Particle Flow Filter and Differentiable Particle Filter

Project Report: Part I

Joowon Lee
Department of Statistics, University of Wisconsin - Madison
ljw9510@gmail.com
https://github.com/ljw9510/2026MLCOE

December 31, 2025

Contents

Introduction

The fundamental goal of data assimilation is to provide a rigorous mathematical framework for estimating the state of a physical system by combining numerical model forecasts with empirical observations. At its core, this process is an exercise in Bayesian inference: we seek to update a prior *Probability Density Function* (pdf), representing our initial state estimate, with a likelihood function derived from new data to obtain a posterior pdf. In high-dimensional systems—characterized by chaotic dynamics and sparse, nonlinear observations—this update presents a formidable challenge that pushes the limits of classical and modern filtering theory.

A comprehensive theoretical survey on Bayesian filters. This report provides a comprehensive overview of both established and emerging Bayesian filters, bridging the gap between theoretical derivation and practical implementation. We examine the evolution of these methods, starting with classical Gaussian-based filters such as the *Kalman Filter* (KF), *Extended Kalman Filter* (EKF), and the *Unscented Kalman Filter* (UKF), which rely on linearization or deterministic sampling to approximate the posterior. We then transition to modern, fully nonlinear approaches, specifically focusing on the *Particle Flow Filter* (PFF). Our investigation of the PFF begins with analytic solutions derived under specific parametric assumptions, such as the *Exact Daum-Huang* (EDH) flow and its Localized (LEDH) variant, which assume the underlying distributions remain (locally) Gaussian. To move beyond these constraints, we investigate more flexible implementations, including particle-based versions and the *Kernel-Embedded Particle Flow Filter* (KPFF). The KPFF, by embedding the flow in a *Reproducing Kernel Hilbert Space* (RKHS), sidesteps rigid functional forms and avoids the weight degeneracy problem typical of standard particle filters by pushing particles toward the posterior with equal weights.

Experimental validation. A significant portion of this work is dedicated to the practical performance and limitations of these filters within a modern computational framework. In Chapter 2, we detail our experimental setup and implementation strategy. While the experiments in this report utilize geophysical benchmarks, the ultimate goal of this research is to adapt these robust filtering techniques to the financial domain. The ability to handle high-dimensional, non-Gaussian, and rapidly changing data is critical in quantitative finance. For this purpose, all filters are implemented using the TensorFlow ecosystem, ensuring the code is "GPU-ready" for large-scale deployment while currently validated on local hardware. We evaluate the filters using a 1,000-dimensional Lorenz 96 model, testing their ability to handle the dimensionality and sparsity typical of atmospheric applications. Also, we investigate the practical limitations of Gaussian approximations when faced with nonlinear observation operators and demonstrate where methods like the PFF excel in capturing multi-modal likelihoods. For the PFF implementation, we explore the use of matrix-valued kernels to prevent marginal collapse in unobserved subspaces, as well as the impact of prior assumptions on filter stability. By stress-testing these filters in a modern computational environment, we aim to provide a clear perspective on which filtering strategies are best suited for the complex, large-scale deployment required in modern financial systems.

Potential financial applications. The transition from geophysical benchmarks to financial applications represents a move toward a domain where the underlying hidden states are not physical coordinates, but rather latent economic drivers, fair values, and risk sensitivities. In the context of modern banking and large-scale asset management, the Bayesian filtering framework provides a rigorous foundation for navigating the high-dimensional and non-stationary nature of global markets. While classical Kalman-based approaches remain popular due to

their speed, the complexity of modern financial instruments necessitates the advanced, non-linear capabilities of the PFF.

One of the most immediate applications lies in *dynamic portfolio management* and the estimation of *intrinsic asset prices*. In this setting, the state-space model treats the "true" value of a portfolio as a latent variable that must be inferred from a stream of noisy, high-frequency transaction records. Unlike traditional filters that struggle with the non-linear relationship between derivative payoffs and their underlying assets, the PFF can push particles toward the true posterior without assuming local linearity. This allows for a more accurate estimation of fair value during periods of market dislocation, where the bid-ask bounce and sudden liquidity droughts would otherwise render Gaussian approximations useless. By maintaining a diverse ensemble of potential price scenarios, the filter enables more robust rebalancing strategies that are less sensitive to the estimation errors that plague static optimization models.

Furthermore, the integration of these filters into *risk management systems* offers a significant advancement over standard *Value-at-Risk* (VaR) methodologies. Traditional risk models often fail during market crashes because they cannot account for the multi-modal distributions and "fat tails" that characterize financial returns. The KPFF, in particular, is well-suited for tracking these heavy-tailed distributions in real-time. Because it avoids the sample impoverishment typical of standard *Sequential Monte Carlo* (SMC) methods, it can simultaneously track multiple "worst-case" scenarios or regime shifts, providing a more comprehensive view of systemic risk. This ability to capture multimodality is essential for calculating Expected Shortfall and other tail-risk metrics that banks must report for regulatory compliance.

Beyond price and risk, the Bayesian framework is indispensable for identifying *hidden market regimes* and tracking *unobservable stochastic volatility*. Market sentiment and institutional liquidity are rarely directly visible; they must be inferred from the volume and speed of trades. By treating these factors as hidden states, practitioners can use the PFF to detect the transition from a stable market to a high-volatility crisis mode nearly instantaneously. When implemented within a modern TensorFlow environment, these filters can process the immense volume of trade-level data generated by modern order books, bridging the gap between micro-scale microstructure dynamics and macro-scale portfolio shifts. This capability is particularly relevant for high-frequency execution algorithms, which rely on the sequential update of "true price" estimates to minimize market impact and optimize trade timing in an increasingly automated financial landscape.

Chapter 1

In-Depth Literature Review

1.1 Overview

The development of Bayesian filtering has been driven by the dual requirements of handling increasing system nonlinearity and scaling to high-dimensional state spaces. This review categorizes the literature into four phases: classical Gaussian approximations, SMC, the emergence of PFFs, and the recent intersection with Deep Learning.

1.1.1 Classical and Approximate Gaussian Filters

The foundational solution to the linear filtering problem was established by Kalman [Kal60], exploiting the closure of Gaussian distributions under linear transformations. Its recursive filter provides the optimal minimum *Mean Squared Error* (MSE) estimator for linear dynamic systems subject to Gaussian noise. However, practical scenarios often fail to adhere to the standard assumptions of linearity and Gaussianity. Financial markets, for example, exhibit stochastic volatility and heavy-tailed distributions that the standard Kalman Filter fails to capture. To address these limitations, the community initially turned to linearization techniques. The EKF linearizes dynamics via first-order Taylor expansions [Sch66]. While computationally efficient, the EKF diverges when the local linearity assumption is violated. Julier and Uhlmann [JU97] proposed the UKF to mitigate linearization errors. By propagating a deterministic set of sigma-points through the true nonlinear functions, the UKF captures posterior moments accurate to the second order. However, both EKF and UKF are fundamentally limited by their parametric assumption. They approximate the posterior as a unimodal Gaussian, rendering them unsuitable for chaotic or highly non-Gaussian systems often found in geophysics and finance.

1.1.2 Sequential Monte Carlo

A paradigm shift occurred with the introduction of the *Bootstrap Particle Filter* (BPF) by Gordon et al. [GSS93]. By approximating the posterior density with a set of weighted samples (particles), these filters theoretically converge to the true posterior regardless of nonlinearity, as established by Doucet et al. [DDFG01]. Later, Doucet and Johansen [DJ11] provided the standard framework for SMC, emphasizing its flexibility in general SSMs.

Despite their versatility, standard PFs face significant computational challenges. As noted in the comprehensive tutorial by Doucet and Johansen [DJ11], PFs suffer from the "curse of dimensionality". In high-dimensional spaces, the importance weights often collapse onto a single particle, a phenomenon known as weight degeneracy. Furthermore, the resampling

step—necessary to prevent weight variance explosion—can lead to sample impoverishment and path degeneracy. Also, Snyder et al. [SBBA08] and Bengtsson et al. [BBL08] demonstrated that the number of particles required to maintain accuracy grows exponentially with the state dimension, limiting standard PFs to low-dimensional applications ($p < 20$) unless the likelihood is extremely diffuse.

1.1.3 Particle Flow Filters

To address degeneracy without resampling, Daum and Huang [DH08, DHN10] introduced the PFF. Rather than relying on discrete importance sampling and resampling, PFFs introduce a continuous-time transport mechanism. By defining a log-homotopy between the prior and posterior densities, the filter derives a velocity field governed by a fundamental PDE. This flow migrates particles smoothly across the probability landscape, effectively pushing the ensemble toward the high-likelihood regions of the measurement.

Within the PFF framework, two primary paradigms have emerged to solve the underlying PDE. The EDH flow provides an analytical solution under a global Gaussian assumption, resulting in an affine velocity field that is both computationally efficient and numerically stable. However, to handle more complex, non-Gaussian distributions, the LEDH flow was developed [DC12]. LEDH relaxes global constraints by applying the flow equations pointwise, utilizing local gradient and Hessian information to allow for nonlinear, particle-specific trajectories. This localized approach allows the filter to navigate multi-modal or skewed posterior structures that would be collapsed by a global affine transformation.

Despite these improvements, the raw flow approximation often suffers from statistical inconsistency due to discretization errors and the simplifying affine flow assumption. To remedy this, Li and Coates [LC17] proposed the *Particle Flow Particle Filter* (PF-PF) framework. This method treats the deterministic flow not as a final estimator, but as a proposal distribution within a standard *Importance Sampling scheme* (ISS). By correcting the particle weights using the Jacobian determinant of the flow map, the PF-PF ensures asymptotic convergence to the true posterior distribution.

Distinct from the parametric affine flows of EDH and LEDH, another class of methods derives the particle flow within an RKHS, utilizing kernel-based gradients to drive particle transport. However, in the context of high-dimensional geophysical systems with sparse observations, Hu and van Leeuwen [HVL21] identified a critical failure mode in these frameworks known as "marginal collapse." They demonstrated that standard scalar kernels typically used in RKHS formulations fail to maintain diversity in unobserved dimensions. To address this, they proposed the *Kernel-Embedded Particle Flow Filter* (KPFF) utilizing diagonal matrix-valued kernels. This formulation decouples the repulsive forces across dimensions, preventing the collapse of unobserved variables while maintaining accurate tracking of the observed subspaces.

Finally, regardless of the specific flow formulation (affine or kernel-based), numerical stability remains a persistent challenge. The differential equations governing particle flow often become numerically "stiff," particularly in scenarios with low measurement noise (high signal-to-noise ratio). In these regimes, the flow dictates rapid particle transport to match precise observations, leading to ill-conditioned equations that cause standard numerical integrators to diverge. Addressing this, Dai et al. [DD22] analyzed the condition number of the flow ODEs and proposed regularization techniques to ensure stable propagation in these near-deterministic limits.

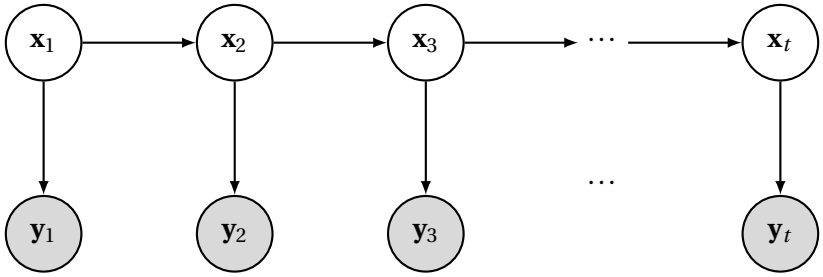
1.1.4 Differentiable and Learning-Based Filtering

The most recent frontier in Bayesian filtering integrates classical filtering structures into differentiable computational graphs, allowing for the optimization of filter parameters using modern deep learning techniques. Jonschkowski et al. [JRB18] and Karkus et al. [KHL18] pioneered the concept of *Differentiable Particle Filters* (DPF), which enables the end-to-end optimization of motion and observation models within a neural network architecture, bridging the gap between data-driven learning and probabilistic inference. A key challenge in this domain is the non-differentiability of the discrete resampling step. Corenflos et al. [CTDD21] addressed this by formulating resampling as an entropy-regularized optimal transport problem.

Building on these differentiable frameworks, current research is shifting toward learning the particle transport map directly from data, bypassing the need for analytic homotopies or explicit PDE solutions. For instance, Chaudhari et al. [CPM25] proposed GradNetOT to approximate optimal transport maps using gradient-based neural networks. In a parallel development within scientific computing, Jha et al. [Jha25] explored the use of Neural Operators to solve the high-dimensional PDEs underlying the flow, paving the way for fully data-driven, mesh-free particle flow filters.

1.2 State Space Models and Bayesian Filtering

The objective of Bayesian filtering is to estimate the hidden state of a stochastic process, $\mathbf{x}_t$, based on a sequence of noisy measurements $\mathbf{y}_{1:t} = (\mathbf{y}_1, \ldots, \mathbf{y}_t)$, where we define $\mathbf{x}_{1:t} := (\mathbf{x}_1, \ldots, \mathbf{x}_t)$. The underlying *State Space Model* (SSM) assumes that the process $(\mathbf{x}_t)_{t\geq 1}$ of hidden states forms a Markov chain and the measurement $\mathbf{y}_t$ depends only on the time-t hidden state $\mathbf{x}_t$.



More precisely, the model is defined by the following two components:

1. **Transition model**: A Markov chain describing the evolution of the hidden states:

$$\mathbf{x}_t \sim p(\mathbf{x}_t \mid \mathbf{x}_{1:t-1}) = p(\mathbf{x}_t \mid \mathbf{x}_{t-1}).$$

2. **Measurement model**: A model describing how observations relate to the state:

$$\mathbf{y}_t \sim p(\mathbf{y}_t \mid \mathbf{x}_{1:t}) = p(\mathbf{y}_t \mid \mathbf{x}_t).$$

A concise way to write the resulting model is the following:

$$\begin{cases} \mathbf{x}_t = f_{t-1}(\mathbf{x}_{t-1}) + \xi_{t-1}, \\ \mathbf{y}_t = g_t(\mathbf{x}_t) + \zeta_t, \end{cases}$$

where f_{t-1} is a deterministic state transition map and g_t is a deterministic measurement map. The terms ξ_{t-1} and ζ_t represent independent noises, respectively.

Based on the SSM defined above, *Bayesian filtering* provides a general framework for recursively estimating the posterior distribution of the state $\mathbf{x}_t$ given the measurements $\mathbf{y}_{1:t}$. This process consists of two alternating steps: *Prediction* and *Update*.

1. **Prediction step**: Using the Chapman-Kolmogorov equation, we compute the prior distribution of the state at time t given measurements up to $t-1$:

$$p(\mathbf{x}_t \mid \mathbf{y}_{1:t-1}) = \int p(\mathbf{x}_t \mid \mathbf{x}_{t-1})\, p(\mathbf{x}_{t-1} \mid \mathbf{y}_{1:t-1})\, d\mathbf{x}_{t-1},$$

   where the integral is taken with respect to the Lebesgue measure on the ambient Euclidean space of the hidden states.

2. **Update step**: Upon receiving the new measurement $\mathbf{y}_t$, we update the prior via Bayes' rule to obtain the posterior:

$$p(\mathbf{x}_t \mid \mathbf{y}_{1:t}) = \frac{1}{Z_t}\, p(\mathbf{y}_t \mid \mathbf{x}_t)\, p(\mathbf{x}_t \mid \mathbf{y}_{1:t-1}),$$

   where the normalization constant (evidence) Z_t is given by:

$$Z_t = p(\mathbf{y}_t \mid \mathbf{y}_{1:t-1}) = \int p(\mathbf{y}_t \mid \mathbf{x}_t)\, p(\mathbf{x}_t \mid \mathbf{y}_{1:t-1})\, d\mathbf{x}_t.$$

Limitations. While the formulation above is theoretically sound, it is analytically intractable for most systems.

- **Closed-form solutions:** Exist only for specific cases, most notably Linear Gaussian systems (Kalman Filter) and finite state spaces (Hidden Markov Models).
- **Intractability:** For nonlinear or non-Gaussian models, the integrals in the prediction and normalization steps cannot be solved analytically.
- **High dimensions:** Numerical integration (quadrature) becomes infeasible as the state dimension increases (curse of dimensionality).

1.3 Linear-Gaussian SSM with Kalman Filter

The *Kalman Filter* provides the closed-form solution to the Bayesian filtering equations under the following two assumptions:

1. **Linearity:** The process and measurement models are linear functions of the state (i.e., the maps f and g are linear).
2. **Gaussianity:** The process noise (ξ_t) and measurement noise (ζ_t) are additive and follow Gaussian distributions.

Accordingly, the model is defined as:

$$\begin{cases} \mathbf{x}_t = \mathbf{A}_{t-1}\mathbf{x}_{t-1} + \xi_{t-1}, & \xi_{t-1} \sim \mathcal{N}(0, \mathbf{Q}_{t-1}), \\ \mathbf{y}_t = \mathbf{H}_t\mathbf{x}_t + \zeta_t, & \zeta_t \sim \mathcal{N}(0, \mathbf{R}_t). \end{cases} \tag{1.1}$$

where $\mathbf{A}_{t-1}$ is the state transition matrix and $\mathbf{H}_t$ is the measurement matrix. We can derive the explicit Kalman Filter equations by applying the general Bayesian recursion (prediction and update steps) to this linear Gaussian model.

Claim 1.1 (KF equations)**.** *Consider the linear-Gaussian model defined in* (1.1)*. The prediction step of Bayesian filtering is given by*

$$\text{(KF Prediction)} \quad p(\mathbf{x}_t \mid \mathbf{y}_{1:t-1}) \sim \mathcal{N}(\mathbf{m}_t^-, \mathbf{P}_t^-), \quad \text{where} \quad \begin{cases} \mathbf{m}_t^- \leftarrow \mathbf{A}_{t-1}\mathbf{m}_{t-1}, \\ \mathbf{P}_t^- \leftarrow \mathbf{A}_{t-1}\mathbf{P}_{t-1}\mathbf{A}_{t-1}^\top + \mathbf{Q}_{t-1}. \end{cases} \tag{1.2}$$

The update step of Bayesian filtering is given by

$$\text{(KF Update)} \quad p(\mathbf{x}_t \mid \mathbf{y}_{1:t}) \sim \mathcal{N}(\mathbf{m}_t, \mathbf{P}_t), \quad \text{where} \quad \begin{cases} \mathbf{S}_t \leftarrow \mathbf{H}_t\mathbf{P}_t^-\mathbf{H}_t^\top + \mathbf{R}_t, & (\triangleright \textit{ Innovation Covariance}) \\ \mathbf{K}_t \leftarrow \mathbf{P}_t^-\mathbf{H}_t^\top\mathbf{S}_t^{-1}, & (\triangleright \textit{ Kalman Gain}) \\ \mathbf{m}_t \leftarrow \mathbf{m}_t^- + \mathbf{K}_t(\mathbf{y}_t - \mathbf{H}_t\mathbf{m}_t^-), \\ \mathbf{P}_t \leftarrow \mathbf{P}_t^- - \mathbf{K}_t\mathbf{S}_t\mathbf{K}_t^\top. \end{cases}$$

In practice, numerical errors can cause the covariance matrix computed via the standard formula $\mathbf{P}_t = \mathbf{P}_t^- - \mathbf{K}_t\mathbf{S}_t\mathbf{K}_t^\top$ to lose symmetry or positive definiteness. The *Joseph stabilized form* provided in the claim below guarantees these properties.

Claim 1.2 (Joseph Stabilized Update)**.** *The covariance update can be written in the Joseph stabilized form, which guarantees symmetry and positive definiteness.*

$$\mathbf{P}_t = (\mathbf{I} - \mathbf{K}_t\mathbf{H}_t)\mathbf{P}_t^-(\mathbf{I} - \mathbf{K}_t\mathbf{H}_t)^\top + \mathbf{K}_t\mathbf{R}_t\mathbf{K}_t^\top. \tag{1.3}$$

Proof. Define the estimation error as $\mathbf{e}_t = \mathbf{x}_t - \mathbf{m}_t$. Recall that the updated mean is $\mathbf{m}_t = \mathbf{m}_t^- + \mathbf{K}_t(\mathbf{y}_t - \mathbf{H}_t\mathbf{m}_t^-)$. Substituting the measurement model $\mathbf{y}_t = \mathbf{H}_t\mathbf{x}_t + \zeta_t$, we get

$$\begin{aligned}\mathbf{e}_t &= \mathbf{x}_t - \left(\mathbf{m}_t^- + \mathbf{K}_t(\mathbf{H}_t\mathbf{x}_t + \zeta_t - \mathbf{H}_t\mathbf{m}_t^-)\right)\\ &= (\mathbf{x}_t - \mathbf{m}_t^-) - \mathbf{K}_t\mathbf{H}_t(\mathbf{x}_t - \mathbf{m}_t^-) - \mathbf{K}_t\zeta_t\\ &= (\mathbf{I} - \mathbf{K}_t\mathbf{H}_t)\mathbf{e}_t^- - \mathbf{K}_t\zeta_t,\end{aligned}$$

where $\mathbf{e}_t^- = \mathbf{x}_t - \mathbf{m}_t^-$ is the prediction error. The updated covariance is $\mathbf{P}_t = \mathbb{E}[\mathbf{e}_t\mathbf{e}_t^\top]$. Since the measurement noise ζ_t is independent of the prediction error $\mathbf{e}_t^-$ (which depends only on past history), the cross-terms vanish, and we get

$$\begin{aligned}\mathbf{P}_t &= \mathbb{E}\left[\left((\mathbf{I} - \mathbf{K}_t\mathbf{H}_t)\mathbf{e}_t^- - \mathbf{K}_t\zeta_t\right)\left((\mathbf{I} - \mathbf{K}_t\mathbf{H}_t)\mathbf{e}_t^- - \mathbf{K}_t\zeta_t\right)^\top\right]\\ &= (\mathbf{I} - \mathbf{K}_t\mathbf{H}_t)\mathbb{E}[\mathbf{e}_t^-(\mathbf{e}_t^-)^\top](\mathbf{I} - \mathbf{K}_t\mathbf{H}_t)^\top + \mathbf{K}_t\mathbb{E}[\zeta_t\zeta_t^\top]\mathbf{K}_t^\top\\ &= (\mathbf{I} - \mathbf{K}_t\mathbf{H}_t)\mathbf{P}_t^-(\mathbf{I} - \mathbf{K}_t\mathbf{H}_t)^\top + \mathbf{K}_t\mathbf{R}_t\mathbf{K}_t^\top.\end{aligned}$$

This form is valid for *any* gain $\mathbf{K}_t$, whereas the standard short form assumes $\mathbf{K}_t$ is the optimal Kalman gain. □

Below, we provide a derivation of the KF equations stated in Claim 1.1. We proceed by analyzing the *Prediction step* and the *Update step* sequentially.

Prediction step. Proceeding by induction, assume that the posterior at time $t-1$ is Gaussian, $p(\mathbf{x}_{t-1} \mid \mathbf{y}_{1:t-1}) = \mathcal{N}(\mathbf{x}_{t-1} \mid \mathbf{m}_{t-1}, \mathbf{P}_{t-1})$. The transition density is given by $p(\mathbf{x}_t \mid \mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t \mid \mathbf{A}_{t-1}\mathbf{x}_{t-1}, \mathbf{Q}_{t-1})$. The predictive distribution is obtained by integrating the product of these two Gaussian densities. Since the system is linear-Gaussian, the result is itself Gaussian, $p(\mathbf{x}_t \mid \mathbf{y}_{1:t-1}) = \mathcal{N}(\mathbf{x}_t \mid \mathbf{m}_t^-, \mathbf{P}_t^-)$. The parameters in (1.2) follow from the linearity of expectation and the independence of the noise ξ_{t-1} from the state $\mathbf{x}_{t-1}$ (conditioned on $\mathbf{y}_{1:t-1}$).

$$\begin{cases}\mathbf{m}_t^- = \mathbb{E}[\mathbf{A}_{t-1}\mathbf{x}_{t-1} + \xi_{t-1}] = \mathbf{A}_{t-1}\mathbf{m}_{t-1},\\ \mathbf{P}_t^- = \mathrm{Var}[\mathbf{A}_{t-1}\mathbf{x}_{t-1} + \xi_{t-1}] = \mathbf{A}_{t-1}\mathbf{P}_{t-1}\mathbf{A}_{t-1}^\top + \mathbf{Q}_{t-1}.\end{cases}$$

Update step. This step updates the prior $\mathcal{N}(\mathbf{x}_t \mid \mathbf{m}_t^-, \mathbf{P}_t^-)$ using the measurement likelihood $p(\mathbf{y}_t \mid \mathbf{x}_t) = \mathcal{N}(\mathbf{y}_t \mid \mathbf{H}_t\mathbf{x}_t, \mathbf{R}_t)$. By Bayes' rule, the product of these two Gaussian densities yields a Gaussian posterior: $p(\mathbf{x}_t \mid \mathbf{y}_{1:t}) = \mathcal{N}(\mathbf{x}_t \mid \mathbf{m}_t, \mathbf{P}_t)$.

We first note that the augmented vector $(\mathbf{x}_t, \mathbf{y}_t)$ is jointly Gaussian conditional on the history $\mathbf{y}_{1:t-1}$.

Claim 1.3. *The joint distribution of* $(\mathbf{x}_t, \mathbf{y}_t)$ *conditional on* $\mathbf{y}_{1:t-1}$ *is given by*

$$(\mathbf{x}_t, \mathbf{y}_t) \,\Big|\, \mathbf{y}_{1:t-1} \sim \mathcal{N}\left(\begin{pmatrix}\mathbf{m}_t^-\\ \mathbf{H}_t\mathbf{m}_t^-\end{pmatrix}, \begin{pmatrix}\mathbf{P}_t^- & \mathbf{P}_t^-\mathbf{H}_t^\top\\ \mathbf{H}_t\mathbf{P}_t^- & \mathbf{H}_t\mathbf{P}_t^-\mathbf{H}_t^\top + \mathbf{R}_t\end{pmatrix}\right).$$

Proof. From the prediction step, we know that the prior $p(\mathbf{x}_t \mid \mathbf{y}_{1:t-1})$ is Gaussian with mean $\mathbf{m}_t^-$ and covariance $\mathbf{P}_t^-$. Since the measurement equation $\mathbf{y}_t = \mathbf{H}_t\mathbf{x}_t + \zeta_t$ is a linear transformation of

$\mathbf{x}_t$ with additive Gaussian noise, the joint vector $(\mathbf{x}_t, \mathbf{y}_t)$ must also be Gaussian. Thus, it suffices to compute the conditional mean and covariance. Conditional on $\mathbf{y}_{1:t-1}$,

$$\begin{aligned}
\mathbb{E}[\mathbf{y}_t \mid \mathbf{y}_{1:t-1}] &= \mathbb{E}[\mathbf{H}_t\mathbf{x}_t + \zeta_t \mid \mathbf{y}_{1:t-1}] = \mathbf{H}_t\mathbf{m}_t^-, \\
\mathrm{Var}(\mathbf{y}_t \mid \mathbf{y}_{1:t-1}) &= \mathrm{Var}(\mathbf{H}_t\mathbf{x}_t + \zeta_t \mid \mathbf{y}_{1:t-1}) = \mathbf{H}_t\mathbf{P}_t^-\mathbf{H}_t^\top + \mathbf{R}_t, \\
\mathrm{Cov}(\mathbf{x}_t, \mathbf{y}_t \mid \mathbf{y}_{1:t-1}) &= \mathrm{Cov}(\mathbf{x}_t, \mathbf{H}_t\mathbf{x}_t + \zeta_t \mid \mathbf{y}_{1:t-1}) = \mathbf{P}_t^-\mathbf{H}_t^\top.
\end{aligned}$$

Note that we used the independence of the measurement noise ζ_t from the state $\mathbf{x}_t$ and the history $\mathbf{y}_{1:t-1}$. This completes the proof. □

Next, to perform the update step, we need to compute the posterior distribution $p(\mathbf{x}_t \mid \mathbf{y}_{1:t})$. Since we have established that $(\mathbf{x}_t, \mathbf{y}_t)$ is jointly Gaussian conditional on $\mathbf{y}_{1:t-1}$, the following standard result regarding conditional Gaussian distributions suffices.

Lemma 1.4 (Conditional of Joint Gaussian)**.** *Suppose two random variables* $\mathbf{x}$ *and* $\mathbf{y}$ *have a joint Gaussian distribution:*

$$\begin{pmatrix}\mathbf{x}\\ \mathbf{y}\end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix}\mathbf{a}\\ \mathbf{b}\end{pmatrix}, \begin{pmatrix}\mathbf{A} & \mathbf{C}\\ \mathbf{C}^\top & \mathbf{B}\end{pmatrix}\right),$$

then the conditional distribution of $\mathbf{x}$ *given* $\mathbf{y}$ *is:*

$$\mathbf{x} \mid \mathbf{y} \sim \mathcal{N}(\mathbf{a} + \mathbf{C}\mathbf{B}^{-1}(\mathbf{y} - \mathbf{b}), \mathbf{A} - \mathbf{C}\mathbf{B}^{-1}\mathbf{C}^\top).$$

Applying Lemma 1.4 with the parameters $\mathbf{a} = \mathbf{m}_t^-$, $\mathbf{b} = \mathbf{H}_t\mathbf{m}_t^-$, $\mathbf{A} = \mathbf{P}_t^-$, $\mathbf{C} = \mathbf{P}_t^-\mathbf{H}_t^\top$, and $\mathbf{B} = \mathbf{H}_t\mathbf{P}_t^-\mathbf{H}_t^\top + \mathbf{R}_t = \mathbf{S}_t$ yields the following.

$$\begin{aligned}
\mathbf{m}_t &= \mathbb{E}[\mathbf{x}_t \mid \mathbf{y}_{1:t}] = \mathbf{a} + \mathbf{C}\mathbf{B}^{-1}(\mathbf{y}_t - \mathbf{b}) = \mathbf{m}_t^- + \mathbf{P}_t^-\mathbf{H}_t^\top\mathbf{S}_t^{-1}(\mathbf{y}_t - \mathbf{H}_t\mathbf{m}_t^-), \\
\mathbf{P}_t &= \mathrm{Var}(\mathbf{x}_t \mid \mathbf{y}_{1:t}) = \mathbf{A} - \mathbf{C}\mathbf{B}^{-1}\mathbf{C}^\top = \mathbf{P}_t^- - \mathbf{K}_t\mathbf{S}_t\mathbf{K}_t^\top.
\end{aligned}$$

To see the last equality for $\mathbf{P}_t$, we substitute the definition of the Kalman Gain $\mathbf{K}_t = \mathbf{P}_t^-\mathbf{H}_t^\top\mathbf{S}_t^{-1}$. Since $\mathbf{S}_t$ is symmetric being the covariance matrix of a random variable, we can write the subtraction term as:

$$\mathbf{C}\mathbf{B}^{-1}\mathbf{C}^\top = (\mathbf{K}_t\mathbf{S}_t)\mathbf{S}_t^{-1}(\mathbf{K}_t\mathbf{S}_t)^\top = \mathbf{K}_t\mathbf{S}_t\mathbf{K}_t^\top.$$

This completes the derivation of the KF update rule from Bayesian filtering.

Optimality. An important property of the KF is its optimality within the class of linear systems.

Theorem 1.5 (Optimality of KF)**.** *For linear Gaussian systems, the Kalman Filter is the minimum Mean Squared Error (MMSE) estimator. If the noise is non-Gaussian but the system is linear, the KF is the Best Linear Unbiased Estimator (BLUE).*

Limitations. The KF is strictly limited to linear systems. In real-world applications (e.g., robotics, finance), the dynamics or measurements are often governed by nonlinear functions (e.g., $\mathbf{x}_t = f(\mathbf{x}_{t-1}) + \xi_{t-1}$). Applying the standard KF to such systems leads to significant estimation errors or filter divergence.

1.4 Extended Kalman Filter

While the Kalman Filter offers an optimal solution for linear systems, many practical problems involve nonlinear dynamics or measurements. Consider the general nonlinear SSM where the state transition and measurement functions can also depend on time:

$$\begin{aligned} \mathbf{x}_t &= f_t(\mathbf{x}_{t-1}) + \xi_{t-1}, \quad \xi_{t-1} \sim \mathcal{N}(0, \mathbf{Q}_{t-1}), \\ \mathbf{y}_t &= g_t(\mathbf{x}_t) + \zeta_t, \qquad \zeta_t \sim \mathcal{N}(0, \mathbf{R}_t). \end{aligned}$$

Here, $f_t : \mathbb{R}^p \to \mathbb{R}^p$ and $g_t : \mathbb{R}^p \to \mathbb{R}^d$ are differentiable nonlinear functions. Because the Gaussian property is not preserved under nonlinear transformations, the posterior distribution $p(\mathbf{x}_t \mid \mathbf{y}_{1:t})$ is generally non-Gaussian and cannot be computed in closed form.

The *Extended Kalman Filter* (EKF) addresses this by applying a local linearization of the nonlinear functions. The core idea is to approximate the nonlinear dynamics with a first-order Taylor series expansion around the current best estimate of the state. This effectively approximates the posterior distribution as a Gaussian at each time step.

For the prediction step, we linearize the state transition function f_t around the previous posterior mean $\mathbf{m}_{t-1}$. For the update step, we linearize the measurement function g_t around the predicted mean $\mathbf{m}_t^-$.

Let $\mathbf{F}_t$ and $\mathbf{G}_t$ denote the Jacobian matrices of f_t and g_t, respectively. The EKF algorithm proceeds recursively as follows:

Claim 1.6 (EKF Equations)**.** *The prediction step propagates the mean through the full nonlinear function f_t, but propagates the covariance using the Jacobian $\mathbf{F}_t$ evaluated at the previous estimate.*

$$\text{(EKF Prediction)} \quad \begin{cases} \mathbf{F}_{t-1} \leftarrow \left.\frac{\partial f_t}{\partial \mathbf{x}}\right|_{\mathbf{x}=\mathbf{m}_{t-1}}, \\ \mathbf{m}_t^- \leftarrow f_t(\mathbf{m}_{t-1}), \\ \mathbf{P}_t^- \leftarrow \mathbf{F}_{t-1}\mathbf{P}_{t-1}\mathbf{F}_{t-1}^\top + \mathbf{Q}_{t-1}. \end{cases}$$

The update step computes the Kalman gain using the Jacobian $\mathbf{G}_t$ evaluated at the predicted mean, while the innovation is computed using the nonlinear measurement function g_t.

$$\text{(EKF Update)} \quad \begin{cases} \mathbf{G}_t \leftarrow \left.\frac{\partial g_t}{\partial \mathbf{x}}\right|_{\mathbf{x}=\mathbf{m}_t^-}, \\ \mathbf{S}_t \leftarrow \mathbf{G}_t\mathbf{P}_t^-\mathbf{G}_t^\top + \mathbf{R}_t, \\ \mathbf{K}_t \leftarrow \mathbf{P}_t^-\mathbf{G}_t^\top\mathbf{S}_t^{-1}, \\ \mathbf{m}_t \leftarrow \mathbf{m}_t^- + \mathbf{K}_t(\mathbf{y}_t - g_t(\mathbf{m}_t^-)), \\ \mathbf{P}_t \leftarrow \mathbf{P}_t^- - \mathbf{K}_t\mathbf{S}_t\mathbf{K}_t^\top. \end{cases}$$

This formulation allows the EKF to track the state of a nonlinear system by continuously re-linearizing the dynamics along the estimated trajectory.

Limitations. The reliance on linearization introduces several key limitations. First, the first-order Taylor approximation ignores higher-order terms, which can lead to significant errors if the nonlinearity is severe or if the state uncertainty is large. Second, the calculation of Jacobian matrices can be computationally expensive or analytically intractable for complex, high-dimensional models. Finally, the implicit Gaussian assumption means the EKF cannot represent multi-modal posterior distributions, which is common in highly nonlinear or ambiguous scenarios.

1.5 Unscented Kalman Filter

1.5.1 The Unscented Transform

To overcome the limitations of the EKF's linearization, the *Unscented Kalman Filter* (UKF) employs a deterministic sampling approach known as the *Unscented Transform* (UT). The philosophy behind the UKF is captured by the intuition that "it is easier to approximate a probability distribution than it is to approximate an arbitrary nonlinear function." [JU97].

Instead of linearizing the functions f_t and g_t (as the EKF does), the UKF deterministically selects a minimal set of sample points, called *sigma points*, that capture the mean and covariance of the state distribution. These points are then propagated through the true nonlinear functions. The mean and covariance of the transformed distribution are then recovered from the propagated sigma points. This approach captures the posterior mean and covariance accurately to the second order of the Taylor series expansion for any nonlinearity, and to the third order for Gaussian inputs, offering a significant improvement over the first-order accuracy of the EKF.

1.5.2 UKF Algorithm

The UKF algorithm for an p-dimensional state involves three main stages: sigma point generation, prediction, and update.

Claim 1.7 (UKF Equations)**.**

1. ***Sigma Point Generation:*** *We generate $2p+1$ points: one center point at the mean, and $2p$ points spread symmetrically along the axes of the covariance ellipse.*

$$\begin{aligned}
\mathscr{X}_{t-1}^{(0)} &= \mathbf{m}_{t-1}, \\
\mathscr{X}_{t-1}^{(i)} &= \mathbf{m}_{t-1} + \sqrt{p+\lambda}\left[\sqrt{\mathbf{P}_{t-1}}\right]_i, \quad i = 1,\dots,p, \\
\mathscr{X}_{t-1}^{(i+p)} &= \mathbf{m}_{t-1} - \sqrt{p+\lambda}\left[\sqrt{\mathbf{P}_{t-1}}\right]_i, \quad i = 1,\dots,p.
\end{aligned}$$

   Here, $\left[\sqrt{\mathbf{P}_{t-1}}\right]_i$ refers to the i-th column of the matrix square root of the covariance (typically computed via Cholesky decomposition). Intuitively, these columns represent the directions and magnitudes of the uncertainty axes. The points are assigned weights for the mean (W_m) and covariance (W_c) reconstruction.

$$\begin{aligned}
W_m^{(0)} &= \frac{\lambda}{p+\lambda}, \quad W_c^{(0)} = \frac{\lambda}{p+\lambda} + (1-\alpha^2+\beta), \\
W_m^{(i)} &= W_c^{(i)} = \frac{1}{2(p+\lambda)}, \quad i = 1,\dots,2p.
\end{aligned}$$

   The behavior of the sigma points is controlled by the scaling parameters α, β, κ, and λ. The parameter α is a small positive constant (typically set to 10^{-3}) that restricts the spread of the points to avoid sampling non-local effects, while κ is a secondary scaling parameter usually set to 0 or $3-p$. These define the composite scaling factor $\lambda = \alpha^2(p+\kappa) - p$, which dictates the precise distance of the sigma points from the center. Finally, the parameter β incorporates prior knowledge about the distribution's shape; for Gaussian distributions, setting $\beta = 2$ is optimal as it minimizes higher-order approximation errors.

2. ***Prediction step:*** *The sigma points are propagated through the nonlinear state transition function f_t:*

$$\mathscr{X}_t^{*(i)} = f_t(\mathscr{X}_{t-1}^{(i)}).$$

The predicted mean and covariance are computed as weighted sums of the transformed points:

$$\mathbf{m}_t^- = \sum_{i=0}^{2p} W_m^{(i)} \mathscr{X}_t^{*(i)},$$

$$\mathbf{P}_t^- = \sum_{i=0}^{2p} W_c^{(i)} (\mathscr{X}_t^{*(i)} - \mathbf{m}_t^-)(\mathscr{X}_t^{*(i)} - \mathbf{m}_t^-)^\top + \mathbf{Q}_{t-1}.$$

3. ***Update step:*** *The propagated sigma points are mapped through the measurement function g_t:*

$$\mathscr{Y}_t^{(i)} = g_t(\mathscr{X}_t^{*(i)}).$$

The predicted measurement $\hat{\mathbf{y}}_t$, innovation covariance $\mathbf{S}_t$, and cross-covariance $\mathbf{C}_t$ are computed as:

$$\hat{\mathbf{y}}_t = \sum_{i=0}^{2p} W_m^{(i)} \mathscr{Y}_t^{(i)},$$

$$\mathbf{S}_t = \sum_{i=0}^{2p} W_c^{(i)} (\mathscr{Y}_t^{(i)} - \hat{\mathbf{y}}_t)(\mathscr{Y}_t^{(i)} - \hat{\mathbf{y}}_t)^\top + \mathbf{R}_t,$$

$$\mathbf{C}_t = \sum_{i=0}^{2p} W_c^{(i)} (\mathscr{X}_t^{*(i)} - \mathbf{m}_t^-)(\mathscr{Y}_t^{(i)} - \hat{\mathbf{y}}_t)^\top.$$

Finally, the Kalman update is performed.

$$\mathbf{K}_t = \mathbf{C}_t \mathbf{S}_t^{-1}, \quad \mathbf{m}_t = \mathbf{m}_t^- + \mathbf{K}_t(\mathbf{y}_t - \hat{\mathbf{y}}_t), \quad \mathbf{P}_t = \mathbf{P}_t^- - \mathbf{K}_t \mathbf{S}_t \mathbf{K}_t^\top.$$

Advantages. The UKF provides a more accurate approximation for nonlinear systems than the EKF, capturing the posterior mean and covariance to the second order (or third order for Gaussian inputs). Crucially, it does not require the derivation of Jacobian matrices, making it a derivative-free method that is significantly easier to implement for complex or non-differentiable models.

Limitations. Despite its higher accuracy, the UKF fundamentally assumes that the posterior distribution can be well-approximated by a Gaussian. Consequently, it cannot capture complex multi-modal behavior or heavy-tailed distributions. Furthermore, the computational cost can be prohibitive for high-dimensional systems. While the number of sigma points scales linearly with the dimension $(2p+1)$, the overall algorithmic complexity scales cubically ($O(p^3)$) due to the matrix square root and covariance reconstruction steps. This can become computationally prohibitive for high-dimensional systems compared to particle-based methods or simpler ensembles.

1.6 Particle Filter

1.6.1 Sequential Monte Carlo (SMC)

When the SSM is highly nonlinear or the noise is non-Gaussian (e.g., multi-modal), the posterior distribution $p(\mathbf{x}_{1:t} \mid \mathbf{y}_{1:t})$ can take complex forms that cannot be adequately described by a

Gaussian approximation. In such cases, Gaussian approximations like the EKF and UKF may fail. The *Particle Filter* (PF), also known as *Sequential Monte Carlo* (SMC), offers a powerful alternative by abandoning the Gaussian assumption.

Intuitively, the Particle Filter works by representing uncertainty as a cloud of random guesses, or *particles*. Consider the example of tracking a robot with an unknown location. We initially generate N random possible locations (particles). When a measurement is received (e.g., a GPS ping), we compare each particle to the data: particles consistent with the measurement are assigned high weights, while those far away are assigned low weights. By focusing computational effort on the high-weight particles and discarding the unlikely ones, the filter converges on the true state over time.

Formally, this intuition corresponds to approximating the complex posterior distribution $p(\mathbf{x}_{1:t} \mid \mathbf{y}_{1:t})$ using a set of N weighted samples $\{\mathbf{x}_{1:t}^{(i)}, w_t^{(i)}\}_{i=1}^N$. As the number of particles $N \to \infty$, this discrete approximation converges to the true posterior. We express the approximated posterior density as a sum of Dirac delta functions centered at the particles.

$$\hat{p}(\mathbf{x}_{1:t} \mid \mathbf{y}_{1:t}) = \sum_{i=1}^{N} w_t^{(i)} \delta(\mathbf{x}_{1:t} - \mathbf{x}_{1:t}^{(i)}).$$

Consequently, any integral of interest (such as the expected state $\varphi(\mathbf{x}_{1:t})$) can be approximated by a weighted sum of the particles:

$$\mathbb{E}[\varphi(\mathbf{x}_{1:t}) \mid \mathbf{y}_{1:t}] \approx \sum_{i=1}^{N} w_t^{(i)} \varphi(\mathbf{x}_{1:t}^{(i)}).$$

1.6.2 Importance Sampling

The challenge is that we cannot generate samples directly from the complicated posterior $p(\mathbf{x}_{1:t} \mid \mathbf{y}_{1:t})$ because we do not know what it looks like yet. To solve this, we use the principle of *Importance Sampling*. The idea is as follows. We draw samples from an alternative distribution that is *easy* to sample from (called the proposal distribution, q). Then we "grade" each sample based on how well it matches the true target distribution p such that $p \ll q$ (absolute continuity). The expectation of a test function $\varphi(\mathbf{x})$ under p can be rewritten as an expectation under q is

$$\mathbb{E}_p[\varphi(\mathbf{x})] = \int \varphi(\mathbf{x}) p(\mathbf{x}) d\mathbf{x} = \int \varphi(\mathbf{x}) \frac{p(\mathbf{x})}{q(\mathbf{x})} q(\mathbf{x}) d\mathbf{x} = \mathbb{E}_q \left[\varphi(\mathbf{x}) \frac{p(\mathbf{x})}{q(\mathbf{x})} \right].$$

The term $w(\mathbf{x}) = p(\mathbf{x})/q(\mathbf{x})$ is the *importance weight*, which acts as the Radon-Nikodym derivative correcting for the discrepancy between our proposal (where we guessed) and the target (where the truth is).

1.6.3 Sequential Importance Sampling (SIS)

In a sequential setting, we want to update these weights recursively as new data comes in, without recalculating the entire history. We decompose the target distribution and the proposal distribution over the trajectory $\mathbf{x}_{1:t}$. By factoring the distributions using Bayes' rule and the Markov property, we derive a recursive update rule for the unnormalized weights $w_t^{*(i)}$

$$w_t^{(i)} = \frac{p(\mathbf{x}_t^{(i)} \mid \mathbf{y}_{1:t})}{q(\mathbf{x}_t^{(i)} \mid \mathbf{y}_{1:t})} \propto w_{t-1}^{(i)} \frac{p(\mathbf{y}_t \mid \mathbf{x}_t^{(i)}) p(\mathbf{x}_t^{(i)} \mid \mathbf{x}_{t-1}^{(i)})}{q(\mathbf{x}_t^{(i)} \mid \mathbf{x}_{t-1}^{(i)}, \mathbf{y}_{1:t})},$$

since the target posterior factors as

$$p(\mathbf{x}_t \mid \mathbf{y}_{1:t}) \propto p(\mathbf{y}_t \mid \mathbf{x}_t)\, p(\mathbf{x}_t \mid \mathbf{x}_{t-1})\, p(\mathbf{x}_{t-1} \mid \mathbf{y}_{1:t-1}),$$

and a proposal distribution also factorizes as

$$q(\mathbf{x}_t \mid \mathbf{y}_{1:t}) = q(\mathbf{x}_t \mid \mathbf{x}_{t-1}, \mathbf{y}_{1:t})\, q(\mathbf{x}_{t-1} \mid \mathbf{y}_{1:t-1}).$$

Choice of Proposal. The choice of the proposal distribution q is critical for the performance of the filter. The *optimal proposal* minimizes the variance of the importance weights, but it is often intractable to sample from. A common and simple choice for the proposal distribution p is to simply use the transition model: $q(\mathbf{x}_t \mid \mathbf{x}_{t-1}^{(i)}, \mathbf{y}_t) = p(\mathbf{x}_t \mid \mathbf{x}_{t-1}^{(i)})$. Substituting this into the equation above cancels out the transition terms, simplifying the weight update to just the likelihood:

$$w_t^{(i)} \propto w_{t-1}^{(i)} \underbrace{p(\mathbf{y}_t \mid \mathbf{x}_t^{(i)})}_{\text{Likelihood}}.$$

This means the weight of a particle is simply multiplied by how likely the current measurement is, given that particle's state.

1.6.4 Sequential Importance Resampling

A major practical issue of standard SIS is particle degeneracy. Over time, the variance of the weights invariably increases: a single particle will eventually accumulate almost all the weight ($w \approx 1$), while all other particles become negligible ($w \approx 0$). This effectively reduces our sample size to 1, wasting computational power.

To address this, we use the *Sequential Importance Resampling* (SIR) algorithm. We monitor the *Effective Sample Size* (ESS), approximated as $N_{\text{eff}} \approx 1/\sum (w_t^{(i)})^2$. When N_{eff} drops below a threshold (e.g., $N/2$), we trigger a resampling step: we duplicate high-weight particles and kill off low-weight particles.

Algorithm 1.8 (Sequential Importance Resampling / Bootstrap Filter)**.** For each time step t:

1. **Prediction step:** Draw N new particles from the transition prior.

$$\mathbf{x}_t^{(i)} \sim p(\mathbf{x}_t \mid \mathbf{x}_{t-1}^{(i)}), \quad i = 1, \ldots, N.$$

2. **Update step:** Compute the importance weights based on the likelihood of the observation $\mathbf{y}_t$.

$$w_t^{(i)} \propto w_{t-1}^{(i)} p(\mathbf{y}_t \mid \mathbf{x}_t^{(i)}).$$

   Normalize weights so that $\sum_{i=1}^{N} w_t^{(i)} = 1$.

3. **Resampling step:** Calculate the effective sample size $N_{\text{eff}} \approx 1/\sum (w_t^{(i)})^2$. If $N_{\text{eff}} < N_{\text{threshold}}$,

   - Draw N new particles $\{\tilde{\mathbf{x}}_t^{(j)}\}$ from the current set $\{\mathbf{x}_t^{(i)}\}$ with probabilities proportional to their weights $w_t^{(i)}$. (High-weight particles are selected multiple times; low-weight ones are dropped).
   - Replace the current set with the resampled set $\{\tilde{\mathbf{x}}_t^{(j)}\}_{j=1}^{N}$ and reset all weights to $1/N$.

Advantages. The PF is highly flexible and capable of representing global, multimodal posteriors for arbitrary nonlinear and non-Gaussian systems, making it superior to the EKF or UKF in complex scenarios, which rely on rigid Gaussian approximations.

Limitations. This flexibility comes at a significant cost. The PF suffers from the "curse of dimensionality", where the number of particles required to adequately cover the state space scales exponentially with the state dimension (d), making it computationally intractable for high-dimensional problems. Furthermore, the resampling step can lead to *sample impoverishment*, a phenomenon where the diversity of the particle set is lost (as high-weight particles are duplicated, potentially causing all particles to collapse to the same few points) if the process noise is too small.

1.7 Exact Daum-Huang Flow

1.7.1 Motivation: Continuous Transport vs. Resampling

While the PF theoretically solves the nonlinear filtering problem, its reliance on discrete resampling introduces stochastic noise and necessitates a prohibitively large number of particles in high dimensions. To mitigate these limitations, Daum and Huang [DHN10, DH11] proposed the *Particle Flow Filter* (PFF).

The fundamental insight of the PFF is to replace the discrete reweighting (update step in Algorithm 1.8) and resampling step of the standard PF with a continuous probability flow. We introduce a pseudo-time parameter $\lambda \in [0,1]$ and construct a flow of particles such that at $\lambda = 0$, the particles are distributed according to the prior $g(\mathbf{x}) := p(\mathbf{x}_t \mid \mathbf{y}_{1:t-1})$, and at $\lambda = 1$, they are distributed according to the posterior $p(\mathbf{x}) := p(\mathbf{x}_t \mid \mathbf{y}_{1:t})$.

The evolution of the *Probability Density Function* (PDF) is governed by a homotopy, which is a smooth interpolation between the two distributions at $\lambda = 0$ and $\lambda = 1$. The particles move according to a deterministic *Ordinary Differential Equation* (ODE)

$$\frac{d\mathbf{x}}{d\lambda} = f(\mathbf{x}, \lambda), \tag{1.4}$$

where $f(\mathbf{x}, \lambda)$ is the velocity field we seek to determine.

1.7.2 Derivation of the Particle Flow PDE

We construct the evolution of the probability distribution $p(\mathbf{x}, \lambda)$ for $0 \le \lambda \le 1$ as follows. Denote the likelihood $h(\mathbf{x}) := p(\mathbf{y}_t \mid \mathbf{x})$. We define the intermediate distribution as

$$p(\mathbf{x}, \lambda) = \frac{1}{Z(\lambda)} g(\mathbf{x}) h(\mathbf{x})^{\lambda},$$

where $Z(\lambda) = \int g(\mathbf{x}) h(\mathbf{x})^{\lambda} \, d\mathbf{x}$ is the normalization constant. Taking the logarithm gives the *log-homotopy* between the prior $g(\mathbf{x})$ and the posterior $\frac{1}{Z(1)} g(\mathbf{x}) h(\mathbf{x})$:

$$\log p(\mathbf{x}, \lambda) = \log g(\mathbf{x}) + \lambda \log h(\mathbf{x}) - \log Z(\lambda), \qquad \lambda \in [0,1].$$

Taking the derivative with respect to λ, we obtain the rate of change of the log-density:

$$\frac{\partial \log p(\mathbf{x}, \lambda)}{\partial \lambda} = \log h(\mathbf{x}) - \frac{d}{d\lambda} \log Z(\lambda). \tag{1.5}$$

Since the particles move according to the deterministic flow (1.4) and mass is conserved, the evolution of the density $p(\mathbf{x}, \lambda)$ must satisfy the *Continuity Equation* (also known as the Fokker-Planck equation with zero diffusion):

$$\frac{\partial p}{\partial \lambda} = -\mathrm{div}(p(\mathbf{x}, \lambda) f(\mathbf{x}, \lambda)). \tag{1.6}$$

where div denotes the divergence operator $\mathrm{div}(f) = \sum_i \frac{\partial f_i}{\partial x_i}$[1]. Using the identity $\frac{\partial p}{\partial \lambda} = p \frac{\partial \log p}{\partial \lambda}$ and expanding the divergence $\mathrm{div}(pf) = p\,\mathrm{div}(f) + \langle \nabla p, f \rangle$, we combine (1.5) and (1.6).

$$p(\mathbf{x}, \lambda) \log h(\mathbf{x}) - p(\mathbf{x}, \lambda) \frac{d}{d\lambda} \log Z(\lambda) = -\left(p(\mathbf{x}, \lambda) \mathrm{div}(f(\mathbf{x}, \lambda)) + \langle \nabla p(\mathbf{x}, \lambda), f(\mathbf{x}, \lambda) \rangle \right).$$

Assuming $p(\mathbf{x}, \lambda)$ is non-vanishing and dividing by p, we arrive at the fundamental *Partial Differential Equation* (PDE) governing the particle flow:

$$\log h(\mathbf{x}) - \frac{d}{d\lambda} \log Z(\lambda) = -\mathrm{div}(f(\mathbf{x}, \lambda)) - \langle \nabla \log p(\mathbf{x}, \lambda), f(\mathbf{x}, \lambda) \rangle. \tag{1.7}$$

This PDE relates the known likelihood $h(\mathbf{x})$ and the current density $p(\mathbf{x}, \lambda)$ to the unknown velocity field f.

Remark 1.9 (Mass Conservation)**.** Early derivations in [DH11] utilized an unnormalized log-homotopy, effectively ignoring the $Z(\lambda)$ term. This leads to a contradiction with the continuity equation, which fundamentally requires mass conservation. The derivation above explicitly accounts for the normalization constant term $\frac{d}{d\lambda} \log Z(\lambda)$, ensuring the flow is exact.

1.7.3 Exact Solution for Gaussian Densities

The PDE derived in (1.7) is under-determined since it gives a single scalar constraint for a p-dimensional vector field. To find a unique solution, we assume a Gaussian prior $g(\mathbf{x}) = \mathcal{N}(\mathbf{x} \mid \bar{\mathbf{x}}(0), \mathbf{P}(0))$ and a Gaussian likelihood

$$h(\mathbf{x}) \propto \exp\left(-\frac{1}{2} (\mathbf{y}_t - \mathbf{H}\mathbf{x})^\top \mathbf{R}^{-1} (\mathbf{y}_t - \mathbf{H}\mathbf{x}) \right).$$

Under the flow, the intermediate density $p(\mathbf{x}, \lambda)$ remains Gaussian (since the log-densities remain quadratic in $\mathbf{x}$ for all λ) with mean $\bar{\mathbf{x}}(\lambda)$ and covariance $\mathbf{P}(\lambda)$ given by

$$\begin{aligned} \mathbf{P}(\lambda) &= \mathbf{P}(0) - \lambda \mathbf{P}(0) \mathbf{H}^\top (\lambda \mathbf{H} \mathbf{P}(0) \mathbf{H}^\top + \mathbf{R})^{-1} \mathbf{H} \mathbf{P}(0), \\ \bar{\mathbf{x}}(\lambda) &= \mathbf{P}(\lambda) \left[\mathbf{P}(0)^{-1} \bar{\mathbf{x}}(0) + \lambda \mathbf{H}^\top \mathbf{R}^{-1} \mathbf{y}_t \right]. \end{aligned} \tag{1.8}$$

Consequently, the gradient of the log-density is

$$\nabla \log p(\mathbf{x}, \lambda) = -\mathbf{P}(\lambda)^{-1} (\mathbf{x} - \bar{\mathbf{x}}(\lambda)).$$

[1] This can be justified as follows: Denote $P_B(\lambda) := \int_B p(\mathbf{x}, \lambda)\, d\mathbf{x}$ for each open ball B. Then

$$\frac{d}{d\lambda} P_B(\lambda) = \int_B \frac{\partial}{\partial \lambda} p(\mathbf{x}, \lambda)\, dx = -\oint_{\partial B} p(\mathbf{x}, \lambda) \langle f(\mathbf{x}, \lambda), n(\mathbf{x}) \rangle\, d\mathbf{x} = -\int_B \mathrm{div}(pf)\, d\mathbf{x},$$

where $n(\mathbf{x})$ denotes the outward normal to the ball B at surface point $\mathbf{x} \in \partial B$. The last equality follows from the divergence theorem. Hence $\int_B \left(\frac{\partial}{\partial \lambda} p(\mathbf{x}, \lambda) + \mathrm{div}(pf) \right) d\mathbf{x} = 0$. This holds for all open balls B, so the integrand must be zero almost surely.

We can justify the claimed Gaussianity of the intermediate pdfs $p(\mathbf{x}, \lambda)$ and justify the above formulas for the mean and the covariance matrix as follow. Inserting the Gaussian densities into the normalized log-homotopy gives:

$$\log p(\mathbf{x}, \lambda) = \log g(\mathbf{x}) + \lambda \log h(\mathbf{x}) - \log Z(\lambda).$$

By substituting the quadratic forms of the Gaussian distributions, we can derive the evolution of the precision matrix and the mean as functions of λ.

The precision matrix (inverse covariance) of the intermediate density is determined by equating the coefficients of the terms quadratic in $\mathbf{x}$: From the log-prior $\log g(\mathbf{x})$, the quadratic term is $-\frac{1}{2}\mathbf{x}^\top \mathbf{P}(0)^{-1}\mathbf{x}$. From the scaled log-likelihood $\lambda \log h(\mathbf{x})$, the quadratic term is $-\frac{1}{2}\lambda \mathbf{x}^\top \mathbf{H}^\top \mathbf{R}^{-1}\mathbf{H}\mathbf{x}$. Equating their sum to the quadratic term of the intermediate density, $-\frac{1}{2}\mathbf{x}^\top \mathbf{P}(\lambda)^{-1}\mathbf{x}$, yields the precision relationship:

$$\mathbf{P}(\lambda)^{-1} = \mathbf{P}(0)^{-1} + \lambda \mathbf{H}^\top \mathbf{R}^{-1}\mathbf{H}.$$

Applying the Woodbury matrix inversion formula allows us to express the covariance $\mathbf{P}(\lambda)$ explicitly as in (1.8).

Similarly, the mean of the intermediate density is determined by equating the coefficients of the terms linear in $\mathbf{x}$: From the log-prior, the linear term is $\mathbf{x}^\top \mathbf{P}(0)^{-1}\bar{\mathbf{x}}(0)$. From the scaled log-likelihood, the linear term is $\lambda \mathbf{x}^\top \mathbf{H}^\top \mathbf{R}^{-1}\mathbf{y}_t$. The intermediate mean $\bar{\mathbf{x}}(\lambda)$ must satisfy

$$\mathbf{P}(\lambda)^{-1}\bar{\mathbf{x}}(\lambda) = \mathbf{P}(0)^{-1}\bar{\mathbf{x}}(0) + \lambda \mathbf{H}^\top \mathbf{R}^{-1}\mathbf{y}_t.$$

Solving for $\bar{\mathbf{x}}(\lambda)$ gives

$$\bar{\mathbf{x}}(\lambda) = \mathbf{P}(\lambda)\left[\mathbf{P}(0)^{-1}\bar{\mathbf{x}}(0) + \lambda \mathbf{H}^\top \mathbf{R}^{-1}\mathbf{y}_t\right].$$

Daum and Huang [DHN10] provided the following explicit affine solution in terms of the initial covariance matrix $\mathbf{P}(0)$ and mean $\bar{\mathbf{x}}(0)$.

Claim 1.10 (Exact Daum-Huang Flow Equations)**.** *Under the Gaussian prior and Gaussian likelihood assumption, the flow PDE* (1.7) *admits an affine solution*

$$\frac{d\mathbf{x}}{d\lambda} = \mathbf{A}(\lambda)\mathbf{x} + \mathbf{b}(\lambda) \tag{1.9}$$

where

$$\mathbf{A}(\lambda) = -\frac{1}{2}\mathbf{P}(0)\mathbf{H}^\top(\lambda \mathbf{H}\mathbf{P}(0)\mathbf{H}^\top + \mathbf{R})^{-1}\mathbf{H},$$
$$\mathbf{b}(\lambda) = (\mathbf{I} + 2\lambda \mathbf{A}(\lambda))\left[(\mathbf{I} + \lambda \mathbf{A}(\lambda))\mathbf{P}(0)\mathbf{H}^\top \mathbf{R}^{-1}\mathbf{y}_t + \mathbf{A}(\lambda)\bar{\mathbf{x}}(0)\right].$$

Here $\mathbf{P}(0)$ *and* $\bar{\mathbf{x}}(0)$ *refer to the prior covariance and mean at* $\lambda = 0$.

Proof. Substituting the ansatz (1.9) and the Gaussian forms into (1.7), we expand terms by powers of $\mathbf{x}$. From the likelihood terms,

$$\begin{aligned}
\text{LHS of (1.7)} &= \log h(\mathbf{x}) - \frac{d}{d\lambda}\log Z(\lambda) \\
&= -\frac{1}{2}(\mathbf{y}_t - \mathbf{H}\mathbf{x})^\top \mathbf{R}^{-1}(\mathbf{y}_t - \mathbf{H}\mathbf{x}) - \frac{d}{d\lambda}\log Z(\lambda) \\
&= -\frac{1}{2}\mathbf{x}^\top \mathbf{H}^\top \mathbf{R}^{-1}\mathbf{H}\mathbf{x} + \mathbf{y}_t^\top \mathbf{R}^{-1}\mathbf{H}\mathbf{x} + \underbrace{\left(-\frac{1}{2}\mathbf{y}_t^\top \mathbf{R}^{-1}\mathbf{y}_t - \frac{d}{d\lambda}\log Z(\lambda)\right)}_{\text{Scalar constants}}.
\end{aligned}$$

Next, recall $\operatorname{div}(f(\mathbf{x},\lambda)) = \operatorname{Tr}(\mathbf{A}(\lambda))$ under the affine flow assumption. The term $\langle \nabla \log p(\mathbf{x},\lambda), f(\mathbf{x},\lambda)\rangle$ becomes

$$\begin{aligned} f(\mathbf{x},\lambda)^\top \nabla \log p(\mathbf{x},\lambda) &= (\mathbf{A}(\lambda)\mathbf{x}+\mathbf{b}(\lambda))^\top \left(-\mathbf{P}(\lambda)^{-1}(\mathbf{x}-\bar{\mathbf{x}}(\lambda))\right) \\ &= -\mathbf{x}^\top \mathbf{A}(\lambda)^\top \mathbf{P}(\lambda)^{-1}\mathbf{x} + \mathbf{x}^\top \mathbf{A}(\lambda)^\top \mathbf{P}(\lambda)^{-1}\bar{\mathbf{x}}(\lambda) - \mathbf{b}(\lambda)^\top \mathbf{P}(\lambda)^{-1}\mathbf{x} + \mathbf{b}(\lambda)^\top \mathbf{P}(\lambda)^{-1}\bar{\mathbf{x}}(\lambda). \end{aligned}$$

Thus,

$$\begin{aligned} \text{RHS of (1.7)} &= -\operatorname{div}(f(\mathbf{x},\lambda)) - \langle \nabla \log p(\mathbf{x},\lambda),\, f(\mathbf{x},\lambda)\rangle \\ &= -\operatorname{Tr}(\mathbf{A}(\lambda)) + \mathbf{x}^\top \mathbf{A}(\lambda)^\top \mathbf{P}(\lambda)^{-1}\mathbf{x} - \mathbf{x}^\top (\mathbf{A}(\lambda)^\top \mathbf{P}(\lambda)^{-1}\bar{\mathbf{x}}(\lambda) - \mathbf{P}(\lambda)^{-1}\mathbf{b}(\lambda)) - \mathbf{b}(\lambda)^\top \mathbf{P}(\lambda)^{-1}\bar{\mathbf{x}}(\lambda). \end{aligned}$$

Since the equation must hold for all $\mathbf{x}$, we equate the coefficients of the quadratic and linear terms separately. Note that the scalar constant term involving $Z(\lambda)$ appear only in the constant terms (order $\mathbf{x}^0$), so it does not affect $\mathbf{A}$ or $\mathbf{b}$. First, equating coefficients of the quadratic terms yields the relationship

$$-\frac{1}{2}\mathbf{H}^\top \mathbf{R}^{-1}\mathbf{H} = \mathbf{A}(\lambda)^\top \mathbf{P}(\lambda)^{-1}.$$

Solving for $\mathbf{A}(\lambda)$ (using symmetry of $\mathbf{P}(\lambda)$ and $\mathbf{R}$) gives

$$\mathbf{A}(\lambda) = -\frac{1}{2}\mathbf{P}(\lambda)\mathbf{H}^\top \mathbf{R}^{-1}\mathbf{H}. \tag{1.10}$$

Plugging in (1.8) gives

$$\begin{aligned} \mathbf{A}(\lambda) &= -\frac{1}{2}\left[\mathbf{P}(0) - \lambda \mathbf{P}(0)\mathbf{H}^\top (\lambda \mathbf{H}\mathbf{P}(0)\mathbf{H}^\top + \mathbf{R})^{-1}\mathbf{H}\mathbf{P}(0)\right]\mathbf{H}^\top \mathbf{R}^{-1}\mathbf{H} \\ &= -\frac{1}{2}\mathbf{P}(0)\mathbf{H}^\top \left[\mathbf{I} - \lambda(\lambda \mathbf{H}\mathbf{P}(0)\mathbf{H}^\top + \mathbf{R})^{-1}\mathbf{H}\mathbf{P}(0)\mathbf{H}^\top\right]\mathbf{R}^{-1}\mathbf{H}. \end{aligned}$$

Inside the brackets, we use the identity $(\mathbf{M}+\mathbf{N})^{-1}\mathbf{M} = \mathbf{I} - (\mathbf{M}+\mathbf{N})^{-1}\mathbf{N}$ with $\mathbf{M} = \lambda \mathbf{H}\mathbf{P}(0)\mathbf{H}^\top$ and $\mathbf{N} = \mathbf{R}$:

$$\mathbf{I} - (\lambda \mathbf{H}\mathbf{P}(0)\mathbf{H}^\top + \mathbf{R})^{-1}(\lambda \mathbf{H}\mathbf{P}(0)\mathbf{H}^\top) = (\lambda \mathbf{H}\mathbf{P}(0)\mathbf{H}^\top + \mathbf{R})^{-1}\mathbf{R}.$$

Substituting this back gives the claimed formula for $\mathbf{A}(\lambda)$.

Next, we derive the claimed formula for $\mathbf{b}(\lambda)$. From the matching of linear coefficients in the fundamental PDE, we have

$$\mathbf{b}(\lambda) = \mathbf{A}(\lambda)\bar{\mathbf{x}}(\lambda) + \mathbf{P}(\lambda)\mathbf{H}^\top \mathbf{R}^{-1}\mathbf{y}_t. \tag{1.11}$$

We first substitute the expression for the evolving mean $\bar{\mathbf{x}}(\lambda)$ into the first term

$$\mathbf{b}(\lambda) = \mathbf{A}(\lambda)\left(\mathbf{P}(\lambda)[\mathbf{P}(0)^{-1}\bar{\mathbf{x}}(0) + \lambda \mathbf{H}^\top \mathbf{R}^{-1}\mathbf{y}_t]\right) + \mathbf{P}(\lambda)\mathbf{H}^\top \mathbf{R}^{-1}\mathbf{y}_t.$$

Expanding and regrouping terms for $\bar{\mathbf{x}}(0)$ and $\mathbf{y}_t$

$$\mathbf{b}(\lambda) = \mathbf{A}(\lambda)\mathbf{P}(\lambda)\mathbf{P}(0)^{-1}\bar{\mathbf{x}}(0) + [\lambda \mathbf{A}(\lambda) + \mathbf{I}]\mathbf{P}(\lambda)\mathbf{H}^\top \mathbf{R}^{-1}\mathbf{y}_t.$$

To express the formula solely in terms of $\mathbf{P}(0)$, we use the precision relationship $\mathbf{P}(\lambda)^{-1} = \mathbf{P}(0)^{-1} + \lambda \mathbf{H}^\top \mathbf{R}^{-1}\mathbf{H}$. Multiplying by $\mathbf{P}(\lambda)$ gives

$$\mathbf{I} = \mathbf{P}(\lambda)\mathbf{P}(0)^{-1} + \lambda \mathbf{P}(\lambda)\mathbf{H}^\top \mathbf{R}^{-1}\mathbf{H}$$

Recalling the instantaneous relationship $\mathbf{A}(\lambda) = -\frac{1}{2}\mathbf{P}(\lambda)\mathbf{H}^\top\mathbf{R}^{-1}\mathbf{H}$, we substitute $\mathbf{P}(\lambda)\mathbf{H}^\top\mathbf{R}^{-1}\mathbf{H} = -2\mathbf{A}(\lambda)$ to obtain:

$$\mathbf{P}(\lambda)\mathbf{P}(0)^{-1} = \mathbf{I} + 2\lambda\mathbf{A}(\lambda),$$

or equivalently,

$$\mathbf{P}(\lambda) = (\mathbf{I} + 2\lambda\mathbf{A}(\lambda))\mathbf{P}(0).$$

Substituting these identities back into the grouped equation for $\mathbf{b}(\lambda)$,

- Term 1 ($\bar{\mathbf{x}}(0)$): $\mathbf{A}(\lambda)(\mathbf{I} + 2\lambda\mathbf{A}(\lambda))\bar{\mathbf{x}}(0) = (\mathbf{I} + 2\lambda\mathbf{A}(\lambda))\mathbf{A}(\lambda)\bar{\mathbf{x}}(0)$.
- Term 2 ($\mathbf{y}_t$): $(\mathbf{I} + \lambda\mathbf{A}(\lambda))(\mathbf{I} + 2\lambda\mathbf{A}(\lambda))\mathbf{P}(0)\mathbf{H}^\top\mathbf{R}^{-1}\mathbf{y}_t$.

Factoring out $(\mathbf{I} + 2\lambda\mathbf{A}(\lambda))$ leads to the claimed formula yields

$$\mathbf{b}(\lambda) = (\mathbf{I} + 2\lambda\mathbf{A}(\lambda))\left[(\mathbf{I} + \lambda\mathbf{A}(\lambda))\mathbf{P}(0)\mathbf{H}^\top\mathbf{R}^{-1}\mathbf{y}_t + \mathbf{A}(\lambda)\bar{\mathbf{x}}(0)\right].$$

□

Remark 1.11 (Linearization of Flow vs. Transformation)**.** It is crucial to distinguish the affine assumption in the *Exact Daum-Huang* (EDH) Flow from the linearization performed in the EKF. The EKF linearizes the measurement function $h(\mathbf{x})$ once at the prior mean, effectively approximating the entire transport from prior to posterior as a single affine transformation. If the likelihood peak is far from the prior, this static first-order approximation can deviate significantly from the true transition map, leading to significant errors.

In contrast, the EDH filter linearizes the *velocity field* $f(\mathbf{x}, \lambda)$ rather than the total transport map. While the instantaneous velocity is affine ($\frac{d\mathbf{x}}{d\lambda} = \mathbf{A}(\lambda)\mathbf{x} + \mathbf{b}(\lambda)$), the accumulated transport map $\Phi : \mathbf{x}_{\lambda=0} \mapsto \mathbf{x}_{\lambda=1}$ obtained by integrating this ODE is generally nonlinear. Furthermore, the linearization coefficients $\mathbf{A}(\lambda)$ and $\mathbf{b}(\lambda)$ are re-evaluated continuously as the particles migrate. This allows the particles to "navigate" the curvature of the probability landscape rather than jumping across it, effectively performing a continuous-time, statistically consistent update.

1.7.4 Numerical Implementation

In practice, the EDH filter is implemented by discretizing the pseudo-time interval $\lambda \in [0, 1]$ into N_λ steps of size $\epsilon = 1/N_\lambda$. For each step $k = 0, \ldots, N_\lambda - 1$,

1. Compute $\lambda_k = k\epsilon$.
2. Evaluate $\mathbf{A}(\lambda_k)$ and $\mathbf{b}(\lambda_k)$ using the fixed prior parameters $\mathbf{P}(0), \bar{\mathbf{x}}(0)$ and the measurement $\mathbf{y}_t$.
3. Update each particle position

$$\mathbf{x}_t^{(i)}(\lambda_{k+1}) = \mathbf{x}_t^{(i)}(\lambda_k) + \epsilon\left(\mathbf{A}(\lambda_k)\mathbf{x}_t^{(i)}(\lambda_k) + \mathbf{b}(\lambda_k)\right).$$

This method avoids resampling entirely, preserving particle diversity and mitigating the curse of dimensionality.

1.8 Local Exact Daum-Huang Flow

1.8.1 Motivation: Handling Non-Gaussianity

The EDH flow relies on a *global* Gaussian assumption. While effective for unimodal distributions, this assumption fails when the posterior is multi-modal or skewed, as the single affine flow forces all particles toward one common mode. To address this, we relax the global assumption and assume the densities are only *locally* Gaussian. This approach, referred to as the *Local Exact Daum-Huang* (LEDH) flow, allows the flow field to be nonlinear and particle-dependent.

1.8.2 Derivation via Local Linearization

While Daum and Huang [DH11] discussed general methods for localizing the flow (such as Direct Integration), the specific formulation of LEDH using pointwise curvature was formalized by Ding and Coates [DC12].

The core idea of LEDH is to relax the global Gaussian assumption by applying the exact flow equations pointwise to each particle. For a particle located at $\mathbf{x}_i$, we approximate the log-density of the intermediate distribution $p(\mathbf{x}, \lambda)$ as a local quadratic function (a local Gaussian):

$$\log p(\mathbf{x}, \lambda) \approx \text{const} + \mathbf{g}_i^\top (\mathbf{x} - \mathbf{x}_i) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_i)^\top \mathbf{H}_i (\mathbf{x} - \mathbf{x}_i), \tag{1.12}$$

where the local gradient $\mathbf{g}_i = \nabla \log p(\mathbf{x}_i, \lambda)$ and local Hessian $\mathbf{H}_i = \nabla^2 \log p(\mathbf{x}_i, \lambda)$ are evaluated at the particle's current position. Crucially, because the intermediate density is defined by the log-homotopy $\log p(\mathbf{x}, \lambda) = \log g(\mathbf{x}) + \lambda \log h(\mathbf{x}) - \log Z(\lambda)$, these local statistics are computed directly from the known prior g and likelihood h. Specifically, the local Hessian is the sum of the prior and scaled likelihood curvatures:

$$\mathbf{H}_i = \nabla^2 \log g(\mathbf{x}_i) + \lambda \nabla^2 \log h(\mathbf{x}_i).$$

Matching the local quadratic approximation in (1.12) to the standard Gaussian form $-\frac{1}{2}(\mathbf{x} - \bar{\mathbf{x}}_i)^\top \mathbf{P}_i^{-1}(\mathbf{x} - \bar{\mathbf{x}}_i)$, we identify the particle-specific local covariance and mean as

$$\mathbf{P}_i = -\mathbf{H}_i^{-1}, \qquad \bar{\mathbf{x}}_i = \mathbf{x}_i - \mathbf{H}_i^{-1}\mathbf{g}_i.$$

By substituting these local parameters into the flow equations, the velocity field becomes nonlinear and particle-dependent, allowing the filter to capture multi-modal or highly skewed posterior structures.

1.8.3 Implementation of LEDH

Following the framework in [DC12], the LEDH algorithm updates each particle $\mathbf{x}_i$ individually based on the *instantaneous* flow equations derived in the previous section's proof:

1. **Local statistics:** At step k, evaluate the Hessian $\mathbf{H}_i(\lambda_k)$ and gradient $\mathbf{g}_i(\lambda_k)$ of the log-density at the current particle position $\mathbf{x}_i(\lambda_k)$:

$$\mathbf{H}_i(\lambda_k) \leftarrow \nabla^2 \log g(\mathbf{x}_i) + \lambda_k \nabla^2 \log h(\mathbf{x}_i), \quad \mathbf{g}_k^{(i)} \leftarrow \nabla \log g(\mathbf{x}_i) + \lambda_k \nabla \log h(\mathbf{x}_i).$$

2. **Local flow parameters:** Compute the local covariance and mean

$$\mathbf{P}_i(\lambda_k) \leftarrow -\mathbf{H}_i(\lambda_k)^{-1}, \quad \bar{\mathbf{x}}_i(\lambda_k) \leftarrow \mathbf{x}_i(\lambda_k) - \mathbf{H}_i(\lambda_k)^{-1}\mathbf{g}_i(\lambda_k).$$

Then, calculate the instantaneous affine flow parameters $\mathbf{A}_i(\lambda_k)$ and $\mathbf{b}_i(\lambda_k)$ using (1.10) and (1.11) respectively:

$$\mathbf{A}_i(\lambda_k) = -\frac{1}{2}\mathbf{P}_i(\lambda_k)\mathbf{J}_i(\lambda_k)^\top \mathbf{R}^{-1}\mathbf{J}_i(\lambda_k), \quad \mathbf{b}_i(\lambda_k) = \mathbf{A}_i(\lambda_k)\bar{\mathbf{x}}_k^{(i)} + \mathbf{P}_i(\lambda_k)\mathbf{J}_i(\lambda_k)^\top \mathbf{R}^{-1}\mathbf{y}_t,$$

where $\mathbf{J}_i(\lambda_k)$ is the linearized measurement matrix (Jacobian of h at $\mathbf{x}_i(\lambda_k)$) and $\mathbf{R}$ is the measurement noise covariance.

3. **Particle update:** Move the particle forward using its specific affine flow by the step size $\epsilon = \lambda_{k+1} - \lambda_k$:
$$\mathbf{x}_i(\lambda_{k+1}) = \mathbf{x}^{(i)}(\lambda_k) + \epsilon\left(\mathbf{A}_i(\lambda_k)\mathbf{x}_i(\lambda_k) + \mathbf{b}_i(\lambda_k)\right).$$

The implementation of the LEDH filter requires a strategy for estimating the local Hessian $\nabla^2 \log g(\mathbf{x})$ and gradient $\nabla \log g(\mathbf{x})$ of the prior density g, despite the algorithm typically maintaining only a discrete set of particles rather than a continuous functional form. One common implementation relies on a *global parametric surrogate*, where the prior is assumed to be globally Gaussian for the specific purpose of calculating these local statistics. In this approach, the algorithm sets the prior Hessian to the negative inverse of a predicted covariance matrix,

$$\nabla^2 \log g(\mathbf{x}_i) \approx -\mathbf{P}_0^{-1},$$

while the prior gradient for each particle is derived from its deviation from the predicted mean

$$\nabla \log g(\mathbf{x}_i) \approx -\mathbf{P}_0^{-1}(\mathbf{x}_i - \bar{\mathbf{x}}_0).$$

Crucially, the prior covariance $\mathbf{P}_0$ is often estimated using the standard prediction equations found in the EKF, which propagates the covariance through a linearized state transition model. While this method is computationally efficient, it is limited by its inability to capture multi-modal distributions, as the entire particle cloud is forced to follow the curvature of a single global mode.

Alternatively, a more sophisticated approach uses an Ensemble-Based Local Approximation, such as *kernel density estimation* (KDE), to estimate the prior density $g(\mathbf{x})$ directly from the current particle distribution. This allows the flow to adapt to the specific shape of the particle cloud by approximating the prior density as a sum of small Gaussian kernels centered at each particle. For a specific particle, the local gradient and Hessian are found by differentiating this sum of kernels, or a subset of kernels from the K-nearest neighbors, at that particle's position. While this ensemble approach enables the flow to capture multi-modal or highly skewed structures, it significantly increases computational complexity because it requires matrix operations for every particle relative to its neighbors.

1.8.4 Comparison: EDH vs. LEDH

The distinction between EDH and LEDH lies primarily in the resolution of the linearization. While EDH operates under a global Gaussian assumption, LEDH relaxes this to a local Gaussian framework, enabling it to track more complex, non-Gaussian probability landscapes.

1. **Flexibility and Modeling Assumptions.** Standard EDH is fundamentally restricted to unimodal posteriors because it utilizes a single affine flow for the entire particle cloud. In contrast, LEDH can capture multi-modal or highly skewed structures because its global

flow field is nonlinear, constructed from piecewise affine flows that adapt to the local geometry of the density around each particle.

The behavior of LEDH depends significantly on how the prior's local statistics are estimated. When implementing LEDH via a Global Parametric Surrogate, the algorithm effectively assumes the prior is globally Gaussian, typically using a covariance $\mathbf{P}_0$ estimated via the standard EKF prediction equations. However, even with this global prior assumption, LEDH retains a significant advantage over EDH: it evaluates the measurement Jacobian $\mathbf{J}_i$ and local likelihood Hessian at every individual particle position rather than at a single global mean. This allows the filter to navigate the curvature of nonlinear measurement manifolds far more accurately than EDH.

For maximum flexibility, the Ensemble-Based Local Approximation (such as Kernel Density Estimation) removes the global Gaussian assumption entirely. By estimating the local prior Hessian and gradient from the neighborhood of each particle, the flow can follow multiple distinct modes or resolve heavy-tailed distributions that a global surrogate would collapse.

2. **Computational Complexity.** The superior flexibility of LEDH comes at a higher computational cost. EDH is $O(p^3)$ per pseudo-time step, requiring only one matrix inversion for the global flow parameters. LEDH, in its most direct form, is $O(Np^3)$ per step, as it necessitates an individual matrix inversion for every particle in the ensemble. If the ensemble-based approximation (KDE) is used, the complexity may increase further due to the $O(N^2)$ or $O(NK)$ neighbor-search and summation requirements for calculating local prior statistics.

3. **Stability and Regularization.** Numerical stability is a critical concern for LEDH. To ensure a valid, positive-definite local covariance $\mathbf{P}_i$, the local Hessian of the log-density must remain negative definite. In regions where the log-density is non-concave—common in the tails of complex distributions—the Hessian may lose this property, leading to numerical divergence. Consequently, regularization techniques, such as adding a diagonal term to the local Hessian or utilizing a "safe" inverse, are often required to maintain filter stability in near-deterministic or highly nonlinear regimes [LC17].

1.9 Particle-flow Particle-filter

1.9.1 Motivation: Consistency

While the EDH and LEDH filters described in the previous sections mitigate particle degeneracy, they suffer from a fundamental theoretical limitation: the resulting particle distribution is only an approximation of the true posterior. The discrepancies arise from the discretization of the pseudo-time integral, the linearization of the measurement model, and the Gaussian assumptions inherent in the derivation of the flow parameters. As a result, the deterministic "raw" particle flow filter is not asymptotically consistent; it introduces a bias that cannot be removed simply by increasing the number of particles.

To address this, Li and Coates [LC17] proposed the *Particle Flow Particle Filter* (PF-PF). The core idea is to treat the particle flow not as the final estimator, but as a mechanism to generate a highly efficient *proposal distribution* $q(\mathbf{x}_t \mid \mathbf{x}_{t-1}, \mathbf{y}_t)$ within a standard SIS framework. By properly reweighting the particles after the flow, we recover the true posterior distribution,

ensuring asymptotic consistency while enjoying the improved effective sample size provided by the flow.

1.9.2 Invertible Particle Flow as a Proposal

Let $\eta_0^{(i)}$ denote the i-th particle drawn from the predictive transition prior $\eta_0^{(i)} \sim p(\mathbf{x}_t \mid \mathbf{x}_{t-1}^{(i)})$, and let $\eta_1^{(i)}$ denote the particle position after the application of the flow (at pseudo-time $\lambda = 1$). We view the flow as a deterministic mapping T

$$\eta_1^{(i)} = T(\eta_0^{(i)}; \mathbf{y}_t).$$

In the standard PF, the importance weight $w_t^{(i)}$ is updated according to the ratio of the target posterior to the proposal density

$$w_t^{(i)} \propto w_{t-1}^{(i)} \frac{p(\mathbf{y}_t \mid \eta_1^{(i)}) p(\eta_1^{(i)} \mid \mathbf{x}_{t-1}^{(i)})}{q(\eta_1^{(i)} \mid \mathbf{x}_{t-1}^{(i)}, \mathbf{y}_t)}.$$

A major challenge in using deterministic flows for proposal generation is evaluating the proposal density $q(\eta_1^{(i)})$. Since η_1 is the result of transforming the random variable η_0 (distributed according to the prior) through the map T, the density transforms according to the change of variables formula involving the Jacobian determinant

$$q(\eta_1^{(i)}) = \frac{p(\eta_0^{(i)} \mid \mathbf{x}_{t-1}^{(i)})}{\left|\det\left(\mathbf{J}_T(\eta_0^{(i)})\right)\right|}, \tag{1.13}$$

where $\mathbf{J}_T = \frac{\partial \eta_1}{\partial \eta_0}$ is the Jacobian matrix of the mapping. Substituting (1.13) into the weight update equation yields

$$w_t^{(i)} \propto w_{t-1}^{(i)} \frac{p(\mathbf{y}_t \mid \eta_1^{(i)}) p(\eta_1^{(i)} \mid \mathbf{x}_{t-1}^{(i)})}{p(\eta_0^{(i)} \mid \mathbf{x}_{t-1}^{(i)})} \left|\det\left(\mathbf{J}_T(\eta_0^{(i)})\right)\right|.$$

Calculating the determinant of the Jacobian for general mappings is computationally expensive ($O(p^3)$). However, Li and Coates demonstrated that for the specific structure of the discretized EDH/LEDH flow, this calculation can be simplified or even eliminated.

1.9.3 Jacobian of the Discretized Flow

Recall that the numerical implementation of the flow updates the particle iteratively over N_λ steps. For a step size ϵ_j, the update is affine

$$\eta_{\lambda_j} = \eta_{\lambda_{j-1}} + \epsilon_j(\mathbf{A}_j \eta_{\lambda_{j-1}} + \mathbf{b}_j) = (\mathbf{I} + \epsilon_j \mathbf{A}_j)\eta_{\lambda_{j-1}} + \epsilon_j \mathbf{b}_j.$$

The Jacobian of this single step is simply $\mathbf{I} + \epsilon_j \mathbf{A}_j$. By the chain rule, the Jacobian determinant of the total mapping T is the product of the determinants of the individual steps

$$|\det \mathbf{J}_T| = \prod_{j=1}^{N_\lambda} \left|\det(\mathbf{I} + \epsilon_j \mathbf{A}_j(\lambda))\right|. \tag{1.14}$$

Claim 1.12 (Invertibility Condition)**.** *The mapping T is invertible if the step sizes ϵ_j are chosen sufficiently small such that $\epsilon_j < 1/\rho(\mathbf{A}_j)$, where $\rho(\cdot)$ is the spectral radius (the largest absolute eigenvalue). This ensures that* $\det(\mathbf{I} + \epsilon_j \mathbf{A}_j) \neq 0$ *for all steps.*

1.9.4 Algorithm Variants: PF-PF (EDH) vs. PF-PF (LEDH)

The computational complexity of the weight update depends critically on whether we use the global (EDH) or local (LEDH) flow parameters.

PF-PF (EDH): The "Free" Weight Update

In the EDH flow, the matrix $\mathbf{A}_j(\lambda)$ is computed globally based on the sample covariance and is identical for all particles. Consequently, the Jacobian determinant term in (1.14) is *constant* across all particles $i = 1, \ldots, N$. Since the weights are normalized such that $\sum w_t^{(i)} = 1$, any particle-independent multiplicative constant cancels out. Therefore, for PF-PF (EDH), the weight update simplifies to

$$w_t^{(i)} \propto w_{t-1}^{(i)} \underbrace{p(\mathbf{y}_t \mid \eta_1^{(i)})}_{\text{Likelihood}} \underbrace{\frac{p(\eta_1^{(i)} \mid \mathbf{x}_{t-1}^{(i)})}{p(\eta_0^{(i)} \mid \mathbf{x}_{t-1}^{(i)})}}_{\text{Transition Ratio}}.$$

This allows us to achieve the benefits of PF without the expensive determinant calculation.

PF-PF (LEDH): High-Precision Correction

In the Local LEDH flow, the matrix $\mathbf{A}_j^{(i)}$ varies per particle. Thus, the Jacobian determinant is different for each particle and must be explicitly calculated

$$w_t^{(i)} \propto w_{t-1}^{(i)} \frac{p(\mathbf{y}_t \mid \eta_1^{(i)}) p(\eta_1^{(i)} \mid \mathbf{x}_{t-1}^{(i)})}{p(\eta_0^{(i)} \mid \mathbf{x}_{t-1}^{(i)})} \prod_{j=1}^{N_\lambda} \left| \det(\mathbf{I} + \epsilon_j \mathbf{A}_j^{(i)}) \right|.$$

While computationally more intensive ($O(N \cdot N_\lambda \cdot p^3)$), this method provides a rigorous correction for the nonlinear local flow, enabling the filter to track complex multi-modal posteriors with high accuracy.

Algorithm 1.13 (PF-PF Pipeline)**.** For each time step k:

1. **Propagate:** Draw $\eta_0^{(i)} \sim p(\mathbf{x}_k \mid \mathbf{x}_{k-1}^{(i)})$ (Standard prediction).
2. **Flow:** Evolve particles $\eta_0^{(i)} \to \eta_1^{(i)}$ using EDH or LEDH equations.
3. **Weight update:**
   - If EDH: Update weights using Likelihood × Transition Ratio. (Jacobian cancels out).
   - If LEDH: Update weights using Likelihood × Transition Ratio × Jacobian Determinant Product $\prod |\det(I + \epsilon \mathbf{A}^{(i)})|$.

   Normalize weights so that $\sum w_k^{(i)} = 1$.
4. **Resample:** If N_{eff} is low, resample $\{\eta_1^{(i)}\}$ according to $\{w_k^{(i)}\}$.

1.10 Kernel-Embedded Particle Flow Filter

1.10.1 Motivation: Handling High-Dimensionality and Sparsity

While affine flow methods such as EDH and LEDH perform well in many settings, they rely on specific parametric assumptions (Gaussianity or local Gaussianity). An alternative class of particle flow filters avoids these assumptions by optimizing the flow within a *Reproducing Kernel Hilbert Space* (RKHS), as exemplified by methods such as *Stein Variational Gradient Descent* (SVGD). However, these kernel-based methods struggle in high-dimensional geophysical systems where observations are often sparse. In these scenarios, the standard scalar kernel commonly adopted in RKHS-based RKHS formulations can lead to a critical failure known as marginal collapse.

As identified by Hu and van Leeuwen [HVL21], when a scalar kernel $K(\mathbf{x},\mathbf{z}) = k(\|\mathbf{x}-\mathbf{z}\|)$ is used, the flow is governed by a single distance metric across the entire state space. In high-dimensional systems with sparse observations, the observed variables converge much faster than the unobserved ones. A scalar kernel, dominated by the large Euclidean distances in the unobserved dimensions, fails to generate a sufficient short-range repulsive force in the observed dimensions. Consequently, particles collapse onto the posterior mode in the observed subspaces, leading to severe sample impoverishment. To address this, Hu and van Leeuwen proposed the *Kernel-embedded Particle Flow Filter* (KPFF) utilizing a matrix-valued kernel.

1.10.2 Derivation of particle flow in Reproducing Kernel Hilbert Space

The derivation begins by defining the particle flow

$$\frac{d\mathbf{x}}{ds} = \mathbf{f}(\mathbf{x}, s)$$

in pseudo-time s (analogous to λ in EDH, but derived via gradient descent). We seek a flow field f_s that minimizes the *Kullback-Leibler* (KL) divergence between the intermediate particle density $q_s(\mathbf{x})$ and the target posterior $p(\mathbf{x}) = p(\mathbf{x}|\mathbf{y})$:

$$\mathrm{KL}(q_s||p) = \int q_s(\mathbf{x}) \log\left(\frac{q_s(\mathbf{x})}{p(\mathbf{x})}\right) d\mathbf{x}.$$

For a given target pdf p, and a given prior from the prediction, the KL divergence is only a function of the intermediate pdf q_s at each pseudo time s. To be efficient, the aim is to find the appropriate flow field $\mathbf{f}(\cdot, s)$ at each pseudo time such that the KL divergence can decrease as fast as possible. For this, note that the rate of change of the KL divergence is given by

$$\begin{aligned}\frac{d}{ds}\mathrm{KL}(q_s||p) &= \int \frac{d}{ds} q_s(\mathbf{x}) \log\left(\frac{q_s(\mathbf{x})}{p(\mathbf{x})}\right) d\mathbf{x} \\ &= \int \frac{d}{ds} q_s(\mathbf{x}) \left(\log\left(\frac{q_s(\mathbf{x})}{p(\mathbf{x})}\right) + 1\right) d\mathbf{x}.\end{aligned} \tag{1.15}$$

Due to conservation of mass, as in the derivation of EDH (1.6), the continuity equation is satisfied for the intermediate pdfs:

$$\frac{\partial q_s}{\partial s} = -\mathrm{div}(q_s(\mathbf{x})\mathbf{f}(\mathbf{x}, s)) = -\nabla_{\mathbf{x}} \cdot (q_s(\mathbf{x})\mathbf{f}(\mathbf{x}, s)).$$

Substituting this back into (1.15) and integrating by parts, we get

$$\begin{aligned}\frac{d}{ds}\mathrm{KL}(q_s||p) &= -\int \nabla_{\mathbf{x}}\cdot(q_s(\mathbf{x})\mathbf{f}(\mathbf{x},s))\left(\log\left(\frac{q_s(\mathbf{x})}{p(\mathbf{x})}\right)+1\right)d\mathbf{x} \\ &= \int (q_s(\mathbf{x})\mathbf{f}(\mathbf{x},s))\nabla_{\mathbf{x}}\cdot\left(\log\left(\frac{q_s(\mathbf{x})}{p(\mathbf{x})}\right)+1\right)d\mathbf{x} \\ &= -\int \mathbf{f}(\mathbf{x},s))\nabla_{\mathbf{x}}\cdot q_s(\mathbf{x}) - q_s(\mathbf{x})\mathbf{f}(\mathbf{x},s)\nabla_{\mathbf{x}}\cdot\log p(\mathbf{x})\,d\mathbf{x}.\end{aligned} \tag{1.16}$$

We have used the fact that q_s should approach zero on the boundary to assure that the integral of q_s in the whole domain is finite.

We now assume that the solution of $\mathbf{f}(\cdot,s)$ is embedded in a *Reproducing Kernel Hilbert Space* (RKHS). Every function in a RKHS can be written as

$$\mathbf{f}(\mathbf{x},s) = \langle \mathbf{K}(\mathbf{x},\cdot), \mathbf{f}(\cdot,s)\rangle \in \mathbb{R}^{n_x}$$

where $\mathbf{K}(\mathbf{x},\cdot)$ is a matrix-valued kernel. The above means that each of the n_x coordinates of $f(\mathbf{x},s)$ is determined by the inner product with the corresponding row of $\mathbf{K}(\mathbf{x},\cdot)$ and the whole vector-valued function $f(\cdot,s)$. Using this reproducing property, the divergence of $\mathbf{f}(\cdot,s)$ can be computed as

$$\nabla_{\mathbf{x}}\cdot[\mathbf{f}(\mathbf{x},s)] = \nabla_{\mathbf{x}}\cdot\langle \mathbf{K}(\mathbf{x},\cdot), \mathbf{f}(\cdot,s)\rangle = \langle \nabla_{\mathbf{x}}\cdot\mathbf{K}(\mathbf{x},\cdot), \mathbf{f}(\cdot,s)\rangle_1 \in \mathbb{R}.$$

Further simplifying (1.16) using the above, we get

$$\frac{d}{ds}\mathrm{KL}(q_s||p) = \left\langle -\int q_s(\mathbf{x})[\nabla_{\mathbf{x}}\cdot\mathbf{K}(\mathbf{x},\cdot) + \mathbf{K}(\mathbf{x},\cdot)\nabla_{\mathbf{x}}\cdot\log p(\mathbf{x})]\,d\mathbf{x}, \mathbf{f}(\cdot,s)\right\rangle_1.$$

We would like to choose $\mathbf{f}(\cdot,s)$ so that the above rate of change in the KL divergence is negative. Choosing

$$f_s(\mathbf{x}) = \mathbf{D}\underbrace{\int q_s(\mathbf{z})\left(\mathbf{K}(\mathbf{z},\mathbf{x})\nabla_{\mathbf{z}}\log p(\mathbf{z}|\mathbf{y}) + \nabla_{\mathbf{z}}\cdot\mathbf{K}(\mathbf{z},\mathbf{x})\right)d\mathbf{z}}_{=:\mathbf{I}_f}$$

where $\mathbf{D}$ is a positive-definite preconditioner matrix (typically the prior covariance) ensuring dimensional consistency, indeed we get

$$\frac{d}{ds}\mathrm{KL}(q_s||p) = -\mathbf{I}_f^{\top}\mathbf{D}\mathbf{I}_f \le 0.$$

Using the Monte Carlo approximation for $q_s(\mathbf{x}) \approx \frac{1}{N}\sum_{j=1}^{N}\delta(\mathbf{x}-\mathbf{x}^{(j)})$, the continuous integral becomes a discrete sum over the particles

$$f_s(\mathbf{x}) = \frac{1}{N}\mathbf{D}\sum_{j=1}^{N}\left(\mathbf{K}(\mathbf{x}^{(j)},\mathbf{x})\nabla_{\mathbf{x}^{(j)}}\log p(\mathbf{x}^{(j)}|\mathbf{y}) + \nabla_{\mathbf{x}^{(j)}}\cdot\mathbf{K}(\mathbf{x}^{(j)},\mathbf{x})\right). \tag{1.17}$$

The first term acts as an *attractive force*, driving particles towards high-probability regions (modes) of the posterior. The second term, the divergence of the kernel, acts as a *repulsive force*, preventing particle collapse by pushing them apart.

1.10.3 Matrix-Valued Kernel vs. Scalar Kernel

The core innovation of the KPFF is the specific form of the kernel $\mathbf{K}$. A standard scalar kernel takes the form $\mathbf{K}(\mathbf{x},\mathbf{z}) = k(\mathbf{x},\mathbf{z})\mathbf{I}$, where k is a scalar function (e.g., Gaussian kernel). Hu and van Leeuwen propose a diagonal matrix-valued kernel to treat state dimensions independently.

$$\mathbf{K}(\mathbf{x},\mathbf{z}) = \operatorname{diag}\big([k_1(x_1,z_1), k_2(x_2,z_2), \ldots, k_p(x_p,z_p)]\big),$$

where each component kernel is a 1D Gaussian

$$k_a(x_a,z_a) = \exp\left(-\frac{(x_a-z_a)^2}{2\sigma_a^2}\right).$$

Here, σ_a is a bandwidth parameter, often scaled by the prior variance of that component.

The primary advantage of this matrix-valued formulation is that it creates a decoupled repulsive force, where the interaction in a specific dimension a is governed solely by the distance $|x_a - z_a|$ in that dimension. Unlike scalar kernels, which dilute the repulsive force based on the total Euclidean distance, this component-wise structure ensures that particles experience strong repulsion in observed dimensions even if they remain far apart in unobserved dimensions. Consequently, this prevents the marginal distributions of fast-converging observed variables from collapsing, effectively maintaining diversity across the entire state space.

1.10.4 Implementation Algorithm

The KPFF is implemented via an iterative pseudo-time integration scheme. For each pseudo-time step s and each particle i,

1. **Gradient Calculation:** Evaluate the gradient of the log-posterior $\nabla \log p(\mathbf{x}^{(i)}|\mathbf{y})$. For Gaussian likelihoods and priors,

$$\nabla \log p(\mathbf{x}^{(i)}|\mathbf{y}) = \mathbf{H}^\top \mathbf{R}^{-1}(\mathbf{y} - \mathbf{H}\mathbf{x}^{(i)}) - \mathbf{P}_{\text{prior}}^{-1}(\mathbf{x}^{(i)} - \bar{\mathbf{x}}_{\text{prior}}).$$

2. **Kernel Interaction:** Compute the kernel matrix $\mathbf{K}(\mathbf{x}^{(j)},\mathbf{x}^{(i)})$ and its divergence $\nabla_{\mathbf{x}^{(j)}} \cdot \mathbf{K}$ relative to all other particles j.

3. **Flow Computation:** Sum the attractive and repulsive terms according to (1.17) to obtain the velocity vector $f_s(\mathbf{x}^{(i)})$.

4. **Update:** Update the particle state

$$\mathbf{x}_{s+1}^{(i)} = \mathbf{x}_s^{(i)} + \Delta s \cdot f_s(\mathbf{x}^{(i)}).$$

   An adaptive step size Δs (e.g., AdaGrad or RMSProp) is often used to ensure stability.

Remark 1.14 (Preconditioning)**.** To accelerate convergence and maintain physical balance in geophysical models, the matrix $\mathbf{D}$ is typically set to the localized prior covariance matrix $\mathbf{P}_{\text{prior}}$ (often denoted as $\mathbf{B}$ in data assimilation literature). This acts as a preconditioner for the gradient descent in probability space.

Chapter 2

Project tasks

2.1 Testing and Validation Strategy

The reliability of advanced filtering algorithms hinges on a robust testing framework that validates not only the code's execution but also its structural integrity. Our implementation is structured around two core modules: `filters.py`, a unified library containing polymorphic implementations of the Kalman Filter, Particle Filter, and Particle Flow Filter families; and `ssm_models.py`, which encapsulates the physics, transition logic, and sensor geometries. Our validation strategy adopts a bottom-up approach, distinguishing between *unit tests* for internal verification and *integration tests* for pipeline stability. This narrative details our verification process across the spectrum of implemented filters.

2.1.1 Unit-Level Physics Verification

Before any filter is instantiated, the underlying *State Space Models* in `ssm_models.py` undergo strict unit testing to validate the mathematical logic governing the system.

- **Jacobian consistency:** For the Stochastic Volatility and Acoustic Tracking models, we validated the analytical Jacobian matrices $\mathbf{H}_x$ against numerical gradients computed via TensorFlow's automatic differentiation (`tf.GradientTape`). This confirmed that our manual linearizations align with the auto-diff baseline.
- **Shape integrity:** We enforced strict shape constraints on data generation, ensuring that the 1D Stochastic Volatility model and 4D Range-Bearing model produce consistent tensor ranks. This prevents the "silent broadcasting" errors common in high-dimensional tensor operations.

2.1.2 Foundational Verification: The Gaussian Regime

Our integration testing begins with the KF in `filters.py`. To accommodate the strict matrix multiplication requirements of the KF implementation, we developed a specialized test harness that reshapes the standard Rank-1 state vectors into Rank-2 column vectors ($\mathbb{R}^{N\times 1}$). We verified that given a linear subset of the Range-Bearing model, the filter's prediction and update logic executes without numerical divergence (NaNs). Crucially, we validated that the pipeline handles the matrix operations for covariance updates using a stabilized inversion method (`safe_inv`) to mitigate Cholesky decomposition failures.

2.1.3 Nonlinear and Particle Filter Validation

For nonlinear estimators, we validated the flexibility of our polymorphic implementation, which seamlessly handles inputs of varying rank.

- **Linearization checks:** For the EKF, we verified that the filter pipeline integrates with the model's analytical Jacobian methods, ensuring the tensor dimensions align during the update steps for both Stochastic Volatility and Range-Bearing models.
- **Pipeline integrity:** For the UKF and *Unscented Particle Filter* (UPF), we verified the execution of the Unscented Transform within the filtering loop, confirming that the sigma-point generation and reconstruction steps produce state estimates of the correct dimensionality.
- **Resampling mechanics:** For standard particle methods (BPF), we validated the normalization of importance weights. We confirmed that after every update step, the weights sum strictly to 1.0, ensuring the probabilistic validity of the particle set.

2.1.4 Stress-Testing High-Dimensional Particle Flows

The most rigorous validation was reserved for the PFF (EDH, LEDH, PFPF), particularly in high-dimensional settings.

- **Flow dynamics:** We validated the execution of the Daum-Huang flow by testing the EDH and LEDH filters on the nonlinear Range-Bearing problem, ensuring the pseudo-time integration steps complete successfully given the provided step-size schedules.
- **High-dimensional stability:** To validate scalability, we subjected the PFPF-LEDH filter to the 16-dimensional Acoustic Tracking model. By explicitly managing device placement (forcing CPU execution), we confirmed that the filter can process complex, multi-target sensor arrays without platform-specific GPU colocation errors.
- **Kernel flow correctness:** For the KPFF, we validated the update logic by asserting that the particle ensemble's mean shifts significantly away from its prior after the flow steps, empirically confirming that the RKHS-based gradients are effectively updating the particle positions.

2.2 Part 1: From Classical Filters To Particle Flows

Problem 1-I: Linear-Gaussian SSM and Kalman Filter.

a) Implement the Kalman filter for a multidimensional linear-Gaussian SSM. (Do not use tfp.distributions. LinearGaussianStateSpaceModel). Use synthetic data from a standard LGSSM, see example 2 in [DJ11].

b) Analyze its filtering optimality and numerical stability: compare filtered means/covariances to the Kalman recursion; use Joseph stabilized covariance updates; discuss conditioning number.

Solution. A TensorFlow implementation of KF is shown below.

A TensorFlow implementation of Kalman filter

```python
class KalmanFilter:
    """
    Standard Kalman Filter with TensorFlow Variables for persistent state.
    """
    def __init__(self, F, H, Q, R, P0, x0):
        # Cast constants to ensuring matching DTYPE
        self.F = tf.cast(F, DTYPE)
        self.H = tf.cast(H, DTYPE)
        self.Q = tf.cast(Q, DTYPE)
        self.R = tf.cast(R, DTYPE)
        self.I = tf.eye(self.F.shape[0], dtype=DTYPE)

        # Initialize State Variables (Persist across tf.function calls)
        # We check if x0/P0 are already variables or tensors
        if isinstance(x0, tf.Variable):
            self.x = x0
        else:
            self.x = tf.Variable(tf.cast(x0, DTYPE), dtype=DTYPE)

        if isinstance(P0, tf.Variable):
            self.P = P0
        else:
            self.P = tf.Variable(tf.cast(P0, DTYPE), dtype=DTYPE)

    @tf.function
    def predict(self):
        # Read current values
        x_curr = self.x.read_value()
        P_curr = self.P.read_value()

        # Prediction Logic
        x_pred = tf.matmul(self.F, x_curr)
        P_pred = tf.matmul(tf.matmul(self.F, P_curr), self.F, transpose_b=True) + self.Q

        # Update State Variables
        self.x.assign(x_pred)
        self.P.assign(P_pred)

        return x_pred, P_pred

    @tf.function
    def update(self, z):
```

```
        z = tf.cast(z, DTYPE)

        # Read current values (post-prediction)
        x_curr = self.x.read_value()
        P_curr = self.P.read_value()

        # Standard KF Update
        S = tf.matmul(tf.matmul(self.H, P_curr), self.H, transpose_b=True) + self.R
        K = tf.matmul(tf.matmul(P_curr, self.H, transpose_b=True), tf.linalg.inv(S))

        y_res = z - tf.matmul(self.H, x_curr)
        x_new = x_curr + tf.matmul(K, y_res)

        # Joseph Form Covariance Update
        I_KH = self.I - tf.matmul(K, self.H)
        term1 = tf.matmul(tf.matmul(I_KH, P_curr), I_KH, transpose_b=True)
        term2 = tf.matmul(tf.matmul(K, self.R), K, transpose_b=True)
        P_new = term1 + term2

        # Update State Variables
        self.x.assign(x_new)
        self.P.assign(P_new)

        return x_new, P_new
```

Consider the following linear Gaussian model given in [DJ11, Ex. 2], where the state space for the hidden states is $\mathscr{X} = \mathbb{R}^{n_x}$, the state space for the measurements is $\mathscr{Y} = \mathbb{R}^{n_y}$, initial hidden state $\mathbf{X}_1 \sim \mathcal{N}(0, \Sigma)$, and

$$\begin{aligned} \mathbf{X}_n &= \mathbf{A}\mathbf{X}_{n-1} + \mathbf{B}\mathbf{V}_n, \\ \mathbf{Y}_n &= \mathbf{C}\mathbf{X}_n + \mathbf{D}\mathbf{W}_n, \end{aligned}$$

where $\mathbf{V}_n \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0, I_{n_v})$, $\mathbf{W}_n \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0, I_{n_w})$ and $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ are matrices of appropriate dimensions.

A TensorFlow implementation of the above linear-Gaussian model is shown below.

A TensorFlow implementation of linear-Gaussian model in [DJ11, Ex. 2].

```
class LinearGaussianSSM:
    """Standard LGSSM Definition using TensorFlow"""
    def __init__(self, dim_x=100, dim_y=10):
        self.dim_x = dim_x
        self.dim_y = dim_y

        # Initialize A in NumPy first for easy spectral radius control
        A_np = np.random.randn(dim_x, dim_x)
        eigvals = np.linalg.eigvals(A_np)
        max_eig = np.max(np.abs(eigvals))
        if max_eig > 1.0:
            A_np /= (max_eig + 0.1)

        # Convert to TF Constants
        self.A = tf.constant(A_np, dtype=DTYPE)
        self.C = tf.random.normal((dim_y, dim_x), dtype=DTYPE)
        self.Q = tf.eye(dim_x, dtype=DTYPE) * 0.1
        self.R = tf.eye(dim_y, dtype=DTYPE) * 1.0
```

```
        self.x0_mean = tf.zeros(dim_x, dtype=DTYPE)
        self.x0_cov = tf.eye(dim_x, dtype=DTYPE) * 5.0

    @tf.function
    def generate(self, T=100):
        """Generates data using TensorFlow TensorArrays."""
        x_ta = tf.TensorArray(DTYPE, size=T)
        y_ta = tf.TensorArray(DTYPE, size=T)

        L_x0 = tf.linalg.cholesky(self.x0_cov)
        x_curr = self.x0_mean + tf.linalg.matvec(L_x0, tf.random.normal([self.dim_x], dtype=DTYPE))

        L_Q = tf.linalg.cholesky(self.Q)
        L_R = tf.linalg.cholesky(self.R)

        for t in tf.range(T):
            # Process Noise
            process_noise = tf.linalg.matvec(L_Q, tf.random.normal([self.dim_x], dtype=DTYPE))
            x_curr = tf.linalg.matvec(self.A, x_curr) + process_noise

            # Measurement Noise
            meas_noise = tf.linalg.matvec(L_R, tf.random.normal([self.dim_y], dtype=DTYPE))
            y_curr = tf.linalg.matvec(self.C, x_curr) + meas_noise

            x_ta = x_ta.write(t, x_curr)
            y_ta = y_ta.write(t, y_curr)

        return x_ta.stack(), y_ta.stack()
```

We report two experimental settings for KF estimation in low-dimension ($n_x = 5$ and $n_y = 5$) and in high-dimension ($n_x = 200$ and $n_y = 5$), both for $T = 100$ time steps.

Optimality validation. The optimality of the KF in Linear-Gaussian SSM is established through two primary theoretical lenses. First, the KF is the minimum MSE estimator under linear-Gaussian systems. This is because the posterior distribution $p(\mathbf{X}_n|\mathbf{Y}_{1:n})$ remains Gaussian for all n. The KF recursively updates the mean $\hat{\mathbf{X}}_{n|n}$ and covariance $\mathbf{P}_{n|n}$, where the mean of a Gaussian posterior is exactly the minimum MSE estimate.

$$\hat{\mathbf{X}}_{n|n} = \mathbb{E}[\mathbf{X}_n|\mathbf{Y}_{1:n}] = \arg\min_{\hat{\mathbf{X}}} \mathbb{E}[\|\mathbf{X}_n - \hat{\mathbf{X}}\|^2|\mathbf{Y}_{1:n}].$$

Second, by the Orthogonality principle, the KF provides the BLUE even if the Gaussian assumption is relaxed. It ensures that the estimation error $\tilde{\mathbf{X}}_{n|n} = \mathbf{X}_n - \hat{\mathbf{X}}_{n|n}$ is orthogonal to the subspace spanned by the observations $\mathbf{Y}_{1:n}$, such that $\mathbb{E}[\tilde{\mathbf{X}}_{n|n}\mathbf{Y}_i^T] = 0$ for $1 \le i \le n$. Consequently, the KF minimizes the trace of the error covariance matrix $\mathbf{P}_{n|n}$ at every discrete time step n.

To validate the KF's optimality empirically, we performed a conditional realization analysis. Recognizing that $\mathbf{P}_{t|t}$ represents the conditional covariance $\text{Cov}(\mathbf{X}_t|\mathbf{Y}_{1:t})$, we verified at each time t whether the KF's analytical covariance matches the empirical covariance of $M = 10,000$ realizations of $\mathbf{X}_t$ conditioned on the observation history. Using a non-parametric Importance Sampling update to approximate the true Bayesian posterior, we computed the Frobenius norm of the difference $\|\mathbf{P}_{\text{KF}} - \mathbf{P}_{\text{MC}}\|_F$. A low-magnitude norm confirms that the KF's internal uncertainty quantification matches the actual statistical spread of state realizations, a requirement for minimum MSE optimality.

For each time t, we compute the empirical error covariance matrix $\hat{\mathbf{P}}_{t|t}$ across the ensemble:

$$\hat{\mathbf{P}}_{t|t} = \frac{1}{M-1} \sum_{i=1}^{M} (\mathbf{X}_t^{(i)} - \hat{\mathbf{X}}_{t|t}^{(i)})(\mathbf{X}_t^{(i)} - \hat{\mathbf{X}}_{t|t}^{(i)})^T.$$

We will plot the Frobenius norm of the difference $\|\hat{\mathbf{P}}_{t|t} - \mathbf{P}_{t|t}\|_F$ over time. A vanishing or low-magnitude norm confirms that the filter's internal uncertainty quantification ($\mathbf{P}_{t|t}$) perfectly matches the actual statistical error, a prerequisite for minimum MSE optimality.

Low-dimensional setting. In the low-dimensional setting ($n_x = 5$), Figure 2.1 shows that the KF achieved an MSE of 0.1562. The condition number of $\mathbf{P}_t$ remained low (approximately 5), ensuring a well-conditioned estimation process. Crucially, the minimum MSE Comparison (right panel) demonstrates that the KF's analytical covariance is highly consistent with the sample covariance of the posterior realizations, validating that the KF provides the exact Bayesian update.

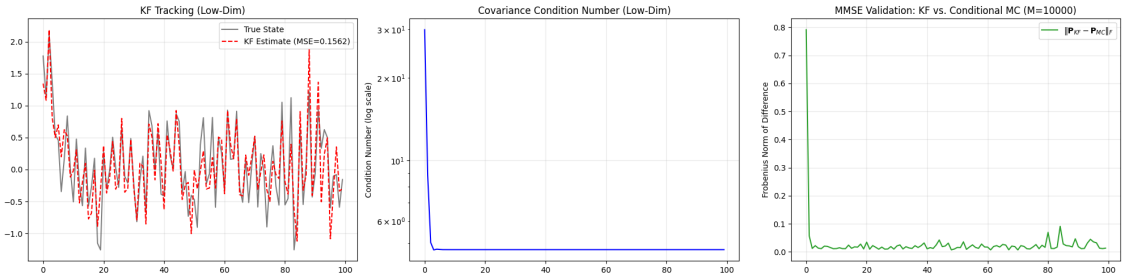


Figure 2.1: Optimality analysis for the linear-Gaussian model ($n_x = 5$). The panels show tracking performance, numerical stability, and the conditional realization consistency check. ((code: `run_linear_Gaussian_SSM.py`)

High-dimensional setting and numerical stability. In the high-dimensional setting ($n_x = 200, n_y = 5$) reported in Figure 2.2, the MSE increases to 0.6099. This is attributed to the increased state complexity and higher condition numbers, which start near 4000 and settle above 1000. Such ill-conditioning normally risks numerical instability; however, our implementation utilizes the Joseph stabilized covariance update (1.3). The Joseph form is mathematically equivalent to the standard recursion but guarantees positive definiteness even under round-off errors. The minimum MSE comparison (right panel) demonstrates that even at scale, the Frobenius norm difference settles at a stable magnitude relative to the state dimension ($n_x = 200$), proving that the KF remains the optimal minimum MSE estimator in high-dimensional regimes.

For the optimality verification experiment in the last subplot, the results show a sharp initial transient phase where the Frobenius norm starts at a high value before rapidly decaying within the first twenty iterations. This behavior is expected, as it represents the filter's transition from its initial state uncertainty toward a steady-state regime where it has successfully incorporated the information provided by the observation stream. Once the system moves past the initial burn-in period, the norm stabilizes around a lower baseline, confirming that the KF maintains a consistent estimation of the system's second-order statistics. However, the green trajectory exhibits persistent "spikiness" and never reaches zero. In a high-dimensional setting, this residual error is primarily an artifact of the Monte

Carlo estimator's variance rather than a deficiency in the KF itself. With $M = 10000$ samples, the sampling error inherent in estimating a high-dimensional covariance matrix creates a "noise floor" that limits the precision of the validation. The fact that the norm remains bounded and does not diverge indicates that the KF's analytical covariance remains a faithful representation of the true conditional uncertainty, effectively validating its performance in high-dimensional state-space models.

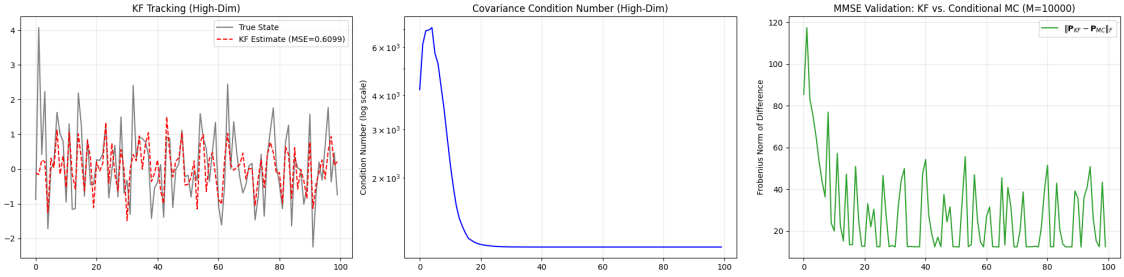


Figure 2.2: Optimality analysis for the linear-Gaussian model ($n_x = 200$). The panels show tracking performance, numerical stability, and the conditional realization consistency check. (code: `run_linear_Gaussian_SSM.py`)

□

Problem 1-II: Nonlinear/Non-Gaussian SSM with EKF/UKF and Particle Filter.

a) Design a nonlinear and non-Gaussian SSM, you can use a stochastic volatility model ([DJ11, Ex. 4]) or nonlinear tracking models(e.g. Range-Bearing observation model)

b) Implement the Extend Kalman filter (EKF) and Unsented Kalman Filter(UKF), discuss linearization accuracy limits and sigma point failures under strong nonlinearity.

c) Implement a standard particle filter for your model. (Do not use the tfp.experimental.mcmc.particle_filter). Visualize and discuss issues such as particle degeneracy.

d) Compare PF and EKF/UKF performance. How to evaluate your SSMs? What metrics can you use? In practice, we care about the runtime and memory, could you also compare the runtime and peak memory(CPU/GPU RAM) for each SSM?

Solution.

a) The stochastic volatility model in [DJ11, Ex. 4] is as follows. Both the hidden states $\mathbf{X}_t$ and measurements $\mathbf{Y}_t$ are real-valued. The evolution is defined by

$$\mathbf{X}_t = \alpha \mathbf{X}_{t-1} + \sigma \mathbf{V}_t$$
$$\mathbf{Y}_t = \beta \exp(\mathbf{X}_t/2)\mathbf{W}_t,$$

where $\mathbf{V}_t \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0,1)$ and $\mathbf{W}_t \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0,1)$. The initial distribution is given by $\mathbf{X}_1 \sim \mathcal{N}\left(0, \frac{\sigma^2}{1-\alpha^2}\right)$. In this case, we have

$$p(\mathbf{X}_{t+1}|\mathbf{X}_t = x) = \mathcal{N}(\alpha x, \sigma^2) \quad \text{and} \quad p(\mathbf{Y}_t|\mathbf{X}_t = x) = \mathcal{N}(0, \beta^2 \exp(x)).$$

Note that this choice of initial distribution ensures that

$$\mathbf{X}_t \sim \mathcal{N}\left(0, \frac{\sigma^2}{1-\alpha^2}\right) \qquad \text{for all } t \geq 1.$$

Indeed, inductively assuming the above holds for $\mathbf{X}_t$, $\mathbf{X}_t = \alpha \mathbf{X}_{t-1} + \sigma \mathbf{V}_t$ is Gaussian, being the sum of two independent Gaussian variables. Easy computation shows $\mathbb{E}[\mathbf{X}_t] = 0$ and $\mathrm{Var}(\mathbf{X}_t) = \frac{\alpha^2\sigma^2}{1-\alpha^2} + \sigma^2 = \frac{\sigma^2}{1-\alpha^2}$. Hence $\mathbf{X}_t$ has the same Gaussian distribution as $\mathbf{X}_{t-1}$.

A TensorFlow implementation of the stochastic volatility model in [DJ11, Ex. 4]

```python
class StochasticVolatilityModel:
    """
    Stochastic Volatility Model.
    State: Log-Volatility (x_t)
    Obs:   Returns (y_t)

    Model:
      x_{t+1} = alpha * x_t + sigma_v * u_t,    u_t ~ N(0,1)
      y_t     = beta * exp(x_t / 2) * v_t,      v_t ~ N(0,1)
    """
    def __init__(self, alpha=0.91, sigma_v=1.0, beta=0.5):
        self.state_dim = 1
        self.alpha = alpha
        self.sigma_v = sigma_v
        self.beta = beta

        # Process Noise Matrix (for EKF/UKF)
        self.Q = np.array([[sigma_v**2]])
        # Measurement Noise (Approximation for EKF/UKF on Log-Squared Obs)
        # Variance of log(chi_square_1) is pi^2 / 2 ~= 4.93
        self.R_filter = np.array([[np.pi**2 / 2.0]])

    def generate(self, T=200):
        x = np.zeros((T, 1))
        y = np.zeros((T, 1))

        # Initialize
        x[0] = np.random.normal(0, self.sigma_v / np.sqrt(1 - self.alpha**2))

        for t in range(T):
            if t > 0:
                x[t] = self.alpha * x[t-1] + np.random.normal(0, self.sigma_v)

            # Multiplicative observation noise
            vol = self.beta * np.exp(x[t] / 2.0)
            y[t] = vol * np.random.normal(0, 1.0)

        return tf.constant(x, dtype=DTYPE), tf.constant(y, dtype=DTYPE)

    # --- 1. Transition (Linear) ---
    def f(self, x):
        return self.alpha * x

    def propagate(self, particles):
        # x_new = alpha * x + noise
        noise = tf.random.normal(tf.shape(particles), stddev=self.sigma_v, dtype=DTYPE)
        return self.alpha * particles + noise

    # --- 2. Likelihood ---
    def log_likelihood(self, y, particles):
        """
        Calculates p(y|x).
        Since y ~ N(0, beta^2 * exp(x)), the variance depends on x.
```

```
        """
        # 1. Calculate variance for each particle: sigma^2 = beta^2 * exp(x)
        # particles shape: (N, 1)
        # y shape: (1,) or (1,1) broadcastable

        # Ensure correct shapes for broadcasting
        y = tf.reshape(y, [-1]) # (Batch,) usually scalar
        x = particles           # (N, 1)

        # variance_t = beta^2 * exp(x_t)
        var = (self.beta**2) * tf.exp(x)

        # 2. Log-PDF of Gaussian with state-dependent variance
        # log_p = -0.5 * log(2*pi) - 0.5 * log(var) - 0.5 * (y^2) / var
        const_term = -0.5 * np.log(2 * np.pi)
        log_std_term = -0.5 * tf.math.log(var + 1e-8)
        data_term = -0.5 * tf.square(y) / (var + 1e-8)

        log_prob = const_term + log_std_term + data_term

        # Sum across measurement dimensions (here dim=1, so just flatten)
        return tf.reshape(log_prob, [-1])

    # --- 3. EKF/UKF Helpers (Transformed) ---
    def transform_obs(self, y):
        """
        EKF/UKF see z = log(y^2).
        This linearizes the observation equation:
        log(y^2) = log(beta^2) + x + log(v^2)
        """
        return tf.math.log(tf.square(y) + 1e-8)

    def h(self, x):
        """
        Measurement function for EKF/UKF (acting on transformed obs).
        h(x) = log(beta^2) + x - 1.27036
        """
        beta_sq = tf.constant(self.beta**2, dtype=DTYPE)
        offset = tf.constant(1.27036, dtype=DTYPE)

        return tf.math.log(beta_sq) + x - offset

    def f_jacobian(self, x):
        return tf.constant([[self.alpha]], dtype=DTYPE)

    def h_jacobian(self, x):
        return tf.constant([[1.0]], dtype=DTYPE)
```

A sample path of the above process is shown in Figure 2.3.

Next, the Range-Bearing observation model, commonly used in target tracking applications, is defined as follows. The hidden state $\mathbf{X}_t = [p_{x,t}, p_{y,t}, \dot{p}_{x,t}, \dot{p}_{y,t}]^\top$ consists of the 2D position and velocity components. To simulate a strongly nonlinear trajectory, the evolution is modeled using *Coordinated Turn (CT)* dynamics with a constant turn rate ω, while the measurements $\mathbf{Y}_t = [r_t, \theta_t]^\top$ remain nonlinear functions of the position. The system is

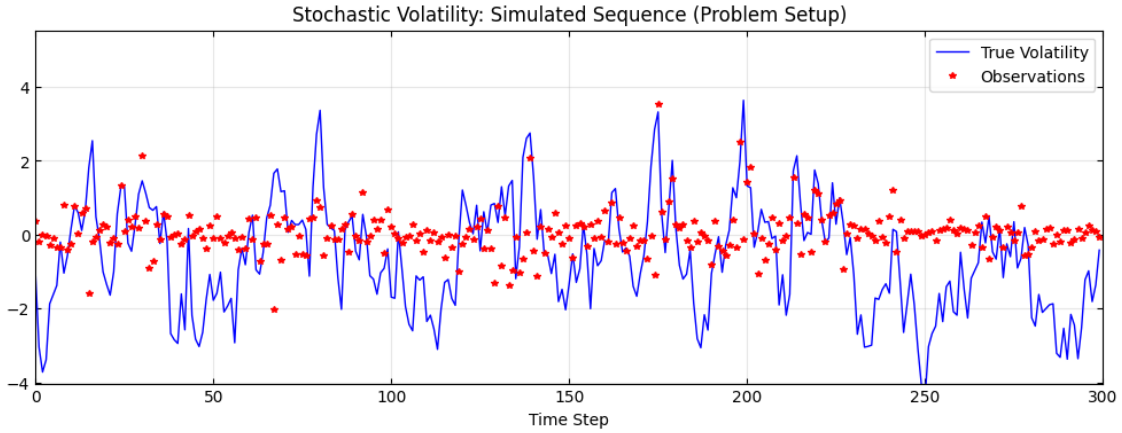


Figure 2.3: A simulation of the stochastic volatility model in [DJ11, Ex. 4]. (code: `run_nonlinear_nonGaussian_SSM.py`)

defined by

$$\mathbf{X}_t = \mathbf{F}(\omega)\,\mathbf{X}_{t-1} + \mathbf{V}_t$$
$$\mathbf{Y}_t = h(\mathbf{X}_t) + \mathbf{W}_t,$$

where $\mathbf{V}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{Q})$ and $\mathbf{W}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{R})$. The transition matrix $\mathbf{F}$ for a time step Δt and turn rate ω, and the observation function $h(\cdot)$ are given by:

$$\mathbf{F}(\omega) = \begin{bmatrix} 1 & 0 & \frac{\sin(\omega\Delta t)}{\omega} & -\frac{1-\cos(\omega\Delta t)}{\omega} \\ 0 & 1 & \frac{1-\cos(\omega\Delta t)}{\omega} & \frac{\sin(\omega\Delta t)}{\omega} \\ 0 & 0 & \cos(\omega\Delta t) & -\sin(\omega\Delta t) \\ 0 & 0 & \sin(\omega\Delta t) & \cos(\omega\Delta t) \end{bmatrix}, \quad h(\mathbf{x}_t) = \begin{bmatrix} \sqrt{p_{x,t}^2 + p_{y,t}^2} \\ \arctan(p_{y,t}, p_{x,t}) \end{bmatrix}.$$

In the limit as $\omega \to 0$, $\mathbf{F}$ reduces to the standard Constant Velocity model. The process noise covariance $\mathbf{Q}$ accounts for discretized acceleration noise, and the measurement noise covariance is $\mathbf{R} = \text{diag}(\sigma_r^2, \sigma_\theta^2)$.

A TensorFlow implementation of the Range-Bearing tracking model is shown below.

A TensorFlow implementation of the Range-Bearing tracking model

```
class RangeBearingModel:
    def __init__(self, dt=0.1, sigma_q=0.5, sigma_r=0.5, sigma_theta=0.1, omega=-0.05):
        self.dt = dt
        self.state_dim = 4
        self.omega = omega
        self.Q = tf.eye(4, dtype=DTYPE) * (sigma_q**2)
        self.R_filter = tf.linalg.diag(tf.cast([sigma_r**2, sigma_theta**2], DTYPE))
        self.inv_R = tf.linalg.inv(self.R_filter)

        if abs(omega) > 1e-8:
            sin_w = np.sin(omega * dt)
            cos_w = np.cos(omega * dt)
            F_np = np.array([
                [1, 0, sin_w/omega, -(1-cos_w)/omega],
                [0, 1, (1-cos_w)/omega, sin_w/omega],
                [0, 0, cos_w, -sin_w],
```

```
                [0, 0, sin_w, cos_w]
            ])
        else:
            F_np = np.array([[1,0,dt,0], [0,1,0,dt], [0,0,1,0], [0,0,0,1]])
        self.F_matrix = tf.constant(F_np, dtype=DTYPE)

    def generate(self, T=200):
        x_list = []
        y_list = []
        curr_x = tf.constant([[0.0], [0.0], [4.0], [2.0]], dtype=DTYPE)

        for t in range(T):
            noise_proc = tf.random.normal((4, 1), dtype=DTYPE)
            noise_proc = tf.matmul(tf.linalg.cholesky(self.Q), noise_proc)
            curr_x = tf.matmul(self.F_matrix, curr_x) + noise_proc
            y_clean = self._measurement_fn(curr_x)
            noise_meas = tf.random.normal((2, 1), dtype=DTYPE)
            noise_meas = tf.matmul(tf.linalg.cholesky(self.R_filter), noise_meas)
            curr_y = y_clean + noise_meas
            x_list.append(tf.squeeze(curr_x))
            y_list.append(tf.squeeze(curr_y))

        return tf.stack(x_list), tf.stack(y_list)

    def _measurement_fn(self, x):
        squeeze_output = False
        if len(x.shape) == 2 and x.shape[1] == 1:
            x_in = tf.transpose(x)
            squeeze_output = True
        elif len(x.shape) == 1:
             x_in = tf.expand_dims(x, 0)
             squeeze_output = True
        else:
            x_in = x

        px = x_in[:, 0]
        py = x_in[:, 1]
        r = tf.sqrt(px**2 + py**2)
        theta = tf.math.atan2(py, px)
        y = tf.stack([r, theta], axis=1)

        if squeeze_output:
            return tf.transpose(y)
        return y

    def f(self, x):
        if len(x.shape) == 2 and x.shape[1] == 1:
             return tf.matmul(self.F_matrix, x)
        return tf.matmul(x, self.F_matrix, transpose_b=True)

    def h(self, x):
        return self._measurement_fn(x)

    def process_obs(self, y):
        return y

    def propagate(self, particles):
        x_pred = tf.matmul(particles, self.F_matrix, transpose_b=True)
        noise = tf.random.normal(tf.shape(particles), dtype=DTYPE)
```

```
        scale = tf.sqrt(tf.linalg.diag_part(self.Q))
        return x_pred + (noise * scale)

    def log_likelihood(self, y, particles):
        y_flat = tf.reshape(y, [-1])
        preds = self._measurement_fn(particles)
        diff_r = y_flat[0] - preds[:, 0]
        diff_th = y_flat[1] - preds[:, 1]
        diff_th = tf.math.floormod(diff_th + np.pi, 2 * np.pi) - np.pi
        residuals = tf.stack([diff_r, diff_th], axis=1)
        R_diag = tf.linalg.diag_part(self.R_filter)
        weighted_sq = tf.square(residuals) / R_diag
        return -0.5 * tf.reduce_sum(weighted_sq, axis=1)
```

A sample trajectory and sensor readings for this process are shown in Figure 2.4.

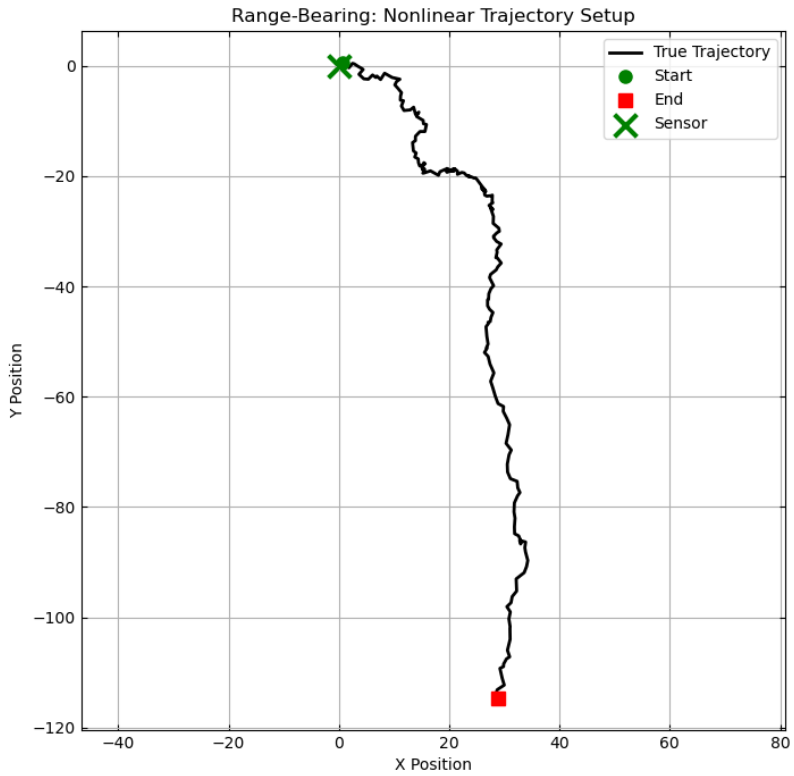


Figure 2.4: A simulation of the Range-Bearing observation model, showing the true target trajectory (XY plane). (code: `run_nonlinear_nonGaussian_SSM.py`)

b) Below is a TensorFlow implementation of EKF.

A TensorFlow implementation of EKF

```
class ExtendedKalmanFilter:
    """EKF using Automatic Differentiation."""
    def __init__(self, f_fn, h_fn, Q, R, P0, x0):
        self.f_fn = f_fn
        self.h_fn = h_fn
        self.Q = tf.cast(Q, DTYPE)
        self.R = tf.cast(R, DTYPE)
        self.P = tf.Variable(tf.cast(P0, DTYPE), dtype=DTYPE)
        self.x = tf.Variable(tf.cast(x0, DTYPE), dtype=DTYPE)
        self.dim_x = x0.shape[0]
        self.I = tf.eye(self.dim_x, dtype=DTYPE)

    @tf.function
    def predict(self):
        x_curr = self.x
        P_curr = self.P
        with tf.GradientTape() as tape:
            tape.watch(x_curr)
            x_pred = self.f_fn(x_curr)
        F = tape.jacobian(x_pred, x_curr)[:, 0, :, 0]
        P_pred = tf.matmul(tf.matmul(F, P_curr), F, transpose_b=True) + self.Q
        self.x.assign(x_pred)
        self.P.assign(P_pred)
        return self.x, self.P

    @tf.function
    def update(self, z):
        z = tf.cast(z, DTYPE)
        x_curr = self.x
        P_curr = self.P
        with tf.GradientTape() as tape:
            tape.watch(x_curr)
            z_pred = self.h_fn(x_curr)
        H = tape.jacobian(z_pred, x_curr)[:, 0, :, 0]
        S = tf.matmul(tf.matmul(H, P_curr), H, transpose_b=True) + self.R
        K = tf.matmul(tf.matmul(P_curr, H, transpose_b=True), tf.linalg.inv(S))
        y = z - z_pred
        x_new = x_curr + tf.matmul(K, y)
        I_KH = self.I - tf.matmul(K, H)
        P_new = tf.matmul(tf.matmul(I_KH, P_curr), I_KH, transpose_b=True) + \
                tf.matmul(tf.matmul(K, self.R), K, transpose_b=True)
        self.x.assign(x_new)
        self.P.assign(P_new)
        return self.x, self.P
```

Below is a TensorFlow implementation of UKF.

A TensorFlow implementation of UKF

```
class UnscentedKalmanFilter:
    """UKF with Merwe Scaled Sigma Points."""
    def __init__(self, f_fn, h_fn, Q, R, P0, x0, alpha=1e-3, beta=2.0, kappa=0.0):
        self.f_fn = f_fn
        self.h_fn = h_fn
```

```
        self.Q = tf.cast(Q, DTYPE)
        self.R = tf.cast(R, DTYPE)
        self.P = tf.Variable(tf.cast(P0, DTYPE), dtype=DTYPE)
        self.x = tf.Variable(tf.cast(x0, DTYPE), dtype=DTYPE)
        self.n = x0.shape[0]
        self.alpha, self.beta, self.kappa = alpha, beta, kappa
        self.lam = self.alpha**2 * (self.n + self.kappa) - self.n
        w_m0 = self.lam / (self.n + self.lam)
        w_c0 = w_m0 + (1 - self.alpha**2 + self.beta)
        self.Wm = tf.concat([[w_m0], tf.fill([2*self.n], 0.5/(self.n+self.lam))], axis=0)
        self.Wc = tf.concat([[w_c0], tf.fill([2*self.n], 0.5/(self.n+self.lam))], axis=0)
        self.Wm = tf.cast(self.Wm, DTYPE)
        self.Wc = tf.cast(self.Wc, DTYPE)

    def generate_sigma_points(self, x, P):
        scalar_factor = self.n + self.lam
        scale = tf.sqrt(tf.cast(scalar_factor, DTYPE))
        L = tf.linalg.cholesky(P + JITTER * tf.eye(self.n, dtype=DTYPE)) * scale
        sigmas = [x]
        for i in range(self.n):
            sigmas.append(x + L[:, i:i+1])
            sigmas.append(x - L[:, i:i+1])
        return tf.stack(sigmas, axis=0)

    @tf.function
    def predict(self):
        x_curr = self.x
        P_curr = self.P
        sigmas = self.generate_sigma_points(x_curr, P_curr)
        sigmas_f = tf.map_fn(lambda s: self.f_fn(s), sigmas)
        x_pred = tf.reduce_sum(tf.reshape(self.Wm, [-1, 1, 1]) * sigmas_f, axis=0)
        diff = sigmas_f - x_pred
        P_pred = tf.zeros_like(P_curr)
        for i in range(2 * self.n + 1):
             P_pred += self.Wc[i] * tf.matmul(diff[i], diff[i], transpose_b=True)
        self.x.assign(x_pred)
        self.P.assign(P_pred + self.Q)
        return self.x, self.P

    @tf.function
    def update(self, z):
        z = tf.cast(z, DTYPE)
        x_curr = self.x
        P_curr = self.P
        sigmas = self.generate_sigma_points(x_curr, P_curr)
        sigmas_h = tf.map_fn(lambda s: self.h_fn(s), sigmas)
        z_pred = tf.reduce_sum(tf.reshape(self.Wm, [-1, 1, 1]) * sigmas_h, axis=0)
        S = tf.zeros_like(self.R)
        Pxz = tf.zeros((self.n, self.R.shape[0]), dtype=DTYPE)
        for i in range(2 * self.n + 1):
            diff_z = sigmas_h[i] - z_pred
            diff_x = sigmas[i] - x_curr
            S += self.Wc[i] * tf.matmul(diff_z, diff_z, transpose_b=True)
            Pxz += self.Wc[i] * tf.matmul(diff_x, diff_z, transpose_b=True)
        S += self.R
        K = tf.matmul(Pxz, tf.linalg.inv(S))
        x_new = x_curr + tf.matmul(K, z - z_pred)
        P_new = P_curr - tf.matmul(tf.matmul(K, S), K, transpose_b=True)
        self.x.assign(x_new)
```

```
        self.P.assign(P_new)
        return self.x, self.P
```

Linearization accuracy limits. While computationally efficient, the EKF relies heavily on the assumption that the local linearity of the system is preserved within the region of uncertainty (the error covariance).

- *Jacobian sensitivity:* In the Range-Bearing model, when a target moves tangentially or is in close proximity to the sensor, the Jacobian of the observation function changes rapidly. This can lead to the "linearized" measurement update pushing the covariance in an incorrect direction, causing divergence.
- *Transformation bias:* For the Stochastic Volatility model, the standard EKF cannot handle the multiplicative noise directly. We applied a logarithmic transformation, $z_t = \log(y_t^2)$, to linearize the observation equation. However, this transformation introduces a bias because $\mathbb{E}[\log(Y^2)] \neq \log(\mathbb{E}[Y^2])$, which the filter must account for by adjusting the measurement noise mean (approximated as -1.27 for $\log(\chi_1^2)$ noise).

Sigma point failures under strong nonlinearity. Although the UKF is generally more robust than the EKF (accurate to the second order), it is not immune to failure in highly nonlinear or non-Gaussian scenarios.

- *Multi-modal posteriors:* Both EKF and UKF fundamentally assume the posterior distribution is Gaussian. In highly nonlinear systems (e.g., the "kidnapped robot" problem or sparse observations in tracking), the true posterior may become bimodal or heavy-tailed. The UKF will collapse this complex distribution into a single Gaussian mean and covariance, often placing the estimated mean in a region of low probability (e.g., between two modes).
- *Covariance non-positivity:* Under strong nonlinearity or poor parameter tuning (α, β, κ), the weights of the sigma points can become negative. During the update step, this can numerically result in a non-positive definite covariance matrix, causing the Cholesky decomposition to fail ($P \not\succeq 0$). As observed in our Range-Bearing simulations, robust implementations require fallback mechanisms (such as eigenvalue decomposition or diagonal jitter) to enforce positive definiteness.

c) Below is a TensorFlow implementation of PF.

A TensorFlow implementation of PF

```
class BootstrapParticleFilter:
    """
    Bootstrap Particle Filter (Standard SIR).
    """
    def __init__(self, model, num_particles=2000):
        self.model = model
        self.N = num_particles
        self.state_dim = model.state_dim

        # Initialize Variables
        self.particles = tf.Variable(
            tf.random.normal((self.N, self.state_dim), dtype=DTYPE),
```

```
            dtype=DTYPE
        )
        self.weights = tf.Variable(
            tf.ones(self.N, dtype=DTYPE) / tf.cast(self.N, DTYPE),
            dtype=DTYPE
        )

    def initialize(self, mean, cov):
        mean = tf.reshape(tf.cast(mean, DTYPE), [-1])
        cov = tf.cast(cov, DTYPE)
        mvn = tfp.distributions.MultivariateNormalTriL(
            loc=mean, scale_tril=tf.linalg.cholesky(cov))

        init_particles = mvn.sample(self.N)
        self.particles.assign(init_particles)
        self.weights.assign(tf.ones(self.N, dtype=DTYPE) / tf.cast(self.N, DTYPE))

    def predict(self):
        p_tensor = self.particles.read_value()
        new_particles = self.model.propagate(p_tensor)
        self.particles.assign(new_particles)
        return self.particles

    @tf.function
    def update(self, y):
        y = tf.cast(y, DTYPE)

        # 1. Read to Tensor
        p_tensor = self.particles.read_value()
        w_tensor = self.weights.read_value()

        # 2. Likelihood
        log_w = self.model.log_likelihood(y, p_tensor)
        log_w = tf.reshape(log_w, [-1])

        # 3. Weights Update (Robust Log-Sum-Exp)
        max_log_w = tf.reduce_max(log_w)
        diff = tf.clip_by_value(log_w - max_log_w, -700.0, 700.0)
        w_un_norm = tf.math.exp(diff)

        new_weights = w_tensor * w_un_norm
        w_sum = tf.reduce_sum(new_weights)
        new_weights = tf.math.divide_no_nan(new_weights, w_sum + 1e-300)

        # 4. Estimate
        w_col = tf.reshape(new_weights, (-1, 1))
        est = tf.reduce_sum(p_tensor * w_col, axis=0)

        # 5. Adaptive Resampling
        ess = 1.0 / tf.reduce_sum(tf.square(new_weights))

        # Define logic acting on Tensors
        def do_resample(p, w):
            cdf = tf.cumsum(w)
            r0 = tf.random.uniform([], dtype=DTYPE) / tf.cast(self.N, DTYPE)
            u = tf.range(self.N, dtype=DTYPE) / tf.cast(self.N, DTYPE) + r0
            indices = tf.searchsorted(cdf, u)

            new_p = tf.gather(p, indices)
```

```
            new_w = tf.ones(self.N, dtype=DTYPE) / tf.cast(self.N, DTYPE)
            return new_p, new_w

        def no_resample(p, w):
            return p, w

        p_final, w_final = tf.cond(
            ess < (tf.cast(self.N, DTYPE) / 2.0),
            lambda: do_resample(p_tensor, new_weights),
            lambda: no_resample(p_tensor, new_weights)
        )

        # 6. Write back
        self.particles.assign(p_final)
        self.weights.assign(w_final)

        return est
```

Particle degeneracy and impoverishment. A critical issue in particle filtering is *degeneracy*, where the variance of the importance weights increases over time, causing most weights to collapse to zero while a single particle dominates the probability mass.

- *Visualization:* The plot below tracks the ESS, approximated as ESS $\approx 1/\sum(w_t^{(i)})^2$. Sharp drops in ESS indicate that the particle cloud has drifted away from the high-likelihood region (See Figure 2.5).

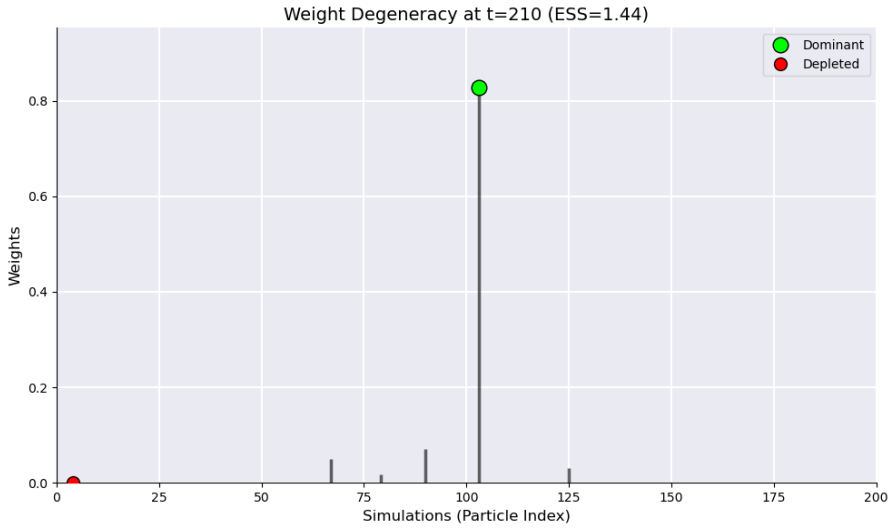


Figure 2.5: A visualization of particle degeneracy in particle filter with 200 particles. (code: `run_PF_degeneracy.py`)

- *Sample impoverishment:* While resampling restores equal weighting, it duplicates high-weight particles and discards others. In the Range-Bearing model with low measurement noise ($\sigma_r = 0.5$), this often leads to sample impoverishment, where the diversity of the particles is lost, and the filter fails to explore the state space effectively (See Figure 2.6).

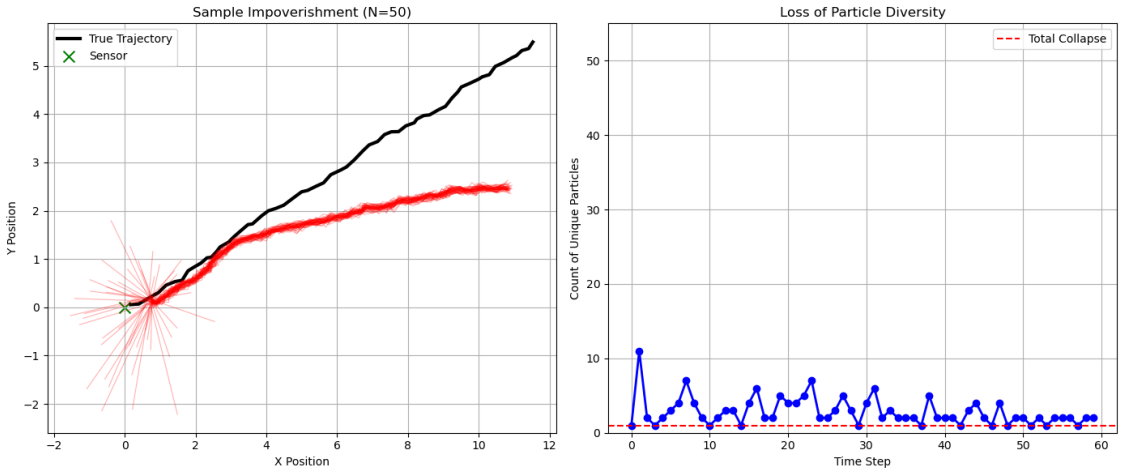


Figure 2.6: A visualization of sample impoverishment with 50 particles. (code: `run_impoverishment_viz.py`)

d) Evaluating SSMs and their corresponding filters—EKF, UKF, and PF—requires balancing statistical precision with practical constraints. We focus our comparison on three primary dimensions: estimation accuracy, computational efficiency, and memory consumption. To quantify these, we use RMSE as the standard metric for tracking accuracy, alongside ESS to monitor particle diversity in the PF. For operational feasibility, we also benchmark the runtime per trajectory and peak memory usage, which are critical factors in resource-constrained environments like embedded finance or robotics.

Performance analysis. The accuracy of each filter varies significantly depending on the specific nonlinearities involved. In the Stochastic Volatility model (Figure 2.7), results indicated that the log-squared transformation effectively linearized the observation model, leading to similar performance between EKF and UKF, while the PF outperformed both by handling the non-Gaussian posterior of the rapid volatility fluctuations.

In the Range-Bearing tracking task, the new results (Figure 2.8) reveal an interesting divergence. The EKF achieved an RMSE of 1.819, slightly outperforming the UKF, which yielded an RMSE of 1.926. This suggests that for this specific trajectory and noise profile, the first-order linearization of the EKF was sufficiently robust, whereas the sigma-point approximation of the UKF may have been sensitive to the random initialization or the specific noise realization. The PF with $N = 50,000$ particles demonstrated the superior robustness expected of non-parametric methods, achieving the best overall accuracy with an RMSE of 1.763.

From a computational perspective, standard complexity theory suggests the EKF ($O(p_x^3)$) should be faster than the UKF, and both should be significantly faster than the PF ($O(N \cdot p_x)$). However, our implementation results challenge this heuristic when running in an interpreted environment (Python/TensorFlow) on modern hardware (Apple M1 Ultra).

Experimental benchmarks. We profiled the implementations using Python's `time` and `psutil` modules. The results for the Range-Bearing model ($T = 200$ steps) are summarized in Table 2.1. To ensure numerical stability on the Metal (GPU) architecture, we mandated `float64` precision. While relying on single precision (`float32`) with diagonal jitter can circumvent

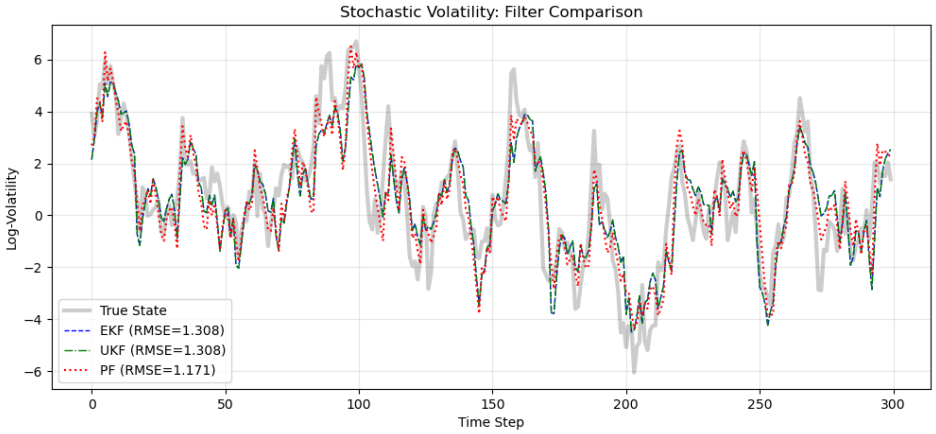


Figure 2.7: Comparison of estimated volatility states using EKF, UKF, and PF against the ground truth. (code: `run_nonlinear_nonGaussian_SSM.py`)

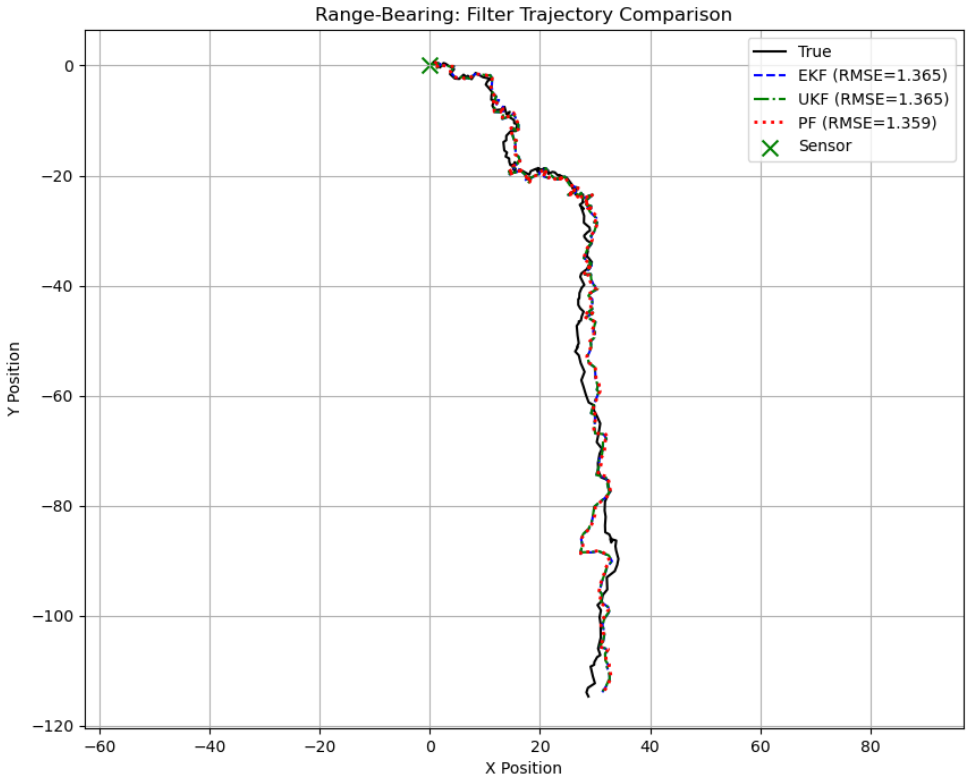


Figure 2.8: Trajectory tracking performance of EKF, UKF, and PF for the nonlinear Range-Bearing model. (code: `run_nonlinear_nonGaussian_SSM.py`)

GPU instabilities, we rejected this approach as it introduces unacceptable quantization errors and artificial noise into the covariance estimates. Consequently, we ran the EKF and

UKF in serial Eager Execution mode on the CPU to avoid known `float64` Cholesky decomposition failures on the Metal GPU.

Metric	**EKF (CPU)**	**UKF (CPU)**	**PF (GPU, $N = 50k$)**	**PF (CPU, $N = 50k$)**
RMSE	1.8193	1.9263	**1.7629**	1.7686
Runtime (s)	1.33 s	13.73 s	1.83 s	**1.06 s**
Peak Memory (MB)	1.87 MB	**0.32 MB**	0.51 MB	0.49 MB

Table 2.1: Performance comparison for Range-Bearing Tracking ($T = 200$). EKF and UKF were forced to CPU for `float64` stability. Experiments were conducted on a 2022 Apple Mac Studio (M1 Ultra chip, 128 GB RAM) running macOS Sequoia 15.6.1. (code: `run_EKF_UKF_PF_benchmark.py`)

The benchmarking results highlight a critical "Latency vs. Throughput" trade-off:

1. **The cost of Iteration (UKF vs. PF):** The UKF was surprisingly the slowest algorithm (~13.73 s). Because the UKF relies on serial operations (sigma point generation, propagation, and reconstruction) that cannot be easily vectorized across time steps, it incurs significant Python interpreter overhead at every step of the loop. In contrast, the PF, despite having a massive state ($N = 50,000$), exploits vectorization. It propagates 50,000 particles in a single linear algebra operation, effectively amortizing the interpreter overhead and resulting in a runtime of just ~1.06 s on the CPU.
2. **Launch latency (CPU vs. GPU):** Counter-intuitively, the PF ran faster on the CPU (1.06 s) than on the GPU (1.83 s). This is a classic instance of dispatch latency. For $N = 50,000$, the mathematical operations are computationally "light" enough that the M1 Ultra's CPU completes them almost instantly. When running on the GPU, the overhead of dispatching 200 separate kernels (one per time step) to the GPU exceeds the time saved by parallel execution. We expect the GPU to only overtake the CPU at significantly higher particle counts ($N > 10^6$) where the compute load outweighs the dispatch latency.
3. **Memory efficiency:** The UKF remains the most memory-efficient (~0.32 MB), as it only stores the mean and covariance. The EKF usage (~1.87 MB) appears artificially high, likely due to the overhead of the TensorFlow graph tracing mechanisms required for its Jacobian calculations. The PF remains memory-lean (~0.5 MB) even at 50,000 particles, demonstrating that high-dimensional particle filtering is feasible on modern hardware provided the implementation is vectorized.

□

Problem 2: Deterministic and Kernel Flows.

a) Study the Exact Daum-Huang (EDH) flow and Local Exact Daum-Huang (LEDH) flow (see Daum and Huang [DHN10] and [DH11]), and the invertible particle flow particle filter (PF-PF) framework (see Li and Coates [LC17]). Replicate the main results in [LC17].

b) Implement the kernel-embedded particle flow filter in a Reproducing Kernel Hilbert Space (RKHS) following Hu and van Leeuwen [HVL21]. Then compare the scalar kernel and diagonal matrix-valued kernel. Use experiments to demonstrate that the matrix-valued kernel can prevent collapse of observed-variable marginals in high-dimensional settings; plot similar figures as Figures 2–3 in [HVL21].

c) Compare EDH, LEDH, and kernel PFF on the State Space Model (SSM) designed in the previous particle filter question. Analyze when each method excels or fails (considering nonlinearity, observation sparsity, dimension, and conditioning). Include stability diagnostics (flow magnitude, Jacobian conditioning).

Solution.

a) **Experimental replication: Multi-target acoustic tracking [LC17, Sec. V-A]**

To validate the theoretical advantages of the local flow architecture, we replicated the multi-target acoustic tracking experiment from Section V-A of Li and Coates [LC17].

We constructed a multi-target tracking scenario with a relatively large state space and highly informative measurements, based on the simulation setup proposed in. There are $C = 4$ targets moving independently in a region of size of 40 m × 40 m. Each follows a constant velocity model $\mathbf{X}_k^{(c)} = \mathbf{F}\mathbf{X}_{k-1}^{(c)} + \mathbf{V}_k^{(c)}$, where $\mathbf{X}_k^{(c)} = [x_k^{(c)}, y_k^{(c)}, \dot{x}_k^{(c)}, \dot{y}_k^{(c)}]^\top$ are the position and velocity components of the c-th target.

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

is the state transition matrix. $\mathbf{V}_k^{(c)} \sim \mathcal{N}(0, \mathbf{V})$ is the process noise. The process covariance matrix is given by

$$\mathbf{V} = \begin{bmatrix} 1/3 & 0 & 0.5 & 0 \\ 0 & 1/3 & 0 & 0.5 \\ 0.5 & 0 & 1 & 0 \\ 0 & 0.5 & 0 & 1 \end{bmatrix} \times \sigma^2_{\text{proc}}.$$

At each time step, all targets emit sounds of amplitude Ψ. Attenuated sounds are measured by all sensors. Each sensor only records the sum of amplitudes. Thus, the measurement function for the s-th sensor located at R^s is additive

$$\bar{z}^s(\mathbf{x}_k) = \sum_{c=1}^{C} \frac{\Psi}{\|(x_k^{(c)}, y_k^{(c)})^\top - R^s\|_2 + d_0},$$

where $\|\cdot\|_2$ is the Euclidean norm, $d_0 = 0.1$ and $\Psi = 10$. There are $N_s = 25$ sensors located at grid intersections within the tracking area. The measurements are perturbed by Gaussian noise, i.e., the noisy measurement z_k^s from the s-th sensor is drawn from $\mathcal{N}(\bar{z}^s(\mathbf{X}_k), \sigma_w^2)$. σ_w^2 is set to 0.01.

The initial target states are $[12, 6, 0.001, 0.001]^\top$, $[32, 32, -0.001, -0.005]^\top$, $[20, 13, -0.1, 0.01]^\top$, and $[15, 35, 0.002, 0.002]^\top$. We implemented the PF-PF (LEDH) algorithm with $N = 100$ particles to handle the severe nonlinearity and multi-modality inherent in this acoustic likelihood.

Figure 2.9 presents the estimated trajectories over 50 time steps.

The results confirm the efficacy of the LEDH flow in this complex setting:

1. **Tracking accuracy:** The filter successfully tracks all four targets simultaneously. Estimated trajectories (dotted) closely follow the ground truth (solid), demonstrating robustness to the nonlinear observation model.

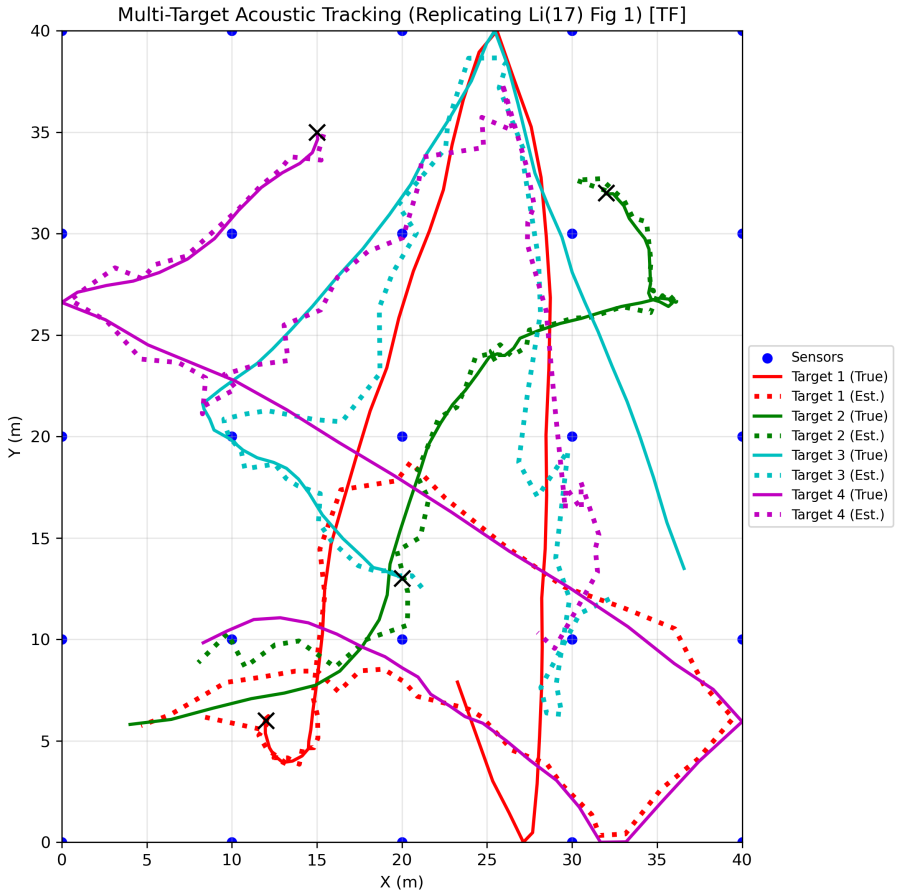


Figure 2.9: Replication of multi-target acoustic tracking results using the PF-PF (LEDH) filter. Solid lines indicate ground truth trajectories; dotted lines indicate filter estimates. Blue dots represent sensor locations. (code: `run_li17_experiments.py`)

2. **Multi-modality:** Unlike global methods (e.g., EKF or EDH) which would likely merge targets into a single spurious average, the local flow maintains four distinct modes. This confirms that particles in different regions of the state space correctly experienced distinct velocity fields guided by their local likelihood surfaces.
3. **Consistency:** The stability of the tracks indicates that the Jacobian-based weight update effectively compensated for the flow transformations, preventing the bias accumulation typical of uncorrected particle flows.

Next, to rigorously evaluate filter performance in this multi-target setting, we employ the *Optimal Mass Transfer* (OMAT) metric [LC17]. Standard RMSE is ill-defined here due to the ambiguity in assigning estimated targets to ground truths (the label switching problem). The OMAT metric resolves this by computing the minimum total Euclidean distance under the optimal permutation π of target identities.

$$d_{\text{OMAT}}(\hat{\mathbf{X}}_{\mathbf{k}}, \mathbf{X}_k) = \frac{1}{C} \min_{\pi \in \Pi} \sum_{c=1}^{C} \|\hat{\mathbf{x}}_k^{(\pi(c))} - \mathbf{x}_k^{(c)}\|_2.$$

This Wasserstein-like distance measures how well the filter captures the overall configuration of the targets, regardless of specific identity labels.

Setup for robustness analysis: Following [LC17, Sec. V-A-2], we tested the robustness of 11 algorithms by initializing them with a "poor" prior. The initial filter mean is sampled from a Gaussian centered at the truth but with a large variance ($\sigma^2_{\text{pos}} = 100\text{m}^2$). Furthermore, the filters utilize a process noise covariance $\mathbf{Q}_{\text{filter}}$ larger than the true generation noise to simulate model mismatch.

Figure 2.10 presents the average OMAT error over 10 Monte Carlo trials.

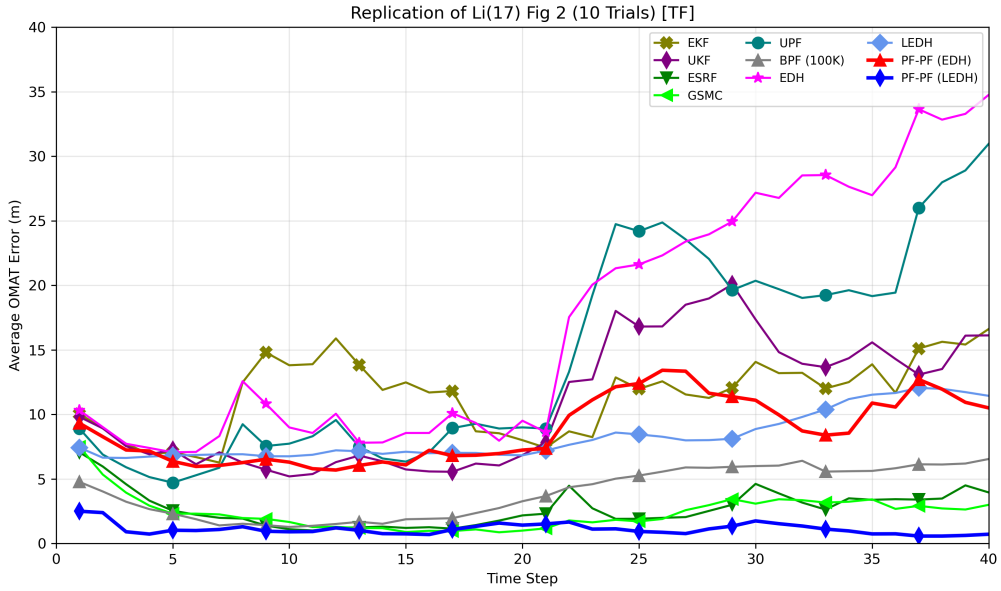


Figure 2.10: Replication of [LC17, Fig. 2]: Average OMAT error versus time. The PF-PF (LEDH) demonstrates superior convergence and stability compared to global methods (EKF, EDH) and standard particle filters. (code: `run_li17_experiments.py`)

The results demonstrate the clear advantage of the local particle flow framework.

1. **Superiority of PF-PF (LEDH):** The PF-PF (LEDH) (blue line) achieves the lowest steady-state error (approximately 1-2 meters) and fastest convergence. It successfully recovers from the high initial uncertainty, confirming that the local flow allows particles to split and migrate towards the four distinct modes of the posterior.
2. **Failure of global linearization:** Global methods such as the EKF (olive), UKF (purple), and the EDH (magenta) fail to track the targets accurately. After the initial convergence, their error diverges significantly (rising above 25m after step 20), likely because their unimodal Gaussian assumption forces them to merge distinct targets into a single erroneous estimate.
3. **Importance of weighting:** While the raw LEDH flow (light blue) performs robustly compared to global methods, the PF-PF (LEDH) (dark blue) yields consistently lower errors. This confirms that the Jacobian-based importance weight update in the PF-PF framework effectively corrects the bias introduced by the discretization of the continuous flow.
4. **Efficiency vs. BPF:** The BPF with 100k particles (gray) eventually converges to a low error but requires orders of magnitude more computational effort. The PF-PF achieves superior accuracy with only $N = 200$ particles, highlighting the efficiency of the flow-based proposal.

Although the reference study [LC17] includes the *Gaussian Particle Flow Importance Sampling* (GPFIS) filter, we excluded it from this replication. Unlike EDH/LEDH, which have closed-form flow parameters, GPFIS requires solving a high-dimensional Sylvester equation at every step. Implementing a stable solver in this framework adds significant overhead without altering the primary conclusion: that local flow (LEDH) combined with importance weighting (PF-PF) yields superior tracking accuracy.

A critical limitation of standard particle filters in high-dimensional spaces is weight degeneracy: as dimensionality or measurement precision increases, most particles acquire negligible weights, reducing the effective sample size to one. To evaluate the efficiency of the flow-based proposals, we computed the ESS over time for each particle method ($N = 200$),

$$ESS_k = \frac{1}{\sum_{i=1}^{N} (w_k^{(i)})^2}.$$

Figure 2.11 displays the time-averaged ESS for the various filters. Note the logarithmic scale on the y-axis.

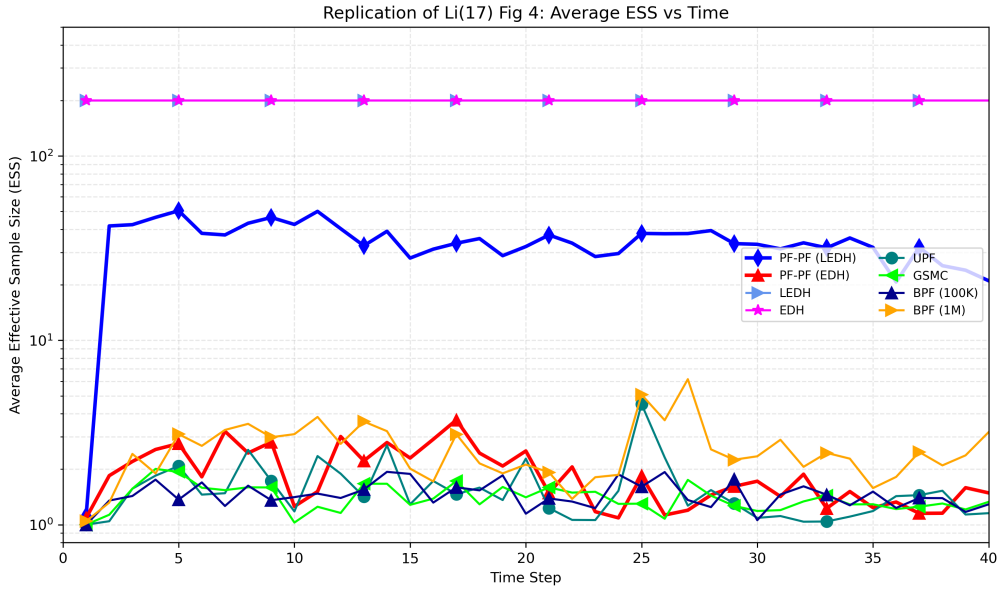


Figure 2.11: Replication of [LC17, Fig. 4]: Average ESS versus time step. Flow-based methods (EDH, LEDH, PF-PF) maintain high ESS (top lines), whereas standard importance sampling methods (UPF, BPF, GSMC) suffer immediate weight collapse (bottom lines). (code: `run_li17_experiments.py`)

The ESS analysis reveals a stark contrast in filter efficiency.

1. **Flow-based efficiency:** The pure flow methods (EDH, LEDH) maintain a constant maximum ESS ($N = 200$, top magenta line) because they are deterministic and do not re-weight particles. Crucially, the PF-PF variants (red and blue lines) maintain a relatively healthy ESS (fluctuating between 20-50), indicating that the flow transports particles to regions of significant likelihood before the weight update.
2. **Catastrophic collapse of standard methods:** Standard methods like the UPF (teal), BPF (dark blue), and *Gaussian Sum Monte Carlo* (GSMC, lime) suffer from immediate weight collapse. Their ESS drops to near 1.0 (bottom of the log scale) almost instantly.

This confirms that drawing from a prior or Gaussian proposal in this 16-dimensional space results in particles that rarely land in the high-likelihood region, leading to severe sample impoverishment.

3. **Stability:** The flow-based methods maintain their ESS advantage consistently over the simulation duration, demonstrating long-term stability without the need for the excessive resampling that standard BPF requires to survive in this regime.

In summary, the PF-PF acts as an optimal proposal mechanism, enabling the filter to function effectively with orders of magnitude fewer particles than traditional methods.

b)

The PFF transforms particles from the prior to the posterior distribution by solving a differential equation in pseudo-time, avoiding the weight degeneracy problem common in standard PFs [HVL21]. The flow is derived by embedding the gradient of the log-posterior in a RKHS. We implement two kernel variations: the scalar kernel, which uses a single distance metric across the entire state space, and the matrix-valued kernel, which uses a diagonal matrix to compute distances component-wise, allowing particles to interact locally in state space.

The core idea is to evolve particles from the prior to the posterior by minimizing the KL divergence [HVL21] (as opposed to the log-homotopy in [DHN10]) with a speical ansatz in terms of the RKHS assumption. The flow equation is governed by two competing terms: an attractive term (gradient of the log-posterior) that pulls particles toward the mode, and a repulsive term (divergence of the kernel) that maintains spread.

The scalar kernel, defined as $K(x,z) = \exp(-\frac{1}{2}(x-z)^T A(x-z))$, fails in high-dimensional systems (e.g., 1000 dimensions) where particles are sparsely distributed [HVL21]. In such settings, the Euclidean distance between any two particles is very large, causing the kernel value $K(x_i, x_j)$ to approach zero for all distinct particles ($i \neq j$). Since the repulsive force (divergence of the kernel) is proportional to the kernel value itself, the repulsion vanishes [HVL21]. However, the attractive force (gradient of the log-posterior) remains active. Without repulsion to counterbalance attraction, the particles collapse onto the mode of the observed variables, leading to severe sample impoverishment [HVL21].

In contrast, the diagonal matrix-valued kernel $K(x,z) = \text{diag}([K_{(1)}, \ldots, K_{(n)}])$ calculates distances independently for each state component [HVL21]. Even if particles are far apart in the full state space due to unobserved variables, they can be close in specific observed dimensions. This ensures that the component-wise kernel values remain significant in observed directions, generating a strong, localized repulsive force [HVL21]. This mechanism prevents the marginal distributions of the observed variables from collapsing, maintaining a spread consistent with the theoretical posterior.

A TensorFlow implementation of Kernel-based PFF

```
class KPFF:
    """
    Kernel Particle Flow Filter (KPFF) using RKHS embedding.
    """
    def __init__(self, ensemble, y_obs, obs_idx, R_var, kernel_type='matrix'):

        self.X = tf.Variable(tf.convert_to_tensor(ensemble, dtype=DTYPE), dtype=DTYPE)
        self.Np = tf.shape(self.X)[0]
```

```
        self.Nx = tf.shape(self.X)[1]

        self.y = tf.convert_to_tensor(y_obs, dtype=DTYPE)
        self.obs_idx = tf.convert_to_tensor(obs_idx, dtype=tf.int32)

        # Cast scalar R_var
        self.R_inv = tf.cast(1.0 / R_var, DTYPE)
        self.kernel_type = kernel_type

        # Kernel width parameter alpha
        # Chosen as 1/Np to balance smoothing and detail
        self.alpha = tf.cast(1.0 / tf.cast(self.Np, DTYPE), DTYPE)

        # Prior stats (Static for the flow duration)
        self.prior_mean = tf.reduce_mean(self.X, axis=0)
        self.prior_var = tf.math.reduce_variance(self.X, axis=0) + 1e-4
        self.D = self.prior_var

    @tf.function
    def get_grad_log_posterior(self, x):
        """
        Computes gradient of log posterior.
        """
        # Likelihood grad
        x_obs = tf.gather(x, self.obs_idx)
        innov = self.y - x_obs
        grad_lik_sub = innov * self.R_inv # Scalar R

        # Scatter back to full dimension
        indices = tf.expand_dims(self.obs_idx, 1)
        grad_lik = tf.scatter_nd(indices, grad_lik_sub, [self.Nx])

        # Prior grad
        grad_prior = -(x - self.prior_mean) / self.prior_var

        return grad_lik + grad_prior

    @tf.function
    def compute_flow(self, x_eval):
        """
        Computes the deterministic particle flow f_s(x).
        f_s(x) = D * (1/Np) * Sum [ K * grad_log_p + grad_K ]
        """

        # Compute gradients for all particles
        grad_log_p_list = tf.map_fn(self.get_grad_log_posterior, self.X)

        # Current particle i is x_eval
        diffs = self.X - x_eval # (Np, Nx)

        if self.kernel_type == 'scalar':
            dist_sq = tf.reduce_sum(tf.square(diffs) / (self.alpha * self.prior_var), axis=1)
            k_val = tf.exp(-0.5 * dist_sq)

            # grad_k: -K * diff / (alpha * var)
            # reshape k_val to (Np, 1)
            grad_k = -tf.expand_dims(k_val, 1) * (diffs / (self.alpha * self.prior_var))

            term = tf.expand_dims(k_val, 1) * grad_log_p_list + grad_k
```

```
        elif self.kernel_type == 'matrix':
            dist_sq_vec = tf.square(diffs) / (self.alpha * self.prior_var)
            k_vec = tf.exp(-0.5 * dist_sq_vec)

            grad_k = -k_vec * (diffs / (self.alpha * self.prior_var))

            term = k_vec * grad_log_p_list + grad_k

        flow_sum = tf.reduce_sum(term, axis=0)
        return (self.D * flow_sum) / tf.cast(self.Np, DTYPE)

    @tf.function
    def update(self, n_steps=50, dt=0.01):
        dt_tf = tf.cast(dt, DTYPE)

        def body(i, current_X):
            # 1. Gradients
            grad_log_p = tf.map_fn(self.get_grad_log_posterior, current_X)

            # 2. Kernel matrix
            X_j = tf.expand_dims(current_X, 0)
            X_i = tf.expand_dims(current_X, 1)
            diffs = X_j - X_i

            scale = self.alpha * self.prior_var

            if self.kernel_type == 'scalar':
                dist_sq = tf.reduce_sum(tf.square(diffs)/scale, axis=2)
                K_mat = tf.exp(-0.5 * dist_sq) # (Np, Np)

                # grad K
                grad_K = tf.expand_dims(K_mat, 2) * (diffs / scale)

                term1 = tf.expand_dims(K_mat, 2) * tf.expand_dims(grad_log_p, 0)
                total = term1 + grad_K

            else: # Matrix
                dist_sq_vec = tf.square(diffs)/scale
                K_vec = tf.exp(-0.5 * dist_sq_vec)
                grad_K = K_vec * (diffs / scale)

                term1 = K_vec * tf.expand_dims(grad_log_p, 0)
                total = term1 + grad_K

            flow = tf.reduce_mean(total, axis=1) * self.D

            return i + 1, current_X + dt_tf * flow

        _, new_X = tf.while_loop(
            lambda i, x: i < n_steps,
            body,
            [0, self.X]
        )

        self.X.assign(new_X)
```

To validate the theoretical advantages of the matrix-valued kernel, we applied both PFF variants to a 1000-dimensional Lorenz 96 system. The results, replicating Figure 3 from [HVL21], are visualized in Figure 2.12.

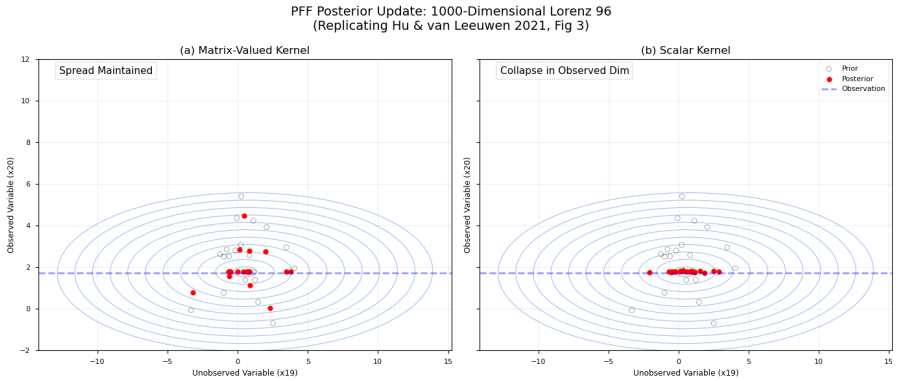


Figure 2.12: Replication of [HVL21, Fig. 3]: Prior (black circles) and posterior (red circles) marginal distributions of the unobserved variable (x_{19}) and observed variable (x_{20}) in the 1000-dimensional Lorenz 96 model. The blue contours represent the reference posterior covariance from the ensemble KF. (a) The matrix-valued kernel maintains the correct posterior spread by applying component-wise repulsive forces. (b) The scalar kernel causes ensemble collapse onto the observation line (dashed blue) due to the vanishing gradients of the kernel in high-dimensional space. (code: `run_hu21_experiments1.py`)

Figure 2.12 plots the prior (black circles) and posterior (red circles) particles for an unobserved variable (x_{19}) versus an observed variable (x_{20}). The blue ellipses represent the theoretical posterior covariance derived from a localized *Ensemble Kalman Filter* (EnKF) for reference [HVL21].

- **Panel (a). Matrix-Valued Kernel**: The posterior particles (red) maintain a healthy spread that aligns well with the EnKF reference contours. The matrix-valued kernel successfully generates a repulsive force in the observed dimension (x_{20}) even when particles are distant in the unobserved dimension (x_{19}). This prevents the ensemble collapse and preserves the correct marginal distribution, demonstrating the necessity of component-wise distance metrics in high-dimensional filtering [HVL21].
- **Panel (b). Scalar Kernel**: The particles exhibit a severe collapse along the observed dimension (y-axis). The posterior particles (red) have clustered tightly onto the observation line (dashed blue), effectively ignoring the uncertainty structure indicated by the reference contours. This confirms the "sample impoverishment" predicted by the theory: the lack of repulsive force allows the likelihood gradient to dominate, driving all particles to the mode [HVL21].

c)

Here, we analyze the specific failure modes of explicit Jacobian-based methods (EDH, LEDH) compared to the proposed matrix-valued kernel PFF. While the problem instructs to use the previous SSM in the last particle filter problem (i.e., Stochastic Volatility or Range-Bearing), it seems that these problems are not hard enough to demonstrate distinct behaviors of the three methods regarding all four aspects of nonlinearity, observation sparsity, dimension,

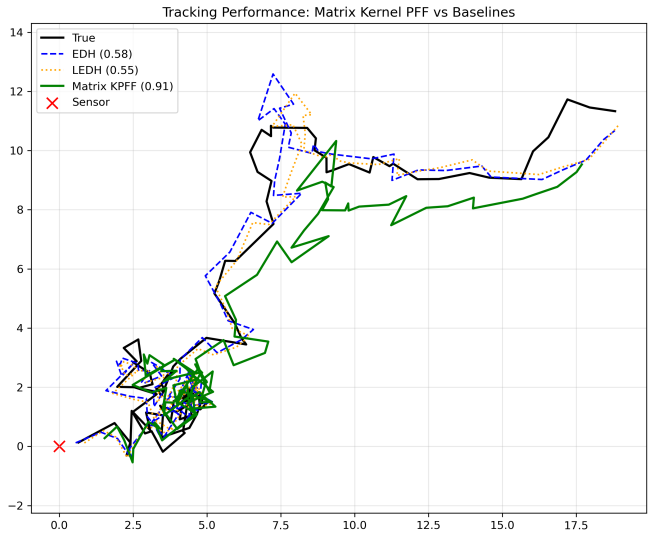


Figure 2.13: Tracking performance comparison on the Range-Bearing model. The plot demonstrates that in this lower-dimensional setting, all three methods—EDH, LEDH, and matrix-valued kernel PFF—maintain a stable lock on the target with comparable RMSEs (0.58, 0.55, and 0.91, respectively). The lack of divergent behavior highlights the necessity of using a higher-dimensional, sparse system (like Lorenz 96) to stress-test the conditioning and marginal collapse issues addressed by the matrix-valued kernel PFF. (code: `run_hu21_experiments2.py`)

and conditioning. The predicted trajectory of the range-bearing problem, for instance, is shown in Figure 2.13.

Instead of using a single SSM for demonstrating all four aspects, we will design appropriate SSMs for each aspect.

To address the limitations of the Range-Bearing model in distinguishing the performance characteristics of the filters, we turn to the Lorenz 96 (L96) model. As established in the literature, the L96 model serves as a robust testbed for high-dimensional, nonlinear, and chaotic system dynamics. By manipulating the dimensionality, observation operator, and noise levels of this system, we can design targeted experiments to isolate the failure modes of the EDH, LEDH, and matrix-valued KPFF. The governing equations for the n_x-dimensional Lorenz 96 model are given by

$$\frac{dx_i}{dt} = (x_{i+1} - x_{i-2})x_{i-1} - x_i + F,$$

where $i = 1, \ldots, n_x$, indices are cyclic, and the forcing term is set to $F = 8$ to ensure chaotic behavior.

Experiment A: Nonlinearity stress test

To evaluate robustness against non-Gaussian likelihoods, we employ a low-dimensional L96 setup ($n_x = 40$) with the highly nonlinear *Square Observation Operator*, defined as $H(x) = x_{(4a)}^2$. This operator induces a bimodal posterior distribution, which challenges the fundamental assumptions of Gaussian-based flow methods. The results are given in Figure 2.14.

- **EDH (Failure mode):** The EDH filter relies on a global linearization of the measurement function at the ensemble mean, $\bar{\eta}_\lambda$. In bimodal distributions (e.g., $x^2 = y \implies x = \pm\sqrt{y}$), the ensemble mean often resides in a low-probability region between modes (near zero). Linearizing at this point yields a near-zero Jacobian, resulting in vanishing flow or incorrect drift directions, causing the filter to fail in capturing the true modes.
- **LEDH (Partial success):** The LEDH improves upon this by linearizing $\mathbf{H}(x)$ locally at each particle η_λ^i. This allows particles in different basins of attraction to flow toward their respective local modes, provided the initialization covers both modes adequately.
- **KPFF (Success):** The Kernel PFF does not require linearization of the observation operator in the flow derivation. Instead, it drives particles using the gradient of the log-posterior directly. As demonstrated in Hu et al. [HVL21], the Kernel PFF successfully captures multi-modal posteriors where standard ensemble methods produce unrepresentative means.

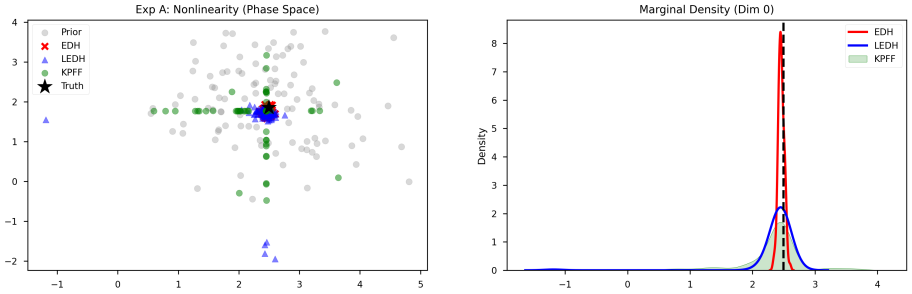


Figure 2.14: Phase space (left) and marginal density (right) comparison for the nonlinear square observation experiment. The EDH (red) clusters incorrectly in the probability trough between modes due to global linearization at the mean. The LEDH (blue) successfully splits into two modes but exhibits over-confident (peaky) distributions. The matrix-valued KPFF (green) correctly maintains the posterior variance around the true state (black star) and the mirror mode, demonstrating robustness to highly nonlinear likelihoods. (code: `run_EDH_LEDH_KPFF_comparison.py`)

Experiment B: The curse of dimensionality and sparsity

This experiment replicates the conditions identified in Hu et al. [HVL21] to demonstrate the specific failure of scalar kernels. We utilize a high-dimensional setup ($n_x = 40$) with sparse observations, observing only 25% of the state variables. See Figure 2.15 for the experiment results.

- **EDH/LEDH (Covariance failure):** Explicit Jacobian methods rely on the term $(\lambda \mathbf{H}\mathbf{P}\mathbf{H}^{\mathbf{T}} + \mathbf{R})^{-1}$ to propagate information. In sparse, chaotic systems, the estimated covariance $\mathbf{P}$ often contains spurious correlations for unobserved variables. EDH, using a global $\mathbf{P}$, propagates these errors uniformly, leading to the higher RMSE and potential divergence. LEDH is computationally prohibitive ($O(N_p)$ inversions) and offers little gain if the underlying covariance estimate is poor.
- **Matrix-valued KPFF (Success):** The Matrix-valued KPFF outperforms explicit methods in this regime. It uses a diagonal matrix-valued kernel to measure distances componentwise, preventing the "weight dilution" problem where noise in unobserved dimensions washes out the signal in observed dimensions. This allows for accurate updates in the observed subspace independent of the unobserved dimensions, preventing marginal collapse.

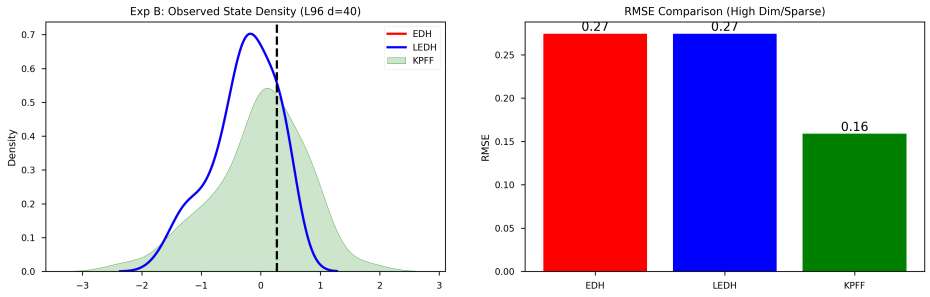


Figure 2.15: Posterior density estimates (left) and RMSE comparison (right) for the high-dimensional sparse L96 experiment ($d = 40$, 25% observed). The matrix-valued KPFF (green) achieves the lowest RMSE (0.16), significantly outperforming EDH and LEDH (both 0.27). The density plot confirms that KPFF maintains a wider, more accurate posterior distribution coverage, whereas explicit Jacobian methods (EDH/LEDH) show signs of variance collapse or bias in the unobserved dimensions. `run_EDH_LEDH_KPFF_comparison.py`)

Experiment C: Conditioning and numerical stability

To analyze numerical stability, we return to a low-dimensional linear observation regime but introduce extreme precision by setting the measurement noise covariance $\mathbf{R} \to 0$ (e.g., $\sigma^2_{obs} = 10^{-12}$). The results are given in Figure 2.16.

- **EDH/LEDH (Stiffness):** The flow in explicit Jacobian methods is governed by the inversion of $S = (\lambda \mathbf{H}(\lambda)\mathbf{P}\mathbf{H}(\lambda)^{\mathbf{T}} + \mathbf{R})$. As $\mathbf{R}$ approaches zero, this matrix becomes ill-conditioned. Even in linear cases where $\mathbf{H}$ is constant (resulting in identical condition numbers for EDH and LEDH), the magnitude of the update vector explodes. This "stiffness" necessitates infinitesimally small time steps to avoid numerical blow-up.
- **KPFF (Stability):** The KPFF formulation avoids the direct inversion of the measurement noise covariance in the flow definition. Instead, the update is driven by the inner product of the kernel with the gradient of the log-likelihood. While the gradient

term $\nabla_x \log p(y|x)$ scales with $\mathbf{R}^{-1}$, the method allows for preconditioning using the system covariance, offering superior stability in stiff regimes without the catastrophic flow magnitudes seen in Jacobian-based methods.

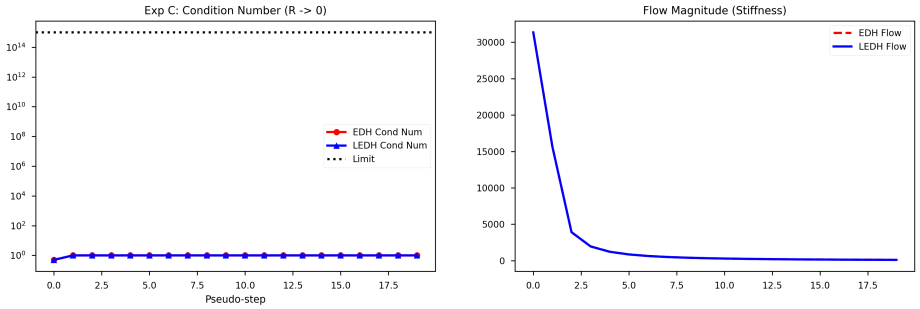


Figure 2.16: Stability diagnostics for Experiment C ($\mathbf{R} \to 0$). Left: The condition number of the innovation matrix remains stable in this specific linear setup ($H = I$), preventing immediate singularity. Right: The Flow Magnitude, however, reveals the instability. Both EDH (red) and LEDH (blue) exhibit massive initial drift magnitudes ($> 3 \times 10^4$), characterizing the "stiffness" problem that requires computationally expensive, tiny integration steps to resolve without divergence. `run_EDH_LEDH_KPFF_comparison.py`)

□

Bibliography

[BBL08] Thomas Bengtsson, Peter Bickel, and Bo Li, *Curse-of-dimensionality revisited: Collapse of the particle filter in very large scale systems*, Probability and statistics: Essays in honor of David A. Freedman **2** (2008), 316–334.

[CPM25] Shreyas Chaudhari, Srinivasa Pranav, and José MF Moura, *Gradnetot: Learning optimal transport maps with gradnets*, arXiv preprint arXiv:2507.13191 (2025).

[CTDD21] Adrien Corenflos, James Thornton, George Deligiannidis, and Arnaud Doucet, *Differentiable particle filtering via entropy-regularized optimal transport*, International Conference on Machine Learning, PMLR, 2021, pp. 2100–2111.

[DC12] Tao Ding and Mark J Coates, *Implementation of the daum-huang exact-flow particle filter*, 2012 IEEE Statistical Signal Processing Workshop (SSP), IEEE, 2012, pp. 257–260.

[DD22] Liyi Dai and Fred Daum, *Stiffness mitigation in stochastic particle flow filters*, IEEE Transactions on Aerospace and Electronic Systems **58** (2022), no. 4, 3563–3577.

[DDFG01] Arnaud Doucet, Nando De Freitas, and Neil Gordon, *Sequential monte carlo methods in practice*, Springer, 2001.

[DH08] Fred Daum and Jim Huang, *Nonlinear filters with particle flow*, Signal Processing, Sensor Fusion, and Target Recognition XVII, vol. 6968, International Society for Optics and Photonics, 2008, p. 696805.

[DH11] ———, *Particle degeneracy: root cause and solution*, Signal Processing, Sensor Fusion, and Target Recognition XX, vol. 8050, SPIE, 2011, pp. 367–377.

[DHN10] Fred Daum, Jim Huang, and Arjang Noushin, *Exact particle flow for nonlinear filters*, Signal processing, sensor fusion, and target recognition XIX, vol. 7697, SPIE, 2010, pp. 92–110.

[DJ11] Arnaud Doucet and Adam M Johansen, *A tutorial on particle filtering and smoothing: Fifteen years later*, The Oxford Handbook of Nonlinear Filtering (Dan Crisan and Boris Rozovskii, eds.), Oxford University Press, 2011, pp. 656–704.

[GSS93] Neil J Gordon, David J Salmond, and Adrian FM Smith, *Novel approach to nonlinear/non-gaussian bayesian state estimation*, IEE Proceedings F (Radar and Signal Processing), vol. 140, IET, 1993, pp. 107–113.

[HVL21] Chih-Chi Hu and Peter Jan Van Leeuwen, *A particle flow filter for high-dimensional system applications*, Quarterly Journal of the Royal Meteorological Society **147** (2021), no. 737, 2352–2374.

[Jha25] Prashant K Jha, *From theory to application: A practical introduction to neural operators in scientific computing*, arXiv preprint arXiv:2503.05598 (2025).

[JRB18] Rico Jonschkowski, Divyam Rastogi, and Oliver Brock, *Differentiable particle filters: End-to-end learning with algorithmic priors*, Robotics: Science and Systems (RSS), 2018.

[JU97] Simon J Julier and Jeffrey K Uhlmann, *New extension of the kalman filter to nonlinear systems*, Signal processing, sensor fusion, and target recognition VI, vol. 3068, Spie, 1997, pp. 182–193.

[Kal60] Rudolf Emil Kalman, *A new approach to linear filtering and prediction problems*, Journal of basic Engineering **82** (1960), no. 1, 35–45.

[KHL18] Peter Karkus, David Hsu, and Wee Sun Lee, *Particle filter networks with application to visual localization*, Conference on Robot Learning, PMLR, 2018, pp. 169–178.

[LC17] Yunpeng Li and Mark J Coates, *Particle filtering with invertible particle flow*, IEEE Transactions on Signal Processing **65** (2017), no. 15, 4102–4116.

[SBBA08] Chris Snyder, Thomas Bengtsson, Peter Bickel, and Jeffrey Anderson, *Obstacles to high-dimensional particle filtering*, Monthly Weather Review **136** (2008), no. 12, 4629–4640.

[Sch66] Stanley F Schmidt, *Application of state-space methods to navigation problems*, Advances in control systems, vol. 3, Elsevier, 1966, pp. 293–340.