Investigation: 2-State Kalman Filter for PLCs (Position + Velocity)

Target Platform: Rockwell Studio 5000 (Logix / AOI, Structured Text)

Document Version: 1.3
Date: 2026-02-16
Author: Taylor Turner <resume.tturner@gmail.com>

Introduction

This document presents an analysis and implementation rationale for a 2-state discrete Kalman filter designed for Rockwell Logix PLCs. The filter estimates position and velocity from a position-only measurement using a constant-velocity model. The implementation is optimized for scan-based execution, deterministic behavior, and numeric safety within PLC constraints.

This implementation emphasizes the mapping between optimal estimation theory and practical industrial constraints such as variable scan time, limited computational primitives, and simplified tuning. A non-optimal velocity "bleed" heuristic is included within the design as an engineering mitigation for constant-velocity model bias at changes in the system discontinuities.

The Industrial Problem

Position signals in automation systems often suffer from:

- Noise from sensors and electrical interference
- Quantization from encoder counts or integer scaling
- Irregular timing due to scan cycles and communications

At the same time, applications require:

- Smooth position for control and visualization
- Reliable velocity estimation
- Deterministic PLC execution
- Simple and predictable commissioning

Traditional approaches (moving averages, differentiation) either introduce lag or amplify noise.

Why This Approach Was Selected

Method	Advantages	Limitations	Decision
Moving average	Simple	Adds lag, no velocity	Rejected
Numerical differentiation	Fast	Amplifies noise	Rejected
α-β filter	Lightweight	Less adaptive	Considered
2-state Kalman	Optimal weighting, velocity estimate	Moderate complexity	Selected

This approach provides the best balance between performance, interpretability, and PLC feasibility.

What Users Will Observe in Operation

Expected Behavior

- Position output becomes smoother without excessive delay
- Velocity estimate stabilizes quickly during motion
- Minor overshoot may occur at sudden stops (bleed factor omits this)
- Filter remains stable during scan-time variations

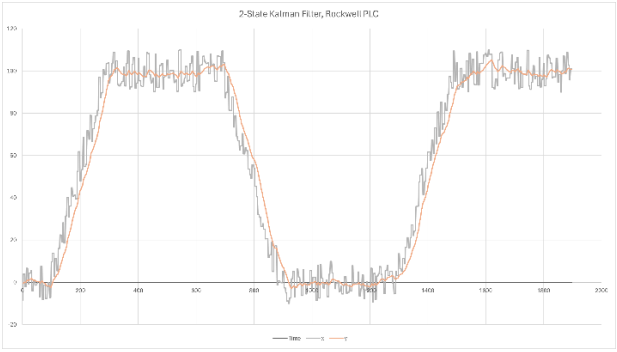

Figure 1: Kalman Filter From Rockwell PLC Trend

Conclusion

The PLC Kalman observer provides a robust and practical method for improving position signals and estimating velocity in industrial automation systems. It bridges optimal estimation theory with PLC execution constraints, delivering improved motion insight without sacrificing determinism or maintainability.

For most applications requiring smooth position and reliable velocity estimation, this observer offers a clear improvement over traditional filtering and differentiation methods.

System Overview

What the AOI (Rockwell's Add on Instruction) is.
This AOI is a *2-state observer* with persistent memory:

- State (persistent estimates):
  - Estimated position, $\hat{x}$
  - Estimated velocity, $\hat{v}$
- Outputs (what other logic consumes):
  - Filtered position
  - Filtered velocity
- Covariance (persistent uncertainty model):
  - 2×2 matrix in row-major order

The AOI is built around two phases that run every scan when valid:

1. PREDICT: advance the estimate forward in time using the motion model
2. UPDATE: correct the estimate using measurement residual and tuned trust values

The code is intentionally implemented without general matrix math—everything is expanded into scalar expressions to remain PLC-friendly and maintainable.

Inputs / Outputs

Inputs

- `EnableIn`: gate execution and reset state cleanly when disabled
- `x`: measured position-like value
- `dt`: timestep in milliseconds (converted to seconds as `dt_s`)
- `q_x`, `q_x_dot`, `r_x`: tuning values treated as per-update variances
- Optional "Bleed" inputs: `BleedEnable`, `BleedThresh`, `BleedFactor`

Outputs

- `y`: filtered estimate of x
- `y_dot`: estimated velocity derived from the model and residual coupling

Execution Flow and PLC-Driven Guards

Disable behavior: deterministic pass-through
When disabled, the AOI does two important things:

1. Re-arms initialization
2. Outputs become deterministic pass-through

This is a deliberate PLC design choice: when the AOI is not enabled, it should not "coast" on stale internal state.

Timebase handling: `dt`

Rockwell scan timing is not guaranteed to be perfectly constant, so a gating condition is added for deterministic pass-through.
This yields two key safety outcomes:

- If `dt` is invalid (0, negative, uninitialized), the AOI does not compute gains or divide.
- Predict/update math only runs with a valid physical timestep.

One-time init: "start at measurement, assume stationary"

On first enable:

- x_pred := x;
- x_dot_pred := 0.0;

This means the observer starts with **no initial bias** in position, and assumes stationary motion until residual dynamics reveal otherwise.
Initial covariance:

- P00 = 1.0 (position uncertainty)
- P11 = 10.0 (velocity uncertainty)
- cross terms 0

This choice intentionally allows velocity to "move" more quickly early on (higher uncertainty) once covariance coupling emerges through the prediction step.

Model Used (Exactly What This Code Implements)

State and measurement assumptions
This AOI implements a **constant-velocity** discrete model with **position-only** measurement:
State:

$$\mathbf{x} = \begin{bmatrix} x \\ \dot{x} \end{bmatrix}$$

Measurement:

$$z = x + \text{noise} \Rightarrow H = [1 \quad 0]$$

This is why the residual uses only position:

- `y_res := x - x_pred`

No velocity sensor is required; velocity is inferred via position residual and covariance coupling.

Predict state: why this is one line in code

Prediction model:

$$\hat{x}_{k+1} = \hat{x}_k + dt \cdot \hat{v}_k, \hat{v}_{k+1} = \hat{v}_k$$

Code:

```
x_pred := x_pred + (dt_s * x_dot_pred);
```

Notice velocity is not modified during prediction. All changes to `x_dot_pred` happen in the update step when a measurement arrives.

Covariance Math: Why Those Exact Lines Exist

Storage mapping (row-major)

The AOI stores covariance as:

- P[0] = P00 var(x)
- P[1] = P01 cov(x,v)
- P[2] = P10 cov(v,x)
- P[3] = P11 var(v)

The code also uses `x_cov[0..3]` as scratch space for predicted covariance P'.

This is a PLC implementation tactic:

- Keep persistent P[] unchanged until update completes
- Use x_cov[] to compute S, K, and new P

Predict covariance: expanded $P' = FPF^T + Q$

For the constant-velocity model:

$$F = \begin{bmatrix} 1 & dt \\ 0 & 1 \end{bmatrix}$$

Expanding FPF^T yields:

$$\begin{aligned} P'_{00} &= P_{00} + dt(P_{01} + P_{10}) + dt^2 P_{11} \\ P'_{01} &= P_{01} + dtP_{11} \\ P'_{10} &= P_{10} + dtP_{11} \\ P'_{11} &= P_{11} \end{aligned}$$

That is implemented directly (arranged for fewer operations):

```
x_cov[0] := (P[0] + dt_s * P[2]) + dt_s * (P[1] + dt_s * P[3]); // P00'
x_cov[1] := (P[1] + dt_s * P[3]);                               // P01'
x_cov[2] := (P[2] + dt_s * P[3]);                               // P10'
x_cov[3] := (P[3]);                                             // P11'
```

Then the AOI adds process noise as per-update uncertainty growth:

```
x_cov[0] := x_cov[0] + q_x;
x_cov[3] := x_cov[3] + q_x_dot;
```

Important implementation note (PLC tuning reality):

`q_x` and `q_x_dot` are treated as additive variance increments per scan/update, not continuous-time spectral densities. This keeps tuning straightforward for commissioning.

Update Step: Residual, Innovation Variance, Gains

Residual (innovation)

```
y_res := x - x_pred;
```

This is $z - \hat{x}'$: measurement minus predicted position.

Innovation variance S: why it is only P00' + r_x

Because $H = [1 \quad 0]$, the predicted measurement variance is just P'_{00}. Therefore:

$$S = P'_{00} + r_x$$

Code:
```
S := x_cov[0] + r_x;
```
The guard:
```
IF S > 0.0 THEN
```
is the PLC safety requirement preventing gain blow-up and division instability.

Kalman gains: why K0 uses x_cov[0] and K1 uses x_cov[2]
For position-only measurement:

$$K = \frac{1}{S}\begin{bmatrix} P'_{00} \\ P'_{10} \end{bmatrix}$$

Code:
```
K0 := x_cov[0] / S; // position gain
K1 := x_cov[2] / S; // velocity gain
```
Key insight tied to the code:
Even though velocity is not measured, K1 is nonzero because P'10 becomes nonzero after prediction. This is the mechanism by which position residual corrects velocity.

Correct state
```
x_pred := x_pred + (K0 * y_res);
x_dot_pred := x_dot_pred + (K1 * y_res);
```
This is the concrete "position pulls velocity" behavior. If position consistently lags, residual drives velocity up.

Optional Bleed: Engineering Heuristic, Not Kalman
This block:
```
IF BleedEnable THEN
    IF ABS(x - x_pred) < BleedThresh THEN
        x_dot_pred := x_dot_pred * BleedFactor;
    END_IF;
END_IF;
```
exists to address a constant-velocity artifact:

- During a stop, the predictor continues to advance position using stale velocity.
- Measurement correction can inject a velocity correction that rings or overshoots at corners.

Bleed is a PLC-friendly, explainable mitigation:

- Only active near-stationary (threshold)
- Multiplies velocity by a decay factor (<1)

Tradeoff: It introduces bias toward zero velocity and may slow re-acceleration response if too aggressive.

Covariance Update: PLC-Optimized Form and Symmetry Enforcement
Why this update form was chosen
The AOI uses:

$$P = (I - KH)P'$$

because it is computationally light and easy to implement in scalar code.
```
P[0] := (1.0 - K0) * x_cov[0];
P[1] := (1.0 - K0) * x_cov[1];
P[2] := x_cov[2] - (K1 * x_cov[0]);
P[3] := x_cov[3] - (K1 * x_cov[1]);
```
Why enforce symmetry (P10 == P01)?
```
P[2] := P[1];
```
This is a numeric drift guard. In long-running PLC runtime, small floating-point differences can accumulate and cause asymmetry, which can distort gain behavior.
Why not Joseph form?
Joseph form is more numerically robust, but it requires more operations and intermediate terms—less PLC-friendly and harder to maintain.
The chosen compromise is:

- use lightweight update
- enforce symmetry

Tuning Interpretation (As Used by This AOI)

This AOI treats tuning as per-update variances:

- `r_x`: measurement variance (trust sensor less as it increases)
- `q_x`: position process variance (allow position drift between updates)
- `q_x_dot`: velocity process variance (allow velocity to change faster)

Practical guidance aligned to code:

- Increasing `r_x` reduces K0 and K1 (since S increases), making output smoother but laggier.
- Increasing `q_x_dot` increases P'11 which increases coupling over time, making K1 more responsive to residuals (faster velocity adaptation).

PLC-specific note: because q_* is per-update, changes in scan time can subtly change behavior. dt-scaling is a valid future enhancement but was intentionally excluded for commissioning simplicity.

Test and Validation Checklist (Code-Oriented)

Functional

- `EnableIn=FALSE` → `init=0`, `y=x`, `y_dot=0`
- Enable transition → state initializes once, no persistent stale state
- `dt` <= 0 → pass-through path, no divide by zero

Signal Behavior

- Ramp input: `y_dot` converges to ramp slope
- Step input: y smooths; `y_dot` spikes then settles
- Stop event: check overshoot; compare Bleed disabled vs enabled

Numeric Sanity

- S never ≤ 0 during normal tuning
- P[0], P[3] remain non-negative (strong indicator tuning is sane)
- Gains remain within reasonable bounds (e.g., `0 ≤ K0 ≤ 1` typically)

Findings and Recommendations (Implementation-Specific)

Findings

- The AOI correctly implements a 2-state discrete Kalman filter for a constant-velocity model with position-only measurement.
- Covariance propagation is correctly expanded in scalar form (`x_cov[]`), matching $FPF^T + Q$.
- `K1 := x_cov[2]/S` is the critical mechanism enabling velocity estimation without velocity measurement.
- Numeric safeguards (`dt_s > 0, S > 0`, symmetry enforcement) are appropriate for long-running PLC execution.
- Bleed is a purposeful non-Kalman heuristic to reduce a common CV corner artifact.

Recommendations

1. Document tuning units on the AOI faceplate (variance per update in engineering units).
2. Consider optional dt-scaled Q mode (future): scale q_* by dt_s / dt_s^2 for scan-rate invariance.
3. Consider optional gain clamp or minimum `r_x` to guard against mis-tuning.

Appendix — Traceability (Code → Theory)

Code Section	Variables	Theory Mapping
Disable behavior	`init, y, y_dot`	deterministic AOI behavior
Timebase	`dt_s`	discrete update interval
Predict state	`x_pred, x_dot_pred`	CV model prediction
Predict covariance	`x_cov[0..3]`	$FPF^T + Q$expansion
Residual	`y_res`	$z - \hat{x}'$
Innovation variance	S	$P'_{00} + R$
Gains	`K0, K1`	$K = P'H^T/S$
State update	`x_pred, x_dot_pred`	$\hat{x} = \hat{x}' + Ky$
Covariance update	`P[]`	$(I - KH)P'$
Symmetry guard	`P[2]:=P[1]`	numerical drift mitigation
Bleed	`Bleed*`	heuristic damping near stationary

Appendix – Code

```
// ===============================================================================
// 2-State Kalman Filter AOI (Position + Velocity Observer)
// Taylor Turner <resume.tturner@gmail.com>
//
// -------------------------------------------------------------------------------
// Summary
// -------------------------------------------------------------------------------
// - Discrete Kalman filter for a constant-velocity model (2-state).
// - Inputs: measured position-like signal `x`, timestep `dt` (ms), tuning.
// - Outputs: filtered estimate `y` (x̂) and estimated velocity `y_dot` (v̂).
//
// Model (constant velocity, position-only measurement)
//   State:      [ x_est ; x_dot_est ]
//   Predict:    x_est(k+1)     = x_est(k) + dt * x_dot_est(k)
//               x_dot_est(k+1) = x_dot_est(k)
//   Measure:    x_meas = x_est
//
// Timebase
//   - `dt` provided in milliseconds (e.g., Timer.PRE), converted to seconds.
//
// Covariance storage (row-major)
//   P[0]=P00 var(x), P[1]=P01 cov(x,v), P[2]=P10 cov(v,x), P[3]=P11 var(v)
//
// Tuning (per-update variances)
//   q_x     : process noise variance for x (position drift allowance)
//   q_x_dot : process noise variance for x_dot (velocity adaptability)
//   r_x     : measurement noise variance for x (trust in measurement)
//   Larger r_x => smoother y but more lag.
//
// Optional "Bleed"
//   - Decays velocity estimate when near-stationary to reduce corner overshoot.
// ===============================================================================

IF NOT EnableIn THEN
    // Disable => re-arm init. Pass-through outputs for deterministic behavior.
    init := 0;
    y := x;
    y_dot := 0.0;

ELSE
    // Convert ms -> seconds. dt_s must be > 0 to run predict/update.
    IF dt <= 0.0 THEN
        dt_s := 0.0;
    ELSE
        dt_s := dt / 1000.0;
    END_IF;

    // One-time initialization after enable
    IF NOT init THEN
        x_pred := x;        // start estimate at measurement
        x_dot_pred := 0.0;  // assume stationary initially
```

```
        // Initial covariance (tweak as needed)
        P[0] := 1.0;   // P00
        P[1] := 0.0;   // P01
        P[2] := 0.0;   // P10
        P[3] := 10.0;  // P11

        K0 := 0.0;
        K1 := 0.0;

        init := 1;
    END_IF;

    IF dt_s > 0.0 THEN

        // =====================
        // PREDICT (time update)
        // =====================
        // Predict state using constant-velocity model
        x_pred := x_pred + (dt_s * x_dot_pred);

        // Predict covariance: P' = F*P*F^T + Q
        // F = [[1, dt],
        //      [0, 1 ]]
        // Q is diagonal: diag(q_x, q_x_dot)
        x_cov[0] := (P[0] + dt_s * P[2]) + dt_s * (P[1] + dt_s * P[3]); // P00'
        x_cov[1] := (P[1] + dt_s * P[3]);                               // P01'
        x_cov[2] := (P[2] + dt_s * P[3]);                               // P10'
        x_cov[3] := (P[3]);                                             // P11'

        // Add process noise (uncertainty growth per update)
        x_cov[0] := x_cov[0] + q_x;
        x_cov[3] := x_cov[3] + q_x_dot;

        // =====================
        // UPDATE (measurement)
        // =====================
        // Innovation (residual): measurement minus prediction
        y_res := x - x_pred;

        // Innovation variance: expected residual variance
        // S = P00' + r_x (H = [1 0])
        S := x_cov[0] + r_x;

        IF S > 0.0 THEN
            // Kalman gains (how much to correct from residual)
            K0 := x_cov[0] / S; // position gain
            K1 := x_cov[2] / S; // velocity gain

            // Correct state
            x_pred := x_pred + (K0 * y_res);
            x_dot_pred := x_dot_pred + (K1 * y_res);

            // Optional: bleed velocity toward 0 when near-stationary
            IF BleedEnable THEN
                IF ABS(x - x_pred) < BleedThresh THEN
                    x_dot_pred := x_dot_pred * BleedFactor;
                END_IF;
            END_IF;

            // Covariance update: P = (I - K*H) * P'
            P[0] := (1.0 - K0) * x_cov[0];
            P[1] := (1.0 - K0) * x_cov[1];
            P[2] := x_cov[2] - (K1 * x_cov[0]);
            P[3] := x_cov[3] - (K1 * x_cov[1]);

            // Enforce symmetry: keep P10 == P01 (numerical drift guard)
            P[2] := P[1];
        END_IF;

        // Outputs
        y := x_pred;
        y_dot := x_dot_pred;
```

```
    ELSE
        // dt invalid -> pass-through
        y := x;
        y_dot := 0.0;
    END_IF;
END_IF;
```