Extensions to the Kalman Filter and Particle Filter for Dynamic System Estimation

Marco Conati
University Of Michigan
Ann Arbor, Michigan
marcoco@umich.edu

Abstract

This study investigates advanced extensions of the Kalman Filter (KF) and Particle Filter (PF) to enhance state and parameter estimation in dynamic systems, focusing on a pendulum model. The Extended Rauch-Tung-Striebel Smoother (ERTSS) outperformed the standard Extended Kalman Filter (EKF) by effectively utilizing future data to correct initial errors. Additionally, the integration of Particle Filters with Markov Chain Monte Carlo (PMCMC) demonstrated promise in estimating unknown measurement noise parameters. These findings highlight the potential of these techniques in dynamic system analysis. Future research could explore computational optimizations and the application of these methods to more complex systems.

1 Background

In the realm of dynamic systems analysis, accurate state estimation under uncertainty is a cornerstone of robust system design and control. Among the most influential approaches to achieving precise state estimation are the Kalman Filter (KF) and Particle Filter (PF). The Kalman Filter provides an efficient recursive solution to the linear Gaussian filtering problem, while the Particle Filter extends the capability to non-linear and non-Gaussian contexts through a sequential Monte Carlo method. Both methods are fundamental in fields ranging from autonomous navigation to financial economics, due to their ability to deliver real-time inference about systems dynamically evolving over time.

Beyond the basic filtering algorithms, extensions are available to solve slight variations on the sate estimation problem. Such extensions include the Extended Rauch-Tung-Striebel Smoother (ERTSS), which builds on the Extended Kalman Filter (EKF) framework to refine state estimates retroactively when future data is available, and hybrid methods combining Particle Filters with Markov Chain Monte Carlo (PMCMC) techniques to estimate uncertain model parameters.

This section lays the groundwork by introducing the basic filtering and smoothing algorithms. The subsequent problem setup section (2) will explore the model used in this report along with motivating examples, and the methods section (3), delves deeper into the proposed extensions, detailing their implementation in the context of dynamic system estimation.

1.1 Gaussian Filtering and Smoothing

1.1.1 Kalman Filtering

Gaussian filters, such as the Kalman Filter (KF), are fundamental in the realm of Bayesian filtering, where the probability distributions involved are assumed Gaussian. The KF operates under the framework of Bayesian inference, utilizing a two-step process: prediction and update. In the prediction phase, the filter projects the current state estimate forward in time using a state transition

Submitted to the Conference of 567 Final Projects (AEROSP 567, 2024).

model. This step is mathematically described by:

$$m_k = A_{k-1} m_{k-1},$$
$$P_k = A_{k-1} P_{k-1} A_{k-1}^\top + Q_{k-1},$$

where A_{k-1} denotes the state transition matrix, P represents the covariance matrix of the estimate, and Q_{k-1} is the process noise covariance matrix. During the update phase, the filter refines this prediction based on new measurements, adjusting the state estimate to minimize the uncertainty:

$$S_k = H_k P_k H_k^\top + R_k,$$
$$K_k = P_k H_k^\top S_k^{-1},$$
$$m_k = m_k + K_k(y_k - H_k m_k),$$
$$P_k = P_k - K_k S_k K_k^\top,$$

where H_k is the measurement matrix, R_k is the measurement noise covariance, y_k is the new measurement, and K_k is the Kalman gain. This iterative process of prediction and update allows the Kalman Filter to effectively track the state of linear dynamic systems.

1.1.2 Rauch-Tung-Striebel Smoother (RTSS)

The RTSS is a backward recursion algorithm that starts with the final state estimates obtained from a Kalman filter and works its way back to the initial state. It utilizes the entire sequence of observations to adjust the state estimates, enhancing accuracy over what is possible with forward filtering alone. The smoother operates under the assumption that the state transitions and observations are governed by Gaussian processes. The equations governing this process are:

Prediction For each state, the predicted next state and its covariance are computed as:

$$m_{k+1} = A_k m_k,$$
$$P_{k+1} = A_k P_k A_k^T + Q_k,$$

where A_k represents the state transition matrix, and Q_k is the process noise covariance matrix.

Gain Calculation The smoothing gain, which incorporates information from future states, is calculated by:

$$G_k = P_k A_k^T [P_{k+1}]^{-1},$$

Smoothing Update Using the smoothing gain, the smoothed state estimates and their covariances are updated as follows:

$$m_k^s = m_k + G_k(m_{k+1}^s - m_{k+1}),$$
$$P_k^s = P_k + G_k(P_{k+1}^s - P_{k+1})G_k^T,$$

where m_k^s and P_k^s are the smoothed mean and covariance, respectively.

1.1.3 Extended Kalman Filter

The Extended Kalman Filter (EKF) is an adaptation of the Kalman Filter (KF) designed to handle nonlinear systems. It achieves this by approximating the nonlinear state and measurement functions using a first-order Taylor series expansion around the current estimate, thereby extending the applicability of the KF to a broader range of applications.

Taylor Series Approximation The EKF linearizes the nonlinear state transition and measurement functions at the current estimate, $\hat{x}$, facilitating the propagation of Gaussian distributions through non-linear systems. This linearization is expressed as:

$$f(x) \approx f(\hat{x}) + F(x - \hat{x}),$$

where F is the Jacobian matrix of partial derivatives evaluated at the estimate $\hat{x}$.

These matrices are essential for implementing the EKF prediction and update steps, adjusting for the system's nonlinear characteristics.

EKF Prediction and Update Equations The EKF extends the Kalman prediction and update equations to account for nonlinearity:

$$\hat{x}_{k|k-1} = \Phi(\hat{x}_{k-1}),$$

$$P_{k|k-1} = F_k P_{k-1|k-1} F_k^\top + Q,$$

where Φ represents the state dynamics, and Q is the process noise covariance. The update phase involves the measurement function h and its Jacobian H_k:

$$S_k = H_k P_{k|k-1} H_k^\top + R,$$

$$K_k = P_{k|k-1} H_k^\top S_k^{-1},$$

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k(y_k - h(\hat{x}_{k|k-1})),$$

$$P_{k|k} = (I - K_k H_k) P_{k|k-1}.$$

Here, I is the identity matrix, R represents the measurement noise covariance, and y_k is the actual measurement. These equations succinctly express how the EKF adapts the traditional Kalman Filter equations to accommodate nonlinear dynamics and measurements.

1.2 Bootstrap Particle Filtering

Particle filters excel in scenarios where systems exhibit high non-linearity, possess multi-modal state distributions, or contain discrete state components. As robust implementations of Bayesian filtering using sequential importance sampling, particle filters adeptly handle complex models that challenge traditional Gaussian assumptions. The following paragraphs give some more details about key pieces.

Operational Framework: Bootstrap Particle filters operate by maintaining a cloud of particles that represent potential system states. These particles are propagated through state dynamics and sequentially updated using new observational data to track the evolution of the state distribution. The full algorithm is given in Algorithm 1, and encapsulates propagation, weighting, and resampling to robustly estimate state distributions.

Empirical Distribution: The state of the system at any given time is estimated by the empirical distribution, represented as a sum of delta functions centered at each particle:

$$\hat{P}(X_n|Y_n) = \sum_{i=1}^{N} \bar{w}_i \delta_{x_i^n}(X_n),$$

where x_i^n are the particle states, $\bar{w}_i$ are their normalized weights, and $\delta_{x_i^n}(X_n)$ is the Dirac delta function.

Particle Propagation In the basic implementation of a Bootstrap Particle Filter, particle propagation is conducted through the system's dynamics model. Particles are sampled according to the transition probabilities defined by:

$$X_{k+1}^{(i)} = f(X_k^{(i)}, \Delta t) + Q_k^{1/2} \epsilon_k^{(i)},$$

where f represents the deterministic part of the system's dynamics, $Q_k^{1/2}$ is the Cholesky decomposition of the process noise covariance matrix, and $\epsilon_k^{(i)}$ is a noise term drawn from a standard Gaussian distribution. While this method effectively captures the system's evolution based on its known dynamics, it may not always integrate new observational data optimally. More sophisticated proposals, such as those that utilize an Unscented Kalman Filter (UKF) for particle proposal, can provide a more refined estimation by incorporating observations more directly into the state prediction. These advanced techniques modify the proposal distribution to reflect not only the dynamics but also the likelihood of the observed data, thereby enhancing the particle filter's ability to converge to the true state distribution, especially in highly nonlinear systems.

Sequential Importance Sampling: Updating particle weights after observations is central to particle filtering, implemented as follows:

$$w_i^n \propto w_i^{n-1} \times \frac{p(y_n|x_i^n) \times p(x_i^n|x_i^{n-1})}{\pi(x_i^n|x_i^{n-1})},$$

where $p(y_n|x_i^n)$ is the likelihood of observing data y_n given the state x_i^n, $p(x_i^n|x_i^{n-1})$ is the state transition probability, and $\pi(x_i^n|x_i^{n-1})$ is the proposal distribution.

Marginal Likelihood and Weight Normalization: The weights are normalized to ensure they sum to one, ensuring that the empirical distribution remains a valid probability distribution:

$$\hat{P}(y_n|Y_{n-1}) \approx \sum_{i=1}^{N} \frac{w_i^n}{\sum_{j=1}^{N} w_j^n}.$$

Resampling: Resampling addresses weight degeneracy (where a few particles dominate the weight distribution) by drawing a new set of particles from the weighted distribution, thereby redistributing probability mass:

$$N_{\text{eff}} = \frac{1}{\sum_{i=1}^{N} (\bar{w}_k^{(i)})^2},$$

where a low N_{eff} indicates a need for resampling, typically set at a threshold such as $N_0 = 0.1N$, which is used in this project.

Algorithm 1 Sequential Importance Resampling

Require: Proposal distributions $\pi(X_k|X_{k-1})$ for each step $k = 1, \ldots, n$; initial empirical distribution $\{(X_0^{(i)}, \bar{w}_0^{(i)})\}_{i=1}^N$; and a resampling threshold N_0.
Ensure: Empirical distribution of the states for each timestep $k = 1, \ldots, n$.

for $k = 1 \to n$ **do**
  for $i = 1 \to N$ **do**
    $X_k^{(i)} \sim \pi(X_k|X_{k-1}^{(i)})$ ▷ Sample from proposal distribution
    $v_k^{(i)} = p(y_k|X_k^{(i)}) \cdot p(X_k^{(i)}|X_{k-1}^{(i)}) / \pi(X_k^{(i)}|X_{k-1}^{(i)})$ ▷ Calculate importance weight
    $w_k^{(i)} = v_k^{(i)} \cdot \bar{w}_{k-1}^{(i)}$ ▷ Update weights
  end for
  $\bar{w}_k^{(i)} = w_k^{(i)} / \sum_{j=1}^{N} w_k^{(j)}$ ▷ Normalize weights
  $N_{\text{eff}} = 1 / \sum_{i=1}^{N} (\bar{w}_k^{(i)})^2$ ▷ Calculate effective sample size
  if $N_{\text{eff}} < N_0$ **then**
    Perform resampling: Draw N samples from $\hat{P}(X_k|Y_k)$ ▷ Resample to avoid degeneracy
    Set $\bar{w}_k^{(i)} = 1/N$ for all i
  end if
end for

2 Problem Setup

In this project, we explore the dynamic behavior of a simple pendulum. The pendulum system is characterized by its angle x_1 from the vertical and its angular velocity x_2, evolving through time according to discrete, nonlinear dynamical equations. The system's state at each time step k is given by:

$$\begin{pmatrix} x_{k+1,1} \\ x_{k+1,2} \end{pmatrix} = \begin{pmatrix} x_{k,1} + x_{k,2}\Delta t \\ x_{k,2} - g\sin(x_{k,1})\Delta t \end{pmatrix} + q_k$$

where Δt is the time increment and g is the acceleration due to gravity. The term q_k introduces process noise into the system, modeled as a zero-mean Gaussian process with covariance matrix:

$$Q = \begin{pmatrix} \frac{q_c \Delta t^3}{3} & \frac{q_c \Delta t^2}{2} \\ \frac{q_c \Delta t^2}{2} & q_c \Delta t \end{pmatrix}$$

where $q_c = 0.1$ is the process noise coefficient. Additionally, measurements of the pendulum's angle x_1 are made every δ timesteps and are subject to measurement noise $r_{\delta k}$, also modeled as a Gaussian process:

$$y_{\delta k} = \sin(x_{\delta k,1}) + r_{\delta k}$$

with $r_{\delta k} \sim N(0, R)$, where R represents the variance of the measurement noise. This integration of process and measurement noise provides a complex environment for testing advanced filtering techniques aimed at accurately estimating and smoothing the pendulum's state trajectories under uncertainty. Simulations are run for 500 timesteps, starting from initial conditions $x_0 = (1.5, 0)$, with a standard Gaussian centered at these initial conditions, using a standard deviation of 1, as the prior for the initial state estimates.

Motivations for Extending Filtering Techniques: The goals of this problem are twofold, relating to extending the EKF to ERTSS and PF to PMCMC respectively:

ERTSS motivation and goals: The EKF offers a method to handle nonlinearities in pendulum dynamics. However, it does so as a purely forward, recursive estimator that does not use future data to inform current state estimates. ERTSS builds on the EKF by incorporating both past and future observations to refine state estimates. This backward smoothing technique is particularly valuable in scenarios where subsequent data is available, such as in trajectory analysis.

For our pendulum system, applying ERTSS is expected to significantly reduce the Root Mean Squared Error (RMSE) of the trajectory estimates, especially in regards to initial estimation errors as those states can be better informed by the backwards recursive process.

PMCMC Motivation and Goals: PFs excel in handling systems with non-Gaussian and nonlinear characteristics, making them ideal for dynamic models like the pendulum. Extending PFs with Markov Chain Monte Carlo (MCMC) techniques, the resulting PMCMC approach allows us to address a new problem altogether: the estimation of unknown parameters within such complex systems. This is done in two steps. First, PFs are utilized to evaluate the system's behavior under various parameter settings, effectively estimating what is termed as the "energy" of the system relative to these parameters. Second, MCMC methods are employed to refine our understanding of how likely different parameter values are, based on how well they explain the observed data.

In the context of our pendulum model, I hope to apply PMCMC to accurately determine the variance of measurement noise R, assuming it is initially unknown.

3 Methods

3.1 Informing EKF and PF with known dynamics

With our known pendulum model, we can first represent our EKF and PF in terms of our system knowledge:

3.1.1 EKF for the known Pendulum dynamics

Jacobian Derivation For systems like the modeled pendulum with dynamics defined by:

$$x_{k+1} = \begin{pmatrix} x_{k,1} + \Delta t \cdot x_{k,2} \\ x_{k,2} - g\Delta t \cdot \sin(x_{k,1}) \end{pmatrix},$$

$$y_k = \sin(x_{k,1}),$$

the Jacobians for state transition F_k and measurement H_k are derived as:

$$F_k = \begin{pmatrix} 1 & \Delta t \\ -g\Delta t \cos(x_{k,1}) & 1 \end{pmatrix},$$

$$H_k = \begin{pmatrix} \cos(x_{k,1}) & 0 \end{pmatrix}.$$

Plugging this into the EKF equations yields explicit representations of the EKF prediction and update equations for the pendulum model:

Prediction Equations: The state prediction incorporates the dynamics of the pendulum, projecting the previous state estimate forward in time:

$$\hat{x}_{k|k-1} = \begin{pmatrix} \hat{x}_{k-1,1} \\ \hat{x}_{k-1,2} \end{pmatrix} + \Delta t \cdot \begin{pmatrix} \hat{x}_{k-1,2} \\ -g \sin(\hat{x}_{k-1,1}) \end{pmatrix},$$

and the covariance prediction updates the uncertainty in the state estimate accounting for the linearized dynamics:

$$P_{k|k-1} = \begin{pmatrix} 1 & \Delta t \\ -g\Delta t\cos(\hat{x}_{k-1,1}) & 1 \end{pmatrix} P_{k-1|k-1} \begin{pmatrix} 1 & -g\Delta t\cos(\hat{x}_{k-1,1}) \\ \Delta t & 1 \end{pmatrix} + Q.$$

Update Equations: The update equations are similarly explicit:

$$S_k = \begin{pmatrix} \cos(\hat{x}_{k|k-1,1}) & 0 \end{pmatrix} P_{k|k-1} \begin{pmatrix} \cos(\hat{x}_{k|k-1,1}) \\ 0 \end{pmatrix} + R,$$

$$K_k = P_{k|k-1} \begin{pmatrix} \cos(\hat{x}_{k|k-1,1}) \\ 0 \end{pmatrix} S_k^{-1},$$

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k \left(y_k - \sin(\hat{x}_{k|k-1,1}) \right),$$

$$P_{k|k} = \left(I - K_k \begin{pmatrix} \cos(\hat{x}_{k|k-1,1}) & 0 \end{pmatrix} \right) P_{k|k-1}.$$

3.1.2 PF with Pendulum Dynamics Proposal

This section outlines the proposal distribution, transition probabilities, and the likelihood model using the system dynamics as a proposal method in a particle filter (PF) applied to a pendulum task.

Proposal Distribution The dynamics of the pendulum are employed directly as the proposal distribution in a "Bootstrap Particle Filter" framework. Each subsequent state X_{k+1} is proposed using the current state X_k evolved through the pendulum's dynamics:

$$X_{k+1} = f(X_k, \Delta t) + L_{\text{proc}}\eta, \quad \eta \sim \mathcal{N}(0, I)$$

where $f(X_k, \Delta t)$ represents the deterministic part of the pendulum's motion over a timestep Δt, and L_{proc} is the Cholesky decomposition of the process noise covariance matrix, adding stochasticity to the state transition.

Transition Probabilities and Log Likelihood The transition probability density function (pdf), or log likelihood of proposing a state X_k given the previous state X_{k-1}, incorporates the dynamics with an additional noise component to address modeling uncertainties:

$$\log p(X_k|X_{k-1}) = -\frac{1}{2} \left\| \frac{f(X_{k-1}, \Delta t) - X_k}{\sigma_{\text{proc}}} \right\|^2$$

where σ_{proc} is the standard deviation of the process noise, indicating the expected variation in the next state due to inherent uncertainties in the dynamics.

Likelihood Model The likelihood model assesses how likely a measured data point y_k is, given a state X_k. Assuming that measurements are noisy observations of the system state, the likelihood is modeled as:

$$\log p(y_k|X_k) = -\frac{1}{2} \left\| \frac{h(X_k) - y_k}{\sigma_{\text{noise}}} \right\|^2$$

where $h(X_k)$ maps the state space to the measurement space, and σ_{noise} is the measurement noise standard deviation, assuming Gaussian distribution of measurement errors.

3.2 Extensions to the EKF and PF

3.2.1 Extended Rauch-Tung-Striebel Smoother (ERTSS) for the Pendulum Model

The extension from RTS to ERTSS is directly analogous to the extension from KF to EKF. In both instances, nonlinear dynamics are accommodated by linearizing the system with a first-order approximation.

State Prediction First, the state prediction equations are given by:

$$m_{k+1} = f(m_k),$$

$$P_{k+1} = F_x(m_k) P_k F_x(m_k)^T + Q_k,$$

where $f(m_k)$ represents the nonlinear state transition function, and $F_x(m_k)$ is its Jacobian matrix.

Now, applying the pendulum dynamics and the previously derived Jacobian $F_x(m_k)$ from Section 3.1.1, the specific state prediction equations are:

$$m_{k+1} = f(m_k),$$

$$P_{k+1} = \begin{pmatrix} 1 & \Delta t \\ -g\Delta t \cos(m_{k,1}) & 1 \end{pmatrix} P_k \begin{pmatrix} 1 & \Delta t \\ -g\Delta t \cos(m_{k,1}) & 1 \end{pmatrix}^T + Q_k,$$

Gain Calculation The generic equation for the smoothing gain is:

$$G_k = P_k F_x(m_k)^T [P_{k+1}]^{-1},$$

Now, inserting the Jacobian for the pendulum dynamics into the gain calculation, we get:

$$G_k = P_k \begin{pmatrix} 1 & \Delta t \\ -g\Delta t \cos(m_{k,1}) & 1 \end{pmatrix}^T [P_{k+1}]^{-1},$$

Smoothing Update Similarly to the RTS, the smoothed estimates are then updated using the following equations:

$$m_k^s = m_k + G_k(m_{k+1}^s - m_{k+1}),$$

$$P_k^s = P_k + G_k(P_{k+1}^s - P_{k+1})G_k^T,$$

This ensures a refined trajectory that accurately reflects the nonlinear dynamics of the pendulum, providing enhanced smoothing over the standard RTS. For all ERTSS experiments:

3.3 Particle Filter for Energy Estimation

Particle filtering is an effective tool not only for state estimation but also for evaluating the energy function necessary for parameter estimation within stochastic models. This technique is especially relevant when dealing with unknown parameters within a model's dynamics and observations, as typically represented by:

$$\theta \in \mathbb{R}^d, \quad x_0 \sim p(x_0|\theta), \quad x_k \sim p(x_k|x_{k-1}, \theta), \quad y_k \sim p(y_k|x_k, \theta),$$

where θ denotes the set of unknown parameters to be estimated.

3.3.1 SIR Based Energy Function Approximation

We can leverage the SIR algorithm from Algorithm 1 in order to estimate the energy function of a system, which quantifies the fit of model parameters θ against observed data. This energy function, in the context of Bayesian inference, corresponds to the negative log of the posterior probability of the parameters given the data, essentially representing a cost function in optimization terms. Here, the energy function is approximated using the particle filter framework to evaluate the marginal likelihood of parameters, thereby enabling parameter estimation in complex models where analytical solutions are impractical.

Algorithm 2 SIR-based Energy Function Approximation

1: **Initialize:** Draw N samples $x_0^{(i)}$ from the prior distribution $p(x_0|\theta)$ for $i = 1, \ldots, N$. Initialize all weights $w_0^{(i)} = \frac{1}{N}$.
2: **for** $k = 1$ to T **do**
3: Draw samples $x_k^{(i)}$ from the importance distribution $q(x_k|x_{k-1}^{(i)}, y_{1:k}, \theta)$ for each particle i.
4: Compute weights for each sample:

$$v_k^{(i)} = \frac{p(y_k|x_k^{(i)}, \theta) \cdot p(x_k^{(i)}|x_{k-1}^{(i)}, \theta)}{q(x_k^{(i)}|x_{k-1}^{(i)}, y_{1:k}, \theta)}$$

5: Estimate $\hat{p}(y_k|y_{1:k-1}, \theta)$ as:

$$\hat{p}(y_k|y_{1:k-1}, \theta) = \sum_i w_{k-1}^{(i)} v_k^{(i)}$$

6: Normalize the weights:

$$w_k^{(i)} \propto w_{k-1}^{(i)} v_k^{(i)}$$

7: Resample if the effective number of particles is too low.
8: **end for**
9: **Compute Energy Function:**

$$E(\theta) = \log p(\theta) - \sum_{k=1}^{T} \log \hat{p}(y_k|y_{1:k-1}, \theta)$$

3.3.2 Integration of MCMC with Particle Filters

Particle Markov Chain Monte Carlo (PMCMC) methods integrate particle filters within each MCMC iteration to estimate the model's likelihood given the observed data. This likelihood estimate is then employed to determine the acceptance probability for parameter updates proposed during the MCMC process.

Algorithm Overview The PMCMC algorithm effectively merges the dynamics of particle filtering with the statistical rigor of MCMC, providing a robust framework for parameter estimation in complex models. Below is a more equation-driven description of the PMCMC algorithm:

Algorithm 3 Particle MCMC Algorithm

1: **Initialization:**
- Start with an initial parameter estimate $\theta(0)$.
- Initialize proposal noise σ and set the target acceptance rate.
- Set initial energy E(0)=inf (indicating a poor initial fit).

2: **Parameter Sampling Loop:** For each iteration $k = 1, 2, \ldots, N$:

1. **Propose New Parameter:**
$$\theta' \sim \mathcal{N}(\theta^{(k-1)}, \sigma^2)$$
2. **Run Particle Filter:** Simulate the particle filter with the proposed parameter θ' to compute the likelihood of the observed data:
$$\hat{E}(\theta') = -\log(\text{ParticleFilter}(\theta', \text{Data}))$$
3. **Calculate Acceptance Probability:**
$$a(\theta^{(k-1)}, \theta') = \min(1, e^{E(\theta^{k-1}) - E(\theta')})$$
where $p(\theta)$ is the prior probability of θ, and $q(\theta'|\theta)$ is the proposal probability density function.
4. **Accept or Reject the Proposal:**
$$\theta^{(k)} = \begin{cases} \theta' & \text{with probability } a(\theta^{(k-1)}, \theta') \\ \theta^{(k-1)} & \text{with probability } 1 - a(\theta^{(k-1)}, \theta') \end{cases}$$
5. **Adjust Proposal Noise:** Optionally adjust σ based on the observed acceptance rate to approach the target acceptance rate.

3: **Output:** After completing all iterations, output the sampled parameters $\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(N)}$ and the acceptance rate.

3.4 Simulation Parameters

These were the fixed values used in all simulations:

Table 1: Parameters Used in PMCMC and ERTSS

Parameter	Value
Initial State	[1.5, 0]
Simulation Steps	500
Timestep	0.01

Table 2: Parameters Used in PMCMC only

Parameter	Value
Number of Particles	100
Initial Logarithm of Parameter (initial_r)	-3
Initial Proposal Noise	0.1
Number of MCMC Iterations	1000

Three main results were collected using my ERTSS and PMCMC implementations:

1. **Comparative Analysis of EKF/ERTSS and EKF:** I conducted a comparative study between the EKF and the EKF/ERTSS system. The study included exploratory plots showcasing the performance of each method, as well as RMSE analysis to quantitatively evaluate their effectiveness.

2. **Exploration of PF Approximate Energy Function:** I explored the Particle Filter approximate energy function for various values of the noise variance in the pendulum system. This exploration involved analyzing plots depicting the behavior of the energy function across different noise variance settings.
3. **PMCMC Analysis for Unknown Noise Measurement Values:** I employed the PMCMC method to generate samples and estimate parameters for a range of unknown noise measurement values in the pendulum system. The main analysis included generating histograms illustrating the distribution of parameter estimates obtained through PMCMC.

Note that for all reported RMSEs, the following expression was used to calculate the difference between two signals of interest x and y:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - y_i)^2}$$

4 Results

4.1 Comparative Analysis of EKF/ERTSS and EKF

I collected the following table over a large sweep of different sample intervals delta and measurement noises R:

Table 3: Comparison of RMSE for EKF and ERTSS under Varying Measurement Noise and time between measurements Delta

Delta	Measurement Noise	ERTSS RMSE	EKF RMSE
5	0.001	0.0140	0.0463
5	0.01	0.0304	0.1135
5	0.1	0.1930	0.4063
5	1	9.5892	9.9830
10	0.001	0.0215	0.0399
10	0.01	0.0372	0.0790
10	0.1	0.1175	0.2034
10	1	0.2818	0.3709
20	0.001	0.0212	0.0502
20	0.01	0.0450	0.1204
20	0.1	5.7470	5.6290
20	1	10.5793	10.2047
40	0.001	0.0324	0.0614
40	0.01	0.0663	0.0807
40	0.1	0.1296	0.1819
40	1	2.3894	2.7270

For various values of Delta, I plotted the EKF and EKF/ERTSS performance side-by-side, with the full figure in Figure 1, and a zoom-in on the first 10% of the performance comparison in Figure 2.

4.2 Exploration of PF Approximate Energy Function

For various true measurement noise values R, I conducted a sweep of measurement noises and estimated the energy function with my PF implementation. These estimated energy sweeps are plotted in Figure 3.

4.3 PMCMC Analysis for Unknown Noise Measurement Values

For various true, but unknown measurement noise values R, I ran PMCMC to estimate the value of R. Figure 4 shows the resulting histograms of MCMC samples, with table 4 reporting the results numerically.

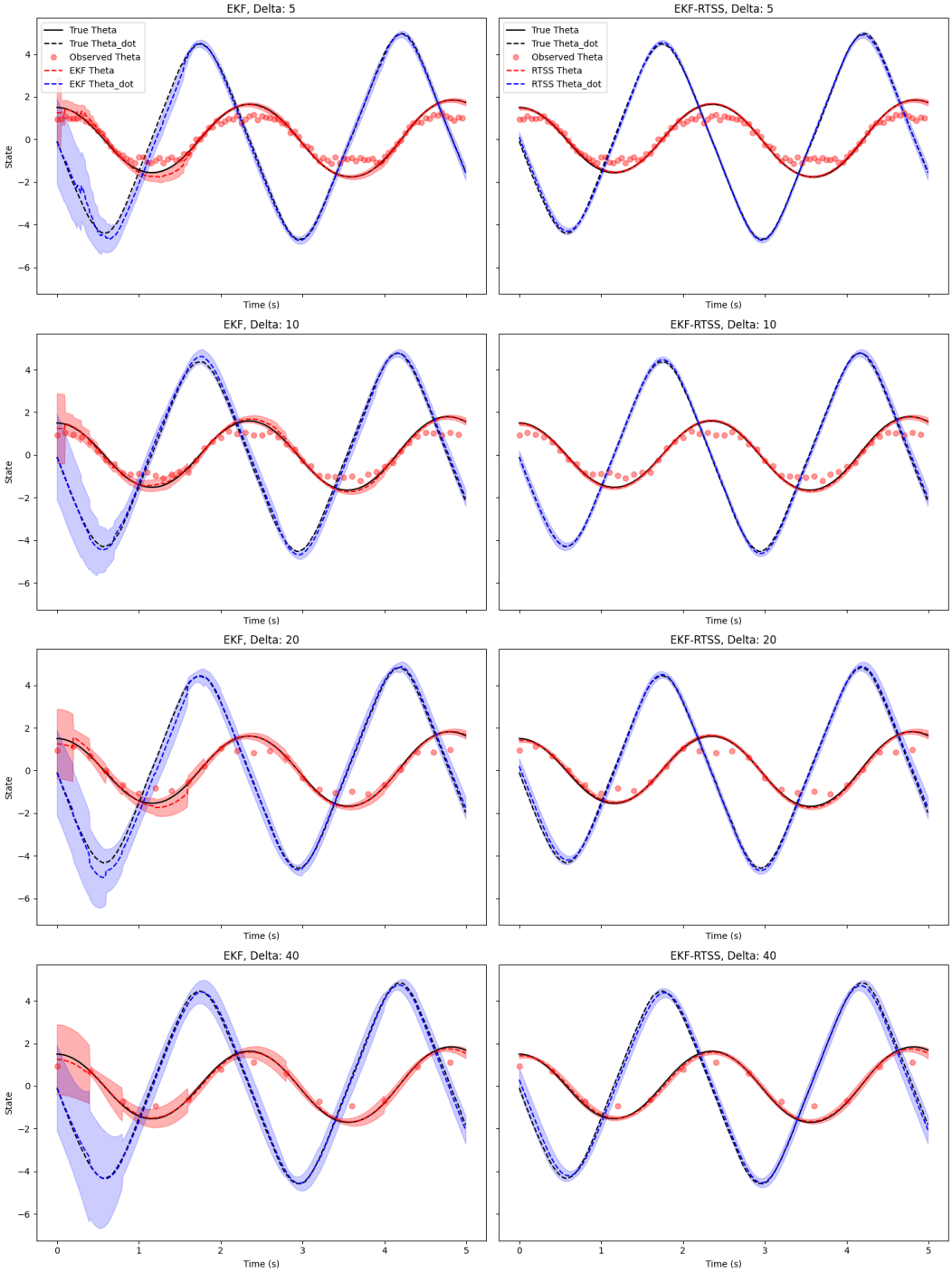


Figure 1: Full comparison of EKF and EKF/ERTSS performance.

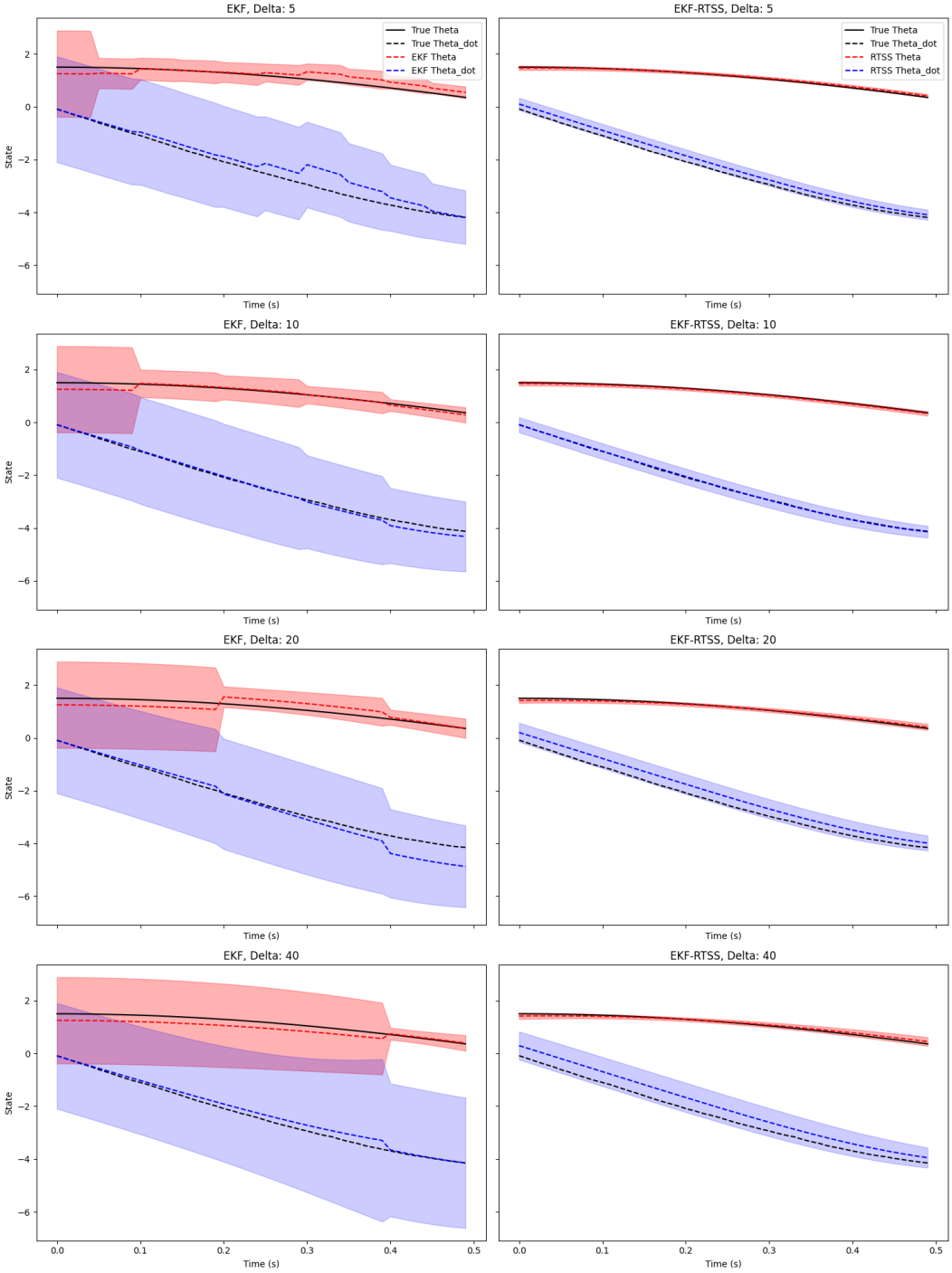


Figure 2: Zoom-in on the first 10% of the EKF and EKF/ERTSS performance comparison.

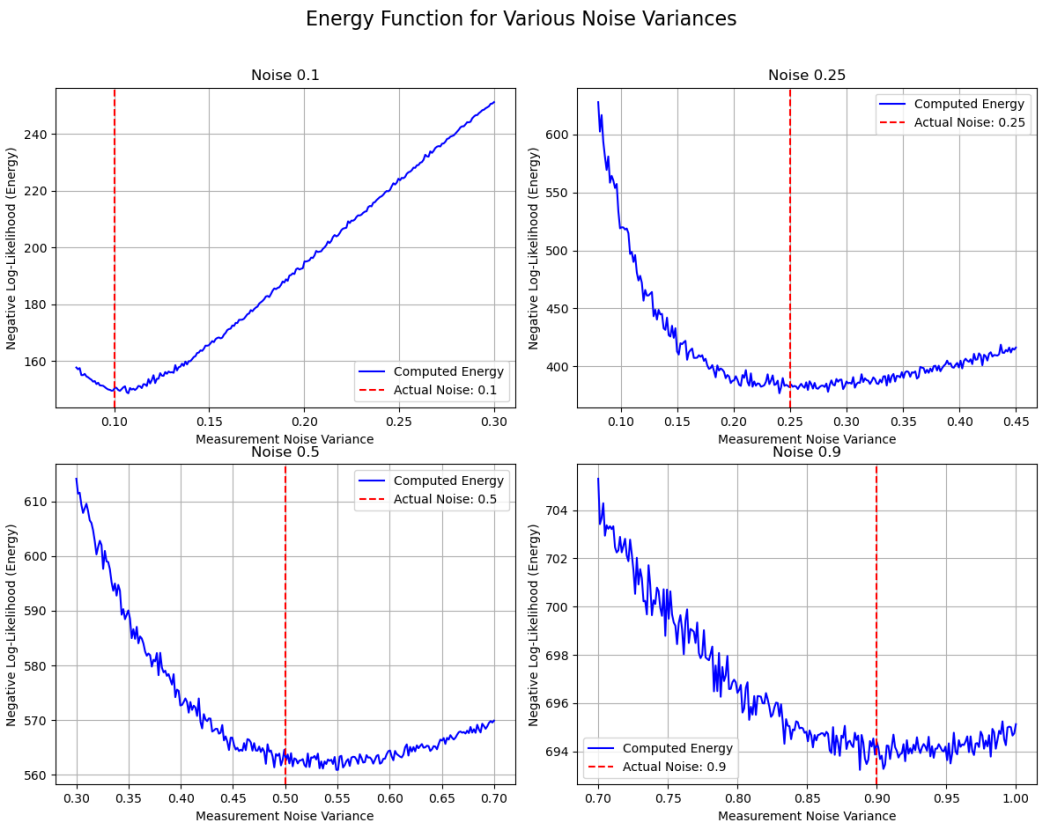


Figure 3: Estimated energy function sweeps using various measurement noises.

Table 4: Estimated Parameters for Different Levels of Measurement Noise

Measurement Noise	Estimated Parameter
0.1	0.1056
0.25	0.2602
0.5	0.5179
0.9	0.9462

5 Discussion

Three main results were collected using my ERTSS and PMCMC implementations:

1. **Comparative Analysis of EKF/ERTSS and EKF:** I conducted a comparative study between the EKF and the EKF/ERTSS system. The study included exploratory plots showcasing the performance of each method, as well as Mean Squared Error analysis to quantitatively evaluate their effectiveness.
2. **Exploration of PF Approximate Energy Function:** I explored the Particle Filter approximate energy function for various values of the noise variance in the pendulum system. This exploration involved analyzing plots depicting the behavior of the energy function across different noise variance settings.
3. **PMCMC Analysis for Unknown Noise Measurement Values:** I employed the PMCMC method to generate samples and estimate parameters for a range of unknown noise measurement values in the pendulum system. The main analysis included generating histograms illustrating the distribution of parameter estimates obtained through PMCMC.

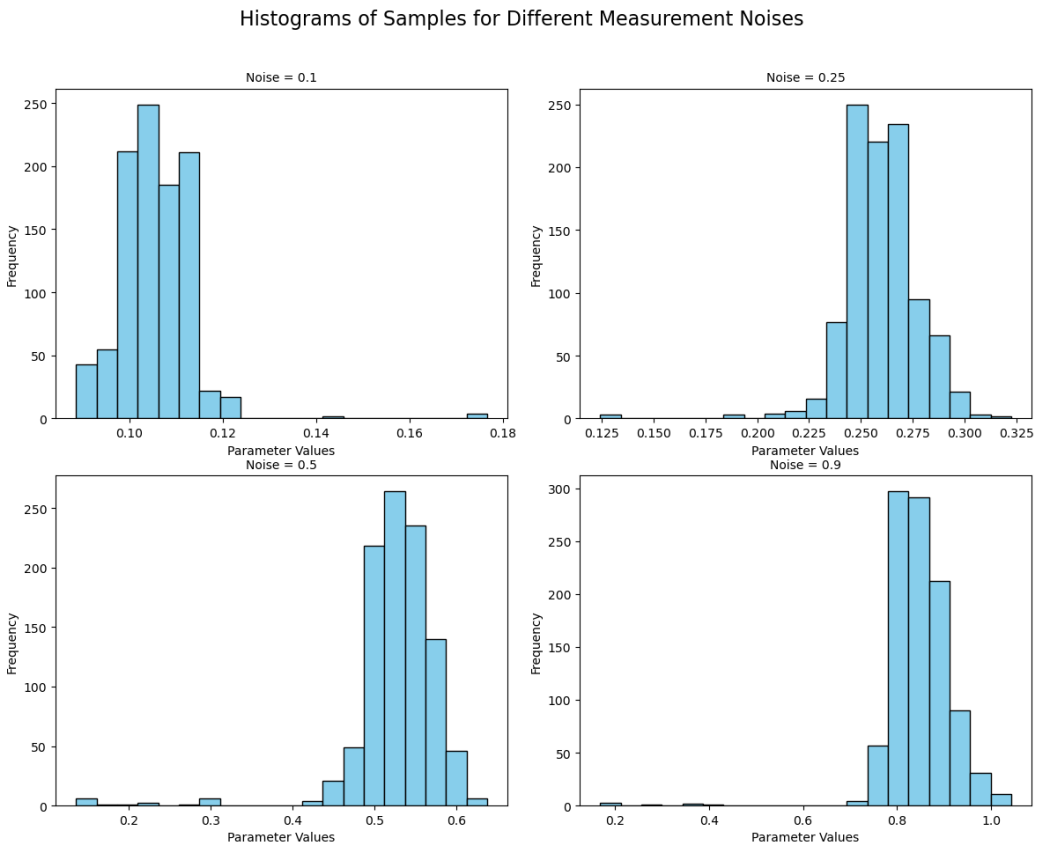


Figure 4: Estimated Parameter histograms for different underlying measurement noises. True measurement noises are listed in each subfigure's title.

6 Discussion

Comparative Analysis of EKF/ERTSS and EKF: The results of comparing EKF/ERTSS to EKF show that EKF/ERTSS consistently achieves lower RMSE values. This is expected as EKF/ERTSS is given more information for each optimal state prediction. In the pendulum example, this improvement is especially pronounced at the start of the trial, where the availability of future data enables ERTSS to quickly correct any initial tracking errors. The difference in performance is particularly evident in the early stages, as shown in Figure 2, where the standard deviation of the filter converges much more quickly in EKF/ERTSS.

Exploration of PF Approximate Energy Function: During the noise variance sweeps, the minima found align well with the true noise value r. This is expected, as the lowest energy should be achieved when precisely matching the true underlying value.

There is significant noise in the results, particularly at higher levels of true measurement noise. This variability makes it challenging to accurately determine the parameters θ. Note that the minimum energy values become much higher as the underlying measurement noise increases. This intuitively makes sense, as we are unable to find as close of a fit for very noisy measurements.

On the computational side, MCMC calls the n-sample, t-step PF for every proposal sample it generates. This is extraordinarily computationally demanding. In order to run this algorithm on my local machine in a reasonable timeframe, the amount of MCMC samples and PF particles is relatively low. The reduced number of particles likely introduces additional noise in the energy function, but this has not significantly affected the results as they still seem promising.

PMCMC Analysis for Unknown Noise Measurement Values: The PMCMC method yielded parameter estimates that closely match the true values, with higher variability noted as the measurement noise increases. This variability complicates the task of discerning the underlying model but is clearly depicted in the histograms in Figure 4. However, despite the spread of these histograms increasing for higher values of measurement noise, they still seem to be reasonably accurate approximations, validating the usefulness of PMCMC in system identification.

7 Conclusion

This study has explored extensions of the base Kalman Filter and Particle Filter techniques to enhance state and parameter estimation in dynamic systems, particularly focusing on a pendulum model. The Extended Rauch-Tung-Striebel Smoother demonstrated significantly improved performance over the standard Extended Kalman Filter, especially noticeable at the start of the state estimation process. This advantage is primarily due to the smoother's capability to utilize future observations, which effectively corrects initial estimation errors more rapidly and robustly.

Additionally, the integration of Particle Filters with Markov Chain Monte Carlo methods has proven effective in estimating unknown measurement noise parameters within the system. The ability of PMCMC to accurately capture the true noise parameters—even under varying levels of noise—illustrates its robustness and adaptability in complex stochastic environments.

For future research, it would be interesting to examine more complicated systems, and attempt to estimate multiple unknown parameters at once. Additionally, it would be interesting to try PMCMC with a more optimal proposal distribution for the PF, such as using an Unscented Kalman Filter proposal.

References

1. A.A. Gorodetsky, 2019. Lecture Notes AEROSP567.
2. S. Särkkä, *Bayesian Filtering and Smoothing*. Cambridge University Press, 2013.

8 Appendix: Code

8.1 Code for EKF/ERTSS

ERTSS

May 1, 2024

[1]:
```
%matplotlib inline
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
LW=5 # linewidth
MS=10 # markersize
```

[2]:
```
from typing import Tuple, List, Optional
from dataclasses import dataclass
def rmse(x, y):
    return np.sqrt(np.mean((x - y)**2))

@dataclass
class Observations:
    times: np.ndarray
    obs_ind: np.ndarray  # index of times that are observed
    obs: np.ndarray
    names: List[str]

@dataclass
class KFTracker:
    means: np.ndarray
    covs: np.ndarray
    stds: np.ndarray

def generate_pendulum(m_0, g, Q, dt, R, steps, observation_interval=10, seed=1):
    np.random.seed(seed)
    m_0 = np.atleast_1d(m_0)
    Q = np.atleast_2d(Q)
    chol_Q = np.linalg.cholesky(Q)
    sqrt_R = np.sqrt(R)

    states = np.zeros((steps, 2))
    observations = []
    observation_times = []
```

```
    times = []
    obs_ind = np.arange(0, steps, observation_interval)
    state = m_0.copy()

    for i in range(steps):
        state = np.array([state[0] + dt * state[1], state[1] - g * dt * np.
↪sin(state[0])])
        state += chol_Q @ np.random.randn(2)
        states[i] = state
        times.append(i * dt)

        if i % observation_interval == 0:
            current_observation = np.sin(state[0]) + sqrt_R * np.random.randn()
            observations.append(current_observation)
            observation_times.append(i * dt)

    observations = np.array(observations)
    observation_times = np.array(observation_times)
    trueData = Observations(np.array(times), np.arange(steps), states,␣
↪['state0', 'state1'])
    obsData = Observations(np.array(observation_times), obs_ind, observations.
↪reshape(-1, 1), ['obs0'])

    return trueData, obsData
```

[3]:
```
from numpy.linalg import solve
def ekf_filter(init_mean, init_cov, gravity, process_noise, timestep,␣
↪measure_noise, measurements, steps, delta=10):
    dim = init_mean.shape[0]
    filtered_means = np.empty((steps, dim))
    filtered_covariances = np.empty((steps, dim, dim))
    filtered_stds = np.empty((steps, dim))

    mean = init_mean.copy()
    covariance = init_cov.copy()
    measurement_index = 0  # Index to track which measurement to use

    for step in range(steps):
        # Prediction step
        F = np.array([[1, timestep], [-gravity * timestep * np.cos(mean[0]),␣
↪1]])
        mean = np.array([mean[0] + timestep * mean[1], mean[1] - gravity *␣
↪timestep * np.sin(mean[0])])
        covariance = F @ covariance @ F.T + process_noise

        # Update step if it's time to measure
```

```
        if step % delta == 0 and measurement_index < len(measurements):
            current_meas = measurements[measurement_index]
            measurement_index += 1

            h = np.sin(mean[0])
            H = np.array([[np.cos(mean[0]), 0]])
            S = H @ covariance @ H.T + measure_noise
            K = solve(S, H @ covariance).T

            mean += K @ (current_meas - h)
            covariance -= K @ S @ K.T

        # Store the results at each step
        filtered_means[step] = mean
        filtered_covariances[step] = covariance
        filtered_stds[step] = np.sqrt(np.diag(covariance))

        kf = KFTracker(filtered_means, filtered_covariances, filtered_stds)

    return kf
```

[4]:
```
def extended_smoother(state_means, state_covariances, gravity, process_noise,␣
↪timestep):
    num_steps, dim = state_means.shape
    smoothed_means = np.zeros_like(state_means)
    smoothed_covariances = np.zeros((num_steps, dim, dim))

    last_mean = state_means[-1]
    last_cov = state_covariances[-1]

    smoothed_means[-1] = last_mean
    smoothed_covariances[-1] = last_cov

    for step in reversed(range(num_steps - 1)):
        current_mean = state_means[step]
        current_cov = state_covariances[step]

        transition_matrix = np.array([[1., timestep],
                                      [-gravity * timestep * np.
↪cos(current_mean[0]), 1.]])

        predicted_mean = np.array([current_mean[0] + timestep * current_mean[1],
                                   current_mean[1] - gravity * timestep * np.
↪sin(current_mean[0])])
        predicted_cov = transition_matrix @ current_cov @ transition_matrix.T +␣
↪process_noise
```

```
        kalman_gain = solve(predicted_cov, transition_matrix @ current_cov).T

        last_mean = current_mean + kalman_gain @ (last_mean - predicted_mean)
        last_cov = current_cov + kalman_gain @ (last_cov - predicted_cov) @␣
 ↪kalman_gain.T

        smoothed_means[step] = last_mean
        smoothed_covariances[step] = last_cov

    return smoothed_means, smoothed_covariances
```

[5]:
```
from itertools import product
x0 = np.array([1.5, 0.0])
gravity = 9.81
timestep = 0.01
process_noise = 0.01 * np.array([[timestep ** 3 / 3, timestep ** 2 / 2],␣
 ↪[timestep ** 2 / 2, timestep]])
measure_noises = [0.001, 0.01, 0.1, 1]
steps = 500
deltas = [5, 10, 20, 40]
order=3

init_mean = np.array([1.5, 0.0])
init_cov = np.eye(2)

# Running the filters
for delta in deltas:
    for measure_noise in measure_noises:
        print()
        print(f"Delta: {delta}", f"Measure Noise: {measure_noise}")
        trueData, obsData = generate_pendulum(x0, gravity, process_noise,␣
 ↪timestep, measure_noise, steps, delta)
        ekf = ekf_filter(init_mean, init_cov, gravity, process_noise, timestep,␣
 ↪measure_noise, obsData.obs, steps, delta)
        smoother_mean, smoother_cov = extended_smoother(ekf.means, ekf.covs,␣
 ↪gravity, process_noise, timestep)
        means_ekf = ekf.means
        covariances_ekf = ekf.covs

        rmse_smoothed = rmse(smoother_mean[:, :1], trueData.obs[:, :1])
        print(f"Smoothed RMSE: {rmse_smoothed}")

        rmse_ekf = rmse(means_ekf[:, :1], trueData.obs[:, :1])
        print(f"ekf RMSE: {rmse_ekf}")
```

```
Delta: 5 Measure Noise: 0.001
```

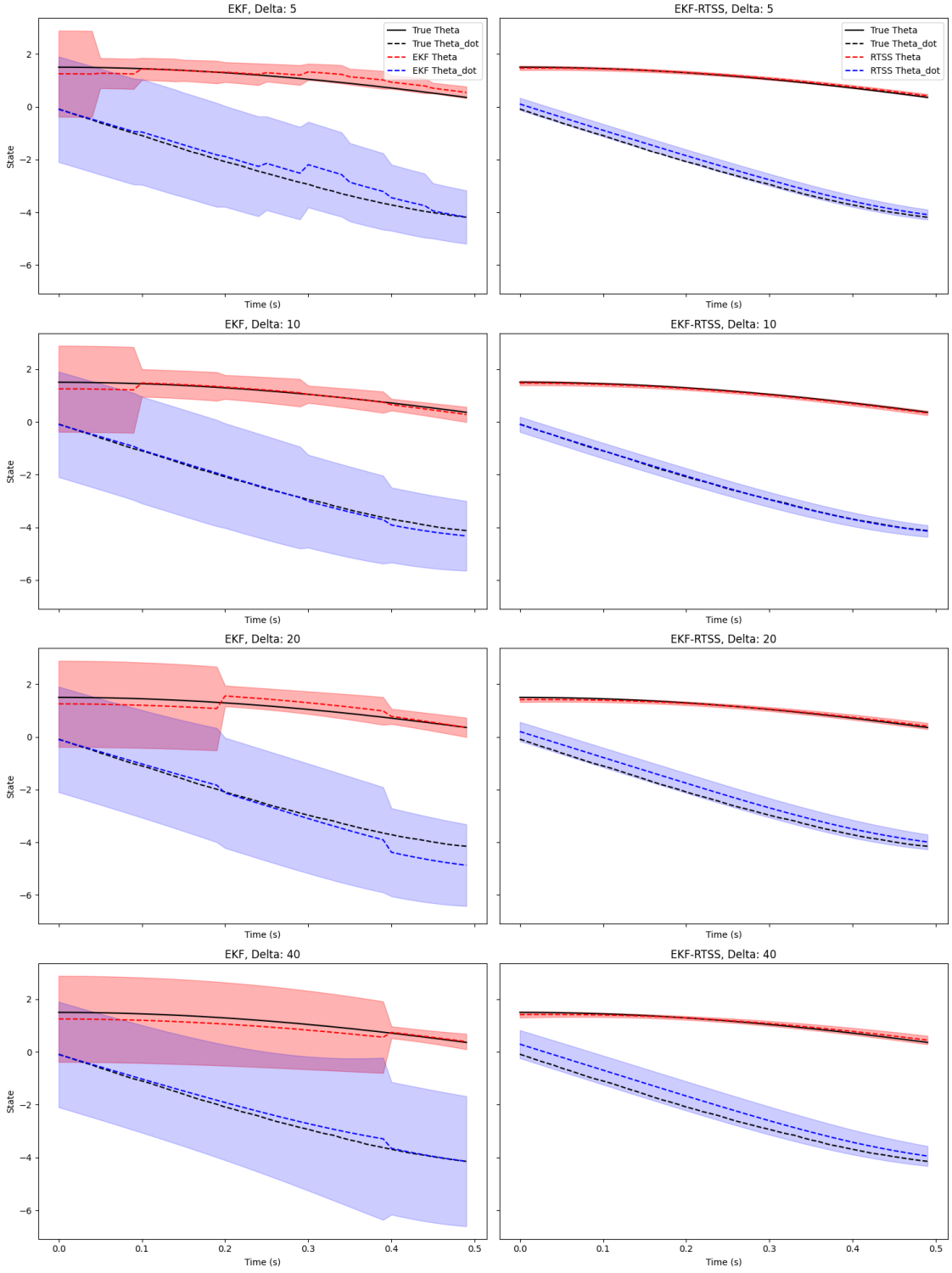
```
ghkf_3 RMSE: 0.014044246183956766
ekf RMSE: 0.046259406815845346

Delta: 5 Measure Noise: 0.01
ghkf_3 RMSE: 0.03044856388842772
ekf RMSE: 0.11349123614485017

Delta: 5 Measure Noise: 0.1
ghkf_3 RMSE: 0.19301202393928338
ekf RMSE: 0.4063150463479567

Delta: 5 Measure Noise: 1
ghkf_3 RMSE: 9.589205915165437
ekf RMSE: 9.982987110795175

Delta: 10 Measure Noise: 0.001
ghkf_3 RMSE: 0.021490467916483198
ekf RMSE: 0.039866362521668905

Delta: 10 Measure Noise: 0.01
ghkf_3 RMSE: 0.037233533694314175
ekf RMSE: 0.07895963097629806

Delta: 10 Measure Noise: 0.1
ghkf_3 RMSE: 0.11745393127431357
ekf RMSE: 0.20340179922169704

Delta: 10 Measure Noise: 1
ghkf_3 RMSE: 0.2817878793816097
ekf RMSE: 0.3709257844529935

Delta: 20 Measure Noise: 0.001
ghkf_3 RMSE: 0.0212044426739208
ekf RMSE: 0.0502060316434209

Delta: 20 Measure Noise: 0.01
ghkf_3 RMSE: 0.04503552854986252
ekf RMSE: 0.12037744164480776

Delta: 20 Measure Noise: 0.1
ghkf_3 RMSE: 5.746997669663604
ekf RMSE: 5.628970943950402

Delta: 20 Measure Noise: 1
ghkf_3 RMSE: 10.579326208819948
ekf RMSE: 10.204744039208945

Delta: 40 Measure Noise: 0.001
```

```
ghkf_3 RMSE: 0.032392467982106016
ekf RMSE: 0.06139312612357028

Delta: 40 Measure Noise: 0.01
ghkf_3 RMSE: 0.0662813770166021
ekf RMSE: 0.0807129551960485

Delta: 40 Measure Noise: 0.1
ghkf_3 RMSE: 0.12961036934122613
ekf RMSE: 0.18194711021686372

Delta: 40 Measure Noise: 1
ghkf_3 RMSE: 2.3893593067260843
ekf RMSE: 2.7269644215151394
```

[ ]:

[6]:
```
fig, axs = plt.subplots(4, 2, figsize=(15, 20), sharex=True, sharey=True)

measure_noise = 0.01
timestep = 0.01
steps = 500

# Iterate through each delta
for i, delta in enumerate(deltas[:4]):  # Assuming there are at least 4 deltas␣
 ↪in the array
    # Generate pendulum data
    trueData, obsData = generate_pendulum(x0, gravity, process_noise, timestep,␣
 ↪measure_noise, steps, delta)

    # EKF
    ekf = ekf_filter(init_mean, init_cov, gravity, process_noise, timestep,␣
 ↪measure_noise, obsData.obs, steps, delta)

    # EKF with RTS Smoother
    rts_m, rts_P = extended_smoother(ekf.means, ekf.covs, gravity,␣
 ↪process_noise, timestep)
    rts_Std = np.zeros((rts_P.shape[0], 2))
    for x in range(rts_P.shape[0]):
        rts_Std[x] = np.sqrt(np.diag(rts_P[x]))

    trueData.times = trueData.times[:50]
    obsData.times = obsData.times[:50]

    # Plotting for EKF
    ax = axs[i, 0]
    ax.plot(trueData.times, trueData.obs[:50, 0], 'k', label='True Theta')
```

```
    ax.plot(trueData.times, trueData.obs[:50, 1], 'k--', label='True Theta_dot')
    #ax.plot(obsData.times, obsData.obs[:50, 0], 'ro', alpha=0.4,␣
 ↪label='Observed Theta')
    ax.plot(trueData.times, ekf.means[:50, 0], 'r--', label='EKF Theta')
    ax.plot(trueData.times, ekf.means[:50, 1], 'b--', label='EKF Theta_dot')
    ax.fill_between(trueData.times,
                         ekf.means[:50, 0] - 2 * ekf.stds[:50, 0],
                         ekf.means[:50, 0] + 2 * ekf.stds[:50, 0],
                         color='red', alpha=0.3)
    ax.fill_between(trueData.times,
                        ekf.means[:50, 1] - 2 * ekf.stds[:50, 1],
                        ekf.means[:50, 1] + 2 * ekf.stds[:50, 1],
                        color='blue', alpha=0.2)
    ax.set_title(f"EKF, Delta: {delta}")
    ax.set_xlabel('Time (s)')
    ax.set_ylabel('State')

    # Plotting for EKF with RTS
    ax = axs[i, 1]
    ax.plot(trueData.times, trueData.obs[:50, 0], 'k', label='True Theta')
    ax.plot(trueData.times, trueData.obs[:50, 1], 'k--', label='True Theta_dot')
    #ax.plot(obsData.times, obsData.obs[:50, 0], 'ro', alpha=0.4,␣
 ↪label='Observed Theta')
    ax.plot(trueData.times, rts_m[:50, 0], 'r--', label='RTSS Theta')
    ax.plot(trueData.times, rts_m[:50, 1], 'b--', label='RTSS Theta_dot')
    ax.fill_between(trueData.times, rts_m[:50, 0] - 2 * rts_Std[:50, 0], rts_m[:
 ↪50, 0] + 2 * rts_Std[:50, 0], color='red', alpha=0.3)
    ax.fill_between(trueData.times, rts_m[:50, 1] - 2 * rts_Std[:50, 1], rts_m[:
 ↪50, 1] + 2 * rts_Std[:50, 1], color='blue', alpha=0.2)
    ax.set_title(f"EKF-RTSS, Delta: {delta}")
    ax.set_xlabel('Time (s)')

    if i == 0:  # Add legend to the first row only
        axs[i, 0].legend()
        axs[i, 1].legend()

plt.tight_layout()
plt.show()
```


EKF, Delta: 5
EKF-RTSS, Delta: 5
EKF, Delta: 10
EKF-RTSS, Delta: 10
EKF, Delta: 20
EKF-RTSS, Delta: 20
EKF, Delta: 40
EKF-RTSS, Delta: 40
True Theta
True Theta_dot
EKF Theta
EKF Theta_dot
RTSS Theta
RTSS Theta_dot
State
Time (s)

8.2 Code for PMCMC

parameter_est

May 1, 2024

[ ]:
```
import numpy as np
from scipy import linalg, stats
import matplotlib.pyplot as plt
import tqdm
import sys



def generate_pendulum(m_0, g, Q, dt, R, steps, seed=0):
    rng = np.random.default_rng(seed)  # Create a new generator

    m_0 = np.atleast_1d(m_0)
    Q = np.atleast_2d(Q)

    states = np.empty((steps, 2))
    observations = np.empty(steps)

    chol_Q = np.linalg.cholesky(Q)
    sqrt_R = np.sqrt(R)

    state = m_0

    for i in range(steps):
        state = np.array([state[0] + dt * state[1],
                          state[1] - g * dt * np.sin(state[0])])
        state += chol_Q @ rng.standard_normal(2)  # Add process noise
        states[i, :] = state

        observations[i] = np.sin(state[0]) + sqrt_R * rng.standard_normal()  #␣
 ↪Add measurement noise

    return np.arange(dt, (steps + 1) * dt, dt), states, observations

def stratified_resampling(weights, rng):
    n_particles = len(weights)
    cumulative_sum = np.cumsum(weights)
```

```
    uniform_samples =  np.random.default_rng(42).random(n_particles) + np.
 ↪linspace(0., n_particles - 1., n_particles)
    indices = np.searchsorted(cumulative_sum, uniform_samples / n_particles)
    return np.clip(indices, 0, n_particles - 1)

def normalize_weights(weights):
    return weights / np.sum(weights)

def particle_filter(initial_mean, initial_cov, gravity, process_noise_cov, dt,␣
 ↪r, observations, n_particles):
    steps = len(observations)
    chol_initial_cov = np.linalg.cholesky(initial_cov)
    chol_process_noise = np.linalg.cholesky(process_noise_cov)
    sqrt_measurement_noise = np.sqrt(r)
    rng = np.random.default_rng(42)

    particles = initial_mean.reshape(2, 1) + np.dot(chol_initial_cov,  rng.
 ↪standard_normal((2, n_particles)))

    log_likelihood = 0.

    norm_dist = stats.norm(0., sqrt_measurement_noise)

    for i in range(steps):
        particles[0, :] += dt * particles[1, :]
        particles[1, :] -= gravity * dt * np.sin(particles[0, :])
        particles += np.dot(chol_process_noise,  rng.standard_normal((2,␣
 ↪n_particles)))
        weights = norm_dist.pdf(np.sin(particles[0, :]) - observations[i])

        log_likelihood += np.log(np.mean(weights))
        weights = normalize_weights(weights)
        indices = stratified_resampling(weights, rng)
        particles = particles[:, indices]

    return log_likelihood



def compute_energy(initial_mean, initial_cov, gravity, process_noise_cov, dt,␣
 ↪measurement_noise, observations, n_particles):
    """Compute the energy using the particle filter for a given noise level."""
    negative_log_likelihood = 0
    particles = np.random.multivariate_normal(initial_mean, initial_cov,␣
 ↪size=n_particles)
    weights = np.ones(n_particles) / n_particles
```

```
    for obs in observations:
        # Predict step
        particles[:, 0] += dt * particles[:, 1]
        particles[:, 1] -= gravity * dt * np.sin(particles[:, 0])
        particles += np.random.multivariate_normal([0, 0], process_noise_cov,␣
↪size=n_particles)

        # Update step
        likelihood = norm.pdf(np.sin(particles[:, 0]), loc=obs, scale=np.
↪sqrt(measurement_noise))
        weights *= likelihood
        weights /= np.sum(weights)  # Normalize weights

        # Resample if necessary
        if (1 / np.sum(weights**2)) < n_particles / 2:  # Effective sample size
            indices = np.random.choice(n_particles, size=n_particles, p=weights)
            particles = particles[indices]
            weights.fill(1/n_particles)

        negative_log_likelihood -= np.log(np.sum(likelihood) / n_particles)

    return negative_log_likelihood




def pmcmc(initial_r, proposal_noise, target_acceptance_rate, num_mcmc,␣
↪initial_mean, initial_cov, gravity, process_noise_cov, dt,␣
↪measurement_noise, observations, n_particles):
    current_energy = float("inf")
    samples = np.empty(num_mcmc)
    energy_history = np.empty(num_mcmc)  # Array to track energy
    acceptance_count = 0

    progress_bar = tqdm.trange(num_mcmc)
    progress_bar.set_description(f'MCMC(accepted=0,␣
↪current_log(r)={initial_r})')
    r = initial_r
    rng = np.random.default_rng(42)

    for i in progress_bar:
        proposed_r = r + proposal_noise * rng.standard_normal()
        negative_log_likelihood = -particle_filter(initial_mean, initial_cov,␣
↪gravity, process_noise_cov, dt, np.exp(proposed_r), observations,␣
↪n_particles)
```

```
        acceptance_probability = min(1, np.exp(current_energy - negative_log_likelihood))
        if rng.random() <= acceptance_probability:
            samples[i] = proposed_r
            r = proposed_r
            current_energy = negative_log_likelihood
            acceptance_count += 1
        else:
            samples[i] = r

        energy_history[i] = current_energy  # Store current energy
        progress_bar.set_description(f'MCMC(accepted={acceptance_count}, current_log(r)={r}, proposal_noise={proposal_noise}, energy={current_energy})')

        if i > 10 and acceptance_count / (i + 1) < target_acceptance_rate:
            proposal_noise_adjustment = 1 / (i ** 0.9)
            proposal_noise *= np.sqrt(1 + proposal_noise_adjustment * (acceptance_probability - target_acceptance_rate))

    return samples, acceptance_count, energy_history


# Constants
TIME_STEP = 0.01
GRAVITY = 9.81
PROCESS_NOISE_COV = 0.01 * np.array([[TIME_STEP ** 3 / 3, TIME_STEP ** 2 / 2],
                                     [TIME_STEP ** 2 / 2, TIME_STEP]])
INITIAL_STATE = np.array([1.5, 0.])

# Simulation Settings
simulation_steps = 500

# Setup for MCMC
num_particles = 100
num_mcmc = 1000
initial_r = -2
initial_proposal_noise = 0.25
target_acceptance = 0.2
import matplotlib.pyplot as plt

# Measurement noise values
noise_values = [0.1, 0.25, 0.5, 0.9]
# Initialize plot for histograms
fig1, axes1 = plt.subplots(nrows=2, ncols=2, figsize=(12, 12))
fig1.suptitle('Histograms of Samples for Different Measurement Noises', fontsize=16)
```

```
# Initialize plot for energy convergence
fig2, axes2 = plt.subplots(nrows=2, ncols=2, figsize=(12, 12))
fig2.suptitle('Energy Convergence for Different Measurement Noises',␣
 ↪fontsize=16)

# Run MCMC for different measurement noises
for i, noise in enumerate(noise_values):
    timeline, true_states, observations = generate_pendulum(INITIAL_STATE,␣
 ↪GRAVITY, PROCESS_NOISE_COV,
                                                            TIME_STEP, noise,␣
 ↪simulation_steps)
    print(f"Running MCMC for MEASUREMENT_NOISE = {noise}")
    samples, accepted, energy_history = pmcmc(initial_r,␣
 ↪initial_proposal_noise, target_acceptance, num_mcmc,
                                              INITIAL_STATE, PROCESS_NOISE_COV,␣
 ↪GRAVITY, PROCESS_NOISE_COV,
                                              TIME_STEP, noise, observations,␣
 ↪num_particles)

    exp_samples = np.exp(samples)  # Exponentiating the samples
    mean_estimate = np.mean(exp_samples)
    print(f"Estimated parameter for MEASUREMENT_NOISE {noise}: {mean_estimate}")

    # Determine subplot indices
    row, col = divmod(i, 2)

    # Plot histogram of exponentiated samples
    ax1 = axes1[row, col]
    ax1.hist(exp_samples, bins=20, color='skyblue', edgecolor='black')
    ax1.set_title(f'Noise = {noise}', fontsize=10)
    ax1.set_xlabel('Parameter Values')
    ax1.set_ylabel('Frequency')
    ax1.axvline(x=noise, color='r', linestyle='--', label=f'Actual Noise:␣
 ↪{noise}')

    # Plot energy convergence
    ax2 = axes2[row, col]
    ax2.plot(energy_history, color='red')
    ax2.set_title(f'Noise = {noise}', fontsize=10)
    ax2.set_xlabel('Iteration')
    ax2.set_ylabel('Energy')

# Adjust layout and display the plots
plt.tight_layout()
fig1.tight_layout(rect=[0, 0.03, 1, 0.95])
```

```
fig2.tight_layout(rect=[0, 0.03, 1, 0.95])

plt.show()

# # Plotting the energy function only
# import matplotlib.pyplot as plt
# from scipy.stats import norm

# # Measurement noise values to be used for data generation
# actual_noises = [0.1, 0.25, 0.5, 0.9]
# num_particles = 1000

# # Setup for plots
# fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(12, 12))
# fig.suptitle('Energy Function for Various Noise Variances', fontsize=16)

# for index, actual_noise in enumerate(actual_noises):
#     # Generate data with the current actual noise
#     timeline, true_states, observations = generate_pendulum(INITIAL_STATE,␣
 ↪GRAVITY, PROCESS_NOISE_COV,
#                                                             TIME_STEP,␣
 ↪actual_noise, simulation_steps)

#     # Sweep over a range of noise variances to compute energy
#     low = max(0.08, actual_noise-0.2)
#     high = min(1, actual_noise+0.2)
#     noise_variances = np.linspace(low, high, 250)
#     energies = []
#     for noise in noise_variances:
#         energy = compute_energy(INITIAL_STATE, PROCESS_NOISE_COV, GRAVITY,␣
 ↪PROCESS_NOISE_COV,
#                                 TIME_STEP, noise, observations, num_particles)
#         energies.append(energy)

#     # Select subplot
#     ax = axes[index // 2, index % 2]


#     ax.plot(noise_variances, energies, linestyle='-', color='b',␣
 ↪label='Computed Energy')
#     ax.axvline(x=actual_noise, color='r', linestyle='--', label=f'Actual␣
 ↪Noise: {actual_noise}')
#     ax.set_xlabel('Measurement Noise Variance')
#     ax.set_ylabel('Negative Log-Likelihood (Energy)')
#     ax.set_title(f'Noise {actual_noise}')
#     ax.grid(True)
#     ax.legend()
```

```
# plt.tight_layout(rect=[0, 0.03, 1, 0.95])
# plt.show()
```