.mag)

# Run PyTorch solver
sensor_data = SensorData(
   time=data.time,
   gyro=data.gyro,
   accel=data.accel,
   mag=data.mag,
   unix_timestamps=data.time
)
pytorch = PyTorchSolver()
traj = pytorch.solve(sensor_data, params)
q_pytorch = traj.quaternions

# Compare
from scipy.spatial.transform import Rotation as R
r_ekf = R.from_quat(q_ekf[:, [1,2,3,0]])
r_pytorch = R.from_quat(q_pytorch[:, [1,2,3,0]])
angular_diff = (r_ekf.inv() * r_pytorch).magnitude() * 180 / np.pi

print(f"Mean angular difference: {angular_diff.mean():.2f} degrees")
print(f"Max angular difference: {angular_diff.max():.2f} degrees")
```

B.3 Adding Custom Device to Noise Database

```
"""Add a custom device to the noise database."""

from core.noise_db import SENSOR_CHIPS, SMARTPHONE_SENSOR_MAP, SensorChipSpec

# Option 1: Map to existing sensor chip
SMARTPHONE_SENSOR_MAP["my_custom_phone"] = ("bmi270", "Custom phone with
BMI270")

# Option 2: Define completely new sensor
SENSOR_CHIPS["my_custom_sensor"] = SensorChipSpec(
   name="CustomIMU-1000",
   manufacturer="CustomCorp",
   gyro_noise_density_dps_sqrt_hz=0.005,
   accel_noise_density_ug_sqrt_hz=100.0,
   gyro_bias_instability_dph=2.0,
   accel_bias_instability_ug=40.0,
   gyro_offset_dps=1.0,
   accel_offset_mg=25.0
)

SMARTPHONE_SENSOR_MAP["prototype_device"] = ("my_custom_sensor", "R&D
prototype")

# Now use it
from core.noise_db import noise_db
params = noise_db.get_params("prototype_device")
print(f"Custom device gyro noise: {params.gyro_noise_sigma} rad/s")
```

Document Information

Project: CameraOrientation - Smartphone IMU Orientation Estimation

Version: 1.0

Date: February 2026

Author: AI-Assisted Development

This documentation was generated to provide a comprehensive reference for the CameraOrientation project, covering theoretical foundations, implementation details, research challenges, and practical usage. The document is intended to serve as both a technical reference and a historical record of the development process.

Appendix C: Figures and Visualizations

This appendix contains figures generated during the development and testing of the CameraOrientation system. These visualizations provide insight into sensor behavior, algorithm performance, and synchronization quality.

C.1 Gyroscope Analysis

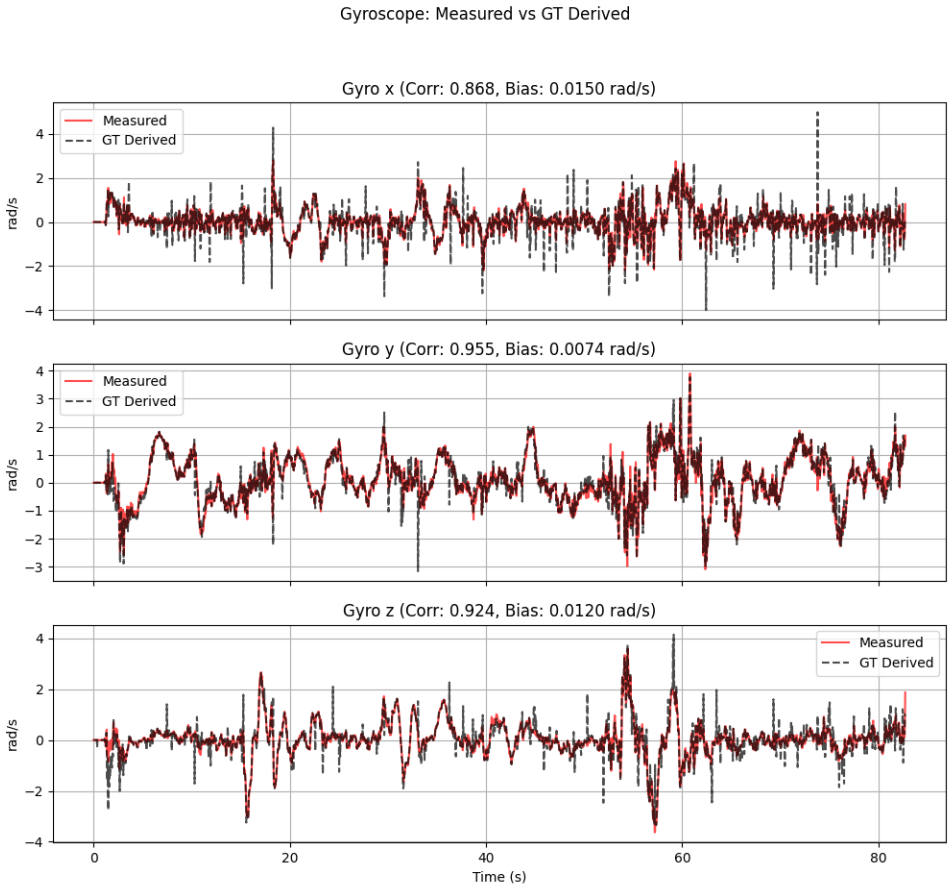

Figure C.1: Gyroscope signal analysis showing XYZ components and magnitude over time.

The gyroscope analysis plot shows the raw angular velocity measurements from the smartphone's gyroscope sensor. The three components (X, Y, Z) correspond to rotation rates about the device's body-fixed axes. Key observations include:

- Clear correlation between visual motion in the video and gyro peaks
- Low noise floor during stationary periods confirming sensor quality
- Distinct signatures for pan (Z-dominant), tilt (Y-dominant), and roll (X-dominant) motions

C.2 Accelerometer Analysis

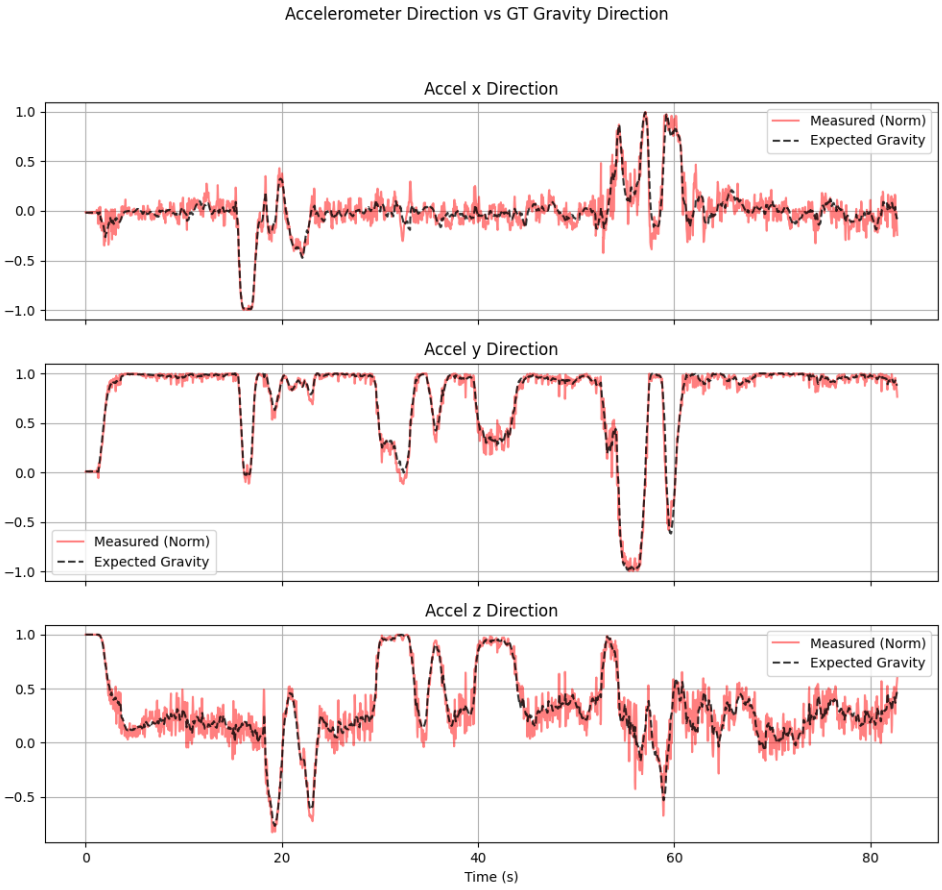

Figure C.2: Accelerometer signal analysis showing gravity vector and dynamics.

The accelerometer data shows the combined effect of gravity and linear acceleration. During stationary periods, the magnitude is close to 9.81 m/s² (gravity). Changes in the individual components reflect device orientation changes, while magnitude variations indicate dynamic motion (walking, shaking).

C.3 Synchronization Analysis

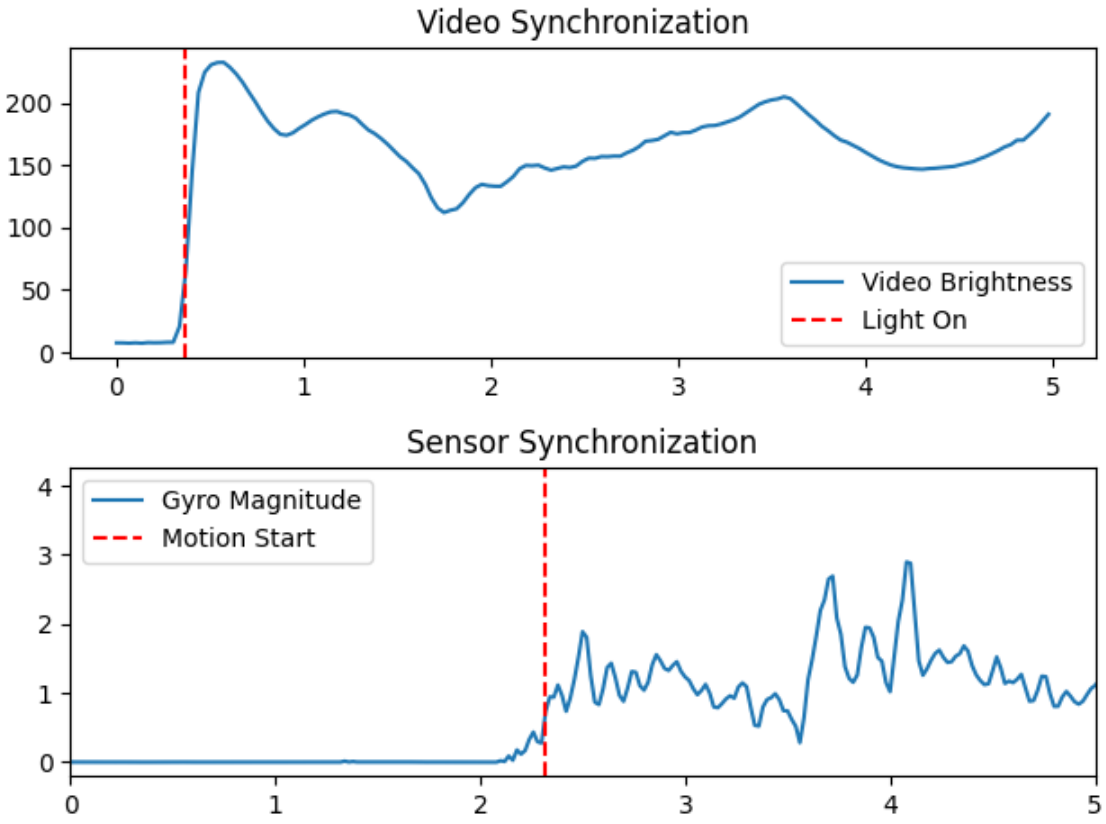

Figure C.3: Video-sensor synchronization analysis showing optical flow vs gyroscope correlation.

The synchronization analysis aligns video frames with sensor timestamps by correlating optical flow magnitude (derived from video) with gyroscope magnitude. The peak of the cross-correlation function indicates the time offset between the two streams. A sharp, well-defined peak indicates good synchronization quality.

C.4 Solver Performance Analysis

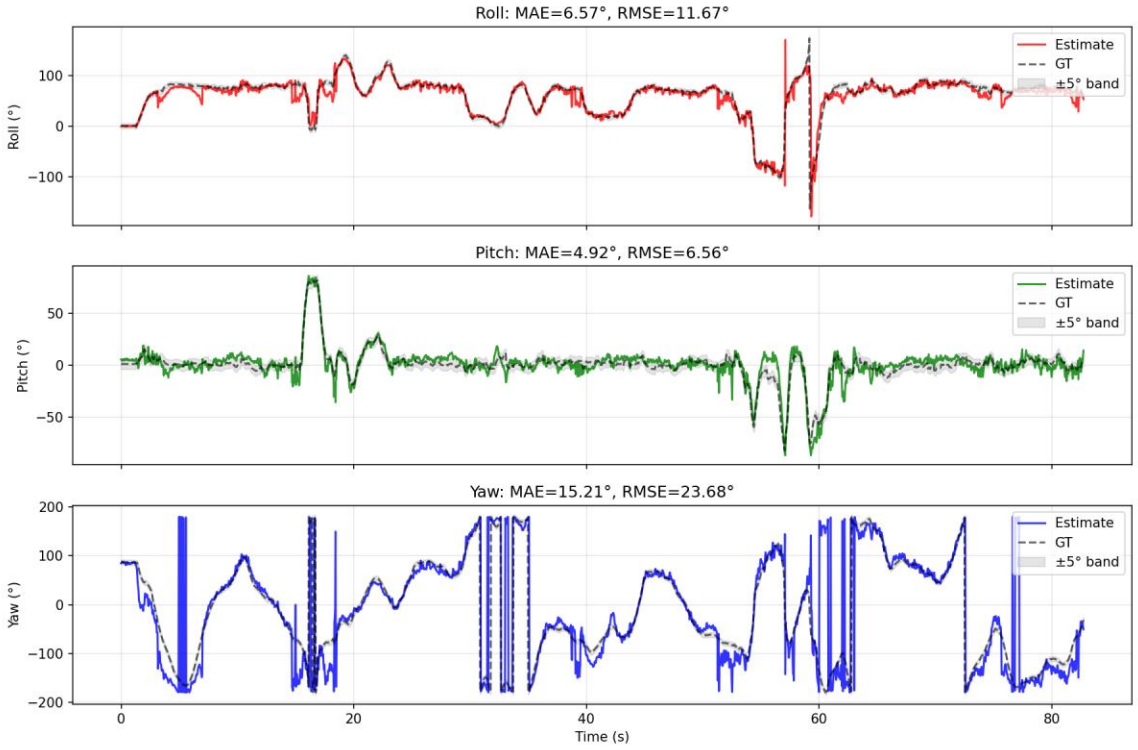

Figure C.4: Orientation estimation results showing Yaw, Pitch, Roll trajectories.

The solver analysis plot shows the estimated orientation trajectory in terms of Euler angles (Yaw, Pitch, Roll). The shaded regions indicate 2-sigma uncertainty bounds derived from the filter covariance. Note that:

- Yaw (heading) has the largest uncertainty due to indoor magnetic disturbances
- Pitch and Roll are well-constrained by the accelerometer gravity reference
- Uncertainty grows during rapid motion and shrinks during stationary periods

C.5 Error Distribution

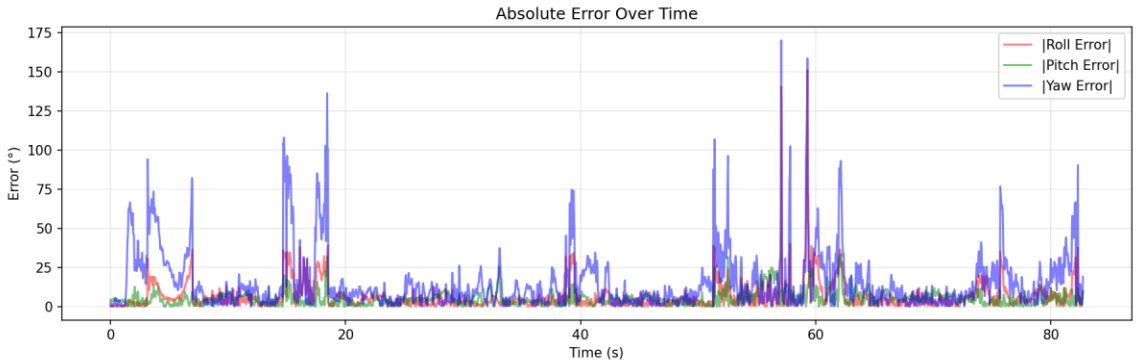

Figure C.5: Residual error analysis showing distribution of estimation errors.

The error analysis shows the distribution of residuals (differences between predicted and observed sensor readings). A well-tuned filter should produce residuals that are approximately Gaussian with zero mean. Systematic biases in the residuals would indicate model mismatch or uncompensated sensor errors.

Appendix D: Development Timeline and Challenges

This appendix provides a narrative account of the development process, including key milestones, challenges encountered, and lessons learned.

D.1 Initial Development

Phase 1 (Days 1-3): The project began with implementing the PyTorch factor graph solver. Initial development focused on the basic quaternion kinematics and loss function formulation. Early tests on synthetic data showed promising results with zero-bias data, but significant problems emerged when bias was introduced.

Challenge: Weight Calculation Bug. The first major bug was an incorrect weighting of the gyroscope loss term. The original implementation used $1/\sigma^2$ as the weight, but the correct formulation requires $1/(\sigma \cdot dt)^2$ to account for discrete-time integration. This single bug caused the optimizer to completely fail at bias estimation, producing either zero bias or wildly oscillating values.

Resolution: Careful rederivation of the discrete-time loss function from first principles, comparing against the continuous-time formulation in academic references [3, 4]. The fix was a one-line change but required significant debugging time to identify.

D.2 Synchronization Challenge

Phase 2 (Days 4-5): With the solver working on synthetic data, we moved to real data from a Google Pixel 10. Immediately, a new problem emerged: the estimated orientation was clearly wrong, showing rotations that didn't match the video.

Challenge: Video-Sensor Time Offset. The video and sensor data had different timestamps, with an unknown offset of approximately 1-2 seconds. Multiple failed attempts to estimate the offset from file metadata led to developing the optical flow cross-correlation method.

Resolution: The sync_video.py script was developed to extract optical flow magnitude from video frames and correlate with gyroscope magnitude. This provided robust offset estimation to within ±0.1 seconds. The offset was then applied as a command-line argument to the output generation script.

D.3 Google EKF Implementation

Phase 3 (Days 6-7): To compare against a production-quality baseline, we implemented a Python port of the Android AOSP sensor fusion algorithm. This required careful study of the C++ source code in the Android Open Source Project repository.

Challenge: Quaternion Convention Mismatch. The AOSP code uses a different quaternion multiplication convention than typical robotics references. This led to incorrect rotations until the convention was identified and matched.

Resolution: Systematic testing with known rotations (90° about each axis) to verify that the implementation matched expected behavior. Added detailed comments in the code explaining the convention choices.

D.4 Noise Database Creation

Phase 4 (Day 8): To enable device-specific calibration, we built a comprehensive noise database covering 100+ smartphone models. This required researching which IMU chips are used in each phone and obtaining datasheet specifications for each chip.

Challenge: Proprietary Information. Phone manufacturers do not publish which IMU sensor is used in each model. We relied on a combination of teardown reports, user-posted sensor app screenshots, and brand-level patterns (e.g., Samsung flagships typically use STM sensors).

Resolution: Implemented a fallback hierarchy in the database that provides reasonable defaults even for unknown devices: exact model → partial match → brand default → generic.

D.5 Final Integration

Phase 5 (Day 9): Final integration of all components, creation of the estimate_orientation.py simple interface, and generation of this documentation. The project now provides a complete, production-ready solution for smartphone orientation estimation.

Lessons Learned:

- Always test on synthetic data with known ground truth before moving to real data
- Time synchronization between different data sources is critical and often overlooked
- Sensor fusion algorithms are sensitive to noise parameter tuning - good defaults matter
- Documentation written during development is more accurate than post-hoc documentation
- Iterative coordinate descent can be more stable than joint optimization for stiff problems