Testing a Modern Inference Framework for POMP Models: A Case Study Using Stochastic Volatility

by

Dae Hyun Kim

Supervisor Professor: Edward Ionides

Graduate Student Mentor: Aaron Abkemeier

Department of Statistics

University of Michigan

April 2025

Abstract

The **pypomp** package is a new Python-based library for modeling likelihood-based inference on partially observed Markov process (POMP) models. Designed with the same goals as the R package **pomp**, **pypomp** offers a flexible environment for modeling nonlinear, non-Gaussian dynamic systems and supports both standard particle filtering and gradient-based estimation methods. By leveraging modern computational tools such as automatic differentiation (AD) and Graphics Processing Unit (GPU) acceleration via JAX, **pypomp** enables efficient inference for POMP models. We test and validate the **pypomp** package by using the Heston stochastic volatility model as a test case. Through this case study, we indicate the potential of **pypomp** to serve as a robust, extensible tool for scientific modeling and the development of advanced inference algorithms.

Table of Contents

Chapter 1. Introduction

Inference for dynamic systems governed by latent processes is a core process in modern statistical science, with applications to various fields such as epidemiology, ecology, and finance. Partially observed Markov process (POMP) models, also known as hidden Markov models or state space models provide a principled framework for modeling such systems, in which a latent stochastic process drives an observed time series. However, only partial and noisy observations of the true underlying state are available in POMP models and general Maximum Likelihood Estimation (MLE) is often infeasible due to the intractability of the likelihood, which involves integrating over high-dimensional latent state spaces.

The improved Iterated Filtering (IF2) algorithm introduced by Ionides et al. [2015], suggests an alternative to approach these challenges. IF2 uses the particle filter, also known as the sequential Monte Carlo (SMC) method, to estimate parameters in POMP models and it iteratively refines parameter estimates by introducing stochastic perturbations to the parameters, which facilitates convergence to the MLE. IF2 is a plug-and-play method that enables parameter estimation without requiring analytic expressions for transition densities, making it applicable to a wide variety of scientific models such as epidemiology [King et al., 2008, Subramanian et al., 2021, Wheeler et al., 2024], ecology [Li et al., 2024], and finance [Szczepocki, 2020], [Sun, 2024].

The R package **pomp** has provided a powerful framework including IF2 for likelihood-based inference on POMP models [King et al., 2016], but with the growing use of Python in scientific computing, there is a need for a library with similar modeling capabilities that also support modern numerical frameworks. In response to these needs, the **pypomp** package is being developed as a Python-based analog to **pomp**, and it supports plug-and-play particle filtering and simulation for POMP models, but it is also designed as a library for new algorithmic development. Key innovations in **pypomp** include the use of automatic differentiation (AD) for efficient gradient computation and the integration of JAX for just-in-time (JIT) compilation and GPU acceleration, enabling significant computational efficiency. One of the central algorithmic contributions of **pypomp** is the Iterated Filtering with Automatic Differentiation (IFAD) algorithm, introduced by Tan et al. [2024]. Unlike IF2, which relies on stochastic perturbations for optimization, IFAD integrates gradient based updates through AD. This approach improves computational efficiency by combining rapid exploration of the parameter space via stochastic perturbations with precise gradient refinement. However, as **pypomp** is currently at early stages of development, it requires rigorous testing to ensure its correctness, stability, and reproducibility across applications.

We evaluate and test the current state of **pypomp** by using the Heston stochastic volatility model as a diagnostic benchmark. Sun [2024] applied the IF2 algorithm using the **pomp** package in R to estimate the parameters of the Heston stochastic volatility model and we attempted to replicate these results using IF2 in **pypomp**. We then apply IFAD within the **pypomp** framework to estimate the same model parameters, comparing the inference quality, stability, and computational efficiency. Rather than focusing on improved inference for the Heston model itself, this research uses the model as a structured test case for evaluating the behavior and correctness of **pypomp**.

Our contributions extend beyond benchmarking by helping **pypomp** to advance through rigorous testing and debugging. Specifically, we identified several critical issues and provided concrete fixes and analysis that contribute to the development of a robust, reliable inference in **pypomp**. As **pypomp** is currently in pre-release, our numerical results indicate its potential for development and provide insights as an innovative library for scientific inference and methodological research in dynamic systems.

Chapter 2. Background

2.1 Partially Observed Markov Process Models

A partially observed Markov process (POMP) model consists of an unobserved latent stochastic process $\{X(t), t \geq t_0\}$ with noisy observations $Y_1, \ldots, Y_N$ collected at discrete observation times $t_1, \ldots, t_N$. We assume:

- $X(t)$ takes values in $\mathbb{X} \subset \mathbb{R}^{\dim(\mathbb{X})}$,
- Y_n takes values in $\mathbb{Y} \subset \mathbb{R}^{\dim(\mathbb{Y})}$
- θ denotes a unknown parameter taking values in $\Theta \subset \mathbb{R}^{\dim(\Theta)}$

Then, we define $y_{m:n} = y_m, y_{m+1}, \ldots, y_n$ for integers $m \leq n$, and let $X_n = X(t_n)$ be the latent state at observation time t_n. Thus, the full set of observations is denoted $Y_{1:N}$. Therefore, the joint density of the latent states and observations factorizes as:

$$f_{X_{0:N},Y_{1:N}}(x_{0:N}, y_{1:N}; \theta) = f_{X_0}(x_0; \theta) \prod_{n=1}^{N} f_{X_n|X_{n-1}}(x_n \mid x_{n-1}; \theta) f_{Y_n|X_n}(y_n \mid x_n; \theta).$$

where:

- $f_{X_0}(x_0; \theta)$ is the initial state density,
- $f_{X_n|X_{n-1}}(x_n \mid x_{n-1}; \theta)$ is the transition density of the latent process,
- $f_{Y_n|X_n}(y_n \mid x_n; \theta)$ is the measurement density connecting latent states to observations.

The data consist of observed realizations $y^*_{1:N}$. The marginal density of the observations, $f_{Y_{1:N}}(y_{1:N}; \theta)$, induces the likelihood function:

$$\ell(\theta) = f_{Y_{1:N}}(y^*_{1:N}; \theta)$$

Inference is typically focused on finding the MLE $\hat{\theta}$, which maximizes $\ell(\theta)$.

2.2 Iterated Filtering Algorithms

The key idea behind IF2 is to embed the estimation of parameters within a particle filter by introducing small random perturbations to the parameters across time steps. These perturbations are gradually reduced over multiple iterations, enabling the algorithm to stochastically explore the likelihood surface and converge toward the MLE. Unlike its predecessor, IF1 introduced by Ionides et al. [2011], which approximates the score function via conditional moments, IF2 performs an iterated Bayes map, providing more robust numerical properties and convergence guarantees. (pseudocode in Appendix A)

The particle filter is a foundational tool for inference in POMP models. It enables likelihood-based inference [Ionides et al., 2006] and Bayesian inference [Chopin et al., 2013], [Andrieu et al., 2010] by providing Monte Carlo estimates of the likelihood function of observed data given model parameters. Specifically, the particle filter approximates the likelihood by simulating a collection of possible latent trajectories (particles) and sequentially weighting them according to how well they explain the observed data. This approach is both flexible and powerful, particularly for models where the latent dynamics are nonlinear and non-Gaussian [Arulampalam et al., 2002], [Kitagawa, 1987]. On top of that, the particle filter produces an unbiased estimate of the likelihood [Doucet et al., 2001], making it suitable for likelihood-based inference as it only requires accurate estimates for the likelihood. As a result, IF2 builds directly on the particle filter to perform plug-and-play likelihood maximization, without losing the statistical rigor of likelihood-based approaches.

The advantages of IF2 are its plug-and-play flexibility and stochastic robustness. Specifically, IF2 does not require the evaluation of transition densities, which allows it to be applied to models defined via simulation and the iterative perturbation process helps avoid early convergence to the local optima, which is common in complex likelihood surfaces. Also, the numerical results from Ionides et al. [2015]

indicate that IF2 can approximate the MLE arbitrarily well, under mild regularity conditions, as the number of particles and iterations increase.

The original implementation of IF2 was developed in the R package **pomp**, but R presents limitations in computational scalability, especially for models requiring GPU acceleration or AD. This is where Python and **pypomp** become critically useful. The field of Python research has grown rapidly due to its integration with high-performance libraries like JAX, as well as GPU-accelerated frameworks. As **pypomp** leverages modern features such as JIT compilation and AD, it makes it particularly well-suited for research using large-scale data.

2.3 Automatic Differentiation Particle Filter

Automatic differentiation (AD) is a computational process that enables the efficient and accurate calculation of derivatives for functions implemented as computer programs. Also, AD computes exact derivatives by systematically applying the chain rule to a sequence of elementary operations [Rall and Corliss, 1996, Verma, 2000]. This capability is particularly powerful in fields such as machine learning, numerical optimization, and scientific computing, where gradient based methods are used.

AD operates by breaking down a function into a finite sequence of elementary operations and computing their derivatives step-by-step. AD can be implemented in two primary modes:

1. **Forward Mode**:
    - Computes derivatives by propagating derivatives from the inputs to the outputs.
    - Efficient for functions with few inputs and many outputs.
2. **Reverse Mode**:
    - Computes derivatives by propagating gradients backwards from the outputs to the inputs.
    - Efficient for functions with many inputs and few outputs.

Both modes rely on the chain rule to propagate derivatives through computation:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y} \cdot \frac{\partial y}{\partial x}$$

Recent work has extended the use of AD to particle filtering and inference for POMP models. The advantage of using AD is the gradient-based optimization for POMP models. Several papers have explored differentiable particle filters. For instance, Jonschkowski et al. [2018] introduced differentiable filtering networks, and Naesseth et al. [2018] proposed variational inference approaches using AD to optimize latent state models. However, these early approaches typically relied on low variance but asymptotically biased estimators of the gradient, raising concerns about their long-term reliability and statistical guarantees.

The Automatic Differentiation Particle Filter (ADPF), introduced by Tan [2023], overcomes these issues by framing the particle filter as a differentiable computational graph. On top of that, Tan et al. [2024] introduces the Measurement Off-Parameter with discount factor α (MOP-α) algorithm (pseudocode in Appendix A) that defines estimators for the gradient of the log-likelihood where the estimators balance the tradeoff between bias and variance. At $\alpha = 0$, low-variance estimators can be highly biased, while at $\alpha = 1$, unbiased estimators tend to suffer from high Monte Carlo variability. MOP-α interpolates between these extremes, allowing users to refine the factor based on precision needs.

The Iterated Filtering with Automatic Differentiation (IFAD) algorithm (pseudocode in Appendix A) introduced by Tan et al. [2024] represents another development of using ADPF. The IFAD algorithm incorporates MOP-based gradient estimators into the structure of the IF2 algorithm. Like IF2, IFAD executes a stochastic perturbation of parameters through a particle filter. However, instead of fully relying on perturbations to update parameter estimates, IFAD supplements this process with gradient-based updates using AD. This process enables IFAD to overcome the two core challenges of IF2 such as Monte Carlo variability from noisy perturbation updates and slow convergence in high-dimensional spaces.

2.4 Stochastic Volatility Models

Stochastic volatility models play a central role in financial modeling, where accurately capturing the evolution of asset price variability is crucial for derivative pricing, risk management, and portfolio allocation. Among these models, the Heston model is widely used due to its ability to represent volatility as a mean-reverting stochastic process. However, parameter estimation for such models is challenging because of the intractability of their likelihood functions and the presence of latent variables.

The estimation of parameters in stochastic volatility models remains a challenging problem, primarily due to the computational complexity of the likelihood function, which involves high dimensional integration. Various methods have been developed to address these challenges. Aït-Sahalia and Kimmel [2007] developed a maximum likelihood estimation method that addresses these challenges by employing closed-form approximations to the likelihood function. Chib et al. [2002] and Chib et al. [2006] developed Markov Chain Monte Carlo (MCMC) algorithms for generalized stochastic volatility models. However, these methods are computationally intensive for high dimensions.

The Heston model introduced by Heston [1993], one of the most widely used stochastic volatility models includes a mean-reverting stochastic process for volatility, revealing the limitations of constant volatility models. Specifically, it assumes that the variance of asset returns follows a Cox-Ingersoll-Ross (CIR) process, expressed as:

$$d\nu_t = \kappa(\theta - \nu_t)\,dt + \xi\sqrt{\nu_t}\,dW_t^{\nu}$$

where

- ν_t is the instantaneous variance of the asset price at time t,
- κ is the rate of mean reversion in variance,
- θ is the long-term mean level of variance ($\lim_{t\to\infty}\mathbb{E}[\nu_t] = \theta$),
- ξ is the volatility of the volatility,
- W_t^{ν} is the Wiener process in the volatility.

Simultaneously, the asset price S_t changes according to the following stochastic differential equation:

$$dS_t = \mu S_t\,dt + \sqrt{\nu_t}S_t\,dW_t^{s}$$

where:

- S_t is the asset price at time t,
- μ is the average growth rate of the asset price
- W_t^s is the Wiener process in the asset price.

We define the correlation between W_t^s and W_t^v as ρ:

$$\rho dt = \mathbb{E}[dW_t^s dW_t^v]$$

The Heston model provides a realistic framework for understanding how volatility changes over time, enabling more accurate pricing of financial instruments and better investment decisions with market uncertainty. Therefore, it is effective for testing and comparing parameter estimation algorithms, such as the IF2 algorithm and the IFAD algorithm.

Chapter 3. Debugging pypomp

As part of evaluating IF2 and IFAD implementations in the **pypomp** package, we encountered several numerical instabilities and inconsistencies while testing the Heston stochastic volatility model. These issues motivated a deeper investigation into the underlying mechanics of **pypomp** and led to insights critical to the future reliability of the package.

3.1 Original Testing

From the outset, the **pypomp** organization incorporated a system of unit tests designed to ensure that core functionalities such as particle simulation, likelihood computation, and filtering are not broken when changes are made to the code. On top of that, unit tests provide early warnings if numerical results shift unexpectedly due to subtle changes in implementation, such as modifications to random number handling or model setup. This unit testing framework greatly aided the debugging process by allowing us to localize problems quickly when inconsistencies were observed during benchmarking. In addition to unit testing, IF2 and the particle filter in **pypomp** were evaluated through performance benchmark tests called quantitative tests, or simply quant tests, named by the **pypomp** organization. Unlike unit tests, quant tests measure the time to complete filtering and optimization tasks, the memory requirements during computation, and the number of iterations needed to achieve convergence for a maximization algorithm. The quant tests were performed using the Linear Gaussian model and the Dhaka cholera model introduced by King et al. [2008]. The Linear Gaussian model offers an analytically tractable benchmark, where exact likelihoods and parameter estimates are known. Since the Kalman filter provides the exact optimal inference for this model, deviations in **pypomp** results could be precisely identified and quantified, making it an ideal diagnostic tool for basic functionality and performance. The Dhaka cholera model represents a complex, nonlinear, epidemiological system. By using this model, it allowed developers to assess scalability to large datasets, robustness under complex latent dynamics with the particle filter and IF2 algorithm. Performance metrics from these two models helped ensure that IF2 and particle filter in **pypomp** could handle both analytically simple and scientifically realistic POMP models with reasonable computational resources.

3.2 Issue in Original R code

Before investigating issues within **pypomp** itself, we first carefully examined the R code provided by Sun [2024] for the Heston stochastic volatility model. During this process, we identified a critical flaw that the latent volatility process V was not correctly updated over time inside the rprocess simulation step.

In the correct implementation of the Heston model, volatility V must evolve dynamically according to its stochastic differential equation. However, in the original R code, it did not update V from one time step to the next. As a result, volatility remained constant or evolved incorrectly, fundamentally altering the latent dynamics of the model. We corrected this by adding a proper stochastic update for V at each time step within the rprocess function in R.

3.3 Initial Value Perturbation Issue in Iterative Filtering

The first major issue was discovered when we were analyzing parameter evolution during IF2 optimization. In the R pomp package, initial value parameters (IVPs) such as the initial state of the latent process are only perturbed during the first filtering iteration, reflecting the fact that IVPs lose their influence over time as the latent process evolves [Ionides et al., 2015], [King et al., 2016]. Perturbing IVPs throughout the entire optimization adds unnecessary noise and destabilizes the inference.

By contrast, **pypomp** initially perturbed IVPs continuously across all iterations as it did not yet distinguish IVPs from other parameters in the filtering process. This behavior reduced optimization performance and introduced instability to our numerical results, as perturbations on IVPs continued to introduce variability into later stages of optimization where the IVPs should have been forgotten by the model dynamics.

This oversight can be understood by considering the origins of the **pypomp** codebase. The initial **pypomp** development was built based on code supporting the experiments in Tan et al. [2024], where the models studied either avoided the IVP issue or were not sensitive to it. In that experimental context, continuous IVP perturbation did not create critical problems. However, **pypomp** aims for a more general and flexible scope, supporting a wide range of POMP models such as the Heston stochastic volatility model, where IVPs play a substantial role in early dynamics but diminish later. The broader generalization intended for **pypomp** exposed the need to properly handle IVPs.

It is common during software development that bugs emerge when generalizing code beyond its original use case. In this case, what worked for specific research applications in Tan et al. [2024] needed further refinement to support a general plug-and-play POMP modeling framework. After diagnosing this issue during our testing, we reported the misbehavior, and it was subsequently corrected by Jun Chen, one of the founding developers of the **pypomp** organization.

Even after the IVPs issue was resolved, the log-likelihood estimates obtained using 1,000 particles were significantly below the anticipated value (around 11849, based on Sun [2024]'s results) and often produced errors. However, when using 10,000 particles, the estimated log-likelihood reached the expected value and errors disappeared, indicating that other numerical stability issues also needed to be addressed.

3.4 Particle Depletion Assumption

This pattern suggested the presence of particle depletion, also known as particle degeneracy, a common problem in particle filter algorithms. Particle depletion occurs when, after resampling, most particles are either discarded or receive extremely low weights [Daum and Huang, 2011]. This leads to a lack of diversity in the particle population and log-likelihood estimates become biased or undefined as likelihood estimates are computed from weighted particles [Gustafsson et al., 2002]. Also, when estimating parameters in IF2, if particles get depleted early, the algorithm struggles to explore different parameter values, resulting in suboptimal estimates. To mitigate particle depletion, increasing the number of particles reduces variance and prevents all weights from collapsing, so the numerical stability gets improved. As the characteristics of particle depletion correspond to the situations that we were experiencing, it supported our assumptions.

To diagnose this, we analyzed the conditional log-likelihoods and particle traces of parameters. In a POMP model, the observations are random, and we want to compute the likelihood of observing the data, given a parameter θ. Instead of looking at the full likelihood at once, we break it down sequentially. At each time point t_k, the conditional likelihood is the probability density of the current observation, given all previous observations. In other words, the conditional log-likelihood is the value of the density of $Y(t_k) \mid Y(t_1), \ldots, Y(t_{k-1})$ evaluated at $Y(t_k) = y_k^*$ where $Y(t_k)$ is the observable process, and y_k^* is the data, at time t_k.

Thus, the conditional log-likelihood at time t_k is

$$\ell_k(\theta) = \log f \left[Y(t_k) = y_k^* \mid Y(t_1) = y_1^*, \ldots, Y(t_{k-1}) = y_{k-1}^* \right]$$

where f is the probability density above.

If particles are depleted at some time t_k, meaning most particle weights are near 0 except for very few, then the sum of the weights becomes very small. This will affect the conditional log-likelihood $\ell_k(\theta)$ to be highly negative and it will show a large negative spike in the plot. In contrast, when particles are not struggling from depletion, $\ell_k(\theta)$ behaves smoothly without huge fluctuations.

For this analysis, at each IF2 iteration, we recorded the conditional log-likelihood at every observation time t_k. Then, for each time index k, we averaged the conditional log-likelihood values across all IF2 iterations. Averaging across iterations helps smooth out random Monte Carlo fluctuations and indicates systematic patterns in the optimization process.

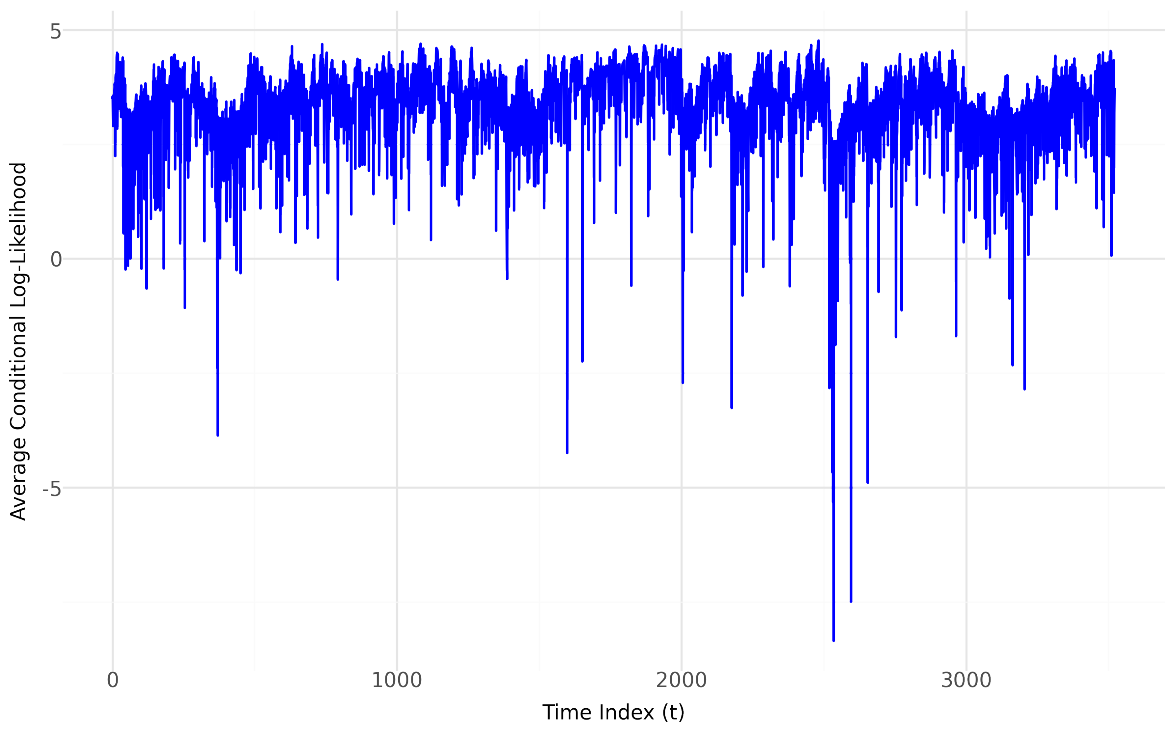

Figure 3.1: Average conditional log-likelihood over time

Figure 3.1 represents the time series of the average conditional log-likelihood over the full dataset in an one repetition. The plot shows that the conditional log-likelihood values fluctuate over time, as expected due to the stochastic nature of the optimization process and variability in particle weights. Some downward excursions are visible, but none of the conditional log-likelihood values fall to levels that would indicate a particle depletion issue. In other words, the lower excursions do not suggest that particle weights collapsed or that the particle population failed to support likelihood evaluation at any time point.

During IF2, the parameters are perturbed at each time point t_k slightly and each particle carries a slightly perturbed parameter $\theta_k^{(i)}$. The particle traces of parameters record $\theta_k^{(i)}$ for each particle i over each time t_k, so tracking particle traces is helpful to analyze whether parameters collapse (particles become identical), or the particle diversity is maintained. Thus, if a particle depletion occurred, then most particle traces will collapse to a few lines and the spread in parameter values will be lost.

By inspecting these traces, we attempted to localize potential failures at particular time points.

However, no obvious collapse or degeneration was observed across the traces, suggesting the problem might lie elsewhere.

3.5 Just-in-time Compilation Issue

A deeper debugging and investigation revealed that the root cause of instability and errors was not particle depletion, but rather faulty random number generation during JIT compilation. Specifically, random perturbations for particles were originally drawn using np.random.normal, a NumPy function. JAX's JIT compilation requires that all random number generations be explicitly managed via jax.random.PRNGKey because JAX uses pure functional programming while NumPy does not. Previously, keeping keys from being split and mixing np.random with JIT-compiled functions led to uncontrolled randomness. In other words, the same random values were reused repeatedly across particles and iterations, corrupting the filtering and optimization process and producing errors in likelihood estimations. After using JAX random number generation such as jax.random.normal rather than np.random and ensuring that keys were explicitly split, the random perturbations became truly random, the errors disappeared, and likelihood estimates stabilized.

Chapter 4. Iterative Filtering in pypomp

Sun [2024] applied the IF2 algorithm implemented in the R package **pomp** to estimate the parameters of the Heston stochastic volatility model using daily Standard & Poor's 500 index (S&P500) log-return data from 2010 to 2024. His study indicated the ability of IF2 to recover volatility dynamics consistent with market benchmarks like the VIX index, and showed the model significantly outperformed classical GARCH-based benchmarks in terms of log-likelihood. His research also conducted profile likelihood estimation on each parameter, initially profiling over μ, and subsequently fixing μ at its smoothed profile MLE to improve optimization stability. In our study, we sought to replicate Sun's results using the **pypomp** package. While Sun performed profiling to construct confidence intervals, we did not require profiling of μ in our setting since our goal was not to conduct inference but to test and validate the numerical behavior of the **pypomp** IF2 implementation.

4.1 Log-likelihood Trace

The log-likelihood trace records the IF2 log-likelihood estimate at each iteration. It is computed using perturbed parameters and is inherently noisy due to stochastic perturbations during optimization. Nonetheless, the overall trend is expected to increase (or equivalently, the negative log-likelihood should decrease) as the algorithm refines parameter estimates.

Tracking the log-likelihood trace serves as an important internal diagnostic for the IF2 optimization process. By analyzing the evolution of log-likelihood values across iterations, we can assess whether IF2 is making consistent progress toward better fitting parameter regions, identify convergence behavior, and detect potential issues such as instability during optimization.

To evaluate the internal optimization process of both **pomp** and **pypomp**, we compared the log-likelihood values estimated at each iteration of the IF2 algorithm. To ensure that differences in starting points did not affect the comparison, we sampled and stored 120 initial parameter sets from Sun [2024]'s model using **pomp** and then used exactly the same starting values when running IF2 optimization in our model using **pypomp**. This guarantees that IF2 of both **pomp** and **pypomp** implementations began from identical initial conditions, allowing us to attribute any differences to algorithmic behavior rather than random initial variability.

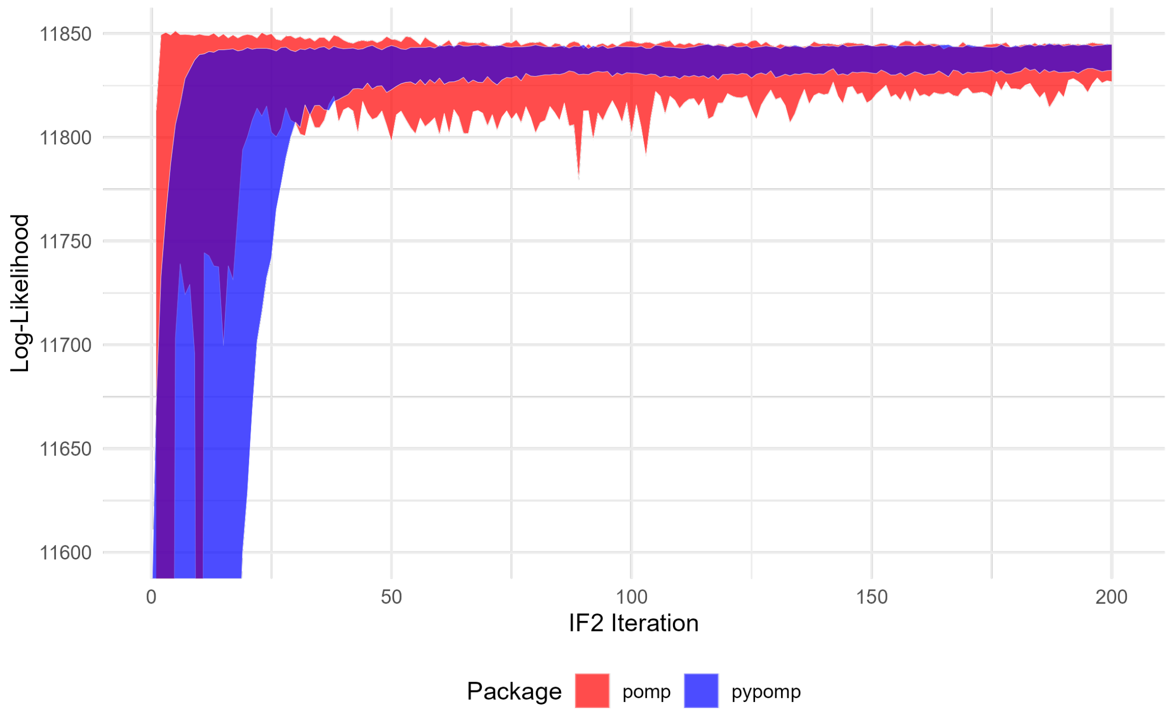

Figure 4.1: IF2 log-likelihood trace across iterations

Each shaded ribbon in Figure 4.1 represents the 10th–90th percentile range of log-likelihood values across 120 IF2 replicates at each iteration. The values were computed using perturbed parameters at each IF2 iteration and represent noisy estimates of model fit throughout optimization. Although stochastic, the log-likelihood trajectories are expected to improve across iterations as the algorithm converges toward high-likelihood regions. The results show that broadly similar convergence behavior in both implementations, with **pypomp** showing noticeably lower variance in its log-likelihood traces compared to **pomp**. This suggests that **pypomp** is not exploring the parameter space as extensively as **pomp**. Thus, while IF2 optimization is functioning and improving log-likelihoods in both packages, there are meaningful differences between the two implementations. Minor differences are always expected when comparing independently developed codebases in different languages, but our findings suggest that **pypomp**'s IF2 implementation is different from **pomp** in ways that affect parameter exploration and optimization dynamics and hence could affect the final parameter estimates.

4.2 Parameter Estimate Trace

The parameter estimation traces show the evolution of each parameter across IF2 iterations. We average across particles at each iteration to visualize the path of optimization. As parameters are iteratively perturbed and adapted toward the MLE, parameter estimation traces are important in the IF2 process. With this trace, we can analyze whether parameter estimates converge or diverge and whether they exhibit similar dynamics in **pomp** and **pypomp**. Also, if both implementations use the same algorithmic parameter values, such as number of particles, iterations, and repetitions, then their traces should be similar. This implies that parameter estimation traces provide qualitative assurance between **pomp** and **pypomp**.

To assess the consistency of the **pomp** implementation and the **pypomp** implementation, we compared the evolution of parameter estimations across IF2 iterations.

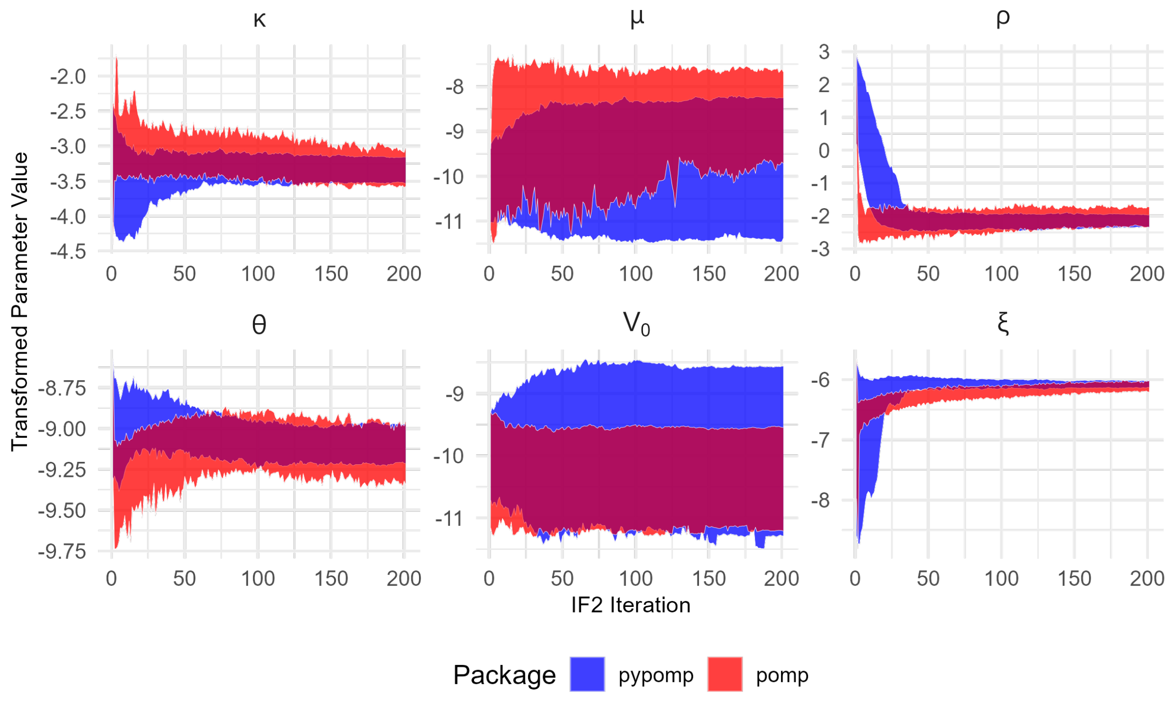

Figure 4.2: Parameter estimates across IF2 iterations

Figure 4.2 displays quantile ribbon plots showing the evolution of parameter estimates over IF2 iterations across **pomp** and **pypomp** implementations. For each parameter, we show the middle 80% (10th to 90th percentile) of transformed parameter values at each iteration. The parameters are transformed to the estimation scale such as log-transformed for positive parameters (μ, κ, θ, ξ, V_0) and logit-transformed for the correlation parameter (ρ). This transformation ensures a reasonable comparison since both implementations optimize in transformed space.

Across all six parameters, we observe similar convergence behavior between the two implementations. Both implementations stabilize similarly to transformed values, indicating agreement in the inferred parameter regions but still there are noticeable differences in the median trajectories.

Then, we assess whether the two implementations arrive at consistent parameter estimate.

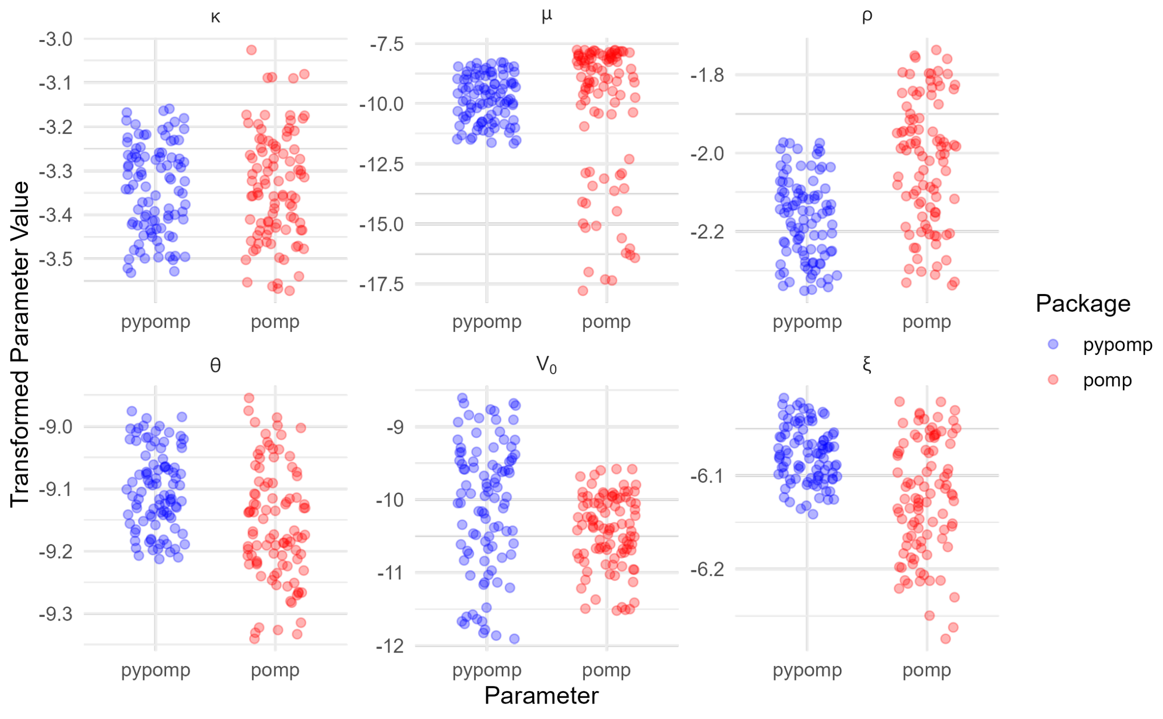

Figure 4.3: Final IF2 parameter estimates

Figure 4.3 displays the transformed parameter estimates at the final IF2 iteration from 120 independent IF2 runs in both **pomp** and **pypomp**. Each point represents the final parameter estimate from one IF2 replicate, transformed to the estimation scale (log or logit). To enhance the interpretability of parameter comparison, extreme parameter estimates were filtered out prior to visualization. For each parameter, only parameter estimates between the 10th and 90th percentiles are shown, to focus on the central distribution and reduce the influence of extreme outliers.

Across parameters, the distributions of final estimates are similarly aligned between the two implementations. Slight differences in spread and central tendency appear in the parameters and these results indicate that while the two implementations do not produce identical traces, they broadly agree in terms of convergence region and variability.

4.3 Particle Filter Log-likelihood

After IF2 finishes, the algorithm uses a particle filter with particles and no perturbation to estimate the log-likelihood at the final estimated parameters. For each IF2 run, the algorithm computes the average and standard deviation of estimated log-likelihoods over particle filter replicates. The average log-likelihood assesses the fit quality of the estimated parameters, and the standard deviation gives a sense of Monte Carlo variability. With these results, given the same number of particles and datasets, IF2 algorithm in both **pomp** and **pypomp** yield likelihood estimates with similar accuracy and variability.

To compare the model fit quality across implementations, we examined the log-likelihood values obtained by applying a particle filter to the final parameter estimates from each IF2 run.

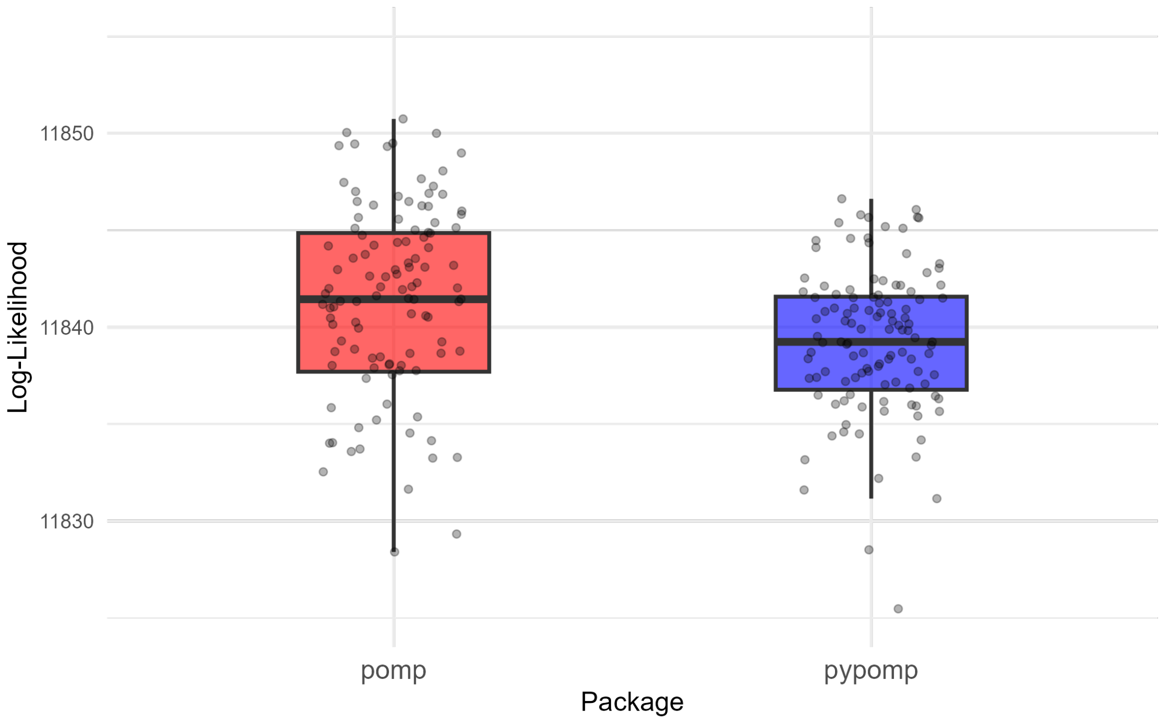

Figure 4.4: Particle filter log-likelihood at IF2 final estimates(outliers excluded)

Figure 4.4 shows a comparison of the log-likelihoods from particle filter evaluations at final IF2 estimates across 120 replicates. Each point represents the mean log-likelihood across iterations for one

IF2 replicate. While **pomp** shows a slightly higher median and broader spread with 11850.74 for the highest log-likelihood, **pypomp** shows lower median and tighter spread with 11847.34 for the highest log-likelihood.

To verify that particle filter in **pypomp** matches the particle filter implementation in **pomp**, we conducted a Monte Carlo-based comparison. We used the same stochastic volatility model and the MLE for each parameter obtained from **pomp** to evaluate the mean log-likelihood using the standard particle filter without parameter perturbation across 24 replicates for both **pypomp** and **pomp**. The mean log-likelihood estimated with the MLE in **pomp** was 11850.74 and the mean log-likelihood estimated with the MLE in **pypomp** was 11848.29 with standard deviation 2.78.

These preliminary results suggested that the particle filter in **pypomp** approximately reproduced the behavior of **pomp** within expected Monte Carlo variation. However, comparing only the means of the log-likelihood estimates is insufficient to fully establish correspondence between the two implementations.

To provide a more rigorous comparison, we generated 1,000 independent particle filter log-likelihood estimates from each package without parameter perturbation, using the same MLE parameter values. To increase the interpretability of the results, we visualized the results using violin plots overlaid with boxplots. This allowed us to examine the full distribution of log-likelihood estimates and assess the density, variability, and central tendency in a much more statistically robust way.

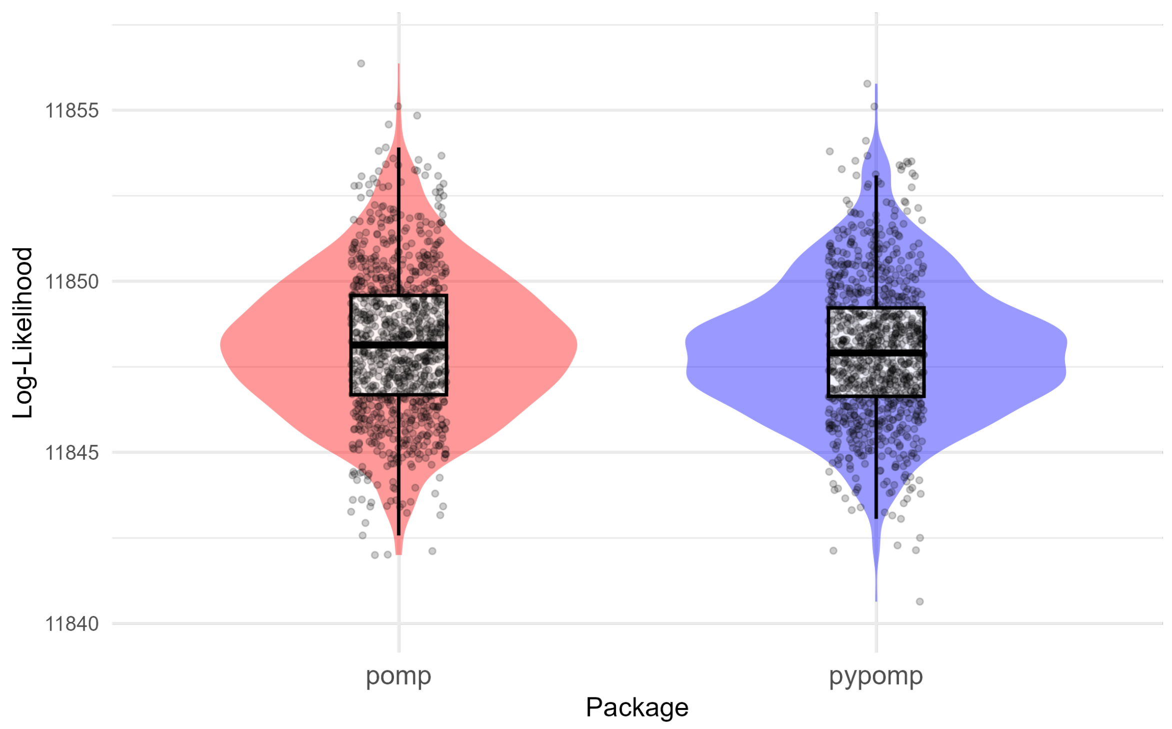

Figure 4.5: Particle filter log-likelihood at MLE

In Figure 4.5, each violin shape shows the estimated density of log-likelihood values, and the embedded boxplot summarizes the interquartile range and median, and the jittered points show individual Monte Carlo replicates. The plot demonstrates that the particle filter in both **pomp** and **pypomp** produce highly overlapping distributions of log-likelihood values, and the medians and interquartile ranges are nearly identical between the two implementations. In addition, the distribution shapes are symmetric and centered around similar values, with minor expected Monte Carlo fluctuations.

Thus, this comparison strongly reinforces the conclusion that the particle filter implementation of **pypomp** accurately replicates the behavior of the particle filter in **pomp**, up to the expected variability due to stochastic simulation. Any observed discrepancies between **pypomp** and **pomp** in IF2 results should therefore be attributed to differences in the parameter perturbation and optimization steps of the IF2 process, rather than to differences in the underlying particle filter algorithm.

Chapter 5. Automatic Differentiation Particle Filter in pypomp

Tan et al. [2024] demonstrate that IFAD outperforms both IF2 and MOP algorithms when applied individually. IF2 alone is prone to slow convergence near the optimum due to diminishing perturbation sizes and MOP alone is sensitive to local minima and saddle points in non-convex likelihood surfaces. The strengths of IFAD are a warm start using IF2 to locate a neighborhood near the MLE and a refinement that uses gradients derived from the MOP-α estimator to perform gradient ascent with AD. This hybrid approach helps IFAD to overcome two critical limitations of IF2 and MOP.

Using the cholera transmission model developed for Dhaka, Bangladesh King et al. [2008], IFAD found log-likelihood values significantly better than those reported using IF2 Ionides et al. [2015]. Notably, IFAD reaches the MLE with fewer iterations and without the assistance of likelihood profiling. These results indicate IFAD's numerical efficiency and statistical accuracy, and its potential as an innovative plug-and-play method for statistical inference in POMP models.

Encouraged by the numerical efficiency and statistical accuracy of IFAD, we attempted to analyze its performance implemented in **pypomp** using the Heston stochastic volatility model from Sun [2024]. However, during the implementation and testing of IFAD and MOP in **pypomp**, we encountered persistent numerical errors and exploding values. Despite initializing the algorithm with plausible parameter values and a reasonable number of particles, errors in gradients and log-likelihoods frequently occurred during the optimization after the first forward mode of AD, suggesting issues within the backward mode of AD execution. As these instabilities do not occur in IF2 implementation of **pypomp**, it reinforces the conclusion that the source of instability comes from the gradient refinement in MOP implemented in **pypomp**.

Chapter 6. Discussion

Our research contributes to the development, testing, and validation of the early-stage **pypomp** package. Using the Heston stochastic volatility model as a test case, we examined the behavior and correctness of the IF2 and IFAD algorithms in **pypomp**, comparing results against the **pomp** package. For IF2, we demonstrated that **pypomp** implementation successfully produces log-likelihood estimates and parameter traces that are consistent with the general behavior of IF2. Particle filter process appears to be stable, and likelihood estimates generally improve across iterations, indicating that the core logic of particle filtering and optimization is in place. Despite these promising results, we identified several important discrepancies and design issues that will require attention to ensure that **pypomp** becomes a fully robust scientific tool.

Although the particle filter behavior matches expectations, there remains meaningful differences between IF2 results from **pypomp** and **pomp**, particularly in log-likelihood traces and parameter estimation across iterations. Based on diagnostic comparisons, we conclude that these differences are not due to the particle filter itself, but rather arise from differences in the IF2 perturbation process. Specifically, lower variance in log-likelihood traces of **pypomp**, compared to **pomp**, suggests that **pypomp** may not be exploring parameter space as much as it intended. In addition, a deeper examination of handling random number generation under JAX's JIT compilation is essential, and PRNG key management must be verified at every stochastic step to ensure true randomness and reproducibility.

For IFAD, although the algorithm is theoretically promising and has shown success in recent research, we encountered persistent numerical instabilities when attempting to apply IFAD to the Heston model in **pypomp**. These issues include errors in gradients and log-likelihoods, and failures in the optimization after the initial iteration. These issues indicate that while IFAD is a prominent method, the current **pypomp** implementation should ensure differentiability of the computational graph across all IFAD components and manage numerical stability in weight scaling and normalization, especially in the MOP-α gradient estimators. Beyond debugging, IFAD must be evaluated through quant tests to assess its scalability and practical utility for statistical inference.

Beyond algorithmic issues, several important design considerations were identified during this research. Although the IVP perturbation behavior was corrected during debugging, the fix currently exists in a separate branch and is not yet fully incorporated into the main **pypomp** IF2 function. Formal integration is necessary to ensure consistent behavior across models.

On top of that, **pypomp** currently does not directly provide access to conditional log-likelihoods at each time point within the particle filter. Conditional log-likelihood traces are extremely valuable for model validation and debugging, allowing researchers to identify specific times where model fit deteriorates or where particle depletion occurs. Adding this feature would greatly enhance the transparency and diagnostic functions of **pypomp**.

At present, **pypomp** supports IF2, MOP-α, and IFAD algorithms but other algorithms available in R **pomp** such as Particle Markov Chain Metropolis-Hastings algorithm [Andrieu et al., 2010], and Ensemble Kalman filters [Evensen, 2003] are not yet implemented. Extending **pypomp** to include these algorithms would enable broader applicability and make the package a forward compatibility with R **pomp**.

Reference

Yacine Aït-Sahalia and Robert Kimmel. Maximum likelihood estimation of stochastic volatility models. *Journal of financial economics*, 83(2):413–452, 2007.

Christophe Andrieu, Arnaud Doucet, and Roman Holenstein. Particle markov chain monte carlo methods. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 72(3):269–342, 2010.

M Sanjeev Arulampalam, Simon Maskell, Neil Gordon, and Tim Clapp. A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking. *IEEE Transactions on signal processing*, 50 (2):174–188, 2002.

Siddhartha Chib, Federico Nardari, and Neil Shephard. Markov chain Monte Carlo methods for stochastic volatility models. *Journal of Econometrics*, 108(2):281–316, 2002.

Siddhartha Chib, Federico Nardari, and Neil Shephard. Analysis of high dimensional multivariate stochastic volatility models. *Journal of Econometrics*, 134(2):341–371, 2006.

Nicolas Chopin, Pierre E Jacob, and Omiros Papaspiliopoulos. SMC2: an efficient algorithm for sequential analysis of state space models. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 75(3):397–426, 2013.

Fred Daum and Jim Huang. Particle degeneracy: root cause and solution. In *Signal Processing, Sensor Fusion, and Target Recognition XX*, volume 8050, pages 367–377. SPIE, 2011.

Arnaud Doucet, Nando De Freitas, Neil James Gordon, et al. *Sequential Monte Carlo methods in practice*, volume 1. Springer, 2001.

Geir Evensen. The ensemble Kalman filter: Theoretical formulation and practical implementation. *Ocean dynamics*, 53:343–367, 2003.

Fredrik Gustafsson, Fredrik Gunnarsson, Niclas Bergman, Urban Forssell, Jonas Jansson, Rickard Karlsson, and P-J Nordlund. Particle filters for positioning, navigation, and tracking. *IEEE Transactions on signal processing*, 50(2):425–437, 2002.

Steven L Heston. A closed-form solution for options with stochastic volatility with applications to bond and currency options. *The review of financial studies*, 6(2):327–343, 1993.

Edward L Ionides, Carles Bretó, and Aaron A King. Inference for nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 103(49):18438–18443, 2006.

Edward L Ionides, Anindya Bhadra, Yves Atchadé, and Aaron King. Iterated filtering. *The Annals of Statistics*, pages 1776–1802, 2011.

Edward L Ionides, Dao Nguyen, Yves Atchadé, Stilian Stoev, and Aaron A King. Inference for dynamic and latent variable models via iterated, perturbed Bayes maps. *Proceedings of the National Academy of Sciences*, 112(3):719–724, 2015.

Rico Jonschkowski, Divyam Rastogi, and Oliver Brock. Differentiable particle filters: End-to-end learning with algorithmic priors. *arXiv preprint arXiv:1805.11122*, 2018.

Aaron A King, Edward L Ionides, Mercedes Pascual, and Menno J Bouma. Inapparent infections and cholera dynamics. *Nature*, 454(7206):877–880, 2008.

Aaron A King, Dao Nguyen, and Edward L Ionides. Statistical inference for partially observed Markov processes via the R package pomp. *Journal of Statistical Software*, 69:1–43, 2016.

Genshiro Kitagawa. Non-Gaussian State-Space Modeling of Nonstationary Time Series. *Journal of the American statistical association*, 82(400):1032–1041, 1987.

Jifan Li, Edward L Ionides, Aaron A King, Mercedes Pascual, and Ning Ning. Inference on spatiotemporal dynamics for coupled biological populations. *Journal of the Royal Society Interface*, 21(216): 20240217, 2024.

Christian Naesseth, Scott Linderman, Rajesh Ranganath, and David Blei. Variational Sequential Monte Carlo. In *International conference on artificial intelligence and statistics*, pages 968–977. PMLR, 2018.

Louis B Rall and George F Corliss. An introduction to automatic differentiation. *Computational Differentiation: Techniques, Applications, and Tools*, 89:1–18, 1996.

Rahul Subramanian, Qixin He, and Mercedes Pascual. Quantifying asymptomatic infection and transmission of COVID-19 in New York City using observed cases, serology, and testing capacity. *Proceedings of the National Academy of Sciences*, 118(9):e2019716118, 2021.

Weizhe Sun. Model Based Inference of Stochastic Volatility via Iterated Filtering. Undergraduate Honors Thesis, 2024.

Piotr Szczepocki. Application of iterated filtering to stochastic volatility models based on non-Gaussian Ornstein-Uhlenbeck process. *Statistics in Transition. New Series*, 21(2):173–187, 2020.

Kevin Tan. Differentiable Plug-And-Play Particle Filtering. Undergraduate Honors Thesis, 2023.

Kevin Tan, Giles Hooker, and Edward L Ionides. Accelerated Inference for Partially Observed Markov Processes using Automatic Differentiation. *arXiv preprint arXiv:2407.03085*, 2024.

Arun Verma. An introduction to automatic differentiation. *Current Science*, pages 804–807, 2000.

Jesse Wheeler, AnnaElaine Rosengart, Zhuoxun Jiang, Kevin Tan, Noah Treutle, and Edward L Ionides. Informing policy via dynamic models: Cholera in Haiti. *PLOS Computational Biology*, 20(4): e1012032, 2024.

Appendix A. Algorithms

Algorithm 1 IF2

```
input: Simulator for $f_{X_0}(x_0;\theta)$
Simulator for $f_{X_n|X_{n-1}}(x_n \mid x_{n-1};\theta)$, $n$ in $1:N$
Evaluator for $f_{Y_n|X_n}(y_n \mid x_n;\theta)$, $n$ in $1:N$
Data, $y^*_{1:N}$
Number of iterations, $M$
Number of particles, $J$
Initial parameter swarm, $\{\Theta^0_j, j \text{ in } 1:J\}$
Perturbation density, $h_n(\theta \mid \varphi;\sigma)$, $n$ in $1:N$
Perturbation sequence, $\sigma_{1:M}$
output: Final parameter swarm, $\{\Theta^M_j, j \text{ in } 1:J\}$
 1: for $m$ in $1:M$ do
 2:     $\Theta^{F,m}_{0,j} \sim h_0(\theta \mid \Theta^{m-1}_j;\sigma_m)$ for $j$ in $1:J$
 3:     $X^{F,m}_{0,j} \sim f_{X_0}(x_0;\Theta^{F,m}_{0,j})$ for $j$ in $1:J$
 4:     for $n$ in $1:N$ do
 5:         $\Theta^{P,m}_{n,j} \sim h_n(\theta \mid \Theta^{F,m}_{n-1,j};\sigma_m)$ for $j$ in $1:J$
 6:         $X^{P,m}_{n,j} \sim f_{X_n|X_{n-1}}(x_n \mid X^{F,m}_{n-1,j};\Theta^{P,m}_{n,j})$ for $j$ in $1:J$
 7:         $w^m_{n,j} = f_{Y_n|X_n}(y^*_n \mid X^{P,m}_{n,j};\Theta^{P,m}_{n,j})$ for $j$ in $1:J$
 8:         Draw $k_{1:J}$ with $\mathbb{P}(k_j = i) = w^m_{n,i} / \sum_{u=1}^{J} w^m_{n,u}$
 9:         $\Theta^{F,m}_{n,j} = \Theta^{P,m}_{n,k_j}$ and $X^{F,m}_{n,j} = X^{P,m}_{n,k_j}$ for $j$ in $1:J$
10:     end for
11:     Set $\Theta^m_j = \Theta^{F,m}_{N,j}$ for $j$ in $1:J$
12: end for
```

The main elements of IF2 are:

- Particles: A collection of J Monte Carlo samples approximating the latent and parameter processes.
- Iterations: A sequence of M optimization loops indexed by m.
- Perturbation density $h_n(\theta \mid \varphi;\sigma)$: A probability density introducing controlled stochastic per-

turbations to parameters at each time step n, with scale σ.

- Prediction step: Evolving particles forward through the latent process.
- Filtering step: Weighting particles by how well they explain the observed data, followed by resampling.

The following notation is used:

- $\Theta_{n,j}^{F,m}$: Parameter vector for the jth particle at time n after filtering at iteration m.
- $\Theta_{n,j}^{P,m}$: Parameter vector for the jth particle at time n after prediction at iteration m.
- $X_{n,j}^{F,m}$ Latent state for the jth particle at time n after filtering at iteration m.
- $X_{n,j}^{P,m}$: Latent state for the jth particle at time n after prediction at iteration m.
- $k_{1:J}$: Resampling indices, drawn multinomially based on particle weights at each filtering step.

Algorithm 2 MOP-α

Input: Number of particles J, timesteps N, measurement model $f_{Y_n|X_n}(y_n^* \mid x_n, \theta)$, simulator process$_n(\cdot \mid x_n; \theta)$, evaluation parameter θ, baseline parameter ϕ, seed ω.
First pass: Set $\theta = \phi$ and fix ω, yielding $X_{n,j}^{P,\phi}$, $X_{n,j}^{F,\phi}$, $g_{n,j}^{\phi}$.
Second pass: Fix ω, and filter at $\theta \neq \phi$:
Initialize particles $X_{0,j}^{F,\theta} \sim f_{X_0}(\cdot; \theta)$, weights $w_{0,j}^{F,\theta} = 1$.

```
1:  for $n = 1, \ldots, N$: do
2:      Accumulate discounted weights, $w_{n,j}^{P,\theta} = \big(w_{n-1,j}^{F,\theta}\big)^{\alpha}$.
3:      Simulate process model, $X_{n,j}^{P,\theta} \sim \text{process}_n\big(\cdot \mid X_{n-1,j}^{F,\theta}; \theta\big)$.
4:      Measurement density, $g_{n,j}^{\theta} = f_{Y_n|X_n}(y_n^* \mid X_{n,j}^{P,\theta}; \theta)$.
5:      Compute $L_n^{B,\theta,\alpha} = \sum_{j=1}^{J} g_{n,j}^{\theta}\, w_{n,j}^{P,\theta} \big/ \sum_{j=1}^{J} w_{n,j}^{P,\theta}$.
6:      Conditional likelihood under $\phi$, $L_n^{\phi} = \frac{1}{J}\sum_{m=1}^{J} g_{n,m}^{\phi}$.
7:      Select resampling indices $k_{1:J}$ with $\mathbb{P}\big(k_j = m\big) \propto g_{n,m}^{\phi}$.
8:      Obtain resampled particles $X_{n,j}^{F,\theta} = X_{n,k_j}^{P,\theta}$.
9:      Calculate corrected weights $w_{n,j}^{F,\theta} = w_{n,k_j}^{P,\theta}\, g_{n,k_j}^{\theta} \big/ g_{n,k_j}^{\phi}$..
10:     Compute $L_n^{A,\theta,\alpha} = L_n^{\phi} \cdot \sum_{j=1}^{J} w_{n,j}^{F,\theta} \big/ \sum_{j=1}^{J} w_{n,j}^{P,\theta}$.
11: end for
```

Return: likelihood estimate $\hat{\mathscr{L}}(\theta) = \prod_{n=1}^{N} L_n^{A,\theta,\alpha}$ or $\hat{\mathscr{L}}(\theta) = \prod_{n=1}^{N} L_n^{B,\theta,\alpha}$, filtering distributions $\{(X_{n,j}^{F,\theta}, w_{n,j}^{F,\theta})\}_{n,j=1}^{N,J}$.

Algorithm 3 IFAD

Input: Number of particles J, timesteps N, IF2 and MOP-α cooling schedules η_m, MOP-α discounting parameter α, θ_0, $m = 0$.
Run initial IF2 search under cooling schedule η_m to obtain $\{\Theta_j, j = 1, ..., J\}$, set $\theta_m := \frac{1}{J}\sum_{j=1}^{J} \Theta_j$.
While procedure not converged:

```
1: Run MOP-$\alpha$ Algorithm to obtain $\hat{\ell}(\theta_m)$.
2: Set $g(\theta_m) := \nabla_\theta(-\hat{\ell}(\theta_m))$, and consider any $H(\theta_m)$ such that $\lambda_{\min}(H(\theta_m)) \geq c$.
3: Update $\theta_{m+1} := \theta_m - \eta_m (H(\theta_m))^{-1} g(\theta_m)$, $m := m + 1$.
```

Return $\hat{\theta} := \theta_m$.