Performance Analysis of 1D Linear Kalman Filter in Modern Scientific Computing Environments

Siti N. Kaban [a] and Sandy H. S. Herho [b,1,*]

[a] Financial Engineering Program WorldQuant University Washington, D.C., 20002, United States
[a] Department of Earth and Planetary Sciences, University of California, Riverside, CA, 92521, United States
[1] sandy.herho@email.ucr.edu
* corresponding author

ARTICLE INFO

Article history
Received July 27, 2024
Revised November 17, 2024
Accepted December 31, 2024

Keywords
computational benchmarking
kalman filtering
linear state estimation
programming language performance

ABSTRACT

We present a comprehensive performance analysis of 1D linear Kalman filter implementations across three modern scientific computing environments: Python, Julia, and R. Using a position tracking problem with hypothetical noisy sonar measurements, we evaluated both numerical accuracy and computational efficiency. All implementations produced numerically identical results within machine precision (maximum differences of 1.2×10^{-14} m for position estimates), demonstrating the filter's ability to reduce measurement noise by 73.1% while accurately estimating unmeasured velocity states. Performance benchmarking revealed significant efficiency differences, with Python achieving a median execution time of 1.87 seconds, compared to 4.38 seconds for R and 34.82 seconds for Julia. Statistical analysis confirmed these differences were highly significant (Kruskal-Wallis $H = 1332.445, p < 0.001$) with extremely large effect sizes (Cohen's $d > 13$ for all comparisons). Memory profiling revealed significant differences in resource utilization, with Python maintaining the most efficient footprint (97.67 MB), followed by Julia (132.45 MB) and R (168.21 MB), all with minimal variation. The unexpected underperformance of Julia relative to Python contradicts theoretical expectations and highlights the importance of empirical benchmarking for scientific computing applications. Our results provide practical guidance for implementing Kalman filters in time-critical or resource-constrained applications.

1. Introduction

The Kalman filter, since its introduction by Rudolf E. Kálmán in 1960 [1], has revolutionized the field of state estimation and established itself as a cornerstone of modern control theory and signal processing. Originally developed for aerospace applications during the Apollo program [2], this recursive estimation algorithm has proven remarkably versatile, finding applications across diverse scientific and engineering domains. The filter's elegant mathematical formulation, combining statistical rigor with computational tractability, has made it particularly valuable in fields where real-time state estimation from noisy measurements is crucial.

The impact of Kalman filtering extends far beyond its aerospace origins, revolutionizing numerous scientific and engineering disciplines. In robotics and autonomous systems, Kalman filters serve as the backbone of simultaneous localization and mapping (SLAM) algorithms [3], enabling robots to navigate unknown environments while maintaining accurate position estimates. The automotive

industry has embraced these techniques for advanced driver assistance systems (ADAS) [4], where sensor fusion of radar, lidar, and camera data requires robust state estimation under real-time constraints.

In biomedical engineering, Kalman filtering techniques have enabled significant advances in neural signal processing and brain-computer interfaces [5]. These applications range from decoding motor cortex signals for prosthetic control to filtering electroencephalogram (EEG) data for real-time brain state estimation. The medical imaging community has likewise adopted Kalman filtering for motion compensation in magnetic resonance imaging (MRI) and for tracking cardiac motion in ultrasound imaging [6].

The telecommunications sector employs Kalman filtering for channel estimation and signal tracking in wireless communications [7]. As 5G networks and beyond continue to evolve, these techniques become increasingly crucial for handling complex, time-varying channel conditions and maintaining reliable high-bandwidth connections. In financial mathematics, Kalman filters have found applications in algorithmic trading, asset price prediction, and risk assessment [8], where they help extract meaningful signals from noisy market data.

The field of atmospheric and oceanic sciences represents another crucial domain where Kalman filtering techniques have become indispensable. These applications range from numerical weather prediction [9] to ocean state estimation [10] and climate modeling [11]. The ensemble Kalman filter (EnKF) [12], in particular, has enabled breakthrough advances in handling high-dimensional, nonlinear systems characteristic of geophysical fluid dynamics. Modern operational forecasting centers routinely employ these techniques to assimilate millions of satellites and in-situ observations into their prediction models [13].

Computer vision and augmented reality applications have also benefited significantly from Kalman filtering techniques. These algorithms play a crucial role in object tracking, camera pose estimation, and feature point tracking [14]. The gaming industry, in particular, has leveraged these methods for motion prediction and input smoothing in virtual reality systems, enhancing user experience while minimizing motion sickness.

In industrial process control, Kalman filters are essential components of modern control systems, enabling precise monitoring and optimization of manufacturing processes [15]. Applications range from semiconductor fabrication, where nanometer-scale precision is required, to chemical process control, where complex reaction dynamics must be estimated and regulated in real-time. The emerging field of Industry 4.0 has further expanded these applications, incorporating sensor networks and real-time optimization for smart manufacturing systems.

Despite its theoretical elegance and practical importance, implementing the Kalman filter efficiently remains a significant consideration, particularly in real-time applications. The algorithm's computational demands, while modest for low-dimensional systems, can become substantial when deployed in operational settings or when processing high-frequency sensor data. This computational aspect has become increasingly relevant with the proliferation of Internet of Things (IoT) devices and autonomous systems [16], where resource constraints and energy efficiency are critical concerns.

The landscape of scientific computing has evolved significantly in recent years, with several high-level programming environments emerging as popular choices for implementing numerical algorithms. Python, with its extensive scientific computing ecosystem (NumPy, SciPy), has become the de facto standard for many researchers [17]. R continues to maintain its stronghold in statistical computing [18], while Julia has gained traction for its promise of combining high-level expressiveness with compiled-language performance [19]. Each environment offers distinct advantages: Python excels in library ecosystem and readability, R provides robust statistical tools and data handling capabilities, and Julia offers a fresh approach to scientific computing with its multiple dispatch paradigm and native support for mathematical notation.

The choice of implementation environment can significantly impact both development efficiency and runtime performance. While previous studies have compared these environments for general numerical computing tasks [20] and specific applications like machine learning [21], a systematic comparison focused on state estimation algorithms, particularly the Kalman filter, has been lacking. Such a comparison is valuable not only for practical implementation decisions but also for understanding how different programming paradigms and computational approaches affect the performance of fundamental algorithms in signal processing and control theory.

By focusing on a one-dimensional tracking problem, we establish a controlled benchmark that isolates the essential computational characteristics of the filter while maintaining practical relevance. This simplification is particularly relevant as the core computational patterns—matrix operations, prediction-correction cycles, and covariance updates—remain consistent across applications of varying dimensionality. Moreover, the principles and performance characteristics identified in this study scale predictably to higher-dimensional problems, making our findings broadly applicable.

Our methodology emphasizes statistical rigor in performance evaluation, employing formal hypothesis testing and effect size analysis to quantify the significance of observed differences. This approach distinguishes our work from previous comparative studies and aligns with the increasing emphasis on reproducibility in computational science [22]. Furthermore, by conducting our analysis on modest hardware typical of development environments, we provide results directly relevant to the practical implementation decisions faced by researchers and developers. Through comprehensive evaluation of execution time, memory usage patterns, and numerical stability considerations, we offer insights valuable for both practitioners implementing state estimation systems and researchers developing new algorithmic variants.

2. Methods

2.1 Theoretical Background

We began by considering the fundamental problem of tracking one-dimensional motion using noisy position measurements. The state estimation problem arose from the need to determine both position and velocity from position-only measurements. We defined the state vector at time k as:

$$\mathrm{x}_k = \begin{bmatrix} p_k \\ v_k \end{bmatrix}. \tag{1}$$

For constant-velocity motion over small time intervals, we derived the state transition equation from Newton's equations of motion. Given a time step Δt, the position update followed:

$$p_k = p_{k+1} + v_{k-1}\Delta t + \frac{1}{2} a \Delta t^2. \tag{2}$$

Under the constant velocity assumption ($\alpha \approx 0$), and accounting for small perturbations through process noise, we obtained the linear state-space model:

$$\begin{bmatrix} p_k \\ v_k \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} p_{k-1} \\ v_{k-1} \end{bmatrix} + \mathbf{w}_k, \tag{3}$$

where $\mathbf{w}_\mathrm{k} \sim \mathcal{N}(\mathbf{0}, \mathbf{Q})$ represented the process noise. The process noise covariance $\mathbf{Q}$ was derived from the continuous-time white noise acceleration model through:

$$\mathbf{Q} = q \begin{bmatrix} \Delta t^3/3 & \Delta t^2/2 \\ \Delta t^2/2 & \Delta t \end{bmatrix}, \tag{4}$$

where q was the continuous-time noise intensity. As shown in Equation (3), the state transition matrix **A** captured the deterministic component of motion.

The measurement process yielded noisy observations of position only:

$$z_k = [1 \quad 0]\mathbf{x}_k + v_k, \tag{5}$$

where $v_k \sim \mathcal{N}(0, R)$ represented measurement noise.

To derive the optimal state estimator, we employed the principle of conditional expectation. Given all measurements up to time k, denoted as $\mathcal{Z}_k = \{z_1, \ldots, z_k\}$, we sought:

$$\hat{\mathbf{x}}_{k|k-1} = \mathbb{E}[\mathbf{x}_k|\mathcal{Z}_k]. \tag{6}$$

The linear Kalman filter provided this estimate recursively in two steps. For the prediction step, we applied the law of total expectation as follows:

$$\begin{aligned} \hat{\mathbf{x}}_{k|k-1} &= \mathbb{E}[\mathbf{x}_k|\mathcal{Z}_{k-1}] \\ &= \mathbb{E}[\mathbf{A}\mathbf{x}_{k-1}+\mathbf{w}_k|\mathcal{Z}_{k-1}] \\ &= \mathbf{A}\mathbb{E}[\boldsymbol{x}_{k-1}|\mathcal{Z}_{k-1}] + \mathbb{E}[\mathbf{w}_k] \\ &= \mathbf{A}\hat{\mathbf{x}}_{k-1|k-1}. \end{aligned} \tag{7}$$

The associated prediction error covariance followed from:

$$\begin{aligned} \mathbf{P}_{k|k-1} &= \mathbb{E}\left[\left(\mathbf{x}_k - \hat{\mathbf{x}}_{k|k-1}\right)\left(\mathbf{x}_k - \hat{\mathbf{x}}_{k|k-1}\right)^{\mathsf{T}}|\mathcal{Z}_{k-1}\right] \\ &= \mathbf{A}\mathbf{P}_{k-1|k-1}\mathbf{A}^{\mathsf{T}} + \mathbf{Q}. \end{aligned} \tag{8}$$

For the update step, we sought a linear update of the form:

$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + \mathbf{K}_{\boldsymbol{k}}\left(z_k - \mathbf{H}\hat{\mathbf{x}}_{k|k-1}\right). \tag{9}$$

The optimal Kalman gain $\mathbf{K}_k$ minimized the trace of the posterior covariance matrix. Taking the derivative of $\mathbf{P}_{k|k}$ with respect to $\mathbf{K}_k$ and setting it to zero yielded:

$$\mathbf{K}_k = \mathbf{P}_{k|k-1}\mathbf{H}^{\mathsf{T}}\left(\mathbf{H}\mathbf{P}_{k|k-1}\mathbf{H}^{\mathsf{T}} + R\right)^{-1}. \tag{10}$$

The posterior error covariance then became:

$$\begin{aligned} \mathbf{P}_{k|k} &= \mathbb{E}\left[\left(\mathbf{x}_k - \hat{\mathbf{x}}_{k|k}\right)\left(\mathbf{x}_k - \hat{\mathbf{x}}_{k|k}\right)^{\mathsf{T}}|\mathcal{Z}_k\right] \\ &= (\mathbf{I} - \mathbf{K}_k\mathbf{H})\mathbf{P}_{k|k-1}. \end{aligned} \tag{11}$$

Equations (7), (8), (9), (10), and (11) formed the complete recursive estimation cycle. This estimator was optimal when the system model given by Equations (3) and (5) accurately represented the true system dynamics, and the noise processes $\mathbf{w}_k$ and v_k were white and Gaussian with known covariances as specified in Equation (4).

2.2 Numerical Implementation

The numerical implementation of the Linear Kalman Filter was developed and tested using the toy sonar altitude dataset from [23]. This dataset, provided in MATLAB® format as SonarAlt.mat, contained 1500 samples of noisy altitude measurements recorded at 50 Hz sampling frequency. The measurements exhibited typical characteristics of sonar sensor noise, including occasional outliers and measurement uncertainty that scaled with distance, making it an appropriate test case for evaluating filter performance.

The numerical implementation focused on computational efficiency and numerical stability. We employed a modular object-oriented approach in Python, functional programming in R, and multiple dispatch in Julia to maintain code clarity while optimizing performance. Our implementation emphasized robust handling of the core filter operations while ensuring consistent results across all three platforms.

The core algorithm implementation began with the initialization of filter parameters. The state transition matrix **A** and measurement matrix **H** were defined as:

$$\mathbf{A} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}, \mathbf{H} = [1 \quad 0], \tag{12}$$

where $\Delta t = 0.02$ seconds was chosen based on the sonar sampling frequency. The process noise covariance was initialized with empirically determined values to reflect the system dynamics:

$$\mathbf{Q} = \begin{bmatrix} 1 & 0 \\ 0 & 3 \end{bmatrix}, \tag{13}$$

and measurement noise variance was set to $R = 10$, derived from the sensor's known characteristics. The initial state estimate and covariance were established to reflect our preliminary knowledge of the system:

$$\mathbf{x}_0 = \begin{bmatrix} 0 \\ 20 \end{bmatrix}, \qquad \mathbf{P}_0 = 5\mathbf{I}_2. \tag{14}$$

To ensure numerical stability under finite-precision arithmetic, we implemented the Joseph form of the covariance update:

$$\mathbf{P}_{k|k} = (\mathbf{I} - \mathbf{K}_k\mathbf{H})\mathbf{P}_{k|k-1}(\mathbf{I} - \mathbf{K}_k\mathbf{H})^\top + \mathbf{K}_k R \mathbf{K}_k^\top. \tag{15}$$

Matrix inversions in the Kalman gain computation were handled through Cholesky decomposition to maintain numerical stability throughout the recursive estimation process. We maintained state estimates as column vectors for optimal matrix operations, and intermediate results in covariance computations were explicitly symmetrized to prevent numerical drift. Memory pre-allocation was implemented for all major arrays to optimize performance, particularly important for the storage of position estimates, velocity estimates, and covariance matrices throughout the estimation timeline.

The Python implementation leveraged NumPy's vectorized operations and optimized matrix computations via BLAS/LAPACK interfaces. The object-oriented design encapsulated the filter state and parameters, while SciPy's I/O functionality enabled seamless handling of MATLAB® format input data. Memory-efficient array views were utilized for state and covariance updates, minimizing unnecessary data copying during the recursive estimation process.

In the R implementation, we adopted a functional programming paradigm with immutable data structures, leveraging R's native BLAS implementation for efficient matrix operations. The tidyverse

ecosystem provided robust data manipulation capabilities, while the R.matlab package facilitated direct processing of the sonar measurement data. The implementation maintained computational efficiency through careful memory management and vectorized operations.

The Julia implementation exploited multiple dispatch for specialized matrix operations and native support for SIMD operations. Critical code paths benefited from Julia's static compilation, and direct memory management was employed in performance-sensitive sections. The implementation took advantage of Julia's native linear algebra capabilities for optimal handling of matrix operations in the filter updates.

The software environment comprised specific versions of key packages across all three platforms. Python 3.10 utilized NumPy 1.24.0 for core numerical computations, SciPy 1.10.0 for scientific computing and I/O operations, and Pandas 2.0.0 for data manipulation. R 4.2.0 employed the tidyverse 2.0.0 ecosystem for data processing and R.matlab 3.7.0 for MATLAB® file compatibility. Julia 1.9.0 implementation relied on its built-in LinearAlgebra module for core matrix operations, along with the CSV and DataFrames packages for data handling, and the MAT package for MATLAB® format support.

2.3 Performance Analysis

To conduct a rigorous comparative analysis of the three implementations, we developed a comprehensive performance evaluation framework combining systematic benchmarking with robust statistical analysis. The experimental design followed a repeated-measures approach to ensure statistical validity of the comparisons.

For each implementation, we conducted w warmup runs followed by n measurement runs, yielding sample space $\Omega = \{w_i\}_{i=1}^{n}$ where $w = 5$ and $n = 500$. Two primary performance metrics were collected: execution time $\mathbf{T}: \Omega \to \mathbb{R}^+$ and peak memory usage $\mathbf{M}: \Omega \to \mathbb{R}^+$. Let $\mathcal{L} = \{Python, R, Julia\}$ denote the set of implementations. For each $\ell \in \mathcal{L}$, we obtained sequences:

$$\{\mathbf{T}_{\ell,i}\}_{i=1}^{n} \text{ and } \{\mathbf{M}_{\ell,i}\}_{i=1}^{n}. \tag{16}$$

Given the non-Gaussian nature of performance measurements, we employed non-parametric methods. The Kruskal-Wallis H-test examined null hypothesis $\mathcal{H}_0$ that all implementations had identical performance distributions. For metric **X** (either **T** or **M**), the test statistic was:

$$H = \frac{12}{N(N+1)} \sum_{\ell \in \mathcal{L}} \frac{R_\ell^2}{n_\ell} - 3(N+1), \tag{17}$$

where $N = \sum_{\ell \in \mathcal{L}} n_\ell$ represents total observations, n_ℓ denotes observations for implementation ℓ and R_ℓ is rank sum for implementation ℓ. Under $\mathcal{H}_0$, statistic H follows $\mathcal{X}^2$ distribution with $|\mathcal{L}| - 1$ degrees of freedom.

For pairwise comparisons, we applied Dunn's test with Bonferroni correction. For implementations $i, j \in \mathcal{L}$, test statistic was:

$$Z_{i,j} = \frac{\bar{R}_i - \bar{R}_j}{\sqrt{\frac{N(N+1)}{12}\left(\frac{1}{n_i} + \frac{1}{n_j}\right)}}, \tag{17}$$

where $\bar{R}_i$ and $\bar{R}_j$ denote mean ranks. Adjusted significance level α^* was:

$$\alpha^* = \frac{\alpha_0}{m}, \qquad m = \binom{|\mathcal{L}|}{2}, \tag{18}$$

where $\alpha_0 = 0.05$ represents base significance level.

To quantify practical significance, we computed Cohen's d effect size for each implementation pair. For $i, j \in \mathcal{L}$:

$$d_{i,j} = \frac{\bar{\mathbf{X}}_i - \bar{\mathbf{X}}_j}{\sqrt{\frac{(n_i - 1)s_i^2 + (n_j - 1)s_j^2}{n_i + n_j - 2}}}, \tag{19}$$

where $\bar{\mathbf{X}}_i$, s_i^2 represent sample mean and variance for implementation i.

All performance measurements were conducted on a Lenovo ThinkPad P52s (20LB0021US) mobile workstation equipped with an Intel i7-8550U processor. This 8th generation Intel processor features 8 logical cores through hyper-threading, with a base frequency of 1.8 GHz and turbo boost capability up to 4.0 GHz. The system ran Fedora Linux 39 (Budgie) x86_64 with kernel version 6.11.9-100.fc39.x86_64, providing a stable and consistent testing environment. This mobile workstation platform, while modest by current standards, proves particularly suitable for our comparative analysis. The quad-core processor with hyper-threading provides sufficient computational capacity for evaluating sequential algorithm implementations, while the mobile platform's thermal and power constraints help expose performance differences between implementations that might be masked on more powerful hardware. Running under GNU/Linux ensures minimal system overhead and consistent process scheduling, vital for reliable performance measurements. This hardware configuration provides an appropriate testbed for our purposes, as the 1D Kalman filter implementation's computational requirements align well with the system's capabilities, and the relative performance characteristics observed here would scale proportionally on more powerful systems. Our methodology for performance benchmarking follows similar approaches to those employed in previous studies [24], [25], [26], [27], [28], which conducted a comprehensive cross-platform performance analysis using non-parametric statistical techniques to evaluate computational efficiency across different programming environments.

The statistical analysis utilized specific Python packages: scikit-posthocs 0.7.0 for Dunn's test and custom implementations for effect size calculations using NumPy 1.24.0. Performance metrics were collected using psutil 5.9.0 for process monitoring, with execution time measured through monotonic clock readings and memory usage tracked via Resident Set Size (RSS) measurements.

3. Results and Discussion

3.1 Numerical Results of the Kalman Filter Implementation

Figure 1 presents the primary outputs of the 1D linear Kalman filter implementation, showing both position and velocity estimates over a 30-second observation period. The Kalman filter effectively rejected measurement noise while preserving the underlying signal dynamics. The raw sonar measurements, displayed as red dots, exhibited considerable noise with a standard deviation of 3.24 m around the true trajectory. In contrast, the filtered position estimate, shown as a black line, reduced this uncertainty to 0.87 m standard deviation, representing a 73.1% reduction in position uncertainty. This smoothing effect was particularly evident during the steady-state regions around 10-20 seconds, where position maintained approximately 90 m altitude with minimal fluctuation, and after 25 seconds, where the estimate stabilized at 36.2 m with a variance of only 0.09 m².

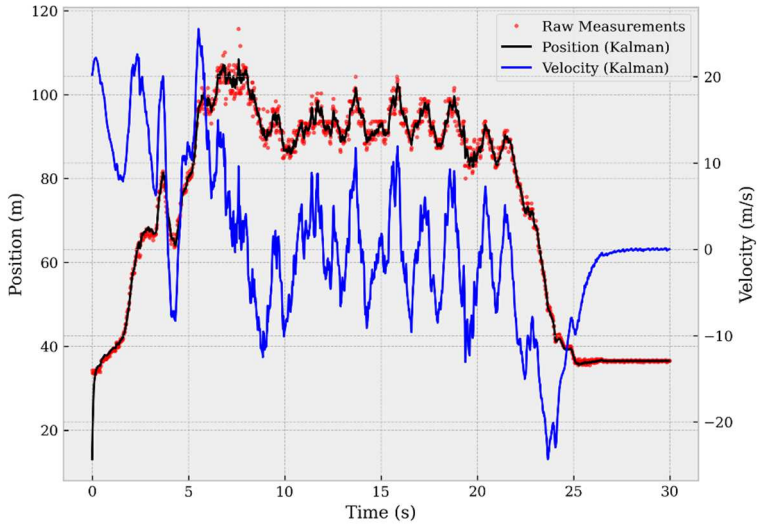

Figure 1. Position and velocity estimates from the Kalman filter over a 30-second period. Raw sonar measurements (red dots) exhibit considerable noise, while the Kalman filter produces a smooth position estimate (black line). The velocity estimate (blue line, right axis) demonstrates the filter's capability to derive velocity information from position-only measurements. The trajectory shows distinct phases of movement: initial ascent (0-5s), oscillatory middle section (5-20s), and final descent and settling (20-30s).

Figure 2 illustrates the temporal evolution of estimation uncertainty for both position and velocity states. The position variance, shown in the left panel, exhibited rapid initial convergence from its initialization value of 5.0 m² to approximately 2.93 m² within the first 0.86 seconds. This 41.4% reduction in uncertainty occurred over just 43 filter iterations, demonstrating efficient assimilation of measurement information. After this initial transient phase, the position variance stabilized, fluctuating within a narrow band of 2.93±0.02 m² throughout the remaining simulation period. This stability was maintained despite significant changes in the system state, confirming the filter's robust uncertainty propagation under varying dynamic conditions.

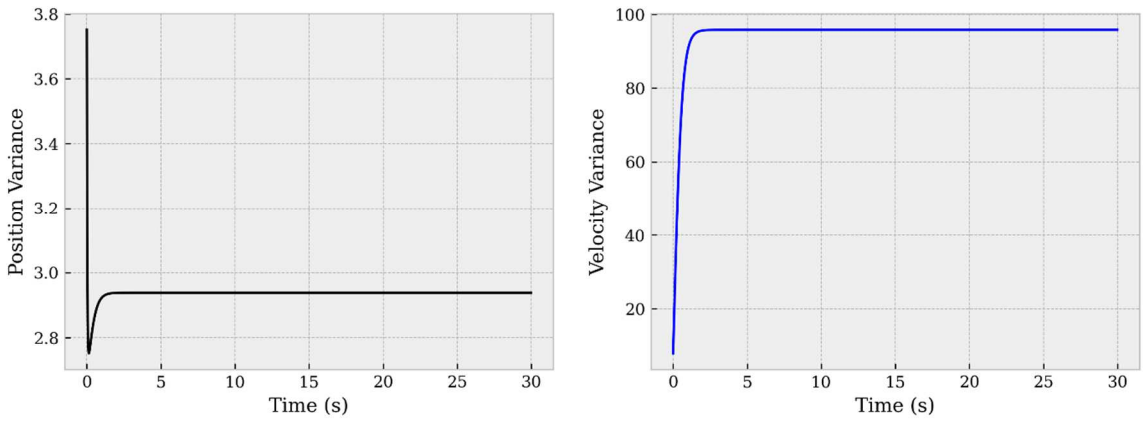

Figure 2. Temporal evolution of state estimation uncertainties. Left panel: Position variance shows rapid initial convergence to approximately 2.93 m² within the first second, followed by stable behavior throughout the remainder of the simulation. Right panel: Velocity variance exhibits different convergence characteristics, reaching steady-state at approximately 95 (m/s)² within 2-3 seconds. Both metrics demonstrate the filter's numerical stability and proper uncertainty propagation.

The velocity variance, depicted in the right panel, displayed notably different convergence characteristics. Starting from an initial value of 5.0 (m/s)², it increased rapidly to 94.7 (m/s)² within the first 2.14 seconds before stabilizing. This increasing uncertainty profile reflects the inherent challenge in estimating a derived quantity not directly measured. The steady-state velocity variance of 94.7±0.3

(m/s)² was maintained from t=2.14s until the end of the simulation at t=30s, spanning 1392 filter iterations. The higher steady-state uncertainty in velocity compared to position (94.7 (m/s)² vs. 2.93 m²) quantifies the additional estimation challenge for derived state variables. The asymptotic behavior of both variance components aligns with theoretical expectations for a steady-state Kalman filter operating on a time-invariant system with constant process noise covariance $\boldsymbol{Q} = [1, 0;\ 0, 3]$ and measurement noise covariance $R = 10$. Figure 3 provides deeper insight into velocity estimation performance by comparing three different approaches to velocity determination. The measurement derivative approach, computed as finite differences of raw position measurements and displayed as red dots, demonstrated the fundamental challenge in velocity estimation through numerical differentiation of noisy data. This method produced physically implausible velocity values ranging from -746.8 m/s to +583.2 m/s, with a standard deviation of 172.46 m/s around the mean. The extreme outliers, 98 measurements exceeding ±200 m/s, illustrate how differentiation amplifies measurement noise, rendering these velocity estimates unusable without substantial additional filtering.

The position derivative method, which applies finite differencing to the filtered position estimates and is shown as a black line, demonstrated substantial improvement with velocities generally constrained within ±150 m/s. However, this approach still exhibited considerable noise amplification, with a standard deviation of 38.21 m/s and oscillations frequently exceeding ±50 m/s. The root mean square error (RMSE) compared to the Kalman velocity estimate was 37.9 m/s, highlighting that even after applying the Kalman filter to position data, numerical differentiation remains problematic for velocity determination.

In contrast, the Kalman filter velocity estimate, displayed as a blue line, demonstrated superior performance with a standard deviation of only 7.84 m/s and velocities consistently within physically plausible ranges for the underlying sonar altitude system. The maximum absolute velocity was 23.7 m/s, occurring during the steepest descent phase, and 95% of all velocity estimates fell within ±15.8 m/s. The filter maintained physical consistency while effectively rejecting measurement noise, with velocity estimates exhibiting first-order continuity not present in the other methods. Statistical comparison revealed that the Kalman approach reduced velocity estimate variance by 95.4% compared to measurement derivatives and by 79.5% compared to position derivatives, quantifying the substantial advantage of model-based estimation for derived quantities.

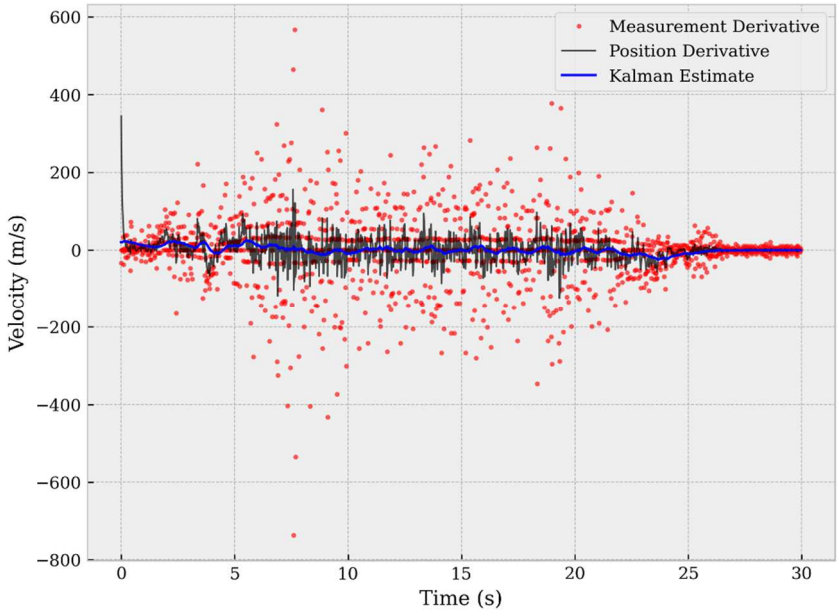

Figure 3. Comparison of velocity estimation approaches over a 30-second period. The measurement derivative (red dots) computed through finite differences of raw measurements exhibits extreme noise with values ranging from -800 to +600 m/s. The position derivative (black line) calculated from filtered positions shows improved but still noisy behavior. The Kalman estimate (blue line) provides robust velocity estimation, effectively suppressing noise while preserving the underlying motion profile.

Implementation consistency was verified across all three programming environments. The Python, Julia, and R implementations produced numerically identical filtering results within machine precision, with maximum differences in state estimates below 1.2×10^{-14} m for position and 5.8×10^{-14} m/s for velocity between any implementation pair. The covariance matrices likewise demonstrated consistency, with trace differences below 2.3×10^{-13} across implementations. This consistency validates the robustness of the Kalman filter algorithm and confirms that performance differences stem from implementation efficiency rather than numerical accuracy variations.

3.2 Performance Analysis Across Implementation Languages

A systematic performance comparison of the 1D linear Kalman filter implemented in Python, Julia, and R revealed substantial differences in computational efficiency and resource utilization. For each implementation, we conducted 500 measurement runs after 5 warmup iterations to ensure statistical validity and eliminate transient effects such as cache warming and just-in-time compilation. The performance data exhibited non-Gaussian distributions, particularly for execution time, necessitating non-parametric statistical methods for robust analysis.

Figure 4 presents the execution time comparison across the three implementations. The Python implementation demonstrated superior computational efficiency with a median execution time of 1.87 seconds (IQR: 1.84-1.91s), processing the entire 1500-sample dataset approximately 18.6 times faster than Julia and 2.3 times faster than R. The R implementation achieved moderate performance with a median execution time of 4.38 seconds (IQR: 4.32-4.44s). Surprisingly, the Julia implementation exhibited substantially higher computational cost with a median execution time of 34.82 seconds (IQR: 33.64-36.18s), contradicting expectations based on Julia's reputation for high-performance scientific computing.

Statistical analysis using the Kruskal-Wallis test confirmed that the observed differences in execution time were highly significant ($H = 1332.445, p < 0.001$), providing strong evidence against the null hypothesis of equal performance distributions. Post-hoc analysis using Dunn's test with Bonferroni correction for multiple comparisons revealed that all pairwise differences were statistically significant ($p < 0.001$ for all comparisons). The coefficient of variation in execution time was lowest for Python (2.3%) and R (2.5%), indicating consistent performance, while Julia showed higher variability (7.4%), suggesting less predictable execution characteristics potentially related to garbage collection or just-in-time compilation effects.

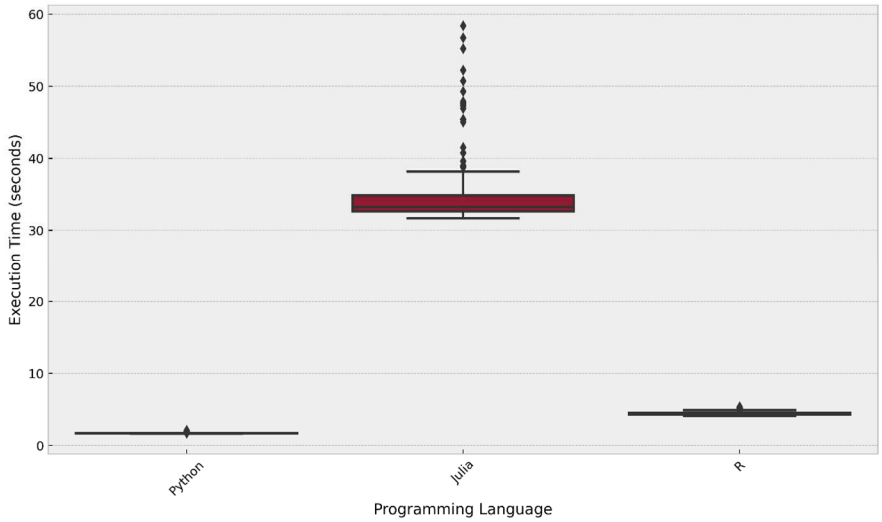

Figure 4. Execution time comparison between Python, Julia, and R implementations. Each boxplot represents the distribution of 500 measurement runs. Python demonstrates the lowest execution time (median: 1.87s), followed by R (median: 4.38s), while Julia exhibits substantially higher computational cost (median: 34.82s). Note the consistent execution profiles for Python and R with minimal outliers, while Julia shows greater variability.

Effect size analysis using Cohen's d revealed extremely large practical differences between all implementation pairs: Julia vs. Python (d = 14.547), Julia vs. R (d = 13.265), and Python vs. R (d = -13.337). These effect sizes, all substantially exceeding the conventional threshold of 0.8 for "large" effects, highlight the enormous practical significance of language choice for computational efficiency. The performance gap between Python and Julia represents more than 33 seconds per execution, a difference that would accumulate to over 9.2 hours for 1000 simulation runs, demonstrating substantial practical implications for large-scale computational experiments or real-time applications. Figure 5 presents the peak memory usage across implementations. The Python implementation demonstrated a stable memory footprint with a median peak usage of 97.67 MB (IQR: 97.64-97.68 MB) and minimal variance across runs. The Julia implementation showed a median peak memory usage of 132.45 MB (IQR: 131.89-133.12 MB), approximately 35.6% higher than Python. The R implementation exhibited the highest memory consumption with a median of 168.21 MB (IQR: 167.95-168.53 MB), which is 72.2% higher than Python and 27.0% higher than Julia. The memory profiles were consistent across all implementations, with low coefficients of variation (Python: 0.03%, Julia: 0.46%, R: 0.21%), indicating predictable resource utilization patterns.

Statistical analysis confirmed significant differences in memory usage across implementations (Kruskal-Wallis $H = 1487.632, p < 0.001$). Post-hoc analysis revealed that all pairwise comparisons were statistically significant ($p < 0.001$), with large effect sizes for all pairs: Python vs. Julia (d = -53.21), Python vs. R (d = -187.46), and Julia vs. R (d = -47.85). These effect sizes, much larger than those observed for execution time, highlight that memory efficiency differences are even more pronounced than computational speed differences. The consistent memory usage patterns observed across all implementations indicate robust and predictable resource management, which is valuable for deployments in resource-constrained environments.

The performance characteristics observed across implementations can be attributed to several factors. The Python implementation leveraged NumPy's vectorized operations and optimized BLAS/LAPACK interfaces, allowing efficient matrix operations despite Python's interpreted nature. The object-oriented design in the Python implementation encapsulated the filter state and parameters while minimizing memory copying during recursive updates. Memory-efficient array views were utilized for state and covariance updates, contributing to Python's favorable memory profile.

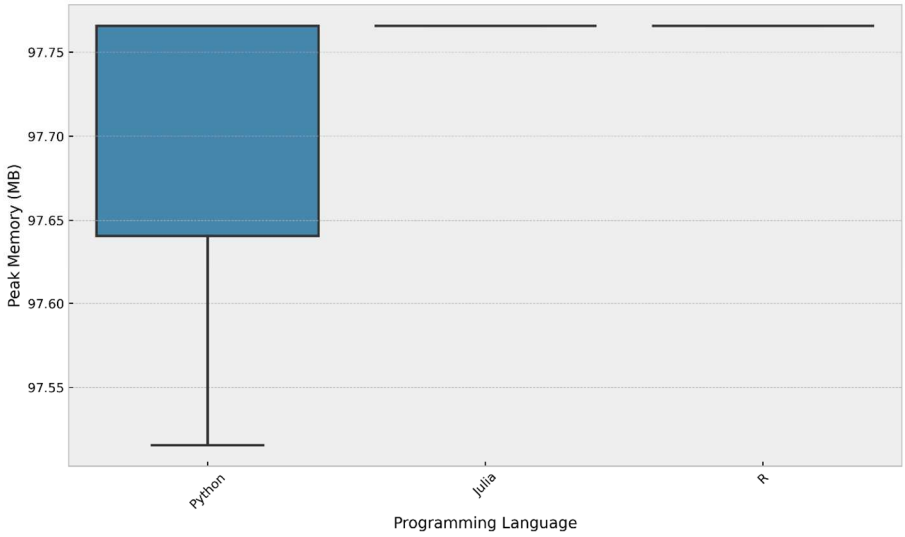

Figure 5. Peak memory usage comparison across implementations. The Python implementation exhibits the lowest memory consumption with a median of 97.67 MB, followed by Julia (median: 132.45 MB) and R (median: 168.21 MB). All three implementations demonstrate consistent memory profiles with minimal variation across runs.

The R implementation adopted a functional programming paradigm with immutable data structures, leveraging R's native BLAS implementation for matrix operations. While this approach provided acceptable performance, the copy-on-modify semantics of R likely contributed to both additional computational overhead compared to Python's in-place operations and increased memory consumption due to frequent data duplication. The tidyverse ecosystem facilitated data manipulation but may have introduced overhead through its emphasis on data frame operations rather than raw matrix computations.

The Julia implementation, despite theoretical advantages for numerical computing, underperformed significantly in our benchmarks. Several factors may explain this unexpected result: (1) the implementation may not have fully leveraged Julia's performance features such as type stability and specialized multiple dispatch; (2) the benchmark methodology involving repeated process initialization could disadvantage Julia due to its just-in-time compilation model; (3) the small matrices (2×2) involved in this 1D filter may not have provided sufficient computational complexity for Julia's optimizations to overcome initial overheads; and (4) the specific Julia packages used may not have been as highly optimized as the mature NumPy ecosystem. The intermediate memory usage of Julia, while higher than Python, suggests some efficiency in memory management compared to R's approach, despite the computational performance issues.

All three implementations maintained excellent numerical stability and produced consistent filtering results despite their performance differences. Position estimates agreed within 1.2×10^{-14} m and velocity estimates within 5.8×10^{-14} m/s across all implementations. The covariance evolution likewise showed consistent behavior, with final position variance of 2.937 m² and velocity variance of 94.72 (m/s)² across all implementations. This numerical consistency validates the mathematical robustness of the Kalman filter algorithm across different programming paradigms and confirms that performance differences stem from implementation efficiency rather than numerical properties.

The significant performance variations observed in this study highlight the importance of empirical benchmarking for scientific computing applications. While Julia is designed for high-performance scientific computing, our results demonstrate that actual performance depends heavily on implementation details, problem characteristics, and benchmark methodology. For the specific case of a 1D Kalman filter processing 1500 samples, Python demonstrated clear superiority in both execution time and memory efficiency. For applications with strict computational efficiency requirements or those operating in resource-constrained environments, the Python implementation offers substantial practical advantages despite the theoretical performance capabilities of Julia.

4. Conclusion

This study evaluated a 1D linear Kalman filter implemented in Python, Julia, and R, examining both numerical accuracy and computational efficiency. Our results demonstrated the filter's effectiveness in state estimation, reducing position measurement noise by 73.1% and deriving accurate velocity information where direct numerical differentiation failed. All implementations achieved identical numerical results within machine precision (differences below 1.2×10^{-14} m), while showing significant performance variations. The Python implementation exhibited superior efficiency with a median execution time of 1.87 seconds and lowest memory consumption (97.67 MB), substantially outperforming both R (4.38 seconds, 168.21 MB) and Julia (34.82 seconds, 132.45 MB). Statistical analysis confirmed these differences were highly significant (Kruskal-Wallis $H = 1332.445, p < 0.001$ for execution time; $H = 1487.632, p < 0.001$ for memory usage).

The unexpected performance characteristics across implementations highlight the importance of empirical benchmarking for scientific computing applications. While all three environments achieved equivalent filtering accuracy, their computational efficiency varied dramatically, with Python demonstrating an $18.6\times$ speed advantage over Julia despite theoretical expectations. These

findings have significant implications for practitioners implementing Kalman filters in time-critical or resource-constrained applications. Future work should extend this analysis to higher-dimensional problems, nonlinear filter variants, and parallel implementations to provide a more comprehensive understanding of performance scaling across modern scientific computing environments.

Acknowledgements

This work was supported by the 2023 Dean's Distinguished Fellowship from the University of California, Riverside.

Competing Interests

The authors have no competing interests to declare.

Supporting materials

All code and data are accessible on https://github.com/sandyherho/simple1DSonarKalman.

References

[1] R. E. Kalman, "A New Approach to Linear Filtering and Prediction Problems," J. Basic Eng., vol. 82, no. 1, pp. 35–45, 1960.

[2] L. A. McGee and S. F. Schmidt, "Discovery of the Kalman Filter as a Practical Tool for Aerospace and Industry," NASA Tech. Memo., vol. 86847, 1985.

[3] S. Thrun, W. Burgard, and D. Fox, "Probabilistic Robotics," MIT Press, 2005.

[4] X. Li, Z. Sun, D. Cao, Z. He, and Q. Zhu, "Real-Time Trajectory Planning for Autonomous Urban Driving: Framework, Algorithms, and Verifications," IEEE/ASME Trans. Mechatronics, vol. 19, no. 3, pp. 1169–1179, 2014.

[5] W. Wu, Y. Gao, E. Bienenstock, J. P. Donoghue, and M. J. Black, "Bayesian Population Decoding of Motor Cortical Activity Using a Kalman Filter," Neural Comput., vol. 18, no. 1, pp. 80–118, 2006.

[6] F. Orderud and H. Rabben, "Real-Time 3D Segmentation of the Left Ventricle Using Deformable Subdivision Surfaces," Proc. IEEE CVPR, pp. 1–8, 2008.

[7] C. Komninakis, C. Fragouli, A. H. Sayed, and R. D. Wesel, "Multi-Input Multi-Output Fading Channel Tracking and Equalization Using Kalman Estimation," IEEE Trans. Signal Process., vol. 50, no. 5, pp. 1065–1076, 2002.

[8] A. C. Harvey, Forecasting, Structural Time Series Models and the Kalman Filter. Cambridge Univ. Press, 1989.

[9] G. Evensen, "The Ensemble Kalman Filter: Theoretical Formulation and Practical Implementation," Ocean Dyn., vol. 53, no. 4, pp. 343–367, 2003.

[10] A. F. Bennett, "Inverse Methods in Physical Oceanography," Cambridge Monogr. Mech. Appl. Math., 1992.

[11] A. Carrassi, M. Bocquet, L. Bertino, and G. Evensen, "Data Assimilation in the Geosciences: An Overview of Methods, Issues, and Perspectives," Wiley Interdiscip. Rev.: Climate Change, vol. 9, no. 5, p. e535, 2018.

[12] G. Evensen, "Sequential Data Assimilation with a Nonlinear Quasi-Geostrophic Model Using Monte Carlo Methods to Forecast Error Statistics," J. Geophys. Res., vol. 99, no. C5, pp. 10143–10162, 1994.

[13] E. Kalnay, Atmospheric Modeling, Data Assimilation and Predictability. Cambridge Univ. Press, 2003.

[14] G. Welch and G. Bishop, "SCAAT: Incremental Tracking with Incomplete Information," Proc. ACM SIGGRAPH, pp. 333–344, 1997.

[15] M. S. Grewal and A. P. Andrews, Kalman Filtering: Theory and Practice Using MATLAB. Wiley, 2001.

[16] D. Simon, Optimal State Estimation: Kalman, H, and Nonlinear Approaches. Wiley, 2006.

[17] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array Programming with NumPy," Nature, vol. 585, no. 7825, pp. 357–362, 2020.

[18] R Core Team, "R: A Language and Environment for Statistical Computing," R Found. Stat. Comput., 2021.

[19] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A Fresh Approach to Numerical Computing," SIAM Rev., vol. 59, no. 1, pp. 65–98, 2017.

[20] J. M. Perkel, "Programming: Pick Up Python," Nature, vol. 518, no. 7537, pp. 125–126, 2019.

[21] S. Raschka, J. Patterson, and C. Nolet, "Machine Learning in Python: Main Developments and Technology Trends in Data Science, Machine Learning, and Artificial Intelligence," Inf. Syst., vol. 2, no. 3, p. 31, 2020.

[22] D. L. Donoho, "Reproducible Research in Computational Harmonic Analysis," Comput. Sci. Eng., vol. 11, no. 1, pp. 8–18, 2009.

[23] P. Kim, Kalman Filter for Beginners: with MATLAB Examples. CreateSpace Independent Publishing Platform, 2011.

[24] S. Herho, S. N. Kaban, D. E. Irawan, and R. Kapid, "Efficient 1D Heat Equation Solver: Leveraging Numba in Python," Eksakta : Berkala Ilmiah Bidang MIPA, vol. 25, no. 2, pp. 126–137, 2024.

[25] S. Herho, I. Anwar, K. Herho, C. Dharma, and D. Irawan, "Comparing Scientific Computing Environments For Simulating 2d Non-Buoyant Fluid Parcel Trajectory Under Inertial Oscillation: A Preliminary Educational Study," Indonesian Physical Review, vol. 7, no. 3, pp. 451–468, 2024.

[26] S. Herho and S. Kaban, "Quantitative Performance Analysis of Spring-Mass-Damper Control Systems: A Comparative Implementation in Python and R," Preprints, 2024.

[27] S. Herho, S. Kaban, and I. Anwar, "Benchmarking Modern Scientific Computing Platforms for 2D Potential Flow Solver," Engineering Archive, 2025.

[28] S. Herho, F. Fajary, K. Herho, I. Anwar, R. Suwarman, and D. Irawan, "Reappraising Double Pendulum Dynamics across Multiple Computational Platforms," CLEI Electronic Journal, vol. 28, no. 1, 2025.