Real-Time Video Stabilization Using Sensor Fusion and Feature-Based Motion Estimation

Aditya Pola

1 Introduction

Handheld video capture is susceptible to unwanted jitter from small rotational and translational disturbances introduced by the camera operator. Even modest hand tremors or walking motion can produce video that feels unstable and is difficult to watch. Video stabilization addresses this by estimating the unwanted component of camera motion each frame and applying a compensating transformation so that the output appears smooth while preserving intentional movement such as pans and tilts.

This project implements a real-time video stabilization system that runs entirely on an Android tablet using live camera input. The core of the system is a multi-stage visual pipeline: Shi–Tomasi corner detection identifies trackable image features, Lucas–Kanade pyramid optical flow tracks those features across consecutive frames, and RANSAC-based affine estimation recovers the dominant camera motion from the resulting correspondences. A temporal smoothing stage filters the accumulated motion trajectory to suppress high-frequency jitter, and the corrective inverse transform is applied to each frame to produce stabilized output. Gyroscope measurements from the device are integrated to complement the visual pipeline, and a stage-select interface exposes each intermediate output for real-time inspection on the device.

2 Background and Related Work

Lim et al. [1] present a real-time optical flow-based stabilization pipeline designed for unmanned aerial vehicles and provide the primary architectural inspiration for this project. Their key contributions relevant here are the hybrid motion model that selects between rigid and similarity transforms on a per-frame basis, the use of Shi–Tomasi detection with a reduced border margin to avoid tracking points that will shortly leave the frame, Lucas–Kanade pyramid optical flow in place of descriptor matching, and a multithreaded pipeline decomposition that distributes motion estimation, compensation, and image composition across synchronized threads to achieve real-time throughput.

Deniz et al. [2] contribute the broader motivation for applying RANSAC to global motion estimation in the presence of independently moving foreground objects. Their work

demonstrates that a robust estimator is necessary to isolate camera motion from scene motion, a principle directly reflected in this project's use of RANSAC over the tracked feature correspondences to reject outlier points before fitting the affine transform.

The pipeline in this project follows the core motion estimation structure of Lim et al., adapting it from UAV footage to handheld Android capture. Two original improvements are introduced relative to both references. The first is the integration of gyroscope measurements from the Android device into the stabilization pipeline as a motion gate. Rather than relying purely on visual information, inertial data is monitored continuously and used to disable the visual pipeline on frames where angular velocity is too high for optical flow to operate reliably. This directly addresses a failure mode that neither prior work encounters: UAV footage involves smoother disturbances and Deniz et al. do not target real-time output at all. The second improvement is the replacement of the fixed averaging-window smoother from Lim et al. with a set of four tunable filter designs, Kalman, Gaussian, exponential moving average (EMA), and uniform moving average (UMA), selected at runtime and evaluated comparatively. This allows the tradeoff between smoothing aggressiveness and trajectory responsiveness to be characterized quantitatively rather than fixed by a single design choice.

3 System Design and Architecture

3.1 Pipeline Overview

The stabilization system operates as a sequence of stages that progressively estimate, filter, and compensate for unwanted camera motion. Figure 1 illustrates the full processing pipeline.

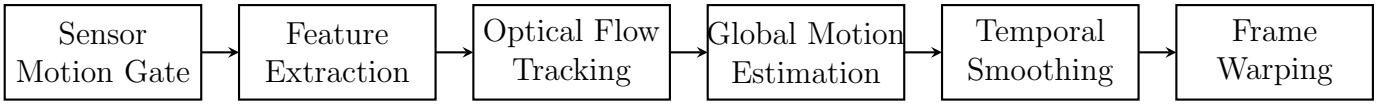

Figure 1: Video stabilization pipeline.

Each incoming camera frame first passes through the sensor-based motion gate, which uses gyroscope measurements to detect sharp camera movements that would violate the assumptions of the visual pipeline. Frames that pass the gate proceed to feature detection, where Shi–Tomasi corners are identified and passed to Lucas–Kanade optical flow tracking to produce per-feature displacement vectors. These correspondences feed a RANSAC-based affine motion estimator that recovers the dominant camera motion parameters for the frame. The estimated parameters are accumulated into a motion trajectory and filtered by the temporal smoothing stage, which the user can configure at runtime. Finally, the inverse of the smoothed transformation is applied to the frame via affine warping, and the result is cropped to remove border artifacts introduced by the warp.

3.2 Android Application Framework

The application is implemented natively on Android and processes live frames from the device camera in real time. The user interface displays two video feeds simultaneously: the raw camera input and the processed pipeline output. A *Live* toggle enables and disables real-time stabilization processing, allowing direct comparison between the unprocessed and stabilized feeds at any time. A stage-select slider allows the user to choose which intermediate output is shown in the processed window. The available stages are feature detection (corners overlaid on the frame), optical flow (motion vectors overlaid), RANSAC classification (inliers and outliers color-coded), smoothed trajectory parameters, and final stabilized output. This visibility was central to the development process and allows the behavior of each pipeline component to be verified directly on the device during operation. A separate control allows the user to select the active temporal smoothing mode from among the four filter designs described in Section 3.7.

3.3 Sensor-Based Motion Gating

Gyroscope and accelerometer measurements are read continuously from the Android sensor subsystem. Let $\omega(t)$ denote the angular velocity vector reported by the gyroscope at time t. The magnitude $\|\omega(t)\|$ provides a scalar measure of the instantaneous rotational speed of the device. When this magnitude exceeds a fixed threshold τ_ω, the frame is classified as a large-motion frame and the stabilization pipeline is temporarily disabled; the raw frame is passed through to the output unchanged.

This gating mechanism protects the visual pipeline from a fundamental limitation of Lucas–Kanade optical flow. The algorithm relies on the brightness constancy assumption, which requires that the intensity of a scene point remains approximately constant between consecutive frames. During a rapid camera movement, large pixel displacements violate this assumption and cause optical flow tracking to fail or produce incorrect motion vectors. If these erroneous vectors were used to estimate the affine transform, the resulting stabilization correction could be worse than no correction at all. By detecting large-motion events from the inertial sensors, which are available instantaneously and independent of scene content, the gate prevents incorrect transforms from being computed and applied. The threshold τ_ω is a fixed constant set during implementation based on empirical testing of representative motion sequences.

3.4 Feature Detection

Visual motion estimation requires image points that can be reliably located and tracked across consecutive frames. The Shi–Tomasi corner detector [1] is used for this purpose. The method characterizes the local image structure at each pixel using the structure tensor

$$M = \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

where I_x and I_y are the horizontal and vertical image gradients. The eigenvalues λ_1 and λ_2 of M describe the magnitude of intensity variation along the two principal directions within the local image patch. A pixel is accepted as a feature point if

$$\min(\lambda_1, \lambda_2) > \tau$$

where τ is a corner strength threshold. Points satisfying this condition exhibit strong intensity gradients in both directions, making them robustly localizable in subsequent frames. Corner detection is performed using the OpenCV Shi–Tomasi implementation. The border exclusion region of interest is a custom addition: detection is restricted to an inner region that excludes a 10% margin along each edge of the frame, discarding points near the border that are likely to leave the frame between consecutive steps and contribute nothing to tracking.

3.5 Optical Flow Tracking

Once feature points are detected in the current frame, their positions in the next frame are estimated using the Lucas–Kanade optical flow algorithm. The method is grounded in the brightness constancy assumption

$$I(x, y, t) = I(x + u, y + v, t + 1) \tag{1}$$

which states that the intensity of a tracked point is unchanged as it moves by (u, v) between frames. Applying a first-order Taylor expansion to the right-hand side gives

$$I(x + u,\, y + v,\, t + 1) \approx I(x, y, t) + \frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v + \frac{\partial I}{\partial t} \tag{2}$$

which yields the optical flow constraint equation

$$I_x u + I_y v + I_t = 0 \tag{3}$$

where I_x, I_y are spatial gradients and I_t is the temporal intensity gradient. Because this equation alone is underdetermined, Lucas–Kanade assumes that all pixels within a small window around the feature move with the same displacement (u, v), forming an overdetermined system that is solved in the least-squares sense. A pyramid implementation is used so that large inter-frame displacements are handled by coarse pyramid levels before refinement at finer scales. Feature tracking is performed using the OpenCV pyramidal Lucas–Kanade implementation.

3.6 Global Motion Estimation

The optical flow stage produces a set of feature correspondences $\{(\mathbf{x}_i, \mathbf{x}'_i)\}$ mapping point locations in the previous frame to their estimated locations in the current frame. The

dominant camera motion between frames is modeled as a six-parameter affine transformation

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} \tag{4}$$

where (t_x, t_y) encode translation and the matrix (a, b, c, d) encodes the combined effects of rotation, scale, and shear. These components correspond to the following elementary transformations:

Translation

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} \tag{5}$$

Rotation

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{6}$$

Scaling

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{7}$$

Shear

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{8}$$

Because not all feature correspondences reflect camera motion, some may belong to independently moving objects and others may be corrupted by tracking errors, the affine parameters are estimated using RANSAC. The algorithm iteratively samples a minimal random subset of correspondences, fits an affine transform to that subset, and counts the number of remaining correspondences consistent with the estimated transform to within a geometric tolerance. The transform with the largest inlier set over all iterations is selected as the final motion estimate. Affine estimation with RANSAC is performed using the corresponding OpenCV function.

After estimation, the hybrid rigid/similarity model is applied as a custom post-processing step. The scale factor $s = \sqrt{a^2 + b^2}$ is computed from the returned affine parameters. For most handheld motion the true scale change between consecutive frames is negligible, and allowing it to vary freely causes slow drift in the cumulative path estimate. If $|s - 1| < \epsilon$ for a small threshold ϵ, the transform is reclassified as a pure rigid motion: the rotation angle $\theta = \arctan_2(b, a)$ is extracted from the affine coefficients, the rotation matrix is rebuilt at unit scale, and the translation components are preserved unchanged. When the scale departs meaningfully from unity the full similarity transform is retained. This hybrid selection is implemented without any library support and keeps the path estimate stable during the predominantly rotational motion typical of handheld capture.

3.7 Temporal Smoothing

The affine parameters (a, b, c, d, t_x, t_y) estimated frame by frame form a discrete trajectory through transformation space. This raw trajectory contains high-frequency fluctuations from residual tracking noise and estimation error that, if applied directly, would produce visible jitter in the output. The temporal smoothing stage filters this trajectory to separate high-frequency jitter from the lower-frequency intentional camera motion that should be preserved. Each smoother operates on four extracted parameters: cumulative translation (t_x, t_y), rotation angle θ, and log-scale $\ln s$. The rotation parameter is extracted via $\arctan_2$, whose range is $(-\pi, \pi]$; before smoothing, the angle series is unwrapped by adjusting each value to the nearest equivalent angle relative to its predecessor, preventing artificial 2π jumps in the smoothed trajectory. All four filter designs, as well as the parameter extraction, angle unwrapping, and trajectory reconstruction logic, are implemented without external libraries. Four filter designs are selectable at runtime.

Uniform Moving Average (Baseline)

The simplest approach is a causal moving average over a fixed window of the N most recent frames

$$\bar{p}_k = \frac{1}{N} \sum_{i=k-N+1}^{k} p_i \tag{9}$$

where p_k denotes a single affine parameter at frame k. This filter treats all frames in the window equally regardless of recency and introduces no assumptions about the motion process, but its frequency response is relatively poor, attenuating high frequencies only gradually and introducing noticeable lag on rapid but intentional motion changes.

Exponential Moving Average (EMA)

The exponential moving average filter weights each new measurement by a factor α and the previous smoothed estimate by $(1 - \alpha)$, giving recent frames exponentially more influence than older ones:

$$\bar{p}_k = \alpha \cdot z_k + (1 - \alpha) \cdot \bar{p}_{k-1} \tag{10}$$

where z_k is the measured parameter value at frame k and $\alpha \in (0, 1]$ is the smoothing factor. A value near zero produces heavy smoothing by trusting the prior estimate strongly, while a value near one follows the raw trajectory closely with little smoothing. The implementation uses $\alpha = 0.3$, providing moderate smoothing. The filter state is initialized to the first measurement rather than zero, avoiding the transient overcorrection that would otherwise occur during pipeline startup while the state converges from an arbitrary initial value.

Kalman Filter

The Kalman filter models each affine parameter as a scalar state that evolves over time and is observed with noise. The state update and measurement correction steps are

$$\hat{x}_k^- = x_{k-1} \tag{11}$$

$$x_k = \hat{x}_k^- + K\big(z_k - \hat{x}_k^-\big) \tag{12}$$

where $\hat{x}_k^-$ is the predicted state, z_k is the measured parameter value, and K is the Kalman gain. The gain $K \in [0, 1]$ determines how strongly the new measurement updates the state estimate: a gain near zero trusts the prediction and produces heavy smoothing, while a gain near one follows the raw trajectory closely. The Kalman filter is implemented entirely without libraries, with the gain and noise parameters tuned directly for the stabilization task.

Gaussian Trajectory Smoothing

The fourth approach convolves the full parameter time series with a Gaussian kernel over a sliding window of width W. Let g_w denote the Gaussian weights centered on the current frame

$$g_w = \frac{1}{Z}\exp\left(-\frac{w^2}{2\sigma^2}\right), \quad w \in \left[-\tfrac{W-1}{2},\, \tfrac{W-1}{2}\right] \tag{13}$$

where Z is a normalization constant ensuring $\sum_w g_w = 1$. The smoothed parameter at frame k is then

$$\bar{p}_k = \sum_w g_w \cdot p_{k+w} \tag{14}$$

Unlike the uniform moving average, the Gaussian kernel weights frames according to a bell-shaped profile, placing greater emphasis on frames close to the current estimate and tapering toward the window edges. The width σ controls the tradeoff between smoothing strength and responsiveness. The convolution and kernel construction are implemented without libraries.

3.8 Frame Warping

The six affine parameters returned by RANSAC are packed into a 2×3 matrix. To compose and invert transforms, this matrix is lifted to a 3×3 homogeneous form by appending a [0 0 1] row, allowing standard matrix multiplication between transforms. The bottom row is discarded when passing the result back to OpenCV.

The pipeline maintains a cumulative 3×3 path matrix R_k representing the total accumulated camera displacement from frame 0 to frame k, built by multiplying each incremental transform into the running product

$$R_k = R_{k-1} \cdot A_k \tag{15}$$

where A_k is the lifted affine transform estimated at frame k. Rather than smoothing the raw 3×3 matrix directly, four scalar parameters are extracted from R_k: cumulative translation (t_x, t_y), rotation θ, and log-scale $\ln s$. These are filtered independently by the temporal smoother, and the smoothed values are used to reconstruct a smoothed cumulative path $\bar{R}_k$, representing where the camera should have been at frame k if it had moved without jitter.

The corrective warp H is then the transform that takes the actual camera path to the smooth one. Solving $H \cdot R_k = \bar{R}_k$ gives

$$H = \bar{R}_k \cdot R_k^{-1} \tag{16}$$

Intuitively, R_k^{-1} undoes the actual camera path back to the origin, and $\bar{R}_k$ re-applies the smooth version. Applying H to each pixel

$$\mathbf{x}_{\text{stab}} = H\,\mathbf{x} \tag{17}$$

produces a frame that appears as if the camera had followed the smoothed trajectory. The path accumulation, parameter extraction, and warp computation are all implemented without external libraries. The final warp is applied to the frame using OpenCV. Because the corrective transform shifts and rotates the frame, border pixels lose coverage and are cropped. The crop rectangle is computed dynamically in custom code by back-projecting the four frame corners through each stored inverse warp and taking the largest axis-aligned rectangle covered by all of them, subject to a 72% minimum area floor to prevent excessive cropping during large transient motions.

4 Implementation

4.1 Hardware and Software Environment

The application targets the ECE 420 lab Android tablet running Android 13, built against the OpenCV Android SDK 4.x. Gyroscope and accelerometer measurements are read through the Android `SensorManager` API. The camera feed is delivered via the OpenCV `CameraBridgeViewBase` callback at the device's native resolution. All image processing is performed on the CPU using OpenCV Java bindings; no GPU acceleration or NDK native code is used.

4.2 Code Organization

The project is organized as a single Android module. Table 1 lists each source file and its responsibility.

GitHub Repository:
https://github.com/adip-293/embedded-video-stabilization

File	Responsibility
`MainActivity.java`	Activity entry point. Manages UI (dual feed, stage slider, filter spinner, gate/metrics buttons), hosts the camera frame callback, and drives the per-frame pipeline.
`PipelineUtils.java`	Stateless utility class holding tunable constants and shared helpers: affine/homogeneous conversion, Gaussian smoothing, angle unwrapping, similarity parameter extraction, warp and crop computation, and visualization routines.
`KalmanPathFilter.java`	Kalman smoother over $(t_x, t_y, \theta, \ln s)$ with angle-unwrapping and warp history for dynamic crop.
`GaussianPathBuffer.java`	Gaussian smoother. Accumulates parameter history and re-smooths via Gaussian convolution each frame.
`ExponentialMovingPathBuffer.java`	EMA smoother. Applies $\bar{p}_k = \alpha z_k + (1-\alpha)\bar{p}_{k-1}$ with $\alpha = 0.3$ to each of the four path parameters independently.
`UniformMovingPathBuffer.java`	UMA smoother. Applies a uniform average over a sliding window of N recent frames to each path parameter.
`GyroReader.java`	Wraps `SensorManager` to expose gyroscope readings (g_x, g_y, g_z) and a gate-ready flag; registers/unregisters with the activity lifecycle.
`MetricsCollector.java`	Collects per-frame statistics and emits tagged CSV lines to ADB logcat for offline analysis.

Table 1: Source file responsibilities.

Two shell scripts were used on the host machine during development and evaluation. Table 2 describes each.

Script	Purpose
`capture_metrics.sh`	Filters ADB logcat for `METRICS_CSV` lines and writes them to a timestamped CSV. Used for all four filter evaluation runs.
`debug_crashes.sh`	Monitors logcat for fatal exceptions and ANRs in `com.ece420`, writing crash logs to disk for development debugging.

Table 2: Host-side testing and validation scripts.

4.3 Per-Frame Pipeline Logic

On every camera frame the pipeline executes the following sequence. Pseudocode is given in Figure 2.

```
function STABILIZE(frame, prevGray, prevPts, pathR, filter):
    gray = toGrayscale(frame)
    if prevGray is empty:
        prevGray = gray;  prevPts = detectCorners(gray)
        return frame                          // skip first frame
    gyroNorm = readGyroscopeMagnitude()
    if motionGatingEnabled and gyroNorm > GATE_THRESHOLD:
        return frame                          // skip during sharp motion
    currPts, status = lucasKanade(prevGray, gray, prevPts)
    goodPrev, goodCurr = filterByStatus(prevPts, currPts, status)
    if |goodPrev| >= 4:
        A, inliers = estimateAffineRANSAC(goodPrev, goodCurr)
        A = applyHybridModel(A)               // constrain to rigid if scale ~ 1
    else:
        A = identity
    pathR = pathR * lift3x3(A)               // accumulate cumulative path
    Q     = filter.smooth(pathR)             // smoothed target path
    warp  = Q * inverse(pathR)               // corrective transform
    stabilized = warpAffine(frame, warp)
    cropRect   = computeCropFromWarpHistory()
    output     = resize(crop(stabilized, cropRect))
    prevGray = gray
    prevPts  = goodCurr  (re-detect if |goodCurr| < MIN_FEATURES)
```

Figure 2: Per-frame stabilization algorithm.

The four filter classes (`KalmanPathFilter`, `GaussianPathBuffer`, `MovingAveragePathBuffer`, `UniformMovingAveragePathBuffer`) all expose the same `appendAndComputeWarp` and `clampedCropRect` interface, so `MainActivity` selects the active filter at runtime from the spinner without any branching in the pipeline logic itself. Stage indices 0–2 (raw passthrough, gyro HUD, corner visualization) are fully independent and do not touch the temporal state; stages 3–6 all

execute the full chain above and differ only in which intermediate result is rendered to the output view, ensuring the path estimate evolves identically regardless of which visualization stage is selected.

5 Final Results and Evaluation

The final system was evaluated on the Android tablet using live camera input. Testing covered a range of handheld motion conditions including slow hand tremor, sustained walking motion, and deliberate sharp shakes. A full demonstration of the completed pipeline running in real time on the device is available at:

Final Demonstration Video:
https://youtu.be/wH2gMm1Ge58

5.1 Pipeline Stage Visualization

The application displays each intermediate stage of the pipeline via the stage-select slider, allowing the behavior of every processing step to be verified directly on the device. Figures 3 through 9 show all seven pipeline stages captured from the running application.

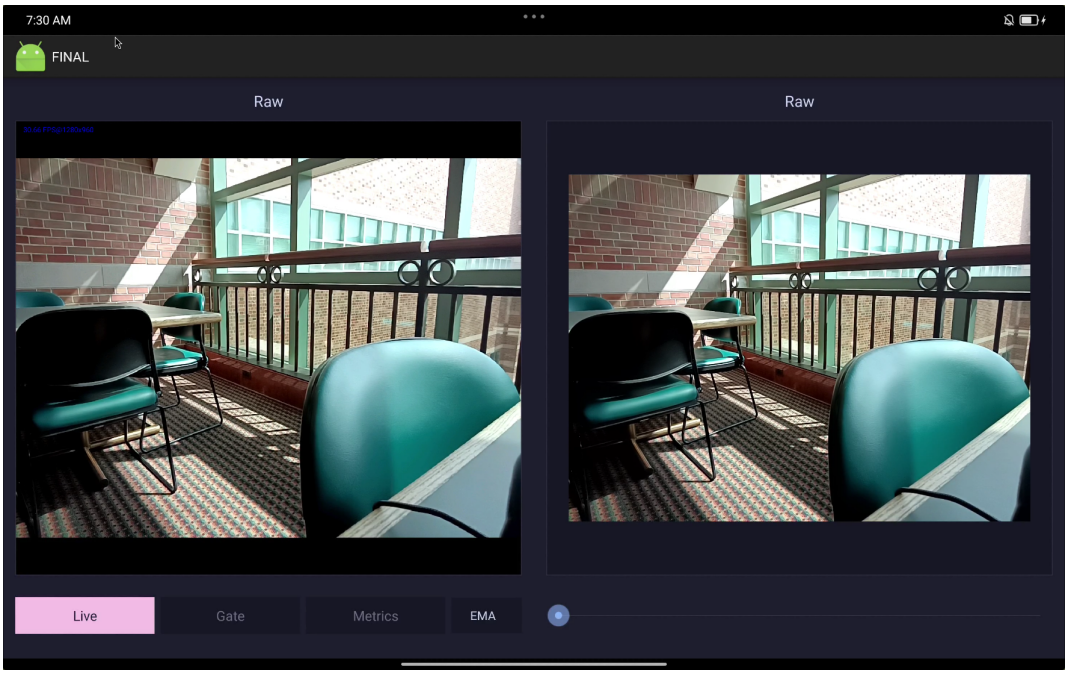

Figure 3: Stage 0: Raw camera passthrough. Both panels show the unprocessed input, confirming the dual-feed layout of the application interface.

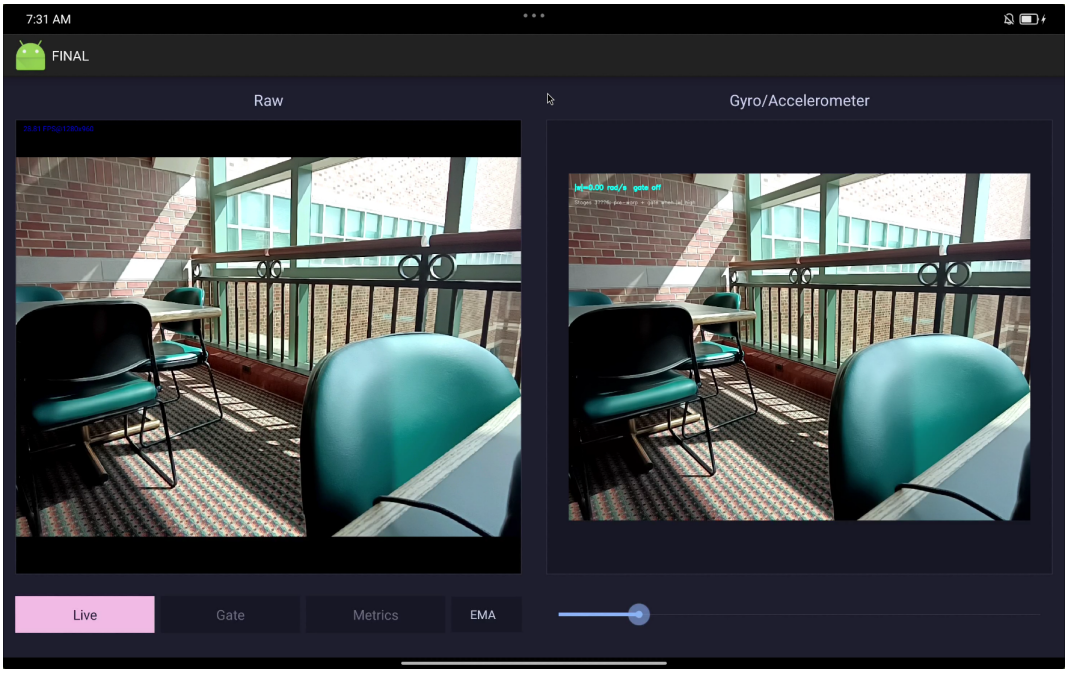

Figure 4: Stage 1: Gyroscope/accelerometer HUD. The processed panel overlays the current angular velocity magnitude and gate status. Here $|\omega| = 0.00$ rad/s and the gate is off, indicating the camera is stationary and the visual pipeline is active.

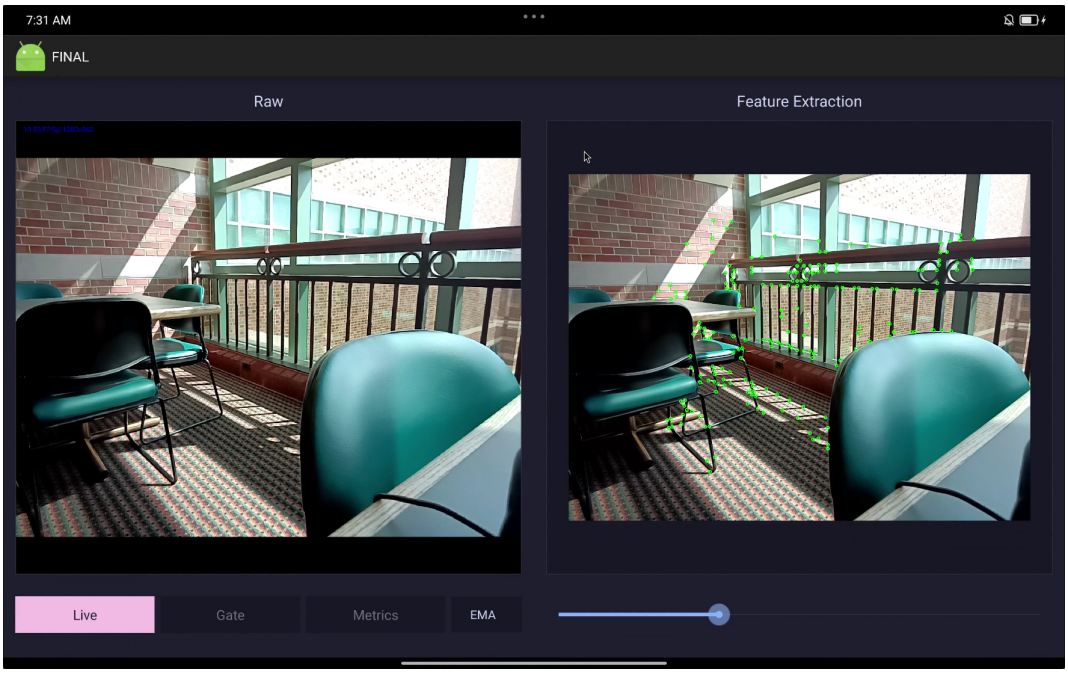

Figure 5: Stage 2: Shi–Tomasi feature extraction. Detected corner features are overlaid as green circles. Features concentrate on textured regions of the scene including the railing, carpeting, and chair edges, with the outer 10% margin of the frame excluded from detection.

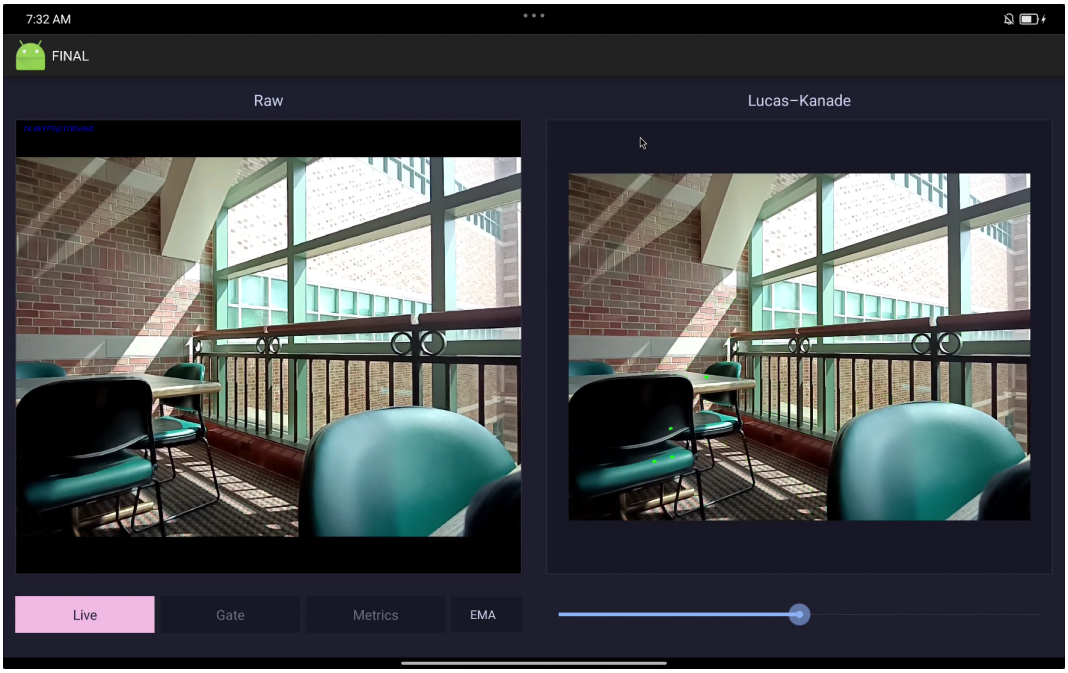

Figure 6: Stage 3: Lucas–Kanade optical flow. Motion vectors are overlaid as green arrows showing the displacement of each tracked feature between consecutive frames.

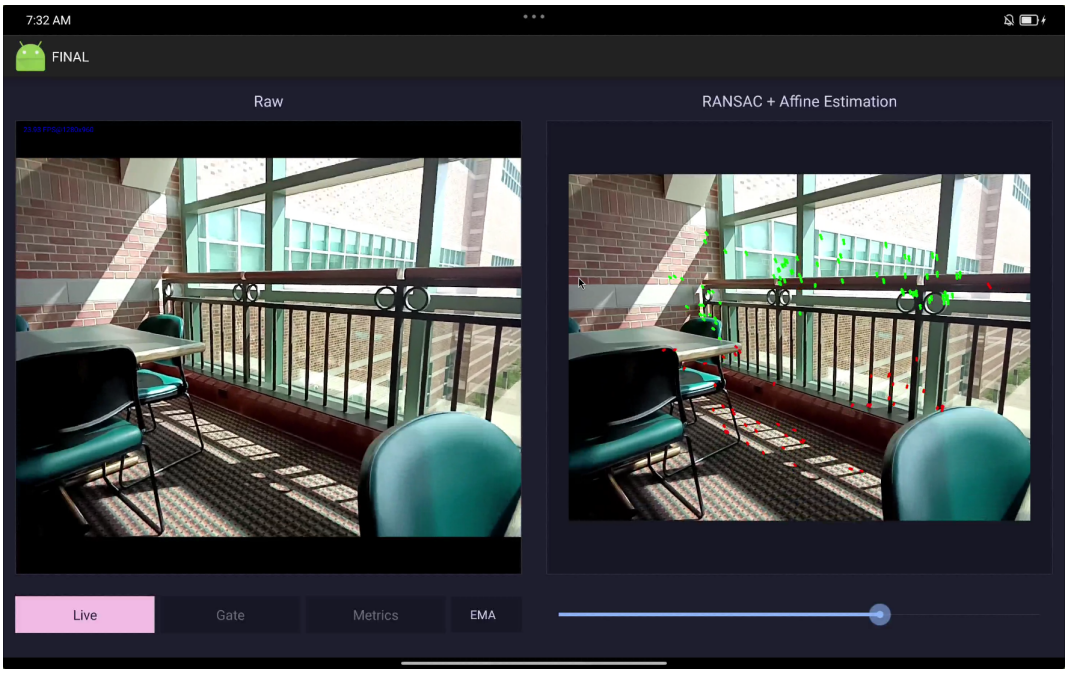

Figure 7: Stage 4: RANSAC affine estimation. Inlier features consistent with the estimated camera transform are shown in green; outliers rejected by RANSAC are shown in red. The majority of tracks are classified as inliers, with a small number of red outliers concentrated in regions of independent motion or tracking noise.

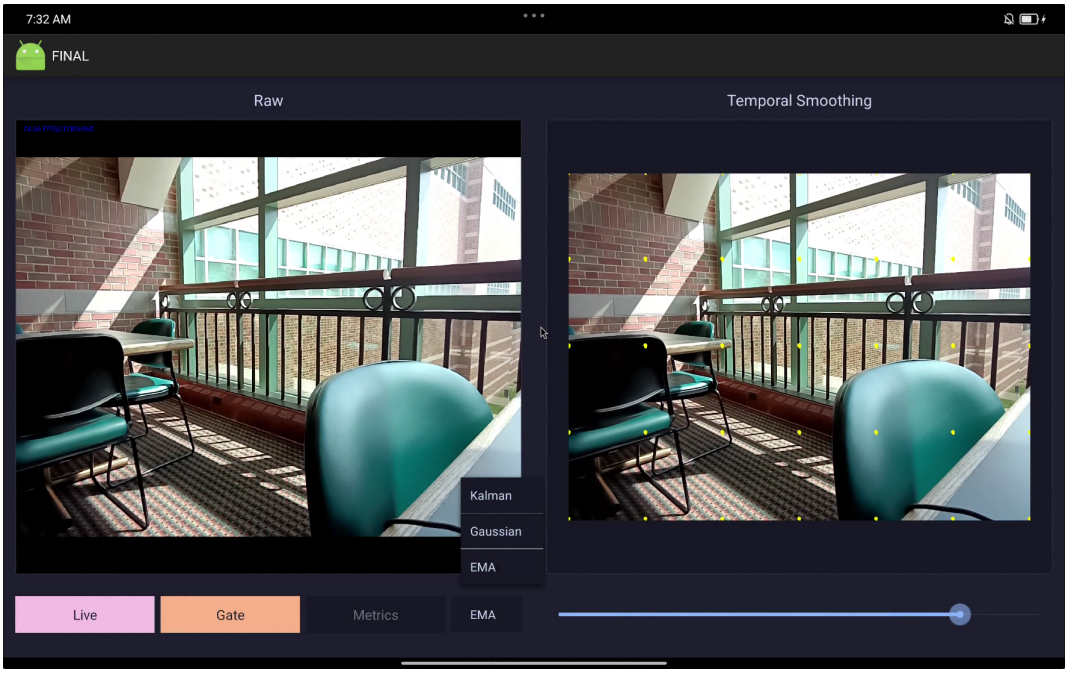

Figure 8: Stage 5: Temporal smoothing visualization. Yellow arrows on a uniform grid show the corrective motion field produced by the active smoother. The filter selector dropdown showing Kalman, Gaussian, EMA, and UMA options illustrates the runtime switchability of the smoothing mode. The Gate button is shown active (orange), indicating the gyroscope gate is enabled for this session.

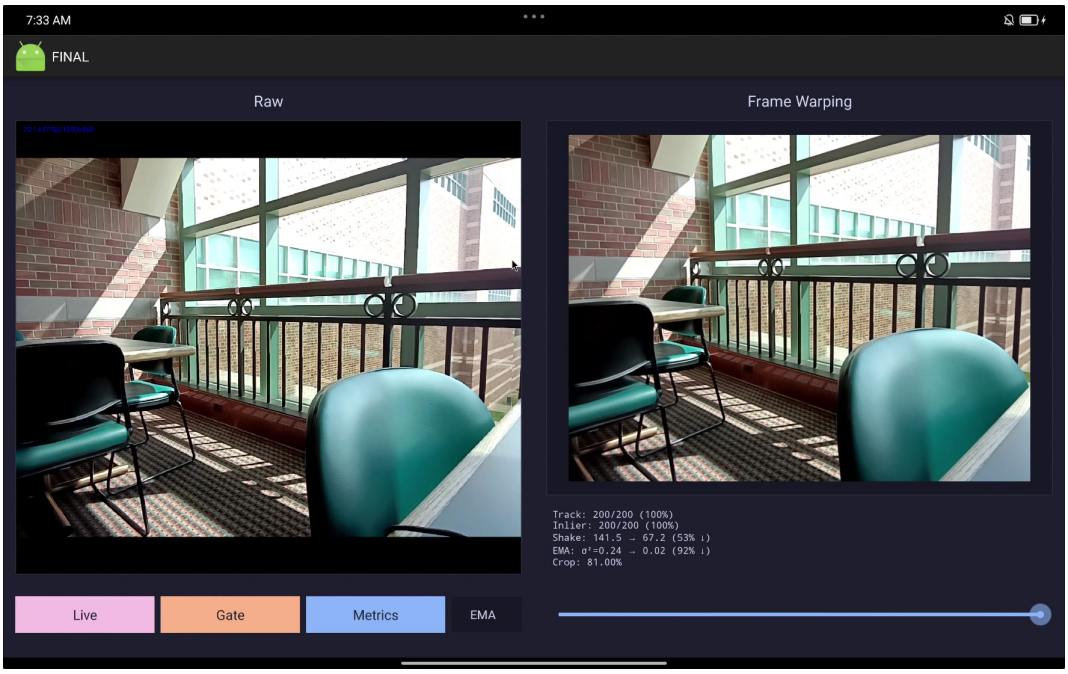

Figure 9: Stage 6: Final stabilized output with live metrics overlay. The raw feed (left) and stabilized output (right) are displayed simultaneously.

5.2 Quantitative Evaluation

Four filter configurations were evaluated on separate handheld test sequences using the live metrics overlay. The same six metrics were recorded for each run: feature tracking success rate, RANSAC inlier rate, shake reduction, jitter variance reduction, and crop efficiency. Figures 10, 11, 12, and 13 show the per-frame metric plots for each filter. Table 3 summarizes the aggregate values extracted from the raw logs.

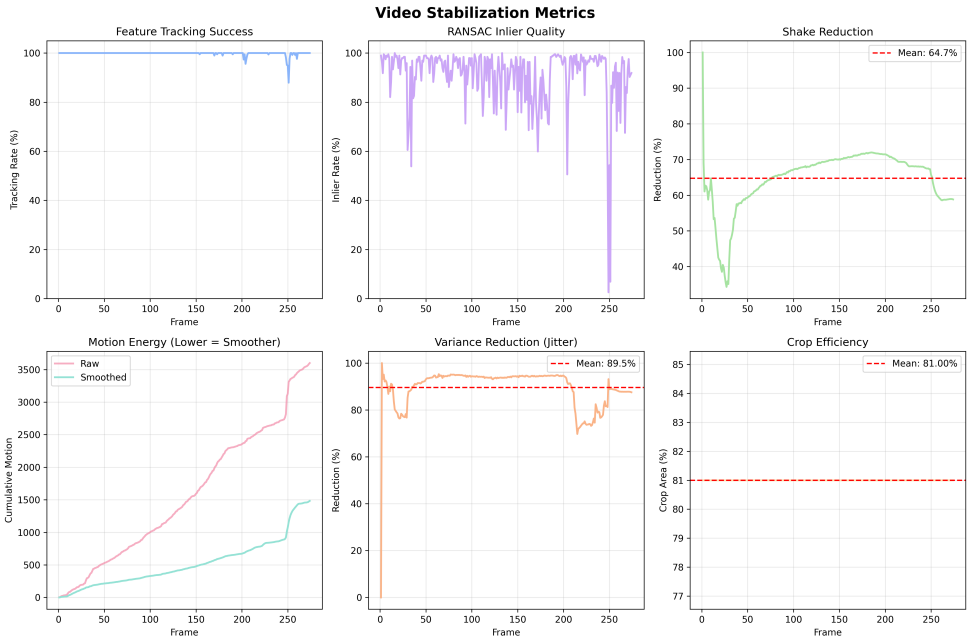

Figure 10: Per-frame metrics for the Kalman filter over a 274-frame test sequence.

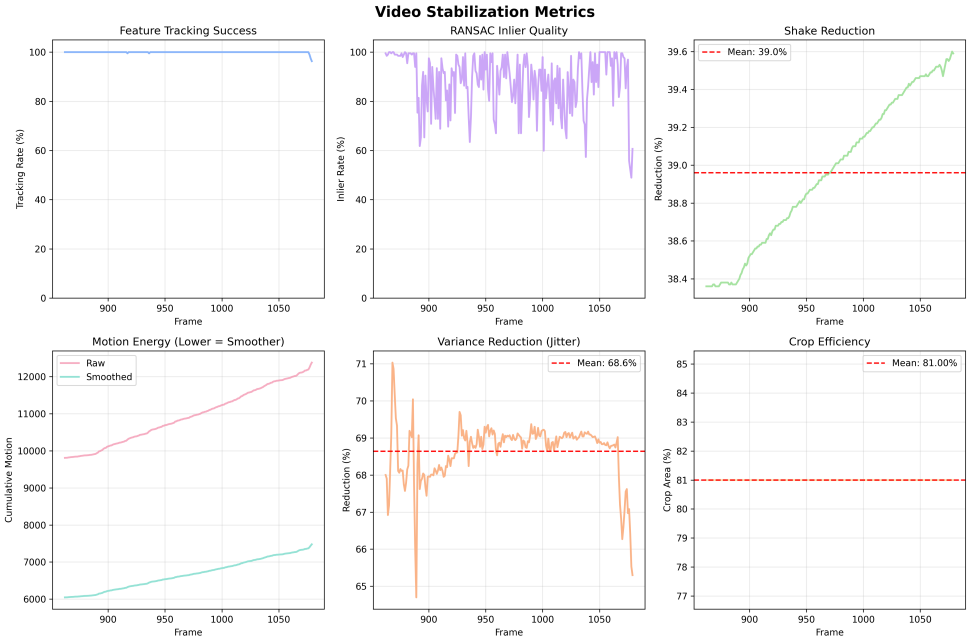

Figure 11: Per-frame metrics for the Gaussian smoother over a 217-frame test sequence.

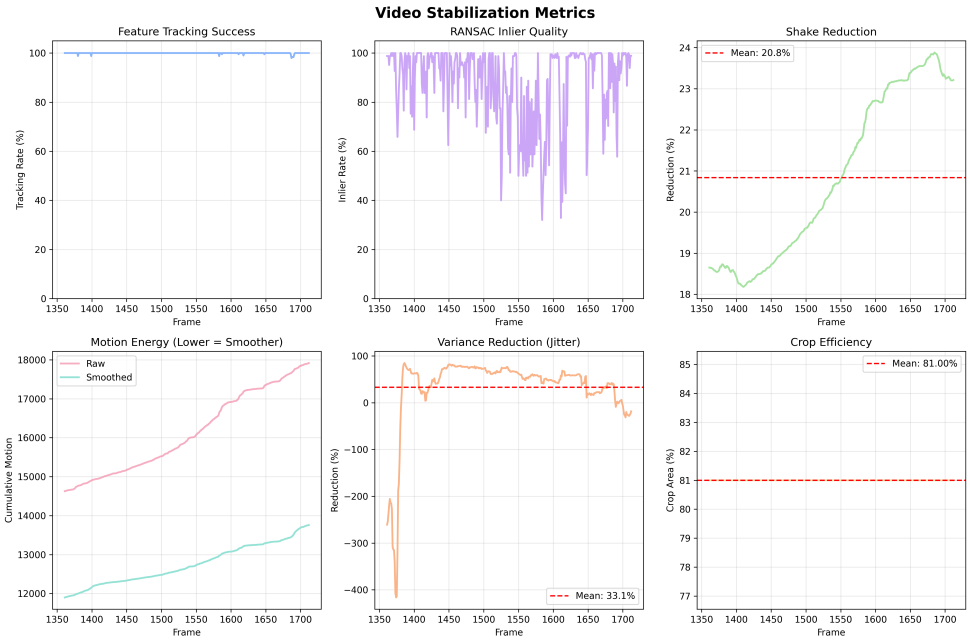

Figure 12: Per-frame metrics for the EMA smoother over a 351-frame test sequence.

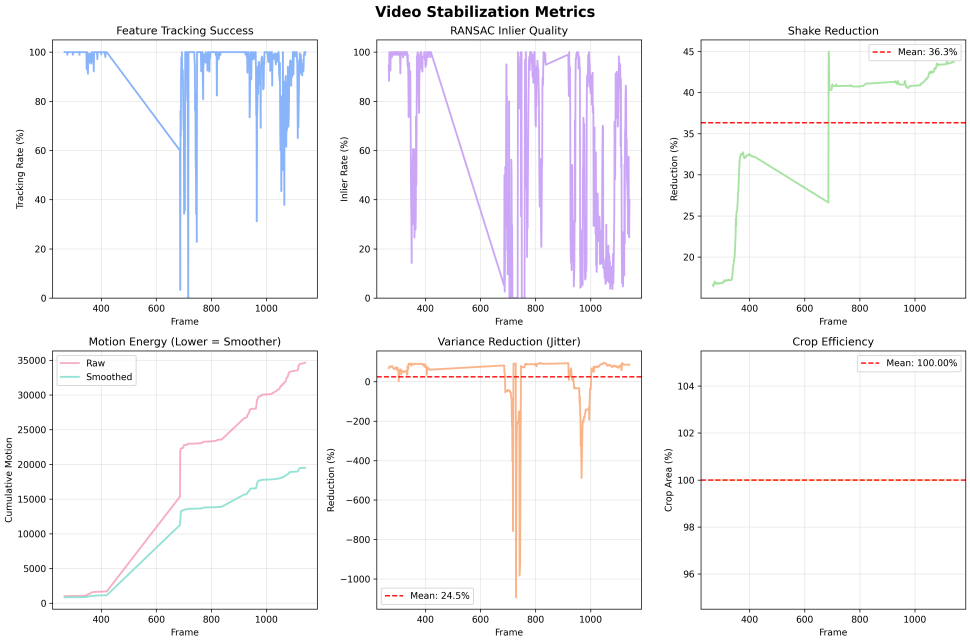

Figure 13: Per-frame metrics for the UMA smoother over a 529-frame test sequence. Large negative variance spikes in the middle correspond to a pipeline restart; the 24.5% mean is computed over the full sequence including these transients.

Filter	Frames	Track Rate	Inlier Rate	Shake Reduction	Var. Reduction	Crop
Kalman	274	99.84%	90.79%	**64.7%**	**89.9%**	81.0%
Gaussian	217	99.96%	88.31%	39.0%	68.6%	81.0%
EMA	351	99.96%	87.50%	20.8%	33.9%	81.0%
UMA	529	94.08%	61.23%	36.3%	24.5%	100.0%

Table 3: Aggregate stabilization metrics across the four filter configurations.

5.3 Analysis

Core Pipeline Validation

Feature tracking success was consistently near 100% across the Kalman, Gaussian, and EMA test runs, with a minimum observed rate of 87.9% during the most challenging motion in the Kalman sequence, and above 96% at all times for Gaussian and EMA. The UMA sequence shows a lower mean tracking rate of 94.08%, reflecting a longer and more varied test sequence that included a pipeline restart mid-run during which the feature count dropped transiently to near zero. RANSAC inlier rates were strong for Kalman, Gaussian, and EMA (averaging between 87.5% and 90.8%), but the UMA sequence averages 61.23%, again attributable to the pipeline restart frames where the inlier count collapsed entirely before recovering. For the stable portions of the UMA sequence, inlier rates track closely with those of the other filters. The motions energy plots confirm that all four filters successfully reduce cumulative trajectory energy relative to the raw estimate, validating the fundamental stabilization mechanism.

Effect of Motion Gating

The gyroscope gate addresses a specific failure mode of the visual pipeline: during rapid camera motion, large pixel displacements violate the brightness constancy assumption of Lucas–Kanade, causing optical flow to fail or produce corrupted vectors. Without the gate, these corrupted vectors propagate through RANSAC and affine estimation into the cumulative path, introducing sudden erroneous corrections that are worse than no stabilization at all. The gate's contribution is visible indirectly in the RANSAC inlier data: even with the gate enabled, transient inlier drops to as low as 2.6% appear in the Kalman test sequence, corresponding to moderate-speed motions that fell below the gate threshold but still degraded optical flow quality. These frames represent the residual failure region that the gate does not cover, and they motivate future work on a more graduated gating response rather than the current hard on/off threshold. When the gate fires on a truly sharp movement, the raw frame is passed through unchanged, which is the correct behavior: an unsmoothable frame is better left as-is than actively miscorrected.

Filter Design Comparison

Shake reduction improves from EMA (20.8%) to UMA (36.3%) to Gaussian (39.0%) to Kalman (64.7%), reflecting increasing filter sophistication. The UMA variance reduction figure (24.5%) is depressed by the mid-sequence pipeline restart; during stable operation it performs comparably to Gaussian, consistent with their similar windowed designs. The EMA's fixed decay factor $\alpha = 0.3$ limits responsiveness to genuine motion changes, while the Gaussian's bell-shaped kernel better emphasises recent frames. The Kalman filter outperforms all others by a wide margin owing to its state-space formulation: it maintains an explicit estimate updated via a principled gain, responding quickly to real motion while rejecting noise — a tradeoff window-based filters cannot achieve simultaneously. The UMA crop efficiency of 100% against 81% for the other filters reflects lower corrective warp magnitudes in that test sequence rather than an algorithmic property. The large negative variance spike in the EMA results is a warmup artifact avoided by the Kalman (first-measurement initialization) and Gaussian (sparse-history downweighting), and absent in UMA because its window fills gradually from the first available frames.

Testing Conditions and Comparability

Each filter was tested on a separately recorded handheld sequence, so metric comparisons are approximate. Handheld shake is inherently non-repeatable across sessions, and a preferable methodology would use a standardised mechanical stimulus to ensure all filters face identical input motion.

5.4 Problems Encountered

Even with the gyroscope gate enabled, frames with moderate angular velocity below the gate threshold still produced degraded optical flow, with RANSAC inlier rates as low as 2.6% in the Kalman sequence. These frames briefly corrupt the cumulative path before the smoother attenuates the error. A confidence-weighted path update that scales the influence of each frame by its inlier ratio would address this more cleanly than the current binary gate.

The EMA filter initializes its state to the first measurement to avoid zero-initialization overcorrection, but any mid-session reset (e.g. rapid stage switching) can briefly reintroduce the transient. The UMA exhibits an analogous sparse-window warmup issue: after a pipeline restart the history buffer refills gradually, producing unreliable warp estimates visible as large negative variance spikes in the UMA metrics plot.

Low-texture scenes caused the tracked feature count to approach the re-detection threshold more frequently. Full re-detections introduce brief path discontinuities because the new feature set differs from the old one, leaving the affine estimate for that frame effectively unconstrained.

Handheld shake induction is not a replicable stimulus across test sessions. Differences in shake magnitude and frequency between runs make it difficult to isolate filter performance

from sequence difficulty, and a standardised mechanical shake source would be preferable for future evaluations.

6 Conclusions and Future Work

The pipeline successfully reduces handheld camera jitter in real time on an Android tablet. Feature tracking exceeded 99.8% and RANSAC inlier rates exceeded 87.5% across the Kalman, Gaussian, and EMA sequences, confirming reliable upstream operation. The Kalman filter achieved the strongest stabilization (64.7% shake reduction, 89.9% variance reduction), followed by Gaussian (39.0% / 68.6%), UMA (36.3% / 24.5%), and EMA (20.8% / 33.9%). UMA variance reduction is penalised by a mid-sequence restart and is not directly comparable; its shake reduction is consistent with Gaussian-level performance on stable footage. Crop efficiency was 81% for Kalman, Gaussian, and EMA, and 100% for UMA due to lower warp magnitudes in that test sequence. The gyroscope gate protected the pipeline from large-motion optical flow failures, though its binary threshold leaves moderate-speed motion partially unhandled.

Future Work

The most impactful near-term improvement is multithreading. The motion estimation, path smoothing, and frame warping stages have largely independent data dependencies and could be distributed across threads in a producer-consumer model, processing frame k through optical flow and RANSAC concurrently with applying the warp from frame $k-1$, as demonstrated by Lim et al. [1]. Reducing the tracked feature count is a complementary approach; stabilization quality is largely insensitive to feature count above roughly 50 points, well below the current 200-point maximum.

Replacing the binary gyroscope gate with adaptive fusion would cover the moderate-motion gap: the gyroscope reading could modulate the Kalman gain directly, scaling down the visual measurement's influence on high-angular-velocity frames rather than disabling the pipeline entirely.

Finally, standardising the evaluation with a controlled mechanical shake source would allow cleaner filter comparisons, and extending the motion model to a full projective homography would handle perspective distortion from large rotations.

References

[1] Lim, A., Ramesh, B., Yang, Y., Xiang, C., Gao, Z., & Lin, F. (2019). Real-Time Optical Flow-Based Video Stabilization for Unmanned Aerial Vehicles. *Journal of Real-Time Image Processing*, 16. doi:10.1007/s11554-017-0699-y

[2] Deniz, O., Bueno, G., Bermejo, E., & Sukthankar, R. (2011). Fast and Accurate Global Motion Compensation. *Pattern Recognition*, 44(12), 2887–2901. doi:10.1016/j.patcog.2010.10.019