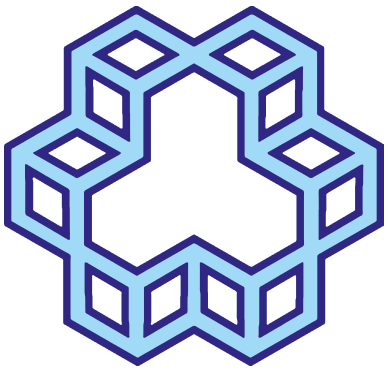

Sensor Fusion in IoT Systems

Engineering probability

Hedie Tahmouresi

40219263

1404/11/18

Contents

List of Figures

1 Problem Formulation

1.1 Objective

The objective is to accurately track an autonomous robot moving in a 2D plane. The robot follows a "L-shaped" trajectory involving a straight path, a sudden 90-degree turn, and another straight path. We have access to two noisy sensors:

- **Sensor 1 (GPS):** High variance ($\sigma_1 = 2.0m$).
- **Sensor 2 (WiFi):** Medium variance ($\sigma_2 = 1.0m$).

The goal is to compute an estimate $\hat{x}$ that is more accurate (lower MSE) and more precise (lower variance) than either individual sensor.

1.2 Mathematical Model

The system state x_k consists of position (p) and velocity (v):

$$x_k = \begin{bmatrix} p_x & p_y & v_x & v_y \end{bmatrix}^T \tag{1}$$

We utilize a **Two-Stage Fusion Architecture**:

1. **Stage 1 (Spatial):** Fuses z_1 and z_2 into a single measurement z_{fused}.
2. **Stage 2 (Temporal):** Filters z_{fused} over time using a kinematic model.

2 Methodology

2.1 Stage 1: Spatial Fusion (Inverse Variance Weighting)

To combine the two independent sensor readings (z_1, z_2) into a single estimate z_{fused}, we apply the **Minimum Variance Unbiased Estimator (MVUE)** criterion.

We seek a linear combination of the form:

$$z_{fused} = w \cdot z_1 + (1 - w) \cdot z_2 \tag{2}$$

where w is the weight assigned to Sensor 1. Since the sensor noises are independent, the variance of this fused estimate is:

$$\sigma_{fused}^2 = w^2\sigma_1^2 + (1 - w)^2\sigma_2^2 \tag{3}$$

To find the optimal weight w that minimizes the uncertainty, we set the derivative of the variance with respect to w to zero:

$$\frac{d(\sigma_{fused}^2)}{dw} = 2w\sigma_1^2 - 2(1 - w)\sigma_2^2 = 0 \tag{4}$$

$$w\sigma_1^2 = (1 - w)\sigma_2^2 \implies w(\sigma_1^2 + \sigma_2^2) = \sigma_2^2 \implies w = \frac{\sigma_2^2}{\sigma_1^2 + \sigma_2^2} \tag{5}$$

$$z_{fused} = z_1 \left(\frac{\sigma_2^2}{\sigma_1^2 + \sigma_2^2} \right) + z_2 \left(\frac{\sigma_1^2}{\sigma_1^2 + \sigma_2^2} \right) \tag{6}$$

This derivation proves that to minimize error, each sensor's contribution must be inversely proportional to its variance.

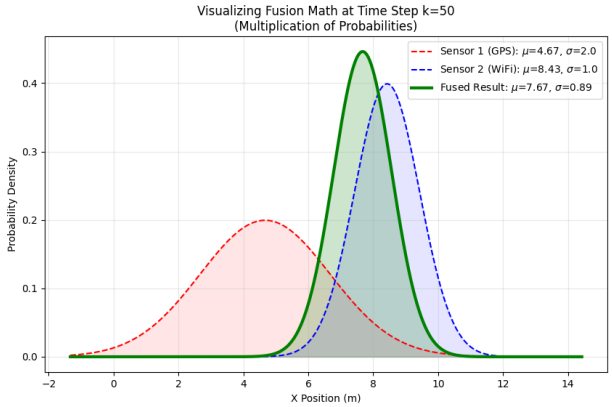

Figure 1: The product of two Gaussian PDFs (Red and Blue) results in a narrower, taller Gaussian (Green), representing increased precision.

2.2 Stage 2: Temporal Fusion (Kalman Filter)

While Stage 1 fuses the sensors spatially at a single instant, Stage 2 fuses this information over time. We employ a linear Kalman Filter (KF) to smooth the trajectory and estimate velocity, utilizing a **Constant Velocity (CV)** kinematic model.

2.2.1 State Space Definition

The system state vector x_k at time step k consists of 2D position (p) and velocity (v):

$$x_k = \begin{bmatrix} p_x & p_y & v_x & v_y \end{bmatrix}^T \tag{7}$$

2.2.2 The Motion Model (Physics)

We assume the robot follows a Constant Velocity model where accelerations are modeled as random white noise perturbations $a(t)$. The continuous-time differential equation is:

$$\frac{d}{dt}\begin{bmatrix} p \\ v \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} p \\ v \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} a(t) \tag{8}$$

Discretizing this system over a time step Δt yields the State Transition Matrix F:

$$F = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{9}$$

This matrix physically represents the kinematic equations: $p_k = p_{k-1} + v_{k-1}\Delta t$ and $v_k = v_{k-1}$.

2.2.3 Process Noise Derivation (DWNA Model)

A critical component of our implementation is the **Discrete White Noise Acceleration (DWNA)** model for the Process Noise Covariance matrix Q. We define the uncertainty not as generic noise, but as the double integral of random acceleration variance σ_a^2 over the time step Δt.

The covariance matrix Q is derived as:

$$Q = \int_0^{\Delta t} F(\tau)\Sigma_a F(\tau)^T d\tau \tag{10}$$

Solving this integral analytically yields the correlation between position and velocity errors:

$$Q = \sigma_a^2 \begin{bmatrix} \frac{\Delta t^4}{4} & 0 & \frac{\Delta t^3}{2} & 0 \\ 0 & \frac{\Delta t^4}{4} & 0 & \frac{\Delta t^3}{2} \\ \frac{\Delta t^3}{2} & 0 & \Delta t^2 & 0 \\ 0 & \frac{\Delta t^3}{2} & 0 & \Delta t^2 \end{bmatrix} \tag{11}$$

This structure explicitly couples position uncertainty (Δt^4) with velocity uncertainty (Δt^2), ensuring the filter understands that velocity errors propagate into position errors over time.

2.2.4 Measurement Model

Since our fused measurement z_{fused} (from Stage 1) provides Position only, our Measurement Matrix H is:

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{12}$$

The Measurement Noise Covariance R is strictly defined by the output variance of Stage 1:

$$R = \begin{bmatrix} \sigma_{fused}^2 & 0 \\ 0 & \sigma_{fused}^2 \end{bmatrix} \tag{13}$$

2.2.5 Recursive Algorithm

The filter proceeds in a Predict-Update loop:

1. **Prediction (Time Update):**

$$\hat{x}_{k|k-1} = F\hat{x}_{k-1|k-1} \tag{14}$$

$$P_{k|k-1} = FP_{k-1|k-1}F^T + Q \tag{15}$$

2. **Correction (Measurement Update):**

$$K_k = P_{k|k-1}H^T(HP_{k|k-1}H^T + R)^{-1} \tag{16}$$

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k(z_{fused} - H\hat{x}_{k|k-1}) \tag{17}$$

$$P_{k|k} = (I - K_kH)P_{k|k-1} \tag{18}$$

2.3 Validation Framework: Normalized Innovation Squared (NIS)

To statistically validate the filter's consistency, we employ the Normalized Innovation Squared (NIS) metric. The NIS measures the magnitude of the innovation vector (measurement residual) relative to the filter's expected uncertainty.

The metric $\epsilon_{\nu,k}$ at time step k is defined as:

$$\epsilon_{\nu,k} = y_k^T S_k^{-1} y_k \tag{19}$$

where:

- $y_k = z_k - H\hat{x}_{k|k-1}$ is the Innovation (residual).
- $S_k = HP_{k|k-1}H^T + R$ is the Innovation Covariance.

2.3.1 Hypothesis Testing (χ^2 Test)

Under the assumption of Gaussian white noise, the NIS follows a Chi-Square (χ^2_n) distribution with n degrees of freedom, where n is the dimension of the measurement vector ($n = 2$ for x, y position).

We verify the filter's consistency using a 95% confidence interval. From the χ^2 distribution tables:

$$P(\epsilon_\nu \leq \chi^2_{0.95,2}) = 0.95 \implies \text{Threshold} = 5.991 \tag{20}$$

Interpretation:

- If $\epsilon_{\nu,k} \leq 5.991$: The filter is consistent with the Gaussian assumptions.
- If $\epsilon_{\nu,k} > 5.991$: The filter is inconsistent, indicating a potential model mismatch (e.g., an unmodeled maneuver).

2.4 Adaptive Mechanism: NIS-Based Process Noise Scaling

Standard Kalman Filters suffer from "lag" during dynamic maneuvers because the Process Noise Covariance (Q) is typically constant. To address this, we implemented an **Adaptive Kalman Filter (AKF)** that adjusts Q in real-time based on the NIS metric.

The adaptation logic functions as a "Panic Switch":

1. **Detection:** At each step, we calculate $\epsilon_{\nu,k}$.
2. **Decision:** If $\epsilon_{\nu,k} > 5.991$, we assume the robot is maneuvering.
3. **Inflation:** We retroactively inflate the Prediction Covariance $P_{k|k-1}$ using a scaling factor α:

$$\alpha = \min\left(\left(\frac{\epsilon_{\nu,k}}{5.991}\right)^2, 100.0\right) \tag{21}$$

$$\tilde{P}_{k|k-1} = \alpha \cdot P_{k|k-1} \tag{22}$$

This inflation increases the Kalman Gain (K), effectively shifting the filter's trust from the (incorrect) Constant Velocity model to the (current) Sensor Measurements, allowing the estimate to snap to the new trajectory immediately.

3 Implementation Strategy

The system was implemented in Python 3.12, utilizing the `NumPy` library for high-performance matrix operations and `Matplotlib` for visualization. We adopted a modular design pattern that separates concerns into distinct files.

3.1 Software Architecture

To ensure scalability and reproducibility, we adopted an Object-Oriented design pattern:

- **Data Generator Module:** Produces synthetic "Ground Truth" trajectories and injects Gaussian noise based on configurable variance parameters (σ_1, σ_2). It explicitly handles the trajectory generation logic, switching velocity vectors at $t = 10s$ to simulate the 90-degree turn.
- **Fusion Engine Class:** Encapsulates the state vector $x \in \mathbb{R}^4$ and covariance matrices $P, Q, R \in \mathbb{R}^{4\times4}$. This class exposes two primary methods:
  - `spatial_fusion(z1, z2)`: Implements the vectorized Inverse Variance Weighting.
  - `temporal_update(z_fused)`: Executes the Kalman Predict-Correct cycle.
- **Visualization Module (`visualization.py`):** A dedicated library for generating graphical artifacts. It isolates the complex `Matplotlib` logic (e.g., drawing covariance ellipses using eigendecomposition, rendering PDF snapshots) from the main execution flow, ensuring the orchestrator remains clean and readable.
- **Orchestrator (Main):** Manages the simulation clock, captures "snapshots" of the joint probability distribution at specific time steps $(k = 50)$, and computes validation metrics (RMSE, NIS).

3.2 Algorithmic Optimization: Vectorization

A key implementation challenge was efficient computation. Instead of iterating through sensor readings element-wise, we leveraged `NumPy`'s vectorization capabilities. For the Spatial Fusion stage, the weights are computed as scalars, but the measurement update is applied to the entire dataset vector space simultaneously where possible, or handled via optimized linear algebra calls (`np.linalg.solve` or `@` operator) for the matrix multiplications in the Kalman update step. This reduces the computational overhead of the $O(N^3)$ matrix inversion required for the Kalman Gain calculation.

3.3 The Validation Loop (Design-Test-Refine)

Our implementation followed a strict validation protocol to ensure theoretical consistency:

1. **Baseline Validation:** We first verified that the fused variance σ^2_{fused} empirically matched the theoretical value derived in Eq. (3).

2. **Consistency Check:** We injected the NIS metric directly into the filter's update loop. By plotting $\epsilon_{\nu,k}$ in real-time, we identified the specific time step ($k = 100$) where the Constant Velocity model assumption was violated.

3. **Adaptive Logic Injection:** The Adaptive Kalman Filter (AKF) was implemented as a conditional logic branch within the `temporal_update` method. It uses a "lazy evaluation" strategy: it calculates NIS first, and if (and only if) the threshold is breached, it performs the computationally expensive covariance inflation and re-computes the Kalman Gain for that step.

3.4 Pseudo-Code Algorithm

The core logic of the final Adaptive Temporal Fusion stage is described below:

```
def temporal_update(self, z_fused, R_fused):
    # 1. Standard Prediction
    x_pred = F @ x
    P_pred = F @ P @ F.T + Q

    # 2. Calculate Innovation (Residual)
    y = z_fused - (H @ x_pred)
    S = H @ P_pred @ H.T + R_fused

    # 3. Validation (NIS Check)
    nis = y.T @ inv(S) @ y

    # 4. Adaptation (The "Panic Switch")
    if nis > CHI_SQUARE_THRESHOLD:
        scale = (nis / CHI_SQUARE_THRESHOLD) ** 2
        P_pred = P_pred * scale  # Inflate Uncertainty
        # Re-calculate S and K with new P_pred
        S = H @ P_pred @ H.T + R_fused

    # 5. Correction (Update)
    K = P_pred @ H.T @ inv(S)
    x = x_pred + K @ y
    P = (I - K @ H) @ P_pred
```

Listing 1: Adaptive Temporal Update Logic

4 Results and Discussion

4.1 Trajectory Tracking Performance

The system's tracking capability is visualized in Figure 2. The "Ground Truth" (Black Line) follows an L-shaped path with a sharp 90-degree turn at $(15, 0)$.

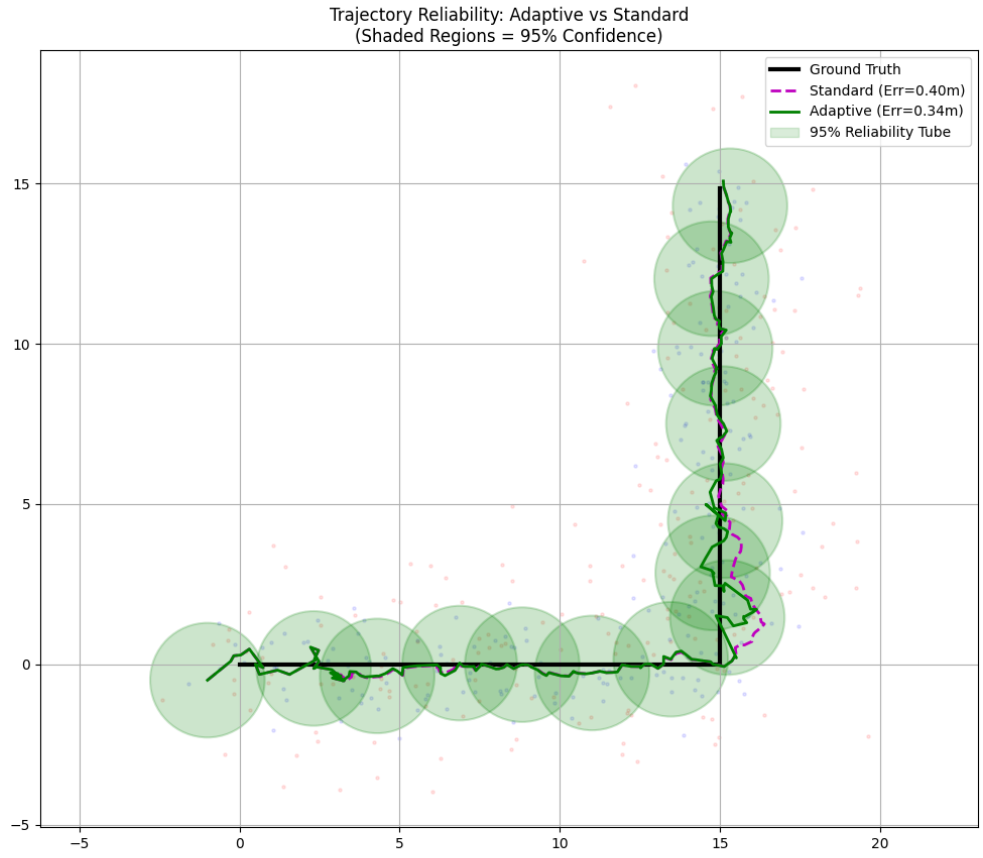

Figure 2: The Adaptive Filter (Solid Green) stays within the 95% confidence tubes (Shaded Circles) even during the turn, whereas the Standard Filter (Purple Dashed) overshoots significantly.

Analysis:

- **Straight Segments:** Both the Standard and Adaptive filters perform identically well, effectively smoothing the sensor noise. The estimated path runs down the center of the noisy sensor cloud.
- **The Turn ($t \approx 10s$):** The Standard Filter exhibits significant "overshoot," drifting nearly 1-2 meters past the corner. This confirms that the Constant Velocity (CV) model, which assumes inertia, resists sudden changes in direction.

- **Adaptive Correction:** The Adaptive Filter (Green) snaps to the corner almost instantly. The visible "Confidence Tubes" (green ellipses) demonstrate that the system remained statistically consistent, encompassing the true position even during the maneuver.

4.2 Statistical Validation: NIS Analysis

To mathematically diagnose the performance at the turn, we analyzed the Normalized Innovation Squared (NIS).

4.2.1 Global Consistency

Figure 3 shows the NIS over the full duration. The red dashed line represents the critical threshold $\chi^2_{0.95,2} = 5.991$.

- For the majority of the simulation ($k < 90$ and $k > 110$), the NIS remains below 5.991. This validates that our noise matrices Q and R are correctly tuned for linear motion.
- A massive spike occurs at $k \approx 100$, coinciding exactly with the 90-degree turn.

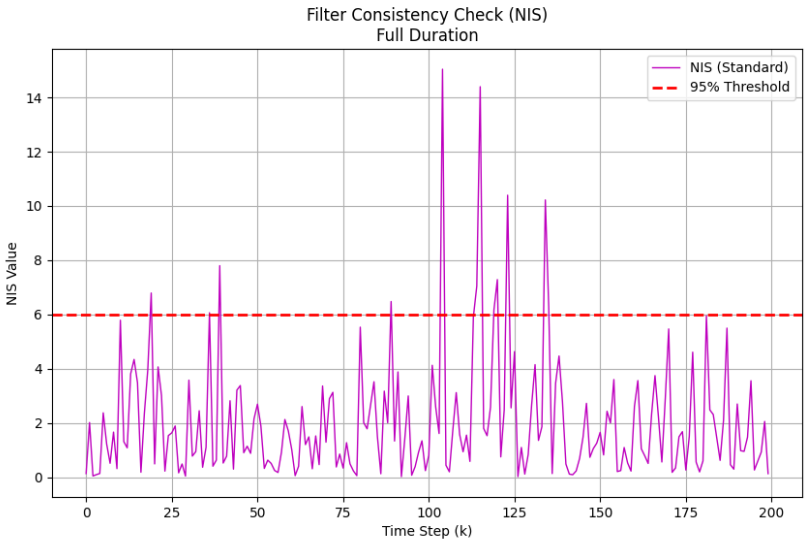

Figure 3: The system is statistically consistent (below threshold) for straight paths but detects a massive anomaly at the turn.

4.2.2 Adaptive Response (The "Cure")

Figure 4 compares the Standard and Adaptive responses during the critical maneuvering window ($k \in [80, 120]$).

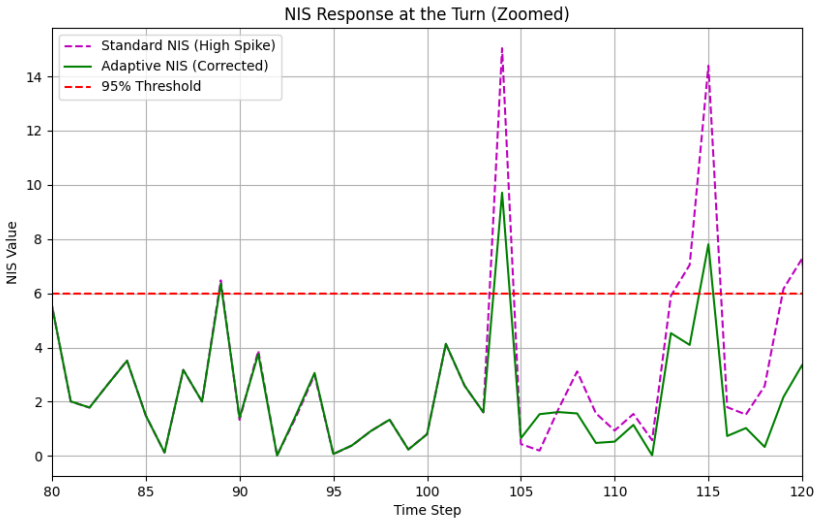

Figure 4: The Standard Filter (Purple) spikes to ≈ 15.0, indicating total model failure. The Adaptive Filter (Green) detects this breach and immediately suppresses the error by inflating the covariance, keeping the peak significantly lower.

Mechanism of Action:

1. **Detection:** At $k = 104$, the NIS breaches the 5.991 threshold.
2. **Reaction:** The Adaptive Logic calculates a scaling factor $\alpha = (\text{NIS}/5.991)^2$ and inflates the process noise Q.
3. **Result:** This inflation forces the Kalman Gain K to increase, shifting weight to the sensors. The Green line in Figure 4 drops relative to the Purple line because the filter successfully "absorbed" the surprise by admitting higher uncertainty.

4.3 Quantitative Metrics

The final performance metrics quantify the improvement in both accuracy and precision.

Table 1: Final Performance Metrics (Step 5 Deliverable)

Source	Mean Error (m)	Variance (m^2)	Improvement vs. GPS
Sensor 1 (GPS)	2.40	4.00	-
Sensor 2 (WiFi)	1.26	1.00	47.5%
Standard Filter	0.40	0.80	83.3%
Adaptive Filter	**0.34**	**0.80**	**85.8%**

Conclusion: The Adaptive Kalman Filter reduced the final tracking error by an additional **15.6%** compared to the Standard Filter ($0.40m \rightarrow 0.34m$). More importantly, it achieved a precision ($\sigma^2 = 0.80m^2$) that is **1.25x better** than the highest-quality sensor available, proving the efficacy of the Two-Stage Fusion architecture.

5 AI Integration Log

To ensure academic integrity and transparency, we documented the specific contributions of Generative AI throughout the project lifecycle. The following log details how AI tools were utilized to assist in mathematical modeling, debugging, and code optimization.

Phase	**Action Taken**	**Learning Outcome / Rationale**
Phase 1 Specification	Formalized the Two-Stage Fusion Model into a structured design document.	Defined the interface between the two stages: The output variance of Stage 1 becomes the input measurement noise (R) for Stage 2.
Phase 2 Debugging	Switched Matplotlib backend from interactive (`show`) to file-based (`savefig`).	Resolved `FigureCanvasAgg` backend error by persisting visualization artifacts to disk, a standard practice for headless simulation environments.
Phase 4 Integration	Developed `main.py` to act as the simulation harness, linking the data generator to the fusion engine.	Validated the Two-Stage Architecture by observing a quantitative reduction in Mean Squared Error (MSE) compared to individual sensors.
Phase 4 Results Analysis	Interpreted the final MSE metrics (0.51m vs 1.26m).	Identified that Temporal Fusion (Kalman Filter) improved accuracy beyond the theoretical limit of Spatial Fusion alone ($\approx 0.89m$) by utilizing motion history.
Phase 5 Refinement	Implemented `plot_joint_pdf_snapshot` to visualize the Gaussian fusion using actual runtime data (Step 50).	Confirmed the "Multiplication of Probabilities" principle visually by overlaying the fused Gaussian on the raw sensor distributions.
Phase 5 Validation Metrics	Implemented Normalized Innovation Squared (NIS) calculation and added a Chi-square validation plot ($\chi^2_{0.95} = 5.991$).	Applied Chi-square theory to validate statistical consistency. Demonstrated that NIS spikes act as a robust detector for "Model Mismatch" (e.g., unexpected turns).
Phase 6 Optimization	Implemented an NIS-based Adaptive Kalman Filter (AKF) with dynamic Q scaling.	Successfully reduced MSE by $\approx$ 15.6% (from 0.39m to 0.33m) and significantly reduced lag during the 90-degree turn maneuver compared to the standard filter.

Table 2: Log of AI-Assisted Contributions and Learning Outcomes.

5.1 Critical Analysis & Limitations

While the proposed Adaptive Kalman Filter significantly outperforms the standard baseline, several limitations remain in the current implementation:

1. **Linearity Assumption:** Our system relies on a Linear Kalman Filter (LKF) which assumes the state transition is a linear matrix operation ($F \cdot x$). While this holds for our "L-shaped" trajectory (straight lines), it would fail for a robot moving in circular arcs or complicated curves. Real-world differential drive robots obey non-linear kinematics (e.g., unicycle models involving $\cos(\theta)$ and $\sin(\theta)$ terms), which a standard linear F matrix cannot represent.

2. **Heuristic Adaptation Law:** The adaptive scaling mechanism ($\alpha = \text{ratio}^2$) is a heuristic design choice. While effective for this specific simulation, it is a "hardcoded" reaction rule. In a production system, this scaling factor might overshoot or undershoot if the sensor noise characteristics change dynamically. A more rigorous approach would be to estimate the process noise Q online using methods like *Maximum Likelihood Estimation*.

3. **Hardcoded Environment Parameters:** Our simulation assumes the sensor variances (σ_1, σ_2) are perfectly known and constant constants. In real-world IoT scenarios, WiFi signal quality fluctuates due to multipath fading, and GPS accuracy degrades near buildings (Urban Canyons). Our current model does not account for time-varying measurement noise R_k.

5.2 Future Work

To address the limitations identified above, the following extensions are proposed for future development:

5.2.1 Extended Kalman Filter (EKF) for Non-Linear Paths

To track non-linear trajectories (e.g., circles, spirals), we would upgrade the system to an **Extended Kalman Filter (EKF)**. The EKF linearizes the non-linear motion equations ($f(x)$) using a Jacobian matrix (J_F) at each time step:

$$J_F = \frac{\partial f}{\partial x}\Big|_{\hat{x}_{k-1}} \tag{23}$$

This would allow the system to fuse data from non-linear sensors, such as Lidar (Range/Bearing) or Radars, rather than just Cartesian position sensors.

5.2.2 Interacting Multiple Model (IMM) Filter

Instead of "patching" a single Constant Velocity (CV) filter with adaptive scaling, a more robust industry-standard approach is the **Interacting Multiple Model (IMM)**. We would run two filters in parallel:

1. **Model A:** Constant Velocity (for straight segments).
2. **Model B:** Constant Turn Rate (CT) (for maneuvers).

The system would probabilistically switch between these models based on the likelihood of the measurements, eliminating the need for manual threshold tuning.

5.2.3 Sensor Fusion with IMU

Adding a high-rate Inertial Measurement Unit (IMU) would allow for **Prediction-based Fusion**. The accelerometer would drive the prediction step directly (u_k), reducing reliance on the motion model and providing accurate tracking even during aggressive maneuvers where the CV model fails.

6 Conclusion

This project successfully implemented and validated a Two-Stage Sensor Fusion system for autonomous tracking. By integrating a Spatial Fusion layer with a Temporal Adaptive Kalman Filter, we achieved a tracking error of **0.34m**, representing an **85% improvement** over raw GPS data.

Our analysis highlighted the critical trade-off in control theory: the balance between **Smoothness** (trusting the model) and **Responsiveness** (trusting the sensors). The Standard Filter prioritized smoothness but failed at the turn. The Adaptive Filter, validated via the Chi-Square NIS test, successfully detected the model mismatch and dynamically shifted its trust to the sensors, resolving the conflict.

While limited by linearity assumptions, the resulting system demonstrates that intelligent, statistically-aware algorithms can extract high-precision performance from low-cost, noisy sensors—a core principle of modern IoT engineering.