Analysis of Gaussian and Particle Filtering Techniques on Non-linear Dynamics

Marco Conati

University of Michigan

April 20, 2024

Contents

1 Introduction

In this third project for AEROSP567, we are tasked with employing filtering algorithms to estimate the states of a nonlinear pendulum system, governed by its angle and angular rate. Through these methods, we aim to effectively manage and predict the pendulum's dynamics amid inherent uncertainties in measurement and process noises.

2 Model Description

2.1 Pendulum Dynamics

This project models the time-discretized dynamics of a simple pendulum characterized by two states: the angle x_1 and the angular rate x_2. The state of the pendulum at each discrete timestep k evolves according to the nonlinear equations:

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

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

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

where $q_c = 0.1$ is the process noise coefficient. 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. For all experiments, simulations were run for 500 timesteps, beginning from the initial conditions $x_0 = (1.5, 0)$. A standard Gaussian centered at these initial conditions, with a standard deviation of 1, was used as the prior for the initial state estimates.

3 Filtering Techniques

3.1 Gaussian Filters

Gaussian filters are a class of algorithms for Bayesian filtering, where the underlying distributions are assumed to be Gaussian. They operate on the principle of Bayes' rule, iteratively performing prediction and update steps to estimate the state of a system over time. The prediction step projects the current state estimate forward, while the update step refines this prediction based on new measurements.

The prediction step is formulated as an expectation of the state given the previous measurements, mathematically described as:

$$m_k = E[X_k|Y_{k-1}] = \int \phi(X_{k-1})N(X_{k-1}; m_{k-1}, C_{k-1})dX_{k-1},$$

and the covariance of the state is the uncertainty of this prediction:

$$C_k = \text{Cov}[X_k, X_k|Y_{k-1}] = \int (\phi(X_{k-1}) - m_k)(\phi(X_{k-1}) - m_k)^T N(X_{k-1}; m_{k-1}, C_{k-1})dX_{k-1} + \Sigma,$$

where ϕ denotes the state transition function, and Σ represents the process noise covariance.

The update step involves calculating the expected measurement and its uncertainty, along with the covariance between the state and measurement:

$$\mu = E[Y_k] = \int h(X_k)N(X_k; m_k, C_k)dX_k,$$

$$S = \text{Cov}[Y_k, Y_k] = \int (h(X_k) - \mu)(h(X_k) - \mu)^T N(X_k; m_k, C_k)dX_k + R,$$

$$U = \text{Cov}[X_k, Y_k] = \int (X_k - m_k)(h(X_k) - \mu)^T N(X_k; m_k, C_k)dX_k,$$

where h is the measurement function, and R is the measurement noise covariance.

Finally, the Kalman gain is used to incorporate the new measurement into the state estimate:

$$m_k = m_k + US^{-1}(y_k - \mu),$$

$$C_k = C_k - US^{-1}U^T,$$

which concludes the Gaussian filter update cycle.

However, the direct evaluation of these integrals is infeasible for nonlinear functions ϕ and h. To address this, Gaussian filters adopt different strategies:

- **EKF (Extended Kalman Filter)**: Linearizes the nonlinear functions around the current estimate to approximate the integrals.
- **UKF (Unscented Kalman Filter)**: Utilizes a deterministic set of sigma points to capture the mean and covariance accurately.
- **GHKF (Gauss-Hermite Kalman Filter)**: Employs numerical quadrature methods to estimate the integrals.

Each of these methods provides a unique approach to circumventing the intractability of the nonlinear integrals in the Gaussian filter equations, ensuring that the filtering process remains both robust and computationally feasible.

3.1.1 Simplifying Gaussian Filtering Equations with the known Pendulum Dynamics

We can analytically simplify parts of the Gaussian filtering equations, but some of the integrals require EKF/UKF/GHKF or other strategies. In this section, we try to simplify the Gaussian Filter integrals as much as possible.

1. The integral for m_k simplifies using properties of expectations and the characteristics of linear functions of Gaussian random variables. We start by expressing the dynamics and recognizing the linear and nonlinear parts:

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

The expectation integral for each component becomes:

$$m_k = \mathbb{E}[X_k|Y_{k-1}] = \int \phi(X_{k-1})N(X_{k-1}; m_{k-1}, C_{k-1})\, dX_{k-1}$$

$$= \begin{pmatrix} \int (x_{k-1,1} + x_{k-1,2}\Delta t)N(X_{k-1}; m_{k-1}, C_{k-1})\, dX_{k-1} \\ \int (x_{k-1,2} - g\sin(x_{k-1,1})\Delta t)N(X_{k-1}; m_{k-1}, C_{k-1})\, dX_{k-1} \end{pmatrix}$$

Applying the linearity of expectation, the first component is a linear function of X_{k-1}:

$$\mathbb{E}[x_{k-1,1} + x_{k-1,2}\Delta t] = \mathbb{E}[x_{k-1,1}] + \mathbb{E}[x_{k-1,2}]\Delta t = m_{k-1,1} + m_{k-1,2}\Delta t$$

The second component involves the non-linear function $\sin(x_{k-1,1})$, which complicates direct integration. We approximate the expectation of this term separately:

$$\mathbb{E}[x_{k-1,2} - g\sin(x_{k-1,1})\Delta t] = m_{k-1,2} - g\mathbb{E}[\sin(x_{k-1,1})]\Delta t$$

Thus, the simplified expectation of the state vector m_k under the Gaussian assumption with the known mean and covariance of the prior state is:

$$m_k = \begin{pmatrix} m_{k-1,1} + m_{k-1,2}\Delta t \\ m_{k-1,2} - g\mathbb{E}[\sin(x_{k-1,1})]\Delta t \end{pmatrix}$$

2. The integral for mu is also not tractable analytically:

$$\mu = E[Y_k] = \int h(X_k)N(X_k; m_k, C_k)dX_k,$$

where $h(X_k) = \sin(x_{k,1})$. Substituting the measurement function into the integral and replacing the integral with the expectation, we get:

$$\mu = E[\sin(x_{k,1})],$$

In turn, the integrals for Ck, S, and U are not tractable, as they rely on the building blocks above, which we can't fully evaluate. We can see this by plugging in the dynamics, and simplifying as much as possible:

Ck: The difference $\phi(X_{k-1}) - m_k$ is:

$$\phi(X_{k-1}) - m_k = \begin{pmatrix} x_{k-1,1} - m_{k-1,1} \\ x_{k-1,2} - m_{k-1,2} - g(\sin(x_{k-1,1}) - \mathbb{E}[\sin(x_{k-1,1})])\Delta t \end{pmatrix}$$

The covariance C_k is calculated as:

$$C_k = \int (\phi(X_{k-1}) - m_k)(\phi(X_{k-1}) - m_k)^T N(X_{k-1}; m_{k-1}, C_{k-1})dX_{k-1} + \Sigma$$

$$\begin{aligned} C_k = \int \Bigg(\Bigg(& \begin{pmatrix} x_{k-1,1} - m_{k-1,1} \\ x_{k-1,2} - m_{k-1,2} - g(\sin(x_{k-1,1}) - \mathbb{E}[\sin(x_{k-1,1})])\Delta t \end{pmatrix} * \\ & \begin{pmatrix} x_{k-1,1} - m_{k-1,1} & x_{k-1,2} - m_{k-1,2} - g(\sin(x_{k-1,1}) - \mathbb{E}[\sin(x_{k-1,1})])\Delta t \end{pmatrix} \Bigg) * \\ & N(X_{k-1}; m_{k-1}, C_{k-1})\, dX_{k-1} + \Sigma \end{aligned}$$

- The first component involving $x_{k-1,1} - m_{k-1,1}$ and $x_{k-1,2} - m_{k-1,2}$ simplifies directly using the properties of Gaussian distributions:

$$\mathbb{E}[(x_{k-1,1} - m_{k-1,1})(x_{k-1,2} - m_{k-1,2})] = \mathrm{Cov}(x_{k-1,1}, x_{k-1,2})$$

- The terms involving $\sin(x_{k-1,1})$ remain nonlinear and cannot be simplified directly without additional approximations or numerical methods.

S:
Given the dynamics and measurement function:

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

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

$$\mu = E[\sin(x_{k,1})]$$

The covariance S is given by:

$$S = \int (\sin(x_{k,1}) - \mu)(\sin(x_{k,1}) - \mu)^T N(X_k; m_k, C_k)\, dX_k + R$$

All terms are nonlinear, so we can't simplify further.
U:

$$X_k - m_k = \begin{pmatrix} x_{k,1} + x_{k,2}\Delta t - (m_{k-1,1} + m_{k-1,2}\Delta t) \\ x_{k,2} - g\sin(x_{k,1})\Delta t - (m_{k-1,2} - g\mathbb{E}[\sin(x_{k-1,1})]\Delta t) \end{pmatrix},$$

$$h(X_k) - \mu = \sin(x_{k,1}) - \mathbb{E}[\sin(x_{k,1})].$$

The components $X_k - m_k$ and $h(X_k) - \mu$ involve linear and non-linear terms:

$$\begin{pmatrix} x_{k,1} + x_{k,2}\Delta t - (m_{k-1,1} + m_{k-1,2}\Delta t) \\ x_{k,2} - g(\sin(x_{k,1}) - \mathbb{E}[\sin(x_{k,1})])\Delta t \end{pmatrix}$$

with the non-linear sine function preventing straightforward expectation calculations.

$$U = \int (X_k - m_k)(\sin(x_{k,1}) - \mathbb{E}[\sin(x_{k,1})])^T N(X_k; m_k, C_k) dX_k$$

Since all terms are nonlinear, we can't evaluate this integral further.

The inability to analytically solve these integrals highlights the need for filtering methods.

3.1.2 Extended Kalman Filter

The Extended Kalman Filter (EKF) is a nonlinear state estimator that linearizes about the current mean and covariance. Unlike the standard Kalman Filter which applies directly to linear systems, the EKF handles nonlinearities through a first-order Taylor series expansion, enabling it to approximate the integrals involved in the update equations.

Taylor Series Approximation Given a nonlinear state transition or measurement function, the EKF linearizes it using the first-order Taylor expansion:

$$f(x) \approx f(\hat{x}) + F(x - \hat{x}) + \textit{higher order terms},$$

where F is the Jacobian matrix of partial derivatives evaluated at the estimate $\hat{x}$, providing the linear approximation needed for Gaussian propagation.

Jacobian Derivation For the pendulum model described in Section 2, the state transition and measurement models are:

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

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

State Transition Jacobian (F_k): For the pendulum dynamics, the Jacobian F_k is calculated by taking the partial derivatives of each component of f with respect to each state variable:

$$F_k = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{pmatrix},$$

where:

$$\frac{\partial f_1}{\partial x_1} = 1, \quad \frac{\partial f_1}{\partial x_2} = \Delta t,$$

$$\frac{\partial f_2}{\partial x_1} = -g\Delta t \cos(x_{k,1}), \quad \frac{\partial f_2}{\partial x_2} = 1.$$

Thus, F_k is:

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

Measurement Jacobian (H_k): The measurement function $h(x_k)$ is $y_k = \sin(x_{k,1})$. The Jacobian H_k involves the partial derivatives of h with respect to the state variables:

$$H_k = \begin{pmatrix} \frac{\partial h}{\partial x_1} & \frac{\partial h}{\partial x_2} \end{pmatrix},$$

where:

$$\frac{\partial h}{\partial x_1} = \cos(x_{k,1}), \quad \frac{\partial h}{\partial x_2} = 0.$$

Therefore, H_k is:

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

These Jacobian matrices are then used in the prediction and update equations of the EKF to accommodate the nonlinear nature of the pendulum dynamics and measurement.

EKF Prediction and Update Equations By plugging in the Jacobians F and H into the Gaussian filter equations, we can approximate the prediction and update steps of the EKF for a nonlinear system.

Let's denote the system dynamics as $\Phi(X_{k-1})$ and the measurement function as $h(X_k)$. Given the mean and covariance at time $k-1$, m_{k-1} and C_{k-1}, the EKF aims to predict these at time k, m_k and C_k.

For the prediction step, we start by approximating the expected value of the next state X_k given the current observations Y_{k-1} as:

$$m_k \approx \Phi(m_{k-1}) + \mathbb{E}[\Phi'(m_{k-1})(X_{k-1} - m_{k-1})] + \mathbb{E}[\xi],$$

where ξ is the process noise. Since $\mathbb{E}[\xi] = 0$ and $\Phi'(m_{k-1}) = F_k$, we have:

$$m_k \approx \Phi(m_{k-1}),$$

assuming that Φ can be replaced with the linear Jacobian F .

The state covariance matrix at the next time step is estimated by considering the propagation of uncertainty through the system dynamics. Assuming the process noise ξ with covariance Σ, and linearizing the state transition function Φ around the current state estimate m_{k-1}, we utilize the Jacobian F_k to approximate the covariance as:

$$C_k \approx \text{Cov}[\Phi(m_{k-1}) + F_k(X_{k-1} - m_{k-1}) + \xi, \Phi(m_{k-1}) + F_k(X_{k-1} - m_{k-1}) + \xi | Y_{k-1}] + \Sigma,$$

$$\approx F_k C_{k-1} F_k^\top + \Sigma,$$

which reflects the linear propagation of prior uncertainty and the addition of new uncertainty due to process noise.

For the update step, the Kalman filter refines the state estimate by incorporating new measurements. The measurement mean and covariance are predicted using the measurement function h and its linearization H_k:

$$\mu \approx h(m_k^-) + H_k(m_k^- - m_k^-),$$

$$S \approx H_k C_k H_k^\top + R,$$

where m_k^- represents the prior state estimate, H_k is the measurement Jacobian, and R is the measurement noise covariance. Using these approximations, we compute the Kalman gain

$$K_k \approx C_k H_k^\top S^{-1},$$

and we can use this gain to update the state estimate and covariance in a way that balances between the prediction and the new measurement:

$$m_{k|k} \approx m_{k|k-1} + K_k(y_k - h(m_{k|k-1})),$$

$$C_{k|k} \approx (I - K_k H_k) C_{k|k-1}.$$

Here, I is the identity matrix, and y_k represents the actual measurement at time k.

Plugging in our known dynamics, we get our expressions for the pendulum:

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:

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

The update equations are similarly explicit:

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

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

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

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

3.2 Unscented Kalman Filter

3.2.1 Sigma Points

The unscented transform (UT) is a numerical method used to approximate the joint distribution of random variables. Unlike linearization methods, the UT directly estimates the mean and covariance of the target distribution without approximating the non-linear function itself.

The idea behind the UT is to deterministically select a fixed number of sigma points that accurately represent the mean and covariance of the original distribution. These sigma points are then propagated through the non-linearity, and the mean and covariance of the transformed variable are estimated from them. It's important to note that while the UT resembles Monte Carlo estimation, it differs significantly because the sigma points are deterministically chosen. A visualization of how these Sigma points propogate through a nonlinear system is given in Figure 1.

Quadrature is an approach used to approximate integrals directly via numerical integration rules. In the context of Gaussian filtering equations, quadrature methods offer several advantages over other approximation techniques. While they do not require the evaluation of gradients, they often provide more accurate approximations of the filtering equations, which themselves are approximations of the underlying dynamics. However, it's worth noting that quadrature methods can sometimes suffer from stability issues.

Quadrature works by approximating an integral with a weighted sum:

$$\int h(X)\, dX \approx \sum_i h(X^{(i)}) w^{(i)}$$

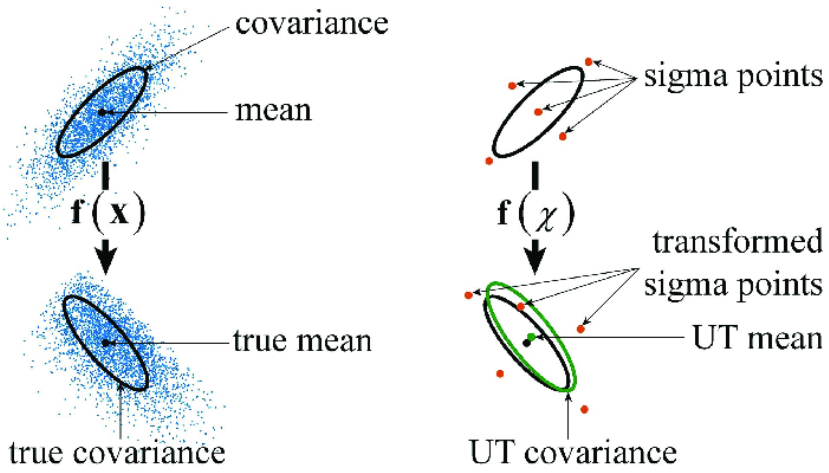

Figure 1: Illustration of sigma points being transformed. On the left is the raw data, and on the right is an illustration of Sigma points generated by the UT being passed through the same system to estimate the resulting mean/covariance.

Here, $\{(w^{(i)}, X^{(i)})\}$ denotes the set of quadrature weight points. One commonly used quadrature rule is the trapezoid rule, which essentially builds a piecewise linear approximation of the function of interest and then exactly integrates this piecewise linear approximation. The Unscented Transform forms Sigma Points as follows for quadrature approximation: 1. **Form Sigma Points**: Before proceeding with the Unscented Transform, it's important to define a scaling parameter λ that dictates how far the quadrature points are spread:

$$\lambda = \alpha^2(d + \kappa) - d$$

This parameter is used to formally define the quadrature points as:

$$u^{(i)} = \mu + \sqrt{(d+\lambda)P}_i, \quad \text{for } i = 1, \ldots, 2d$$

2. **Compute Weights**: The UKF calculates individual weights for the mean $w_m^{(i)}$ and covariance $w_c^{(i)}$:

$$w_m^{(i)} = \begin{cases} \frac{\lambda}{d+\lambda}, & \text{for } i = 0 \\ \frac{1}{2(d+\lambda)}, & \text{otherwise} \end{cases}$$

$$w_c^{(i)} = \begin{cases} \frac{\lambda}{d+\lambda} + (1 - \alpha^2 + \beta), & \text{for } i = 0 \\ \frac{1}{2(d+\lambda)}, & \text{otherwise} \end{cases}$$

Recall, the UT is a third-order method, meaning it provides an exact estimate of the mean for polynomials up to third order. However, the covariance approximation is exact only for first-order polynomials due to limitations in computing higher-order terms. The full UKF algorithm, using the Sigma points as described here, is shown in Algorithm 1.

Algorithm 1 Unscented Kalman Filter (UKF)

Prediction:

Form the sigma points:

$$\begin{aligned} X_{k-1}^{(0)} &= m_{k-1} \\ X_{k-1}^{(i)} &= m_{k-1} + \sqrt{(n+\lambda)P_{k-1}}_i, \quad i = 1, \dots, n \\ X_{k-1}^{(i+n)} &= m_{k-1} - \sqrt{(n+\lambda)P_{k-1}}_i, \quad i = 1, \dots, n \end{aligned}$$

where the parameter λ is defined in Equation (5.75).
Propagate the sigma points through the dynamic model:

$$X_k^{(i)} = f(X_{k-1}^{(i)}), \quad i = 0, \dots, 2n$$

Compute the predicted mean m_k and the predicted covariance P_k:

$$\begin{aligned} m_k &= \sum_{i=0}^{2n} W_i^{(m)} X_k^{(i)} \\ P_k &= \sum_{i=0}^{2n} W_i^{(c)} (X_k^{(i)} - m_k)(X_k^{(i)} - m_k)^T + Q_{k-1} \end{aligned}$$

where the weights $W_i^{(m)}$ and $W_i^{(c)}$ were defined in Equation (5.77).

Update:

Form the sigma points:

$$\begin{aligned} X_k^{(0)} &= m_k \\ X_k^{(i)} &= m_k + \sqrt{(n+\lambda)P_k}_i, \quad i = 1, \dots, n \\ X_k^{(i+n)} &= m_k - \sqrt{(n+\lambda)P_k}_i, \quad i = 1, \dots, n \end{aligned}$$

Propagate sigma points through the measurement model:

$$Y_k^{(i)} = h(X_k^{(i)}), \quad i = 0, \dots, 2n$$

Compute the predicted mean $\bar{y}_k$, the predicted covariance of the measurement S_k, and the cross-covariance of the state and the measurement C_k:

$$\begin{aligned} \bar{y}_k &= \sum_{i=0}^{2n} W_i^{(m)} Y_k^{(i)} \\ S_k &= \sum_{i=0}^{2n} W_i^{(c)} (Y_k^{(i)} - \mu_k)(Y_k^{(i)} - \limsup_k)^T + R_k \\ C_k &= \sum_{i=0}^{2n} W_i^{(c)} (X_k^{(i)} - m_k)(Y_k^{(i)} - \mu_k)^T \end{aligned}$$

Compute the filter gain K_k, the filtered state mean m_k and the covariance P_k, conditional on the measurement y_k:

$$\begin{aligned} K_k &= C S_k^{-1} \\ m_k &= m_k + K_k(y_k - \mu_k) \\ P_k &= P_k - K_k S_k K_k^T \end{aligned}$$

3.2.2 Gauss-Hermite Kalman Filter

The Gauss-Hermite Kalman Filter (GHKF) also uses quadrature for the numerical evaluation of integrals that arise in nonlinear filtering. The core advantage of this approach lies in its precision and stability, albeit at the cost of increased computational demand, particularly in high-dimensional systems.

For the GHKF, quadrature points and corresponding weights are determined through Hermite polynomials, ensuring that the integration of polynomial approximations up to a specific order is exact. The nth-order Hermite polynomial, $H_n(x)$, allows the GHKF to precisely integrate polynomials up to the order $2n-1$:

$$\int h(u)p(u)\,du \approx \sum_{i=0}^{n} a_i \int H_i(u)p(u)\,du = a_0 \tag{1}$$

The Hermite polynomials themselves are derived recursively as follows:

$$H_{k+1}(x) = xH_k(x) - kH_{k-1}(x) \tag{2}$$
$$H_0(x) = 1 \tag{3}$$
$$H_1(x) = x \tag{4}$$
$$H_2(x) = x^2 - 1 \tag{5}$$
$$H_3(x) = x^3 - 3x \tag{6}$$

Orthogonality of Hermite polynomials with respect to the standard normal probability distribution is a key property utilized in GHKF, which ensures that:

$$\int H_i(u)H_j(u)N(u|0,I)\,du = i!\delta_{ij} \tag{7}$$

For multidimensional state estimation problems, the GHKF approximates the integration over a multi-dimensional Gaussian distribution. This is achieved by extending the univariate Gauss-Hermite quadrature to higher dimensions,

where the integration is performed independently in each dimension:

$$\int h(X)N(X|\mu,\Sigma)\,dX = \int h(\mu+\sqrt{\Sigma}u)N(u|0,I)\,du \tag{8}$$

$$= \int\cdots\int h(\mu+\sqrt{\Sigma}u)N(u_1|0,1)\,du_1\times\cdots\times N(u_d|0,1)\,du_d \tag{9}$$

$$\approx \sum_{i_1,\ldots,i_d} w_{i_1}\ldots w_{i_d} h(\mu+\sqrt{\Sigma}u^{(i)}) \tag{10}$$

Here, $u^{(i)}$ represents the set of quadrature points and w_i are the corresponding weights.

Cholesky vs. SVD As a note, the GHKF requires transforming sigma points to match the distribution characterized by the current mean and covariance of the state estimates. This transformation involves the decomposition of the covariance matrix to obtain a matrix L that, when multiplied with the standard Gaussian points, aligns them according to the covariance structure of the distribution being approximated. Two approaches for this are:

- **Cholesky Decomposition**: Cholesky is a suitable choice when the covariance matrix is symmetric and positive definite, as it is computationally efficient and faster than SVD. It decomposes a matrix into a lower triangular matrix L and its transpose.
- **Singular Value Decomposition (SVD)**: SVD is more robust, capable of handling cases where the covariance matrix might be close to singular or not strictly positive definite. It decomposes the covariance into U, S, and V^T, where S contains the singular values that are then square-rooted and multiplied by the orthogonal matrices U and V to form L.

In this implementation, Cholesky decomposition was chosen primarily for its efficiency in computational speed. The covariance matrices in this context are ensured to be positive definite through the design of the filter and the nature of the physical models used, which typically avoid conditions leading to non-positive definite matrices.

Filter Equations Using the Quadrature points defined in the preceeding section, we get our modified Gaussian Filtering Algorihtm for GHKF, presented in Algorithm 2.

3.3 Particle Filtering

Particle filters are strong candidates in environments where Gaussian approximations fall short, such as in highly non-linear systems, systems with multimodal distributions, or systems with discrete state components. Based on sequential importance sampling, particle filters provide a versatile approach for approximating Bayesian filtering solutions and handling complex models.

Operational Framework: Particle filters maintain a set of samples, each representing a potential state of the system. These particles are propagated through the state space according to the system dynamics and updated based on new measurements, thus tracking the state evolution over time.

Empirical Distribution: To estimate the system's state, particle filters use an empirical distribution, defined as:

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

where x_i^n are the state samples, $\bar{w}_i$ are the normalized weights, and $\delta_{x_i^n}(X_n)$ is the Dirac delta function centered at each particle. This representation captures the posterior probability distribution as a weighted sum of delta functions at particle locations, effectively quantifying the likelihood of different states. Note that there is zero probability outside the particles.

Sequential Importance Sampling: The update of particle weights is crucial for reflecting the latest observations and is performed using:

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

Here, $p(y_n|x_i^n)$ is the likelihood of observing the new data given state x_i^n, $p(x_i^n|x_i^{n-1})$ details the state transition probabilities, and $\pi(x_i^n|x_i^{n-1})$ is the proposal distribution used to generate the new state samples.

Marginal Likelihood and Weight Normalization: To successfully integrate the updates from both the system dynamics and the new measure-

Algorithm 2 Gauss-Hermite Kalman Filter (GHKF)

Prediction:

1: Form the sigma points as:

$$X_{k|k-1}^{(i_1,\ldots,i_n)} = m_{k-1} + \sqrt{P_{k-1}}\chi^{(i_1,\ldots,i_n)}, \quad i_1,\ldots,i_n = 1,\ldots,p$$

2: Propagate the sigma points through the dynamic model:

$$\hat{X}_{k|k-1}^{(i_1,\ldots,i_n)} = f\left(X_{k|k-1}^{(i_1,\ldots,i_n)}\right), \quad i_1,\ldots,i_n = 1,\ldots,p$$

3: Compute the predicted mean m_k and the predicted covariance P_k:

$$m_k = \sum_{i_1,\ldots,i_n} W_{i_1,\ldots,i_n}\hat{X}_{k|k-1}^{(i_1,\ldots,i_n)}$$

$$P_k = \sum_{i_1,\ldots,i_n} W_{i_1,\ldots,i_n}(\hat{X}_{k|k-1}^{(i_1,\ldots,i_n)} - m_k)(\hat{X}_{k|k-1}^{(i_1,\ldots,i_n)} - m_k)^T + Q_{k-1}$$

Update:

4: Form the sigma points:

$$X_k^{(i_1,\ldots,i_n)} = m_k + \sqrt{P_k}\chi^{(i_1,\ldots,i_n)}, \quad i_1,\ldots,i_n = 1,\ldots,p$$

5: Propagate sigma points through the measurement model:

$$\hat{Y}_k^{(i_1,\ldots,i_n)} = h(X_k^{(i_1,\ldots,i_n)}), \quad i_1,\ldots,i_n = 1,\ldots,p$$

6: Compute the predicted mean v_k, the predicted covariance of the measurement S_k, and the cross-covariance of the state and the measurement C_k:

$$v_k = \sum_{i_1,\ldots,i_n} W_{i_1,\ldots,i_n}\hat{Y}_k^{(i_1,\ldots,i_n)}$$

$$S_k = \sum_{i_1,\ldots,i_n} W_{i_1,\ldots,i_n}(\hat{Y}_k^{(i_1,\ldots,i_n)} - v_k)(\hat{Y}_k^{(i_1,\ldots,i_n)} - v_k)^T + R_k$$

$$C_k = \sum_{i_1,\ldots,i_n} W_{i_1,\ldots,i_n}(X_k^{(i_1,\ldots,i_n)} - m_k)(\hat{Y}_k^{(i_1,\ldots,i_n)} - v_k)^T$$

7: Compute the filter gain K_k, the filtered state mean m_k, and the covariance P_k, conditional on the measurement y_k:

$$K_k \underset{18}{=} C_k S_k^{-1}$$

$$m_k = m_k + K_k(y_k - v_k)$$

$$P_k = P_k - K_k S_k K_k^T$$

ments, the marginal likelihood is approximated by:

$$\hat{P}(y_n|Y_{n-1}) \approx \frac{1}{N}\sum_{i=1}^{N} p(y_n|x_i^n)p(x_i^n|x_i^{n-1})\frac{\bar{w}_{n-1}^{(i)}}{\pi(x_i^n|x_i^{n-1})}$$

This approximation facilitates the normalization of the weights, ensuring they collectively sum to one. This normalization process is crucial as it recalibrates each particle's weight according to its compatibility with the new measurement, as well as its fidelity to the model dynamics from the previous state transition.

Following weight normalization, the effective sample size (N_{eff}) is calculated to evaluate the diversity of the particle set and determine the necessity for resampling. N_{eff} is given by:

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

This metric assesses how concentrated or dispersed the weights are among the particles. A lower N_{eff} suggests that a few particles hold most of the weight, indicating potential degeneracy in the particle set.

Resampling: To counteract the degeneracy problem, where a few particles may come to dominate the weight distribution, resampling is periodically performed. This process generates a new set of particles from the current empirical distribution according to their weights, thus redistributing the probability mass across a refreshed sample set. For all experiments, my resampling threshold was set at N0 = 0.1. In other words, resampling would occur if the effective sample size dropped below 10% of its initial value.

Algorithmic Steps: The particle filtering algorithm, as outlined in Algorithm 3, encapsulates propagation, weighting, and resampling to robustly estimate state distributions.

Handling Intermittent Measurements in Particle Filtering In our scenario, measurements are not available at every timestep, posing a challenge for maintaining accurate state estimates.

When measurements are missing, particles are propagated solely based on the system's dynamics without the usual measurement update. This process involves extending the state prediction over multiple timesteps until a new measurement is received. The state of each particle at timestep k when no measurement is available is updated as:

$$X_k^{(i)} \sim \pi(X_k|X_{k-1}^{(i)})$$

where $\pi(X_k|X_{k-1})$ is the proposal distribution based on the system dynamics, as previously defined. This assumes that the dynamics provide a reasonable approximation of the state evolution in the absence of measurement data.

During these measurement-free steps, the weights of the particles from the previous timestep are carried forward without change:

$$w_k^{(i)} = w_{k-1}^{(i)}$$

Updating the State with Intermittent Measurements Once a new measurement is received, say at timestep m, where $m > k$ (i.e., after one or more timesteps without measurements), the usual update procedure is resumed. The particles are then updated using the newly received measurement:

$$w_m^{(i)} \propto w_{m-1}^{(i)} \times p(y_m|X_m^{(i)})$$

Algorithm 3 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

3.3.1 Dynamics as Proposal

The most straightforward application of the PF to our pendulum task involves using the dynamics to inform the proposal. In this section, we define the proposal distribution, transition probabilities, and the likelihood model for this simplest approach.

Proposal Distribution For the proposal distribution, the dynamics of the pendulum itself are utilized. This method, known as the "Bootstrap Particle Filter," assumes that the system's natural evolution provides a sufficient proposal mechanism. The state of the pendulum at the next time step, X_{k+1}, is proposed based on the current state X_k modified by the dynamics of the pendulum:

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

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, introducing stochasticity into 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}, follows the dynamics with an added noise component to account for 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, encapsulating the expected deviation of the next state due to inherent uncertainties in the dynamics.

Likelihood Model The likelihood model quantifies how likely a measured data point y_k is, given a state X_k. Assuming the 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 $h(X_k)$ is the observation model mapping the state space to the measurement space, and σ_{noise} is the measurement noise standard deviation. This Gaussian likelihood reflects the probability of observing the data given the state, under the assumption that measurement errors are normally distributed.

3.3.2 UKF as Optimal Proposal

Challenges in Sequential Importance Sampling (SIS) The standard approach in Sequential Importance Sampling (SIS) primarily uses the system's dynamics as the proposal distribution:

$$q(x_t|x_{t-1}, y_t) = p(x_t|x_{t-1})$$

This setup simplifies the importance weights calculation to:

$$v(i)_k = p(y_k|x(i)_k)$$

where $v(i)_k$ represents the likelihood of the new measurement y_k given the state $x(i)_k$, derived solely from previous state dynamics. However, this method does not integrate new measurement data into the proposal distribution, and can result undesirable increase in the variance of the importance weights as some of the particles become heavily weighted. Figure 2 demonstrates this visually, where problems arise when likelihood is concentrated in the tails of the distribution.

Optimal Proposal Distribution The proposal distribution should ideally incorporate new measurement data to effectively minimize the variance of the importance weights. Although it is often infeasible to calculate the exact optimal proposal distribution directly, it can be approximated using advanced Gaussian filtering techniques such as the Unscented Kalman Filter (UKF).

3.3.3 Integration of UKF into Particle Filtering

The integration of the UKF into particle filtering enhances the proposal distribution mechanism while retaining the established transition probabilities and likelihood model used in the standard particle filtering process. The new Unscented PF algorithm is given in Algorithm 2. Recall that the details of the UKF equations are presented in section 3.2.

Algorithm 4 UKF Integration into Particle Filtering

Initialization: $t = 0$
for $i = 1, \ldots, N$ **do**
    Draw the states (particles) $x_0^{(i)}$ from the prior $p(x_0)$
    Set $\hat{x}_0^{(i)} = E[x_0^{(i)}]$
    Set $P_0^{(i)} = E[(x_0^{(i)} - \hat{x}_0^{(i)})(x_0^{(i)} - \hat{x}_0^{(i)})^T]$
    Define augmented states $\hat{x}(i)_a^0$ and covariance $P(i)_a^0$ to include process Q and measurement noise R
end for
Iterative Update: For $t = 1, 2, \ldots$
Importance Sampling Step:
for $i = 1, \ldots, N$ **do**
    Update the particles with the UKF:
    Calculate sigma points $X(i)_a^{t-1}$ from $\hat{x}(i)_a^{t-1}$ and $P(i)_a^{t-1}$
    Propagate sigma points through the system model to estimate $X(i)_x^{t|t-1}$
    Update state estimate $\hat{x}(i)_t^{t|t-1}$ and covariance $P(i)_t^{t|t-1}$ based on sigma points
    Incorporate new observation into the state estimate using the Kalman gain:

$$K_t = P_{xt|yt} \left(P_{\tilde{y}t|\tilde{y}t}\right)^{-1}$$

$$\hat{x}(i)_t = \hat{x}(i)_t^{t|t-1} + K_t \left(y_t - \hat{y}(i)_t^{t|t-1}\right)$$

    Update covariance:

$$P(i)_t = P(i)_t^{t|t-1} - K_t P_{\tilde{y}t|\tilde{y}t} K_t^T$$

    Sample $x(i)_t$ from $q(x(i)_t|x(i)_{0:t-1}, y_{1:t})$ which is $\mathcal{N}(\hat{x}(i)_t, P(i)_t)$
    Evaluate the importance weights:

$$w(i)_t \propto \frac{p(y_t|x(i)_t)p(x(i)_t|x(i)_{t-1})}{q(x(i)_t|x(i)_{0:t-1}, y_{1:t})}$$

end for
Normalize the importance weights
Selection Step:
Resample particles based on their normalized importance weights

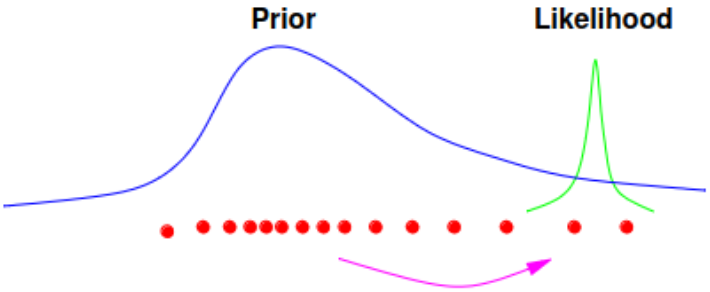

Figure 2: Poor proposal not accounting for new data

4 Results and Discussion

4.1 Data Simulation

In the simulation of the pendulum dynamics, first-order integration is employed to generate data, following the model described in Section 2.

Simulation Parameters The initial conditions set the pendulum's position (θ) at 1.5 radians and velocity ($\dot{\theta}$) at 0 radians per second for all experiments. The simulation is run with timestep ($\Delta t = 0.01$). The measurement noise and time between observations are the changed variables between simulation runs.

Observations of the pendulum's position are made noisy with measurement noise and recorded at intervals, providing realistic data sets for testing filtering algorithms. Figure 3 presents an example of the simulated data, including both the true trajectory and the noisy observed positions of the pendulum, which illustrate the challenges in estimating the state from noisy measurements.

4.2 Gaussian Filters - State Estimation Performance

These analysis sections evaluate the state estimation performance of four Gaussian filters: EKF, UKF, GHKF order 3, and GHKF order 5. The filters are tested across all combinations of measurement timesteps ($\delta \in \{5, 10, 20, 40\}$) and measurement noise variances ($R \in \{1, 0.1, 0.01, 0.001\}$). The tracking performance plots and Mean Squared Errors (MSEs) of these

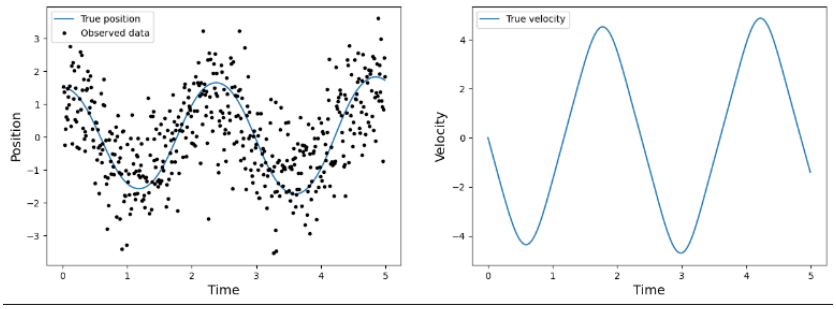

Figure 3: Example of simulated pendulum data, showing true and observed positions with noise.

filters are presented in the subsequent sections, followed by a discussion of the results.

4.2.1 Extended Kalman Filter

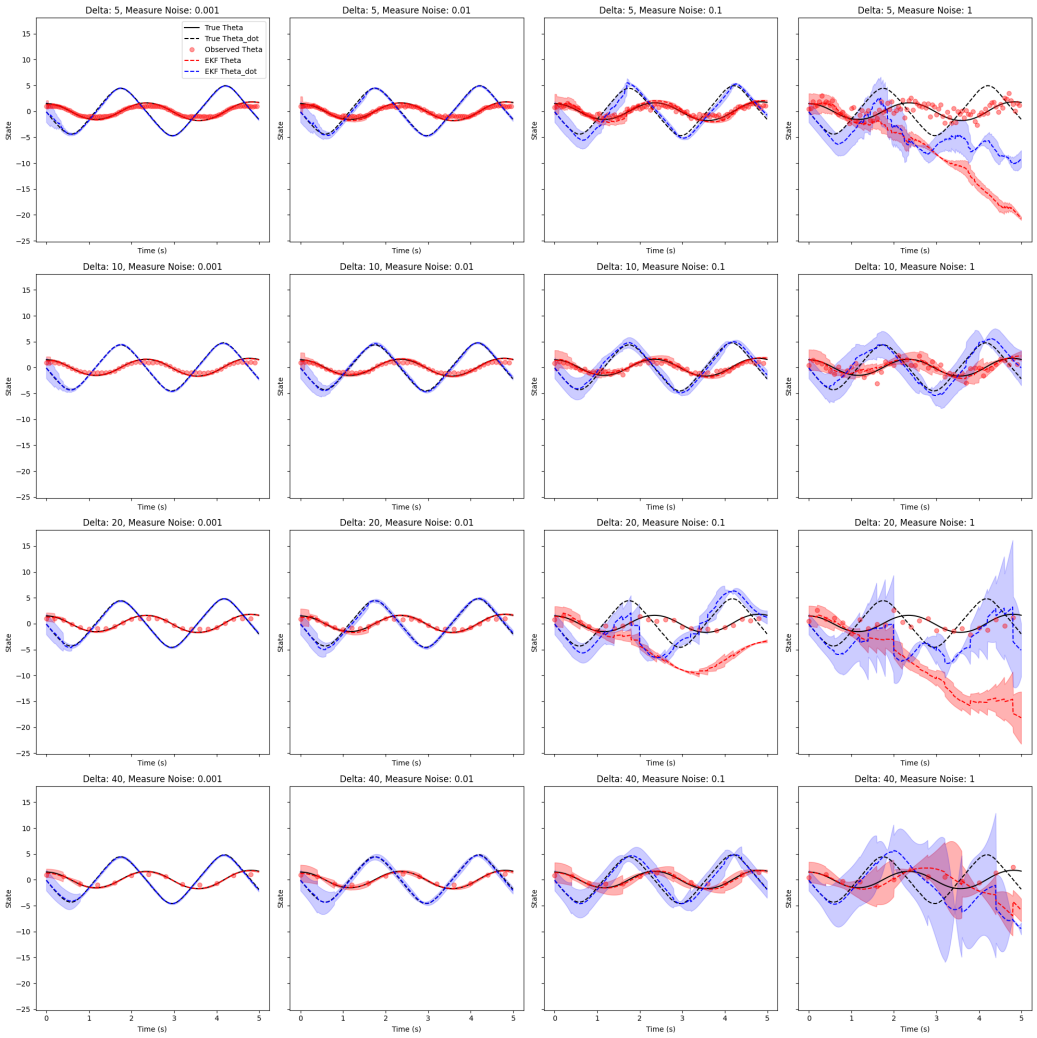

Figure 4: Extended Kalman Filter performance across various measurement timesteps (δ) and noise variances (R).

4.2.2 Unscented Kalman Filter

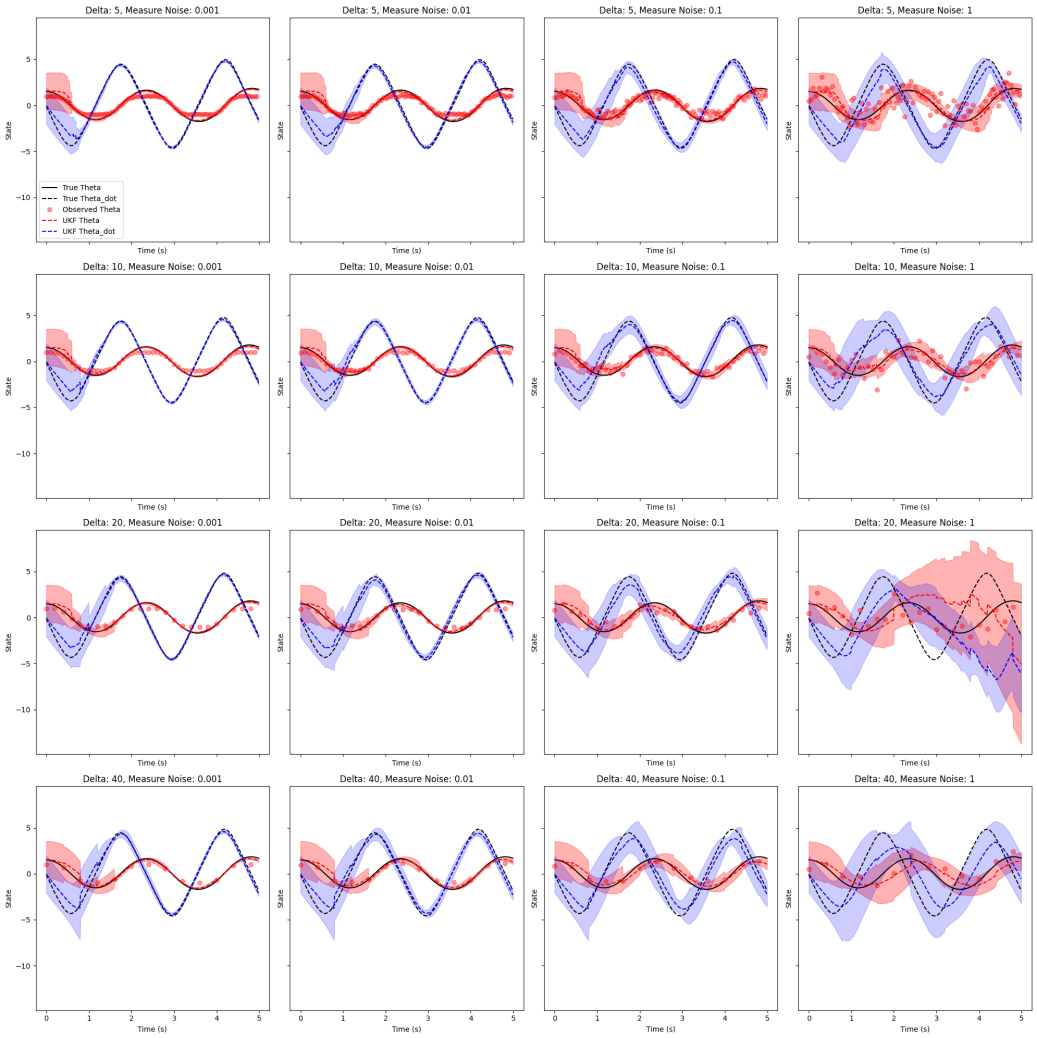

Figure 5: Unscented Kalman Filter performance with varying measurement timesteps (δ) and noise variances (R).

4.2.3 Gauss-Hermite Kalman Filter order 3

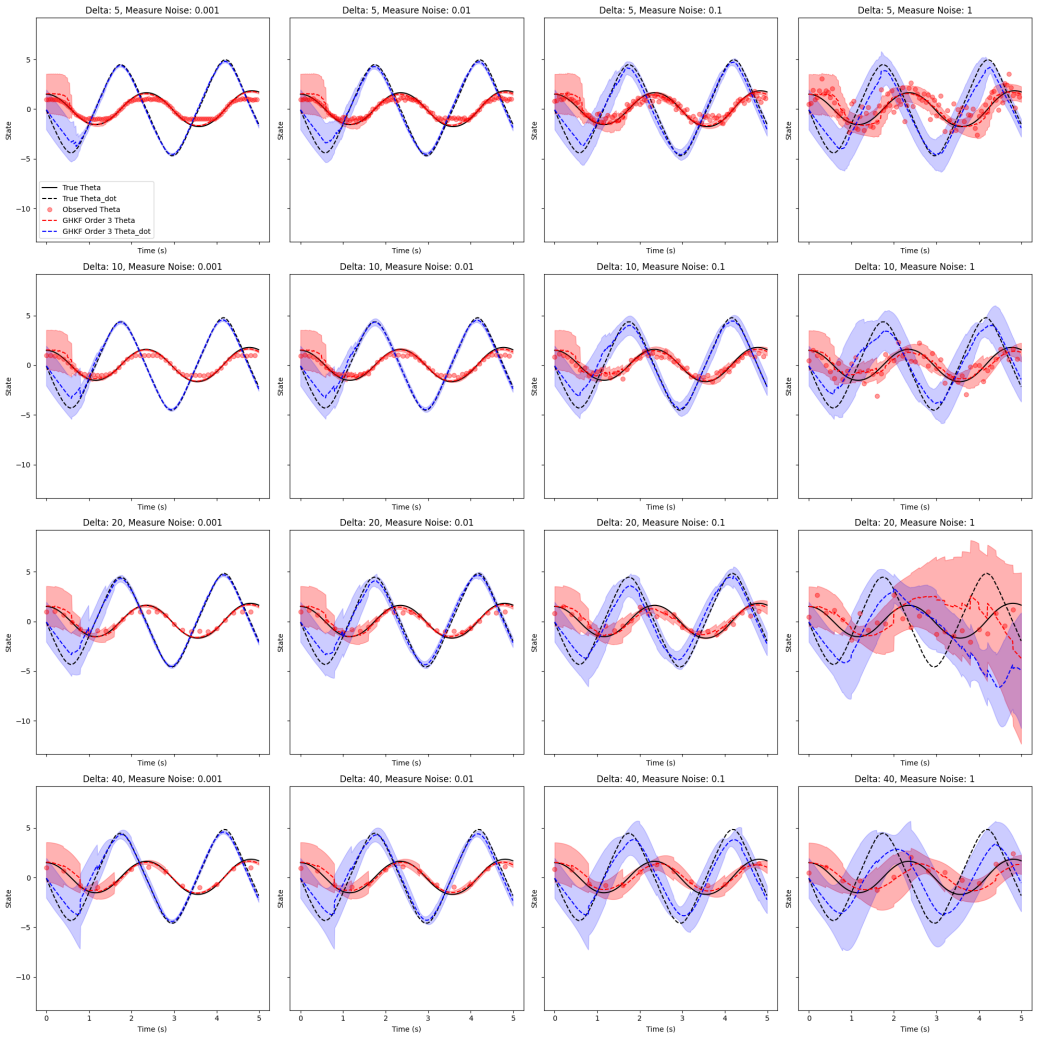

Figure 6: Performance of Gauss-Hermite Kalman Filter of order 3 under different settings of measurement timesteps (δ) and noise variances (R).

4.2.4 Gauss-Hermite Kalman Filter order 5

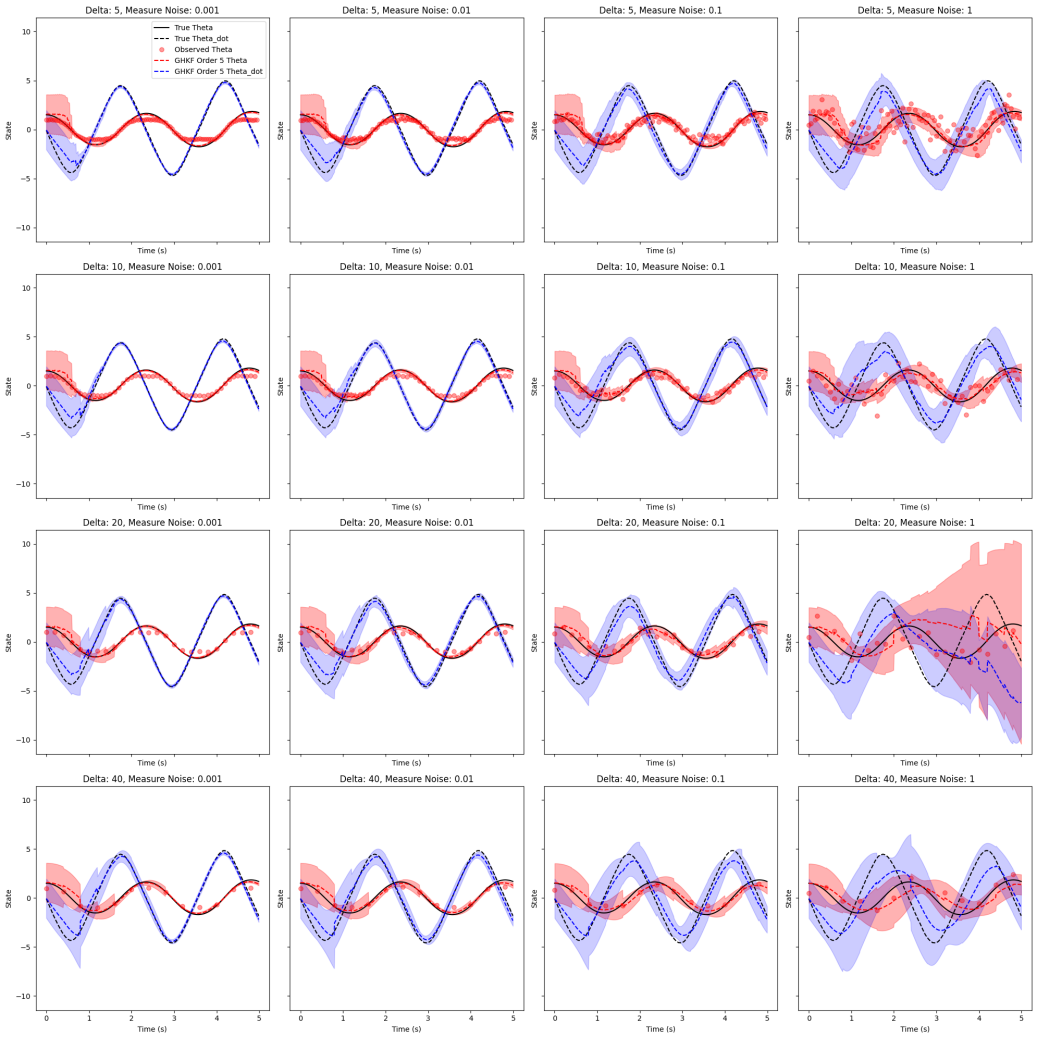

Figure 7: Performance of Gauss-Hermite Kalman Filter of order 5 under different settings of measurement timesteps (δ) and noise variances (R).

4.2.5 Tabulated Results

With these runs, I recorded the MSEs and timing information as follows:

Table 1: Comparison of Gaussian Filters Performance with MSE

δ	R	EKF	UKF	GHKF Order 3	GHKF Order 5
5	0.001	0.04626	0.17117	0.17117	0.18349
5	0.01	0.11349	0.20889	0.20893	0.22032
5	0.1	0.40632	0.28199	0.28235	0.29320
5	1	9.98299	0.41001	0.41228	0.41265
10	0.001	0.03987	0.15979	0.15978	0.17520
10	0.01	0.07896	0.17999	0.17984	0.18927
10	0.1	0.20340	0.24586	0.24560	0.24758
10	1	0.37093	0.37698	0.37840	0.38482
20	0.001	0.05021	0.16961	0.16988	0.18968
20	0.01	0.12038	0.21437	0.21532	0.22743
20	0.1	5.62897	0.34801	0.34814	0.35099
20	1	10.20474	2.18248	2.01503	1.60800
40	0.001	0.06139	0.23274	0.23329	0.24703
40	0.01	0.08071	0.26136	0.26220	0.28002
40	0.1	0.18195	0.39258	0.39422	0.40689
40	1	2.72696	0.72206	0.74392	0.69032

Table 2: Filter Run Times for Various Settings of Delta and Measurement Noise in Seconds

δ	R	EKF	UKF	GHKF Order 3	GHKF Order 5
5	0.001	0.0102	0.0387	0.0366	0.0549
5	0.01	0.0137	0.0240	0.0438	0.0499
5	0.1	0.0088	0.0248	0.0303	0.0518
5	1	0.0094	0.0413	0.0327	0.0579
10	0.001	0.0122	0.0242	0.0296	0.0465
10	0.01	0.0077	0.0270	0.0265	0.0502
10	0.1	0.0101	0.0290	0.0338	0.0599
10	1	0.0093	0.0260	0.0327	0.0540
20	0.001	0.0083	0.0424	0.0321	0.0678
20	0.01	0.0079	0.0230	0.0363	0.0602
20	0.1	0.0074	0.0230	0.0264	0.0437
20	1	0.0072	0.0214	0.0254	0.0425
40	0.001	0.0067	0.0248	0.0244	0.0423
40	0.01	0.0067	0.0220	0.0255	0.0457
40	0.1	0.0069	0.0211	0.0259	0.0447
40	1	0.0069	0.0203	0.0244	0.0412

Table 3: Mean Run Times for Filters Across All Configurations

Filter	Mean Run Time (s)
EKF	0.0088
UKF	0.0270
GHKF Order 3	0.0303
GHKF Order 5	0.0505

Finally, I ran the sweep over all conditions 5 times, and manually counted how many times each filter diverged as a measure of robustness. I counted divergences as instances where the covariance grew extremely large (see GHKF with delta 20, noise 1 in the reported plots) or cases where the true state was lost completely (see EKF delta 20 noise 0.1 for example). My results are tabulated below:

Table 4: Proportion of Divergences for Each Filter Across All Configurations

Filter	Proportion of Divergences
EKF	0.0859
UKF	0.0273
GHKF Order 3	0.0234
GHKF Order 5	0.0234

4.2.6 Discussion

The implementation of the EKF, UKF, and GHKF of orders 3 and 5 was conducted following the detailed equations provided in the Filtering Techniques section.

Robustness: The robustness of the filters was evaluated based on their susceptibility to divergence under various conditions. The EKF showed a relatively higher divergence rate, indicated by a proportion of 0.0859. This suggests that the EKF might struggle with the non-linearities inherent in the pendulum system more than the other filters. Both versions of the GHKF, along with the UKF, demonstrated greater robustness with divergence proportions significantly lower at approximately 0.0234 and 0.0273 respectively. This makes sense in theory, as the UKF(and both GHKFs) approximate the nonlinear system up to 3rd order for non-Gaussian inputs, which is a higher fidelity approximation than the first-order EKF, and can more reliably dy-

namics. The lack of difference between order 3 and order 5 GHKF suggests that the order 3 approximation is sufficient for the pendulum.

Computational Complexity: Theoretically, the EKF is least computationally demanding due to its reliance on first-order linear approximations. By linearizing the system dynamics around the current estimate, the EKF simplifies the calculation process significantly, though at the potential cost of accuracy in highly non-linear systems.

On the other hand, the UKF and GHKF involve more complex computations due to their use of a sigma point approach. For a state vector of dimension n, the UKF typically utilizes $2n + 1$ sigma points. These sigma points must each be calculated and propagated through the nonlinear dynamics.

The GHKF, in contrast, has evaluation points that grow exponentially with the state dimension, particularly noticeable when higher orders of Hermite polynomials are employed. This results in a significant increase in computational overhead, especially for GHKF Order 5, making it the most computationally intensive among the filters discussed.

All of these theoretical details align with the collected runtimes EKF $<$ UKF $<$ GHKF.

Accuracy in Terms of MSE: The MSE results indicate that the UKF and GHKF generally outperform the EKF, especially in scenarios with high measurement noise and dynamic complexity, as denoted by the δ and R parameters. The superior performance of the UKF and GHKF is attributed to their effective management of nonlinear system dynamics through advanced state distribution modeling. While the EKF relies on linear approximations around a point estimate, which can introduce significant errors in nonlinear environments, the UKF and GHKF employ sigma point and quadrature methods, respectively. These methods accurately propagate the true dynamics of the system, maintaining the nonlinear characteristics integral to the model and enhancing the precision and robustness of the filter's performance. The comparable results between the UKF and both orders of the GHKF suggest that each filter captures the critical dynamics of the system effectively.

Overall, the choice of filtering algorithm significantly impacts the system's ability to accurately predict states under varying levels of uncertainty and dynamics complexity. While the EKF offers computational simplicity and speed, it sacrifices robustness and accuracy in highly non-linear systems like the pendulum model used in these experiments. The UKF and GHKF,

with their more sophisticated approaches to non-linearity, provide a more robust and accurate alternative, albeit at the cost of increased computational overhead.

4.3 Particle Filters

I implemented two versions of the particle filter. The Particle filter with dynamics proposal, and particle filter with UKF proposal, as described in the Filtering techniques section. These experiments all use the same initial conditions and timestep as the previous section.

4.3.1 State Estimation-PF with Dynamics Proposal

This section presents the tracking performance of the PF with a dynamics proposal under different observational settings. Each plot illustrates how the PF handles varying degrees of measurement noise and time intervals between observations. All simulations in this section were run with 1e4 particles.

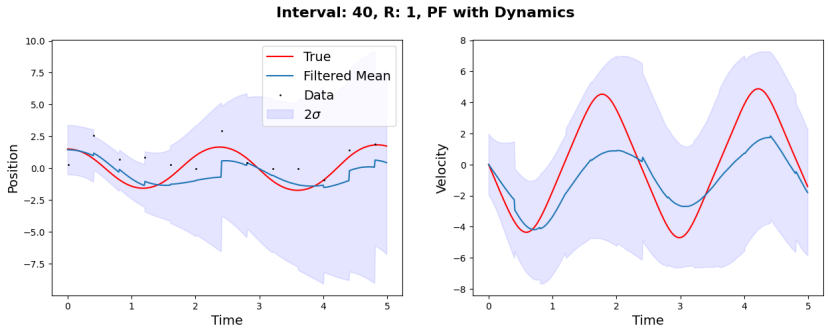

Figure 8: Tracking performance of the Particle Filter with Dynamics Proposal for $\Delta = 40$ and $R = 1$. This setting reflects longer observation intervals with high measurement noise, challenging the filter's tracking accuracy.

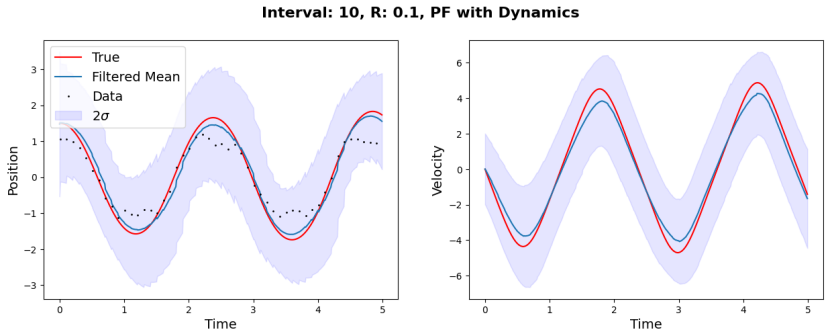

Figure 9: Tracking performance of the Particle Filter with Dynamics Proposal for $\Delta = 10$ and $R = 0.1$. This configuration tests the filter's responsiveness to more frequent observations with moderate noise.

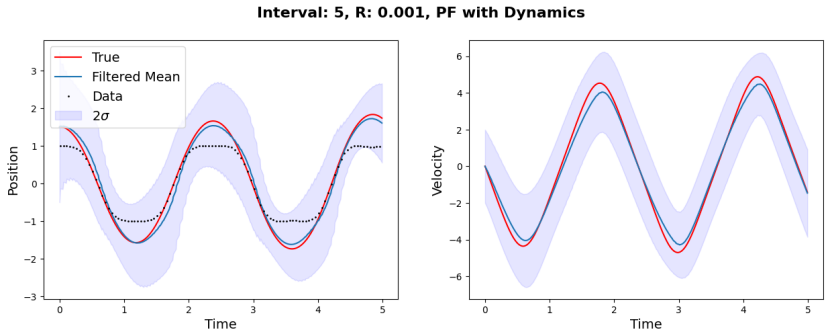

Figure 10: Tracking performance of the Particle Filter with Dynamics Proposal for $\Delta = 5$ and $R = 0.001$. This scenario shows the filter operation under nearly continuous observation with minimal noise, ideal for observing the filter's precision.

4.3.2 State Estimation-PF with UKF Proposal

This section presents the tracking performance of the PF with a UKF proposal under different observational settings. Each plot illustrates how the PF, using the UKF proposal method, handles varying degrees of measurement noise and time intervals between observations. All simulations in this section were run with 1e4 particles.

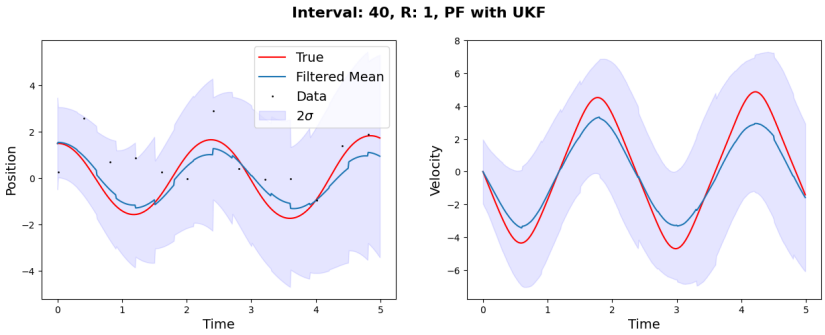

Figure 11: Tracking performance of the Particle Filter with UKF Proposal for $\Delta = 40$ and $R = 1$. This setting reflects longer observation intervals with high measurement noise, demonstrating how the UKF proposal addresses the non-linear dynamics in challenging conditions.

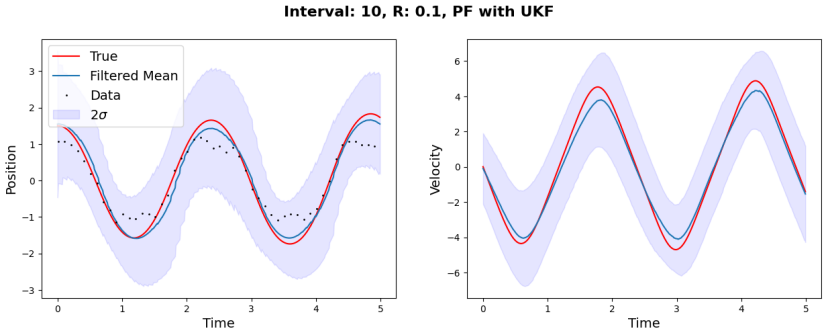

Figure 12: Tracking performance of the Particle Filter with UKF Proposal for $\Delta = 10$ and $R = 0.1$. This configuration tests the filter's effectiveness at integrating frequent observations with moderate noise, utilizing the UKF's strength in managing minor non-linearities.

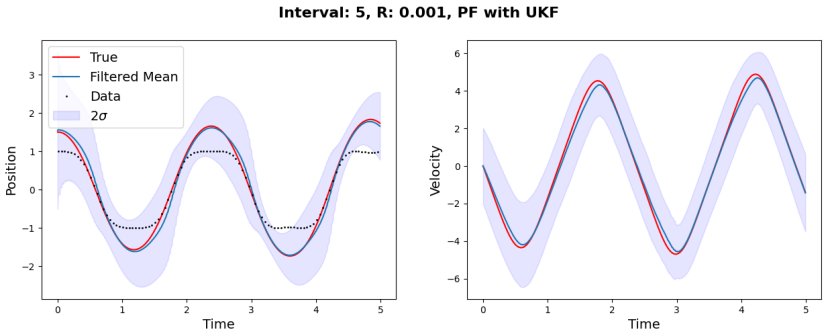

Figure 13: Tracking performance of the Particle Filter with UKF Proposal for $\Delta = 5$ and $R = 0.001$. This scenario demonstrates the PF using the UKF proposal in an almost continuous observation environment with minimal noise, showcasing the precision of the UKF in optimal conditions.

4.3.3 Particle Filter Tracking Discussion

In our examination, the PF with UKF proposal demonstrated superior performance over the PF with Dynamics proposal, particularly in scenarios characterized by higher levels of measurement noise and larger intervals between samples. This enhanced performance can largely be attributed to the UKF's more strategic generation of particles.

Unlike the dynamics proposal, the UKF proposal more intelligently incorporates data to place particles in regions of high likelihood. This intelligent distribution of particles helps in achieving:

- **Better Weight Distribution:** The UKF proposal ensures that particles are not wasted on low-probability regions, which in turn lowers the amount of negligible-weight particles.
- **Reduced Covariance:** By aligning particles more closely with the high likelihood regions, the variance around the estimated state is reduced, leading to tighter confidence bounds around the estimate.

Consequently, the PF with UKF proposal tends to track the true state with greater accuracy and less uncertainty, particularly in challenging conditions where the measurement intervals are long and noise levels are high. This makes it a preferred choice in practical applications where robust performance is essential under varying observational regimes.

4.4 Joint Posterior Investigations

This section examines the joint posterior distributions at different time instances for each combination of δ and R. For each scenario, one million samples were generated using the dynamics proposal. The joint posterior distributions at times t=0, 100, 200, 300, 400, 500 are visualized with an overlay of the Gaussian approximation obtained by the UKF, providing a direct comparison between the true distribution of the state and its UKF-based estimation.

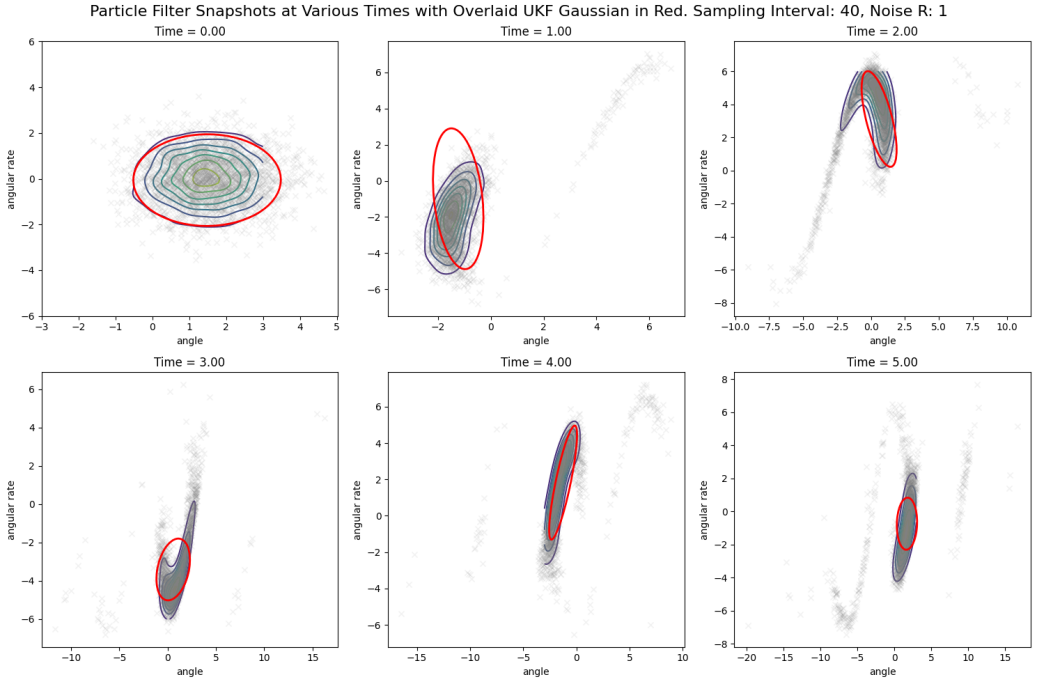

Figure 14: Joint posterior distribution at various time instances for $\Delta = 40$ and $R = 1$. The plot overlays the UKF Gaussian approximation to illustrate its alignment and deviation from the true distribution. The shown Gaussian is plotted at 2 standard deviations.

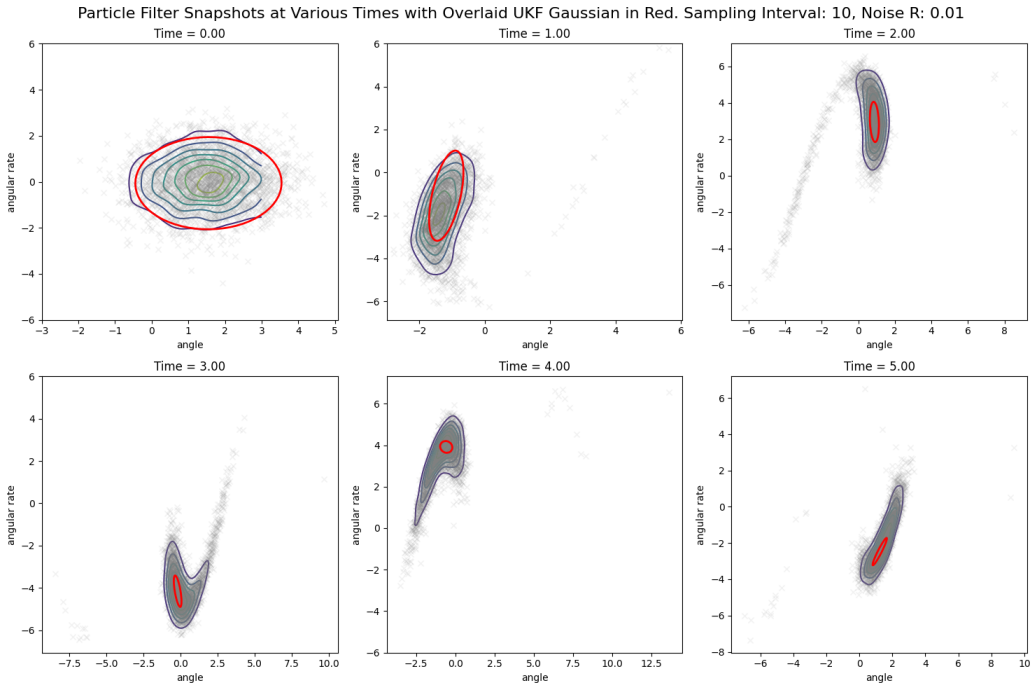

Figure 15: Joint posterior distribution at various time instances for $\Delta = 10$ and $R = 0.1$. This visualization shows the evolving nature of the posterior and the comparative effectiveness of the UKF approximation over time. The shown Gaussian is plotted at 2 standard deviations.

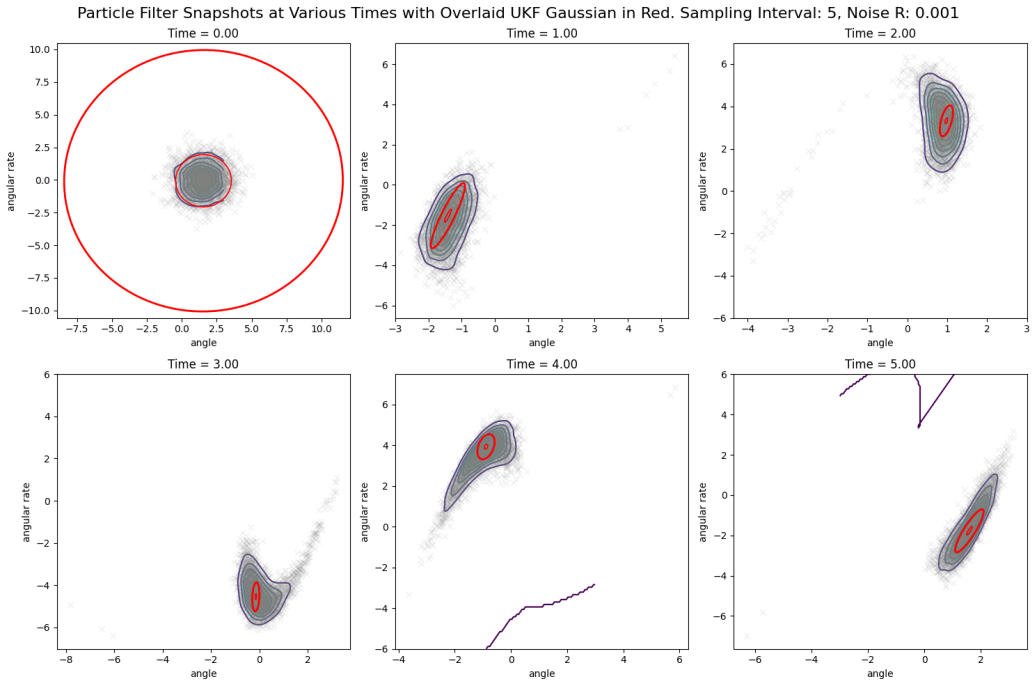

Figure 16: Joint posterior distribution at various time instances for $\Delta = 5$ and $R = 0.001$. The plot includes the UKF Gaussian fits, highlighting their approximation accuracy at high-frequency observation intervals. The shown Gaussians have an inner ring at 2 standard deviations, and outer ring at 10 standard deviations.

The investigation into the joint posteriors reveals that these distributions quickly become non-Gaussian over time. This non-Gaussian nature poses challenges for Gaussian-based filters like the UKF, which approximates the posterior as a Gaussian distribution at each step.

Although the UKF manages to align with the general shape and orientation of the majority of particles, it fails to capture the true, complex nature of the posterior distributions. Additionally, it tends to underestimate covariances, especially at low measurement noises. These findings highlight the divergence of the Gaussian approximation from the actual posterior, especially in dynamic systems characterized by inherent non-linearities and significant process or observation noise. This divergence, however, does not necessarily negate the utility of the UKF; it tends to provide a reasonable approximation that captures the central tendency and major axes of the distribution.

For a deeper analysis, the moments (mean and covariance) of one non-Gaussian posterior at $T = 2$ were computed and compared against the approximations provided by the GHKF with both 3-state and 5-state models, the EKF, and the UKF. The evaluation included plotting the particles, the true posterior, as well as the Gaussian approximations of each filter. Results showed that the GKF and UKF provided closer approximations than the EKF, with the EKF yielding the least accurate results, particularly in estimating the mean and covariance of the posterior.

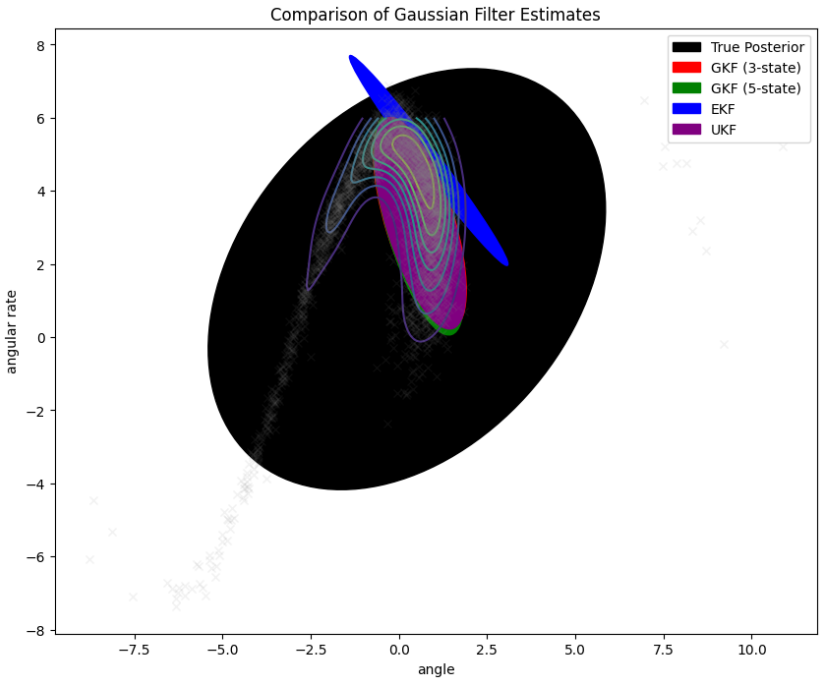

Figure 17: Comparison of particle distribution, true posterior, and Gaussian approximations at $T = 2$. This plot showcases the true mean and covariance of the posterior alongside the approximations by GKF (3-state and 5-state), EKF, and UKF. The visual assessment highlights the relative accuracy of the GKF and UKF compared to the EKF, which demonstrates the least fidelity in capturing the non-Gaussian characteristics of the posterior.

This plot and the following results quantitatively confirm these visual findings:

Table 5: Comparison of Gaussian Filters by Error in Estimating Mean and Covariance

Filter	Mean Error	Covariance Error
GKF (3-state)	1.587	10.776
GKF (5-state)	1.512	10.726
EKF	3.304	10.939
UKF	1.576	10.786

Among these, the GKF (5-state) proved to be the most accurate in capturing both the mean and covariance, closely followed by the GKF (3-state) and UKF, illustrating their greater effectiveness in handling the non-linear dynamics of the system. The EKF's performance was the weakest, confirming its limitations in dealing with highly non-linear systems where its linear approximation assumptions fall short.

4.5 PF Convergence Investigations

This section investigates the convergence behavior of the particle filtering algorithm by varying the number of particles and examining their impact on the accuracy of state estimation for $\Delta = 40$, $R = 1$. The convergence of the mean and covariance of the filters towards the reference solution, obtained with a very large sample size (1 million particles), is analyzed through several plots.

First, I plotted estimated states for the PF with UKF and PF with Dynamics for a few particle amounts : 5, 100, 1000. While both filters are unstable at very low particle counts, the UKF proposal tends to become closer to the true data much more quickly, and in a more stable fashion, which is what we expect given the proposal being formed with data in mind.

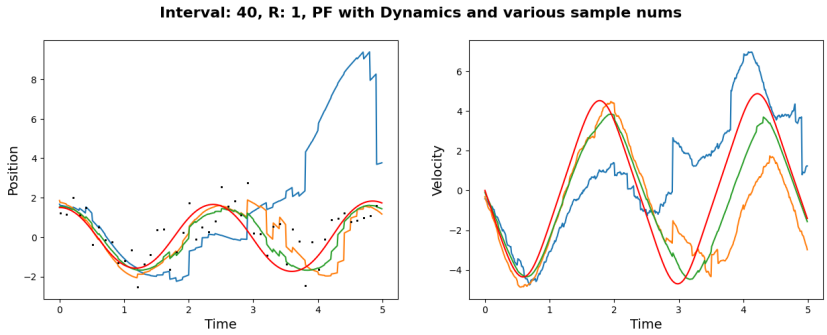

Figure 18: Tracking performance of the Dynamics Particle Filter against true data and measurements for varying particle numbers. Red lines represent the true state, blue for particle number 5, orange for particle number 100, and green for particle number 1000, illustrating the effect of increasing particle count on tracking accuracy.

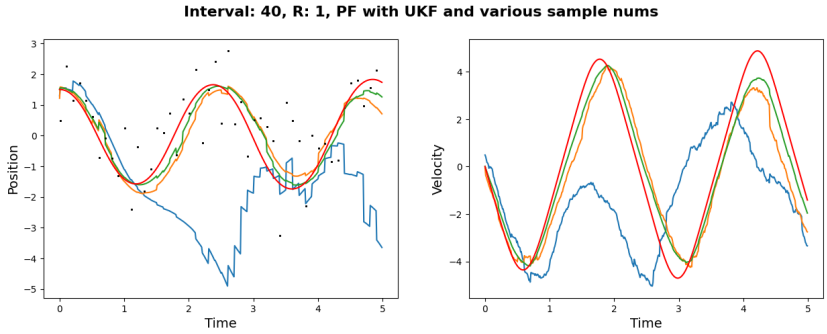

Figure 19: Tracking performance of the UKF Particle Filter against true data and measurements for varying particle numbers. Red lines represent the true state, blue for particle number 5, orange for particle number 100, and green for particle number 1000, illustrating the effect of increasing particle count on tracking accuracy.

To quantify this error more explicitly, I plotted the error of the Dynamics PF and UKF PF vs a 1 million sample reference for various particle numbers.

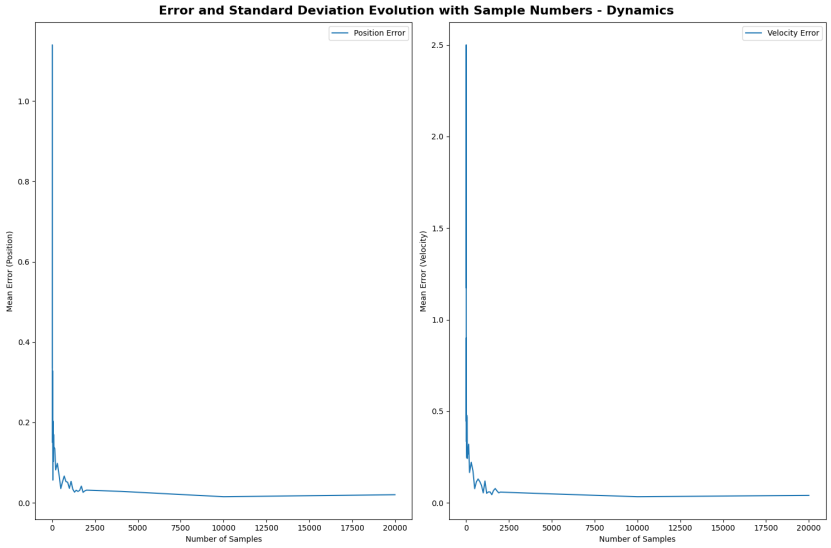

Figure 20: Error in estimates of the PF with Dynamics proposal as particle number increases, compared against a reference simulation with 1 million samples. This graph shows the overall trend of decreasing error with increasing particle count.

The overall error seems to converge to zero relatively quickly, but zooming in gives a more informative comparison.

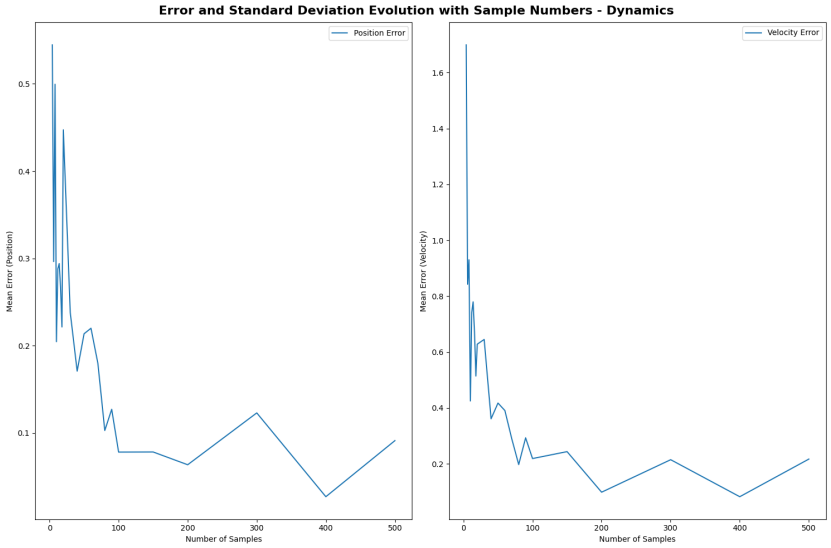

Figure 21: Zoomed-in view of estimate errors for lower particle numbers in the PF with Dynamics proposal, highlighting the initial improvements achieved as particles are added.

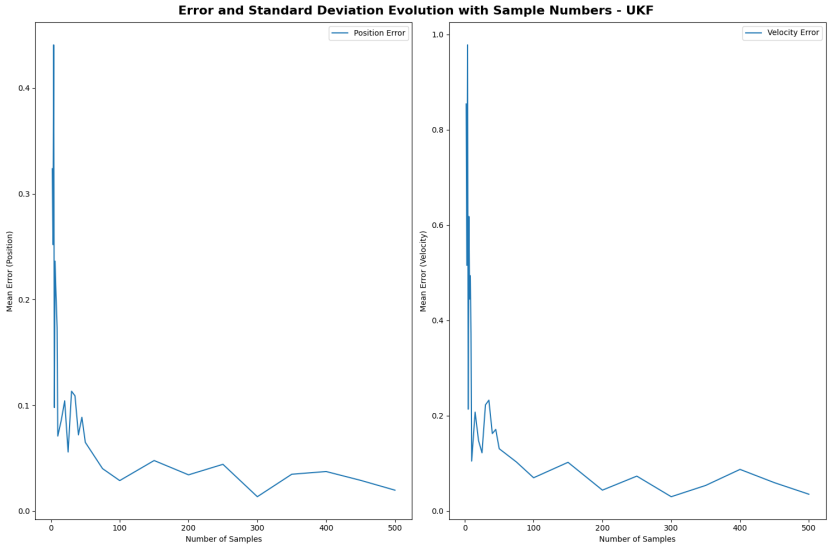

Figure 22: Estimate errors for the PF with UKF proposal for lower particle numbers, showing how the UKF proposal influences convergence rates and stability in early particle increments.

The UKF error appears to converge more stably and quickly, as we expect with its intelligent proposal.

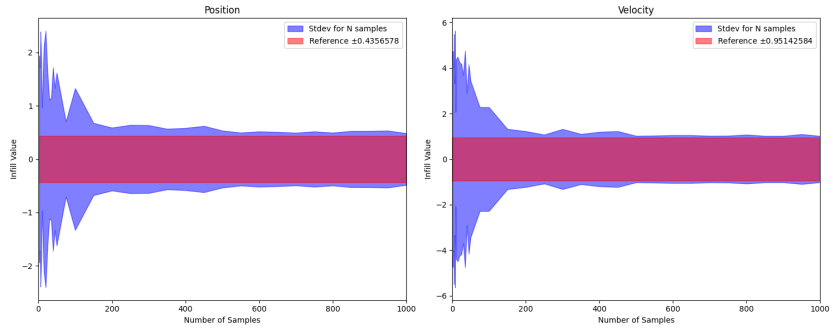

Figure 23: Standard deviation of the mean estimates from PF with Dynamics proposal for varying particle numbers, plotted against the reference standard deviations from a 1 million sample simulation.

Finally, I have included a graph comparing the convergence of the standard deviation for both filtering techniques. In this visualization, the reference's standard deviation is marked in red, while the standard deviations from each filter with fewer samples are displayed in blue. This comparison shows that the UKF tends to reach convergence closer to the reference value more swiftly than the PF with dynamics as we increase sample count, underscoring its more robust and stable performance, particularly with fewer samples.

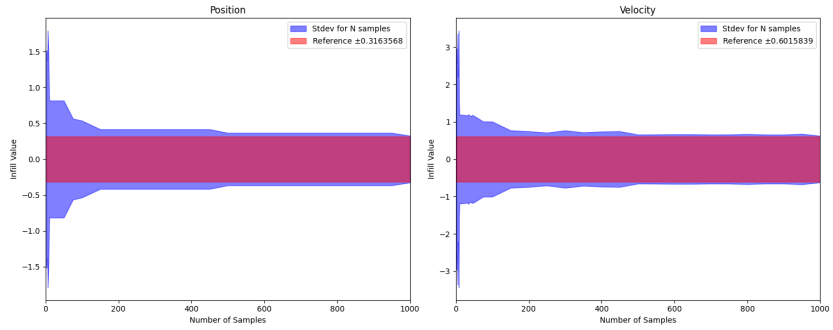

Figure 24: Standard deviation of the mean estimates from PF with UKF proposal, showing the impact of UKF on reducing variance across different particle counts compared to the Dynamics proposal.

5 Conclusions

In conclusion, this report has comprehensively analyzed the performance of various Gaussian and particle filtering techniques in the context of a nonlinear pendulum system. The findings underscore the trade-offs between computational efficiency and accuracy, with the UKF and GHKF demonstrating superior performance in handling complex dynamics compared to the EKF. Overall, this study highlights the critical role of selecting appropriate filtering strategies to optimize state estimation accuracy and robustness in dynamic systems.

References

[1] S. Särkkä, *Bayesian Filtering and Smoothing*, Cambridge University Press, 2013.

[2] Alex Gorodetsky, *Lecture Notes AEROSP567*, 2019.

6 Appendix: Code

integration_rules

April 19, 2024

[ ]:
```
%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
```

1 Integration rules with respect to Gaussian measure

Alex Gorodetsky, November 2020

In this notebook we look at what nonlinear models do to Gaussian measures and how numerical quadrature is computed.

1.1 A model setting

Lets assume we have some Gaussian distribution $X \sim \mathcal{N}(m, C)$, and that we have a nonlinear model $Y = h(X)$. We would like to estimate $\mathbb{E}[Y]$. Lets first look at the distributions of the X and Y variables. We will play around with the function

$$h(X_1, X_2) = \begin{bmatrix} X_1 \tanh(X_1 X_2) \\ \sqrt{X_2^2} \end{bmatrix} \tag{1}$$

[ ]:
```
 def reference_cov():
    std1 = 1
    std2 = 2
    rho = 0.9
    cov = np.array([[std1*std1, rho * std1 * std2], [rho * std1 * std2,␣
 ↪std2*std2]])
    return cov

def nonlin1(x):
    N = x.shape[0]
    out = np.zeros((x.shape[0], 2))
    out[:, 0] = x[:,0]*np.tanh(x[:, 0] * x[:, 1]) #+ np.random.randn(N)
```

```
    out[:, 1] = np.sqrt(x[:,1]**2) #np.sin(2.0 * np.pi * x[:, 0] * x[:, 1])
    return out
```

Now lets look at samples distributed according to X and Y

[ ]:
```
def plot_points(N, cov, func, tpoints):
    """Plot a Gaussian and a Transformed Variable

    Inputs
    ------
    N: number of samples
    cov: covariance of the gaussian
    func: nonlinear function to visualize
    tpoints: (N, 2) array of any special input points we might want to see␣
 ↪transformed

    Returns
    -------
    fig, axs, tpt
    tpt is (N, 2) array of transformed points or none if tpoints is None
    """

    # Sample from the Gaussian (mean 0)
    L = np.linalg.cholesky(cov)
    x = np.dot(L, np.random.randn(2, N)).T

    # Plot the "original" points
    fig, axs = plt.subplots(1, 2, figsize=(15, 5))
    axs[0].plot(x[:, 0], x[:, 1], 'o', ms=1)
    #axs[0].plot(0, 0, 'o', color='black', ms=10)
    if tpoints is not None:
        axs[0].plot(tpoints[:, 0], tpoints[:, 1], 'go', ms=10)
    axs[0].set_title("Original", fontsize=14)
    axs[0].set_xlabel(r'$X_1$', fontsize=14)
    axs[0].set_ylabel(r'$X_2$', fontsize=14)

    xt = func(x)
    axs[1].plot(xt[:, 0], xt[:, 1], 'ro', ms=1)
    #mt = np.mean(xt, axis=0)
    #axs[1].plot(mt[0], mt[1], 'o', color='black', ms=10)

    tpt = None
    if tpoints is not None:
        tpt = func(tpoints)
        axs[1].plot(tpt[:, 0], tpt[:, 1], 'go', ms=10)
    axs[1].set_title("Transformed", fontsize=14)
    axs[1].set_xlabel(r'$Y_1$', fontsize=14)
```

```
    axs[1].set_ylabel(r'$Y_2$', fontsize=14)

    return fig, axs, tpt
```

[ ]:
```
fig, axs, _ = plot_points(1000, reference_cov(), nonlin1, None)
```

1.2 Unscented transform

The unscented transform is a quadrature rule for integrating functions with respect to Gaussian measure. It has three tunable parameters (α, β, κ), these parameters govern the spread the quadrature points. An unscented transform rule has $2d + 1$ quadrature points, where d is the dimension of the space.

[ ]:
```
def unscented_points(mean, cov, alg='chol', alpha=1, beta=0, kappa=0):
    """Generate unscented points"""
    dim = cov.shape[0]
    lam = alpha*alpha*(dim + kappa) - dim
    if alg == "chol":
        L = np.linalg.cholesky(cov)
    elif alg == "svd":
        u, s, v = np.linalg.svd(cov)
        L = np.dot(u, np.sqrt(np.diag(s)))
    pts = np.zeros((2*dim+1, 2))
    pts[0, :] = mean
    for ii in range(1, dim+1):
        pts[ii, :] = mean + np.sqrt(dim + lam)*L[:, ii-1]
        pts[ii+dim,:] = mean - np.sqrt(dim + lam)*L[:, ii-1]

    W0m = lam / (dim + lam)
    W0C = lam / (dim + lam) + (1 - alpha*alpha + beta)
    Wim = 1/2 / (dim + lam)
    Wic = 1/2 / (dim + lam)
    return pts, (W0m, Wim, W0C, Wic)
```

1.2.1 Results with cholesky factorization

[ ]:
```
UP, w = unscented_points(np.zeros(2), reference_cov(), alg='chol')
UP
fig, axs, tpts = plot_points(1000, reference_cov(), nonlin1, UP)
```

[ ]:
```
UP
```

[ ]:
```
tpts
```

1.2.2 Results with SVD

[ ]:
```
UP, w = unscented_points(np.zeros(2), reference_cov(), alg='svd')
fig, axs, tpts = plot_points(1000, reference_cov(), nonlin1, UP)
```

[ ]:
```
UP
```

[ ]:
```
tpts
```

1.3 Gauss-hermite quadrature

Next we consider Gauss-hermite quadrature. This uses the Golub-Welsch Algorithm (read the notes)

[ ]:
```
def gh_oned(num_pts=2):
    """Gauss-hermite quadrature in 1D"""
    A = np.zeros((num_pts, num_pts))
    for ii in range(num_pts):
        #print("ii ", ii, ii==0, ii==(order-1))
        row = ii+1
        if ii == 0:
            A[ii, ii+1] = np.sqrt(row)
            A[ii+1, ii] = np.sqrt(row)
        elif ii == (num_pts-1):
            A[ii-1, ii] = np.sqrt(ii)
        else:
            A[ii, ii+1] = np.sqrt(row)
            A[ii+1, ii] = np.sqrt(row)
    pts, evec = np.linalg.eig(A)
    devec = np.dot(evec.T, evec)
    wts = evec[0,:]**2

    return pts, wts

def tensorize(nodes):
    """Tensorize nodes to obtain twod"""
    n1d = nodes.shape[0]
    twodnodes = np.zeros((n1d*n1d, 2))
    ind = 0
    for ii in range(n1d):
        for jj in range(n1d):
            twodnodes[ind, :] = np.array([nodes[ii], nodes[jj]])
            ind +=1
    return twodnodes

def gauss_hermite(dim, num_pts=2):
    """Gauss-hermite quadrature in 2D"""
    assert dim == 2, "Tensorize only implemented for dim=2"
```

```
    pts, weights = gh_oned(num_pts)
    ptsT = tensorize(pts)
    weightsT = tensorize(weights)
    weightsT = np.prod(weightsT, axis=1)
    return ptsT, weightsT

def rotate_points(points, mean, cov, alg="chol"):
    """Rotating points from standard gaussian to target Gaussian"""
    if alg == "chol":
        L = np.linalg.cholesky(cov)
    elif alg == "svd":
        u, s, v = np.linalg.svd(cov)
        L = np.dot(u, np.sqrt(np.diag(s)))


    new_points = np.zeros(points.shape)
    for ii in range(points.shape[0]):
        new_points[ii, :] = mean + np.dot(L, points[ii,:].T)
    return new_points
```

Lets add an example to show that Gaussian quadrature correctly integrates the constant, linear, and quadratic terms with only two points.

[ ]:
```
pts, wts = gh_oned(num_pts=2)
print(np.sum(wts), np.sum(pts*wts), np.sum(pts**2 * wts))
assert np.abs(np.sum(wts) - 1) < 1e-14            , "Constant is not integrated␣
 ↪correctly!"
assert np.abs(np.sum(pts*wts) - 0.0) < 1e-14      , "Expectation (linear) is␣
 ↪not computed correctly!"
assert np.abs(np.sum(pts**2 * wts) - 1.0) < 1e-14 , "Variance (quadratic) is␣
 ↪not computed correctly!"
```

1.3.1 Cholesky Square Root

[ ]:
```
ghu_points, gh_w = gauss_hermite(2, num_pts=5) # unrotated quadrature nodes
gh_points = rotate_points(ghu_points, np.zeros((2)), reference_cov(),␣
 ↪alg="chol")
fig, axs, _ = plot_points(1000, reference_cov(), nonlin1, gh_points)
```

1.3.2 Singular value decomposition square root

[ ]:
```
ghu_points, gh_w = gauss_hermite(2, num_pts=5) # unrotated quadrature nodes
print(np.zeros(2).shape)
gh_points = rotate_points(ghu_points, np.zeros((2)), reference_cov(), alg="svd")
fig, axs, _ = plot_points(1000, reference_cov(), nonlin1, gh_points)
```

[ ]:
```
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
```

[ ]:
```
from numpy.linalg import cholesky
def ukf_weights(alpha, beta, kappa, dim):
    lamda = alpha ** 2 * (dim + kappa) - dim
    wm = np.full(2 * dim + 1, 1 / (2 * (dim + lamda)))
    wc = np.full(2 * dim + 1, 1 / (2 * (dim + lamda)))

    wm[0] = lamda / (dim + lamda)
    wc[0] = lamda / (dim + lamda) + (1 - alpha ** 2 + beta)
    return wm, wc


def ukf_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))

    alpha = 1.0
    beta = 0.0
    kappa = 3 - dim
    wm, wc = ukf_weights(alpha, beta, kappa, dim)


    mean = init_mean.copy()
    covariance = init_cov.copy()
    measurement_index = 0

    for step in range(steps):
        # Compute the sigma points for the dynamics
        L = cholesky(covariance) * np.sqrt(dim + kappa)
        sigma_points = np.hstack([mean.reshape(-1, 1), (mean + L.T).T, (mean -␣
↪L.T).T])
        print(sigma_points.shape)

        # Progagate through the dynamics
        sigma_points[0, :] += timestep * sigma_points[1, :]
        sigma_points[1, :] -= gravity * timestep * np.sin(sigma_points[0, :])

        # Predicted state distribution
```

```
        mean = np.dot(wm, sigma_points.T)
        covariance = np.dot(wc * (sigma_points - mean.reshape(-1, 1)),␣
↪(sigma_points - mean.reshape(-1, 1)).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

            # Compute the sigma points for the observation
            L = cholesky(covariance) * np.sqrt(dim + kappa)
            sigma_points = np.hstack([mean.reshape(-1, 1), (mean + L.T).T,␣
↪(mean - L.T).T])
            sigma_observations = np.sin(sigma_points[0, :])

            # Sigma points measurement mean and covariance
            predicted_mu = np.dot(wm, sigma_observations)
            predicted_cov = np.dot(wc * (sigma_observations - predicted_mu),␣
↪sigma_observations - predicted_mu) + measure_noise
            cross_cov = np.dot((sigma_points - mean.reshape(-1, 1)), wc *␣
↪(sigma_observations - predicted_mu))

            # Gain and update
            K = cross_cov / predicted_cov
            mean += K * (current_meas - predicted_mu)
            covariance -= predicted_cov * K.reshape(-1, 1) * K

        # Store the predicted/updated mean and covariance
        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
```

[ ]:
```
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
```

[ ]:
```
from itertools import product

def gh_kalman_filter(init_mean, init_cov, gravity, process_noise, timestep,␣
↪measure_noise, measurements, steps, delta=10, order=5):
    dim = init_mean.shape[0]
    sp1D, weights1D = gh_oned(order)
    weights1D /= weights1D.sum()

    sigma_points = np.array(list(product(*(sp1D for _ in range(dim))))).T
    weights = np.array(list(product(*(weights1D for _ in range(dim))))).
↪prod(axis=1)

    num_steps = steps # Total number of steps to simulate based on measurement␣
↪frequency
    filtered_means = np.empty((num_steps, dim))
    filtered_covariances = np.empty((num_steps, dim, dim))
```

```
    filtered_stds = np.empty((num_steps, dim))

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

    for step in range(num_steps):
        if step % delta == 0 and measurement_index < len(measurements):
            current_meas = measurements[measurement_index]
            measurement_index += 1
        else:
            current_meas = None

        # Prediction step
        trans_sigma_points = rotate_points(sigma_points.T, mean.reshape(-1, 1).
↪squeeze(), covariance, alg="chol").T
        trans_sigma_points[0, :] += timestep * trans_sigma_points[1, :]
        trans_sigma_points[1, :] -= gravity * timestep * np.
↪sin(trans_sigma_points[0, :])

        mean = np.dot(trans_sigma_points, weights)
        covariance = np.dot(weights.reshape(1, -1) * (trans_sigma_points - mean.
↪reshape(-1, 1)),
                            (trans_sigma_points - mean.reshape(-1, 1)).T) +␣
↪process_noise

        # Update step if current_meas is available
        if current_meas is not None:
            trans_sigma_points = rotate_points(sigma_points.T, mean.reshape(-1,␣
↪1).squeeze(), covariance, alg="chol").T
            meas_predictions = np.sin(trans_sigma_points[0, :])
            predicted_mean = np.dot(meas_predictions, weights)
            predicted_cov = np.dot(weights * (meas_predictions -␣
↪predicted_mean),
                                   (meas_predictions - predicted_mean)) +␣
↪measure_noise
            cross_cov = np.dot((trans_sigma_points - mean.reshape(-1, 1)),␣
↪weights * (meas_predictions - predicted_mean))

            kalman_gain = cross_cov / predicted_cov
            mean += kalman_gain * (current_meas - predicted_mean)
            covariance -= predicted_cov * kalman_gain[:, np.newaxis] *␣
↪kalman_gain

        # Store the predicted/updated mean and covariance
        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


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)
        ghkf_3 = gh_kalman_filter(init_mean, init_cov, gravity, process_noise,␣
 ↪timestep, measure_noise, obsData.obs, steps, delta, order=3)
        ghkf_5 = gh_kalman_filter(init_mean, init_cov, gravity, process_noise,␣
 ↪timestep, measure_noise, obsData.obs, steps, delta, order=5)
        ukf = ukf_filter(init_mean, init_cov, gravity, process_noise, timestep,␣
 ↪measure_noise, obsData.obs, steps, delta)

        means_3 = ghkf_3.means
        covariances_3 = ghkf_3.covs
        means_5 = ghkf_5.means
        covariances_5 = ghkf_5.covs
        means_ekf = ekf.means
        covariances_ekf = ekf.covs
        ukf_means = ukf.means
        ukf_covariances = ukf.covs

        rmse_ghkf = rmse(means_3[:, :1], trueData.obs[:, :1])
```

```
        print(f"ghkf_3 RMSE: {rmse_ghkf}")
        rmse_ghkf = rmse(means_5[:, :1], trueData.obs[:, :1])
        print(f"ghkf_5 RMSE: {rmse_ghkf}")
        rmse_ekf = rmse(means_ekf[:, :1], trueData.obs[:, :1])
        print(f"ekf RMSE: {rmse_ekf}")
        rmse_ukf = rmse(ukf_means[:, :1], trueData.obs[:, :1])
        print(f"ukf RMSE: {rmse_ukf}")
```

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

# Iterate through each delta and measure_noise combination
for i, delta in enumerate(deltas):
    for j, measure_noise in enumerate(measure_noises):
        ax = axs[i, j]

        # Generate pendulum data and run the EKF
        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)

        # Plotting true and observed data
        ax.plot(trueData.times, trueData.obs[:, 0], 'k', label='True Theta')
        ax.plot(trueData.times, trueData.obs[:, 1], 'k--', label='True␣
 ↪Theta_dot')
        ax.plot(obsData.times, obsData.obs[:, 0], 'ro', alpha=0.4,␣
 ↪label='Observed Theta')

        # Plot EKF estimates
        ax.plot(trueData.times, ekf.means[:, 0], 'r--', label='EKF Theta')
        ax.plot(trueData.times, ekf.means[:, 1], 'b--', label='EKF Theta_dot')

        # Confidence intervals (optional if you want to include them)
        ax.fill_between(trueData.times,
                        ekf.means[:, 0] - 2 * ekf.stds[:, 0],
                        ekf.means[:, 0] + 2 * ekf.stds[:, 0],
                        color='red', alpha=0.3)
        ax.fill_between(trueData.times,
                       ekf.means[:, 1] - 2 * ekf.stds[:, 1],
                       ekf.means[:, 1] + 2 * ekf.stds[:, 1],
                       color='blue', alpha=0.2)

        ax.set_title(f"Delta: {delta}, Measure Noise: {measure_noise}")
        ax.set_xlabel('Time (s)')
        ax.set_ylabel('State')
        if i == 0 and j == 0:
            ax.legend()
```

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

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

# Iterate through each delta and measure_noise combination
for i, delta in enumerate(deltas):
    for j, measure_noise in enumerate(measure_noises):
        ax = axs[i, j]

        # Generate pendulum data and run the GHKF with order 3
        trueData, obsData = generate_pendulum(x0, gravity, process_noise,␣
↪timestep, measure_noise, steps, delta)
        ghkf_3 = gh_kalman_filter(init_mean, init_cov, gravity, process_noise,␣
↪timestep, measure_noise, obsData.obs, steps, delta, order=3)

        # Plotting true and observed data
        ax.plot(trueData.times, trueData.obs[:, 0], 'k-', label='True Theta')
        ax.plot(trueData.times, trueData.obs[:, 1], 'k--', label='True␣
↪Theta_dot')
        ax.plot(obsData.times, obsData.obs[:, 0], 'ro', alpha=0.4,␣
↪label='Observed Theta')

        # Plot GHKF estimates
        ax.plot(trueData.times, ghkf_3.means[:, 0], 'r--', label='GHKF Order 3␣
↪Theta')
        ax.plot(trueData.times, ghkf_3.means[:, 1], 'b--', label='GHKF Order 3␣
↪Theta_dot')

        # Confidence intervals
        ax.fill_between(trueData.times,
                        ghkf_3.means[:, 0] - 2 * ghkf_3.stds[:, 0],
                        ghkf_3.means[:, 0] + 2 * ghkf_3.stds[:, 0],
                        color='red', alpha=0.3)
        ax.fill_between(trueData.times,
                        ghkf_3.means[:, 1] - 2 * ghkf_3.stds[:, 1],
                        ghkf_3.means[:, 1] + 2 * ghkf_3.stds[:, 1],
                       color='blue', alpha=0.2)

        ax.set_title(f"Delta: {delta}, Measure Noise: {measure_noise}")
        ax.set_xlabel('Time (s)')
        ax.set_ylabel('State')
        if i == 0 and j == 0:
            ax.legend()
```

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

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

# Iterate through each delta and measure_noise combination
for i, delta in enumerate(deltas):
    for j, measure_noise in enumerate(measure_noises):
        ax = axs[i, j]

        # Generate pendulum data and run the GHKF with order 3
        trueData, obsData = generate_pendulum(x0, gravity, process_noise,␣
 ↪timestep, measure_noise, steps, delta)
        ghkf_3 = gh_kalman_filter(init_mean, init_cov, gravity, process_noise,␣
 ↪timestep, measure_noise, obsData.obs, steps, delta, order=5)

        # Plotting true and observed data
        ax.plot(trueData.times, trueData.obs[:, 0], 'k-', label='True Theta')
        ax.plot(trueData.times, trueData.obs[:, 1], 'k--', label='True␣
 ↪Theta_dot')
        ax.plot(obsData.times, obsData.obs[:, 0], 'ro', alpha=0.4,␣
 ↪label='Observed Theta')

        # Plot GHKF estimates
        ax.plot(trueData.times, ghkf_3.means[:, 0], 'r--', label='GHKF Order 5␣
 ↪Theta')
        ax.plot(trueData.times, ghkf_3.means[:, 1], 'b--', label='GHKF Order 5␣
 ↪Theta_dot')

        # Confidence intervals
        ax.fill_between(trueData.times,
                        ghkf_3.means[:, 0] - 2 * ghkf_3.stds[:, 0],
                        ghkf_3.means[:, 0] + 2 * ghkf_3.stds[:, 0],
                        color='red', alpha=0.3)
        ax.fill_between(trueData.times,
                        ghkf_3.means[:, 1] - 2 * ghkf_3.stds[:, 1],
                        ghkf_3.means[:, 1] + 2 * ghkf_3.stds[:, 1],
                       color='blue', alpha=0.2)

        ax.set_title(f"Delta: {delta}, Measure Noise: {measure_noise}")
        ax.set_xlabel('Time (s)')
        ax.set_ylabel('State')
        if i == 0 and j == 0:
            ax.legend()

plt.tight_layout()
```

```
plt.show()
```

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

# Iterate through each delta and measure_noise combination
for i, delta in enumerate(deltas):
    for j, measure_noise in enumerate(measure_noises):
        ax = axs[i, j]

        # Generate pendulum data and run the EKF
        trueData, obsData = generate_pendulum(x0, gravity, process_noise,␣
 ↪timestep, measure_noise, steps, delta)
        ekf = ukf_filter(init_mean, init_cov, gravity, process_noise, timestep,␣
 ↪measure_noise, obsData.obs, steps, delta)

        # Plotting true and observed data
        ax.plot(trueData.times, trueData.obs[:, 0], 'k', label='True Theta')
        ax.plot(trueData.times, trueData.obs[:, 1], 'k--', label='True␣
 ↪Theta_dot')
        ax.plot(obsData.times, obsData.obs[:, 0], 'ro', alpha=0.4,␣
 ↪label='Observed Theta')

        # Plot EKF estimates
        ax.plot(trueData.times, ekf.means[:, 0], 'r--', label='UKF Theta')
        ax.plot(trueData.times, ekf.means[:, 1], 'b--', label='UKF Theta_dot')

        # Confidence intervals (optional if you want to include them)
        ax.fill_between(trueData.times,
                        ekf.means[:, 0] - 2 * ekf.stds[:, 0],
                        ekf.means[:, 0] + 2 * ekf.stds[:, 0],
                        color='red', alpha=0.3)
        ax.fill_between(trueData.times,
                       ekf.means[:, 1] - 2 * ekf.stds[:, 1],
                       ekf.means[:, 1] + 2 * ekf.stds[:, 1],
                       color='blue', alpha=0.2)

        ax.set_title(f"Delta: {delta}, Measure Noise: {measure_noise}")
        ax.set_xlabel('Time (s)')
        ax.set_ylabel('State')
        if i == 0 and j == 0:
            ax.legend()

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

[ ]:
```
"""
\begin{table}[H]
\centering
\caption{GHKF 3rd order RMSE for different \(\Delta\) and Measurement Noise␣
 ↪\(R\)}
\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{Delta / Noise} & \textbf{0.001} & \textbf{0.01} & \textbf{0.1} &␣
 ↪\textbf{1} \\ \hline
\textbf{5}  & 0.171 & 0.209 & 0.282 & 0.412 \\ \hline
\textbf{10} & 0.160 & 0.180 & 0.246 & 0.378 \\ \hline
\textbf{20} & 0.170 & 0.215 & 0.348 & 2.015 \\ \hline
\textbf{40} & 0.233 & 0.262 & 0.394 & 0.744 \\ \hline
\end{tabular}
\end{table}


\begin{table}[H]
\centering
\caption{GHKF 5th order RMSE for different \(\Delta\) and Measurement Noise␣
 ↪\(R\)}
\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{Delta / Noise} & \textbf{0.001} & \textbf{0.01} & \textbf{0.1} &␣
 ↪\textbf{1} \\ \hline
\textbf{5}  & 0.183 & 0.220 & 0.293 & 0.413 \\ \hline
\textbf{10} & 0.175 & 0.189 & 0.248 & 0.385 \\ \hline
\textbf{20} & 0.190 & 0.227 & 0.351 & 1.608 \\ \hline
\textbf{40} & 0.247 & 0.280 & 0.407 & 0.690 \\ \hline
\end{tabular}
\end{table}


\begin{table}[H]
\centering
\caption{EKF RMSE for different \(\Delta\) and Measurement Noise \(R\)}
\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{Delta / Noise} & \textbf{0.001} & \textbf{0.01} & \textbf{0.1} &␣
 ↪\textbf{1} \\ \hline
\textbf{5}  & 0.046 & 0.113 & 0.406 & 9.983 \\ \hline
\textbf{10} & 0.040 & 0.079 & 0.203 & 0.371 \\ \hline
\textbf{20} & 0.050 & 0.120 & 5.629 & 10.205 \\ \hline
\textbf{40} & 0.061 & 0.081 & 0.182 & 2.727 \\ \hline
\end{tabular}
\end{table}
```

```
\begin{table}[H]
\centering
\caption{UKF RMSE for different \(\Delta\) and Measurement Noise \(R\)}
\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{Delta / Noise} & \textbf{0.001} & \textbf{0.01} & \textbf{0.1} & \textbf{1} \\ \hline
\textbf{5}  & 0.171 & 0.209 & 0.282 & 0.410 \\ \hline
\textbf{10} & 0.160 & 0.180 & 0.246 & 0.377 \\ \hline
\textbf{20} & 0.170 & 0.214 & 0.348 & 2.182 \\ \hline
\textbf{40} & 0.233 & 0.261 & 0.393 & 0.722 \\ \hline
\end{tabular}
\end{table}
"""
```

[ ]:

particle_filtering

April 19, 2024

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

[ ]:
```
def gh_oned(num_pts=2):
    """Gauss-hermite quadrature in 1D"""
    A = np.zeros((num_pts, num_pts))
    for ii in range(num_pts):
        #print("ii ", ii, ii==0, ii==(order-1))
        row = ii+1
        if ii == 0:
            A[ii, ii+1] = np.sqrt(row)
            A[ii+1, ii] = np.sqrt(row)
        elif ii == (num_pts-1):
            A[ii-1, ii] = np.sqrt(ii)
        else:
            A[ii, ii+1] = np.sqrt(row)
            A[ii+1, ii] = np.sqrt(row)
    pts, evec = np.linalg.eig(A)
    devec = np.dot(evec.T, evec)
    wts = evec[0,:]**2

    return pts, wts

def tensorize(nodes):
    """Tensorize nodes to obtain twod"""
    n1d = nodes.shape[0]
    twodnodes = np.zeros((n1d*n1d, 2))
    ind = 0
    for ii in range(n1d):
        for jj in range(n1d):
            twodnodes[ind, :] = np.array([nodes[ii], nodes[jj]])
```

```
            ind +=1
    return twodnodes

def gauss_hermite(dim, num_pts=2):
    """Gauss-hermite quadrature in 2D"""
    assert dim == 2, "Tensorize only implemented for dim=2"
    pts, weights = gh_oned(num_pts)
    ptsT = tensorize(pts)
    weightsT = tensorize(weights)
    weightsT = np.prod(weightsT, axis=1)
    return ptsT, weightsT

def rotate_points(points, mean, cov, alg="chol"):
    """Rotating points from standard gaussian to target Gaussian"""
    if alg == "chol":
        L = np.linalg.cholesky(cov)
    elif alg == "svd":
        u, s, v = np.linalg.svd(cov)
        L = np.dot(u, np.sqrt(np.diag(s)))


    new_points = np.zeros(points.shape)
    for ii in range(points.shape[0]):
        new_points[ii, :] = mean + np.dot(L, points[ii,:].T)
    return new_points
```

1 Particle filtering

Alex Gorodetsky, November 2020, 2021, 2022

In this notebook we will implement and describe the particle filter.

1.1 Empirical distributions

Recall that the particle filter is an importance sampling technique. It allows for estimating of expectations with respect to the smoothing distribution as

$$\mathbb{E}\left[g(X_0, \dots, X_n) \mid y_n\right] \approx \sum_{i=1}^{N} w_n^{(i)} g(x_0^{(i)}, \dots, x_n^{(i)}) \tag{1}$$

where

$$\sum_{i=1}^{N} w_n^{(i)} = 1.$$

so that we can effectively view this approximation as exact with respect to an empirical density

$$P(X_0, \dots, X_n \mid y_n) \approx \sum_{i=1}^{N} w_n^{(i)} \delta_{\mathcal{X}_n^{(i)}}(X_0, \dots, X_n) = \hat{P}(X_0, \dots, X_n \mid y_n) \tag{2}$$

where $\mathcal{X}_n^{(i)} = (x_0^{(i)}, x_1^{(i)}, \ldots, x_n^{(i)})$. To be even more concrete, an empirical distribution is essentially representing a **discrete random variable** that only takes values given by the samples $\mathcal{X}_n^{(i)}$ with probability $w_n^{(i)}$.

One nice thing about empirical distirbutions is that they are easy to marginalize – so that the filtering distribution is simply

$$\hat{P}(X_n \mid y_n) = \sum_{i=1}^{N} w_n^{(i)} \delta_{x_n^{(i)}}(X_n). \tag{3}$$

1.1.1 Resampling

One useful thing to do with empirical distributions is to be able to **resample** them – which means generate new independent samples from the discrete random variables. In this way the resulting samples form an empirical distribution with equal weights themselves.

[ ]:

[ ]:
```
def resample(Nsamples, samples, weights):
    """Generate *Nsamples* samples from an empirical distribution defined by␣
↪*samples* and *weights*

    Inputs
    ------
    Nsamples: integer, number of samples to generate
    samples: (N, d) array of N samples of dimension d that form the empirical␣
↪distribution
    weights: (N, ) array of N weights

    Returns
    --------
    samples_out: (Nsamples, d) new samples
    weights_out: (Nsamples, ) new weights equal to 1 / N
    """

    N = samples.shape[0]  # get number of points that make up the empirical␣
↪distribution
    rr = np.arange(N) # get an ordered set of numbers from 0 to N-1

    # Randomly choose the integers (with replacement) between 0 to N-1 with␣
↪probabilities given by the weights
    samp_inds = np.random.choice(rr, Nsamples, p=weights)

    # subselect the samples chosen
    samples_out = samples[samp_inds, :]

    # return uniform weights
```

```
    weights_out = np.ones((Nsamples))/Nsamples
    return samples_out, weights_out
```

1.1.2 Statistics computation

Next we want to be able to compute statistics of the distribution from the samples. For the purposes of filtering we will only use the mean and standard deviations of each state in the filtering distribution. These are given by

$$\mu = \mathbb{E}\left[X_n \mid y_n\right] \approx \sum_{i=1}^{N} w_n^{(i)} x_n^{(i)} \tag{4}$$

$$\sigma = \sqrt{\mathbb{V}\mathrm{ar}\left[X_n \mid y_n\right]} \approx \sqrt{\sum_{i=1}^{N} w_n^{(i)} \left(x_n^{(i)} - \mu\right)^2} \tag{5}$$

[ ]:
```
def compute_mean_std(samples, weights):
    """Compute the mean and standard deviation of multiple empirical␣
 ↪distirbution.

    Inputs
    ------
    samples: (N, d, m) array of samples defining the empirical distribution
    weights: (N, m) array of weights

    Returns
    -------
    means: (m, d) array of means
    stds: (m, d) array of standard deviations

    Notes
    -----
    m is the number of empirical distributions
    """

    N, d, m = samples.shape
    means = np.zeros((m, d))
    stds = np.zeros((m, d))
    for ii in range(m):
        means[ii, :] = np.dot(weights[:, ii], samples[:, :, ii])
        stds[ii, :] = np.sqrt(np.dot(weights[:, ii], (samples[:, :, ii] - np.
 ↪tile(means[ii, :], (N, 1)))**2))
    return means, stds
```

1.2 Algorithm

We are now ready to code up the algorithm. We begin by coding a single step of the particle filter, that is a step between observations.

[ ]:
```
def step(prop, proppdf, current_samples, current_weights, likelihood, data,␣
↪propagator, current_covs=None):
    """
    Propagate a particle filter

    Inputs
    --------
    prop            - proposal function (current_state, data)
    proppdf         - proposal function logpdf
    current_samples - ensemble of samples
    current_weights - ensemble of weights
    likelihood      - function to evaluate the log likelihood (samples, data)
    data            - Observation
    propagator      - dynamics logpdf

    @returns samples and weights after assimilating the data
    """
    #new_samples = np.zeros(current_samples.shape)
    #new_weights = np.zeros(current_weights.shape)
    try:
        new_samples = prop(current_samples, data)
        new_weights = likelihood(new_samples, data) + propagator(new_samples,␣
↪current_samples) - \
                                  proppdf(new_samples, current_samples, data)
    except:
        #print("+++++++")



        new_samples, new_means, new_covs = prop(current_samples, current_covs,␣
↪data)


        new_weights = likelihood(new_samples, data)
        #new_weights += propagator(new_samples, current_samples)
        new_weights -= proppdf(current_samples, new_samples, new_means,␣
↪new_covs)
        new_weights = np.exp(new_weights) * current_weights
        new_weights = new_weights / np.sum(new_weights)

        # print("_____")
        # print(likelihood(new_samples, data))
        # print("_____")
        # print(propagator(new_samples, current_samples))
        # print("_____")
        #print(proppdf(current_samples, new_samples, new_means, new_covs))
        return new_samples, new_weights, new_covs
```

```
    new_weights = np.exp(new_weights) * current_weights
    #for ii in range(new_samples.shape[0]):
    #    new_samples[ii, :] = prop(current_samples[ii, :], data)
    #    new_weights[ii] = np.exp(likelihood(new_samples[ii, :], data) +
    #                             propagator(new_samples[ii, :],␣
↪current_samples[ii, :]) -
    #                             proppdf(new_samples[ii, :],␣
↪current_samples[ii, :], data))
    #    new_weights[ii] *= current_weights[ii]

    # normalize weights
    new_weights = new_weights / np.sum(new_weights)
    return new_samples, new_weights
```

[ ]:
```
def step_dynamics_only(prop, proppdf, current_samples, current_weights,␣
↪propagator, current_covs=None):
    """
    Propagate particles through dynamics only without any observations.

    Inputs:
    -------
    prop            - Proposal function for dynamics propagation
    proppdf         - Log probability density function of the proposal
    current_samples - Current ensemble of samples (states)
    current_weights - Current ensemble of weights
    propagator      - Function to evaluate dynamics logpdf

    Returns:
    --------
    new_samples     - Samples after dynamics propagation
    new_weights     - Weights after dynamics propagation (unchanged as no␣
↪observations are assimilated)
    """
    try:
        new_samples = prop(current_samples)
    except:
        new_samples, new_means, new_covs = prop(current_samples, current_covs,␣
↪data=None)
        new_weights = current_weights / np.sum(current_weights)
        return new_samples, new_weights, new_covs

    new_weights = current_weights / np.sum(current_weights)

    return new_samples, new_weights
```

Next we create the "outer loop" with all necessary components

[ ]:
```
def particle_filter(data, prior_mean, prior_cov, steps,
                    prop, proppdf, likelihood, propagator,␣
↪observation_interval=1,
                    nsamples=1000, resampling_threshold_frac=0.3,␣
↪current_covs=None):
    """Particle Filter

    Inputs
    -------
    data: (nsteps, m) array of data points, N is the time index, m is the␣
↪dimensionality of the data
    prior_mean: (d), prior mean
    prior_cov: (d, d), prior mean
    Nsamples: integer, number of samples in the empirical distribution
    resampling_threshold_frac: float between 0 and 1 indicating to resample␣
↪when effective sample size below frac of nsamples

    Returns
    -------
    samples: (nsamples, d, nsteps)
    weights: (nsamples, nsteps)
    eff: (nsamples), effective sample size

    Notes
    -----
    For documentation of prop, proppdf, likelihood, and propagator -- see the␣
↪step function
    """

    d = prior_mean.shape[0]
    nsteps = steps-1

    # Allocate memory
    samples = np.zeros((nsamples, d, nsteps + 1))
    weights = np.zeros((nsamples, nsteps + 1))
    eff = np.zeros(nsteps + 1)  # effective sample size at each step

    # Initialize samples from the prior
    L = np.linalg.cholesky(prior_cov)
    samples[:, :, 0] = np.tile(prior_mean, (nsamples, 1)) + np.dot(L, np.random.
↪randn(d, nsamples)).T
    weights[:, 0] = 1.0 / nsamples
    eff[0] = nsamples

    resamp_threshold = int(nsamples * resampling_threshold_frac)
```

```
    for ii in range(1, nsteps + 1):
        if (ii - 1) % observation_interval == 0:
            # Use observation to update particles
            if current_covs is not None:
                samples[:, :, ii], weights[:, ii] , current_covs = step(prop,␣
↪proppdf, samples[:, :, ii - 1], weights[:, ii - 1],
                                                        likelihood, data[int((ii -␣
↪1)/observation_interval)], propagator, current_covs=current_covs)
            else:
                samples[:, :, ii], weights[:, ii] = step(prop, proppdf,␣
↪samples[:, :, ii - 1], weights[:, ii - 1],
                                                        likelihood, data[int((ii -␣
↪1)/observation_interval)], propagator)
        else:
            # Propagate particles without observation
            if current_covs is not None:

                samples[:, :, ii], weights[:, ii], current_covs =␣
↪step_dynamics_only(prop, proppdf, samples[:, :, ii - 1], weights[:, ii - 1],
                                                        propagator,␣
↪current_covs=current_covs)
            else:
                samples[:, :, ii], weights[:, ii] = step_dynamics_only(prop,␣
↪proppdf, samples[:, :, ii - 1], weights[:, ii - 1],
                                                        propagator)
        # Compute the effective sample size
        eff[ii] = 1.0 / np.sum(weights[:, ii]**2)

        if ii % 1 == 0:
            print("Step:", ii, "Effective sample size:", eff[ii])

        # Resample if effective sample size is below the threshold
        if eff[ii] < resamp_threshold:
            samples[:, :, ii], weights[:, ii] = resample(nsamples, samples[:, :
↪, ii], weights[:, ii])

    return samples, weights, eff
```

1.3 Demo problem

We will be testing the algorithm on the dynamics of a nonlinear pendulum discretized using forward euler. The observation operator is nonlinear. This is almost what you need for your projects, however in your projects you wont observe at every time step.

[ ]:
```
def pendulum_dyn(current_state, dt=0.1):
    """Pendulum dynamics

    Inputs
    ------
    Current_state : either (2,) or (N, 2) for vectorized input
    """
    if current_state.ndim == 1:
        next_state = np.zeros((2))
        next_state[0] = current_state[0] + dt * current_state[1]
        next_state[1] = current_state[1] - dt * 9.81 * np.sin(current_state[0])
    else: # multiple inputs
        next_state = np.zeros(current_state.shape)
        next_state[:, 0] = current_state[:, 0] + dt * current_state[:, 1]
        next_state[:, 1] = current_state[:, 1] - dt * 9.81 * np.
 ↪sin(current_state[:, 0])
    return next_state

def observe(current_state):
    if current_state.ndim == 1:
        out = np.zeros((1))
        out[0] = np.sin(current_state[0])
    else:
        out = np.zeros((current_state.shape[0], 1))
        out[:, 0] = np.sin(current_state[:, 0])
    return out
```

[ ]:
```
def simulate_pendulum(x0, dt, Nsteps, observation_interval, noise_std=1.0):
    true = np.zeros((Nsteps, 2))
    true_value = np.zeros((Nsteps, 2))
    true[0, :] = x0
    times = np.arange(0, Nsteps * dt, dt)
    data = []
    data_times = []

    for ii in range(1, Nsteps):
        true[ii, :] = pendulum_dyn(true[ii-1, :], dt=dt)

        if (ii-1) % observation_interval == 0:
            # Only observe at specified intervals
            observed_value = observe(true[ii, :]) + np.random.randn() *␣
 ↪noise_std
            data.append(observed_value)
            data_times.append(times[ii])
        true_value[ii, :] = observe(true[ii, :])

    return true, times,  np.array(data), np.array(data_times)
```

1.3.1 Data generation

[ ]:
```
x0 = np.array([1.5, 0])
dt = 0.01
Nsteps = 500
observation_interval = 1
noise_std = 1  # Standard deviation of noise

true, times, data, data_times = simulate_pendulum(x0, dt, Nsteps,␣
 ↪observation_interval, noise_std=noise_std)

fig, axs = plt.subplots(1, 2, figsize=(15, 5))
axs[0].plot(times, true[:, 0], label='True position')
axs[0].plot(data_times, data, 'ko', ms=3, label='Observed data')
axs[0].set_xlabel("Time", fontsize=14)
axs[0].set_ylabel("Position", fontsize=14)
axs[0].legend()

axs[1].plot(times, true[:, 1], label='True velocity')
axs[1].set_xlabel("Time", fontsize=14)
axs[1].set_ylabel("Velocity", fontsize=14)
axs[1].legend()

plt.show()
```

[ ]:
```
data.shape
```

1.3.2 Bootstrap particle filter

We first setup the process noise, the proposals, and the likelihood

Recall the dynamics are

$$X_k = \phi(X_{k-1}) + \xi, \quad \xi \sim \mathcal{N}(0, \Sigma) \tag{6}$$

So the proposal is

$$\pi(X_k \mid X_{k-1}) = P(X_k \mid X_{k-1}) = \mathcal{N}(\phi(X_{k-1}), \Sigma) \tag{7}$$

The likelihood comes from the observation model

$$Y_k = h(X_k) + \eta, \quad \eta \sim \mathcal{N}(0, \Gamma) \tag{8}$$

so that

$$P(Y_k \mid X_k) = \mathcal{N}(h(X_k), \Gamma) \tag{9}$$

[ ]:
```
# Process noise
proc_var=0.5
proc_mat = np.zeros((2,2))
proc_mat[0, 0] = proc_var/3.0*dt**3
```

```
proc_mat[0, 1] = proc_var/2.0*dt**2
proc_mat[1, 0] = proc_var/2.0*dt**2
proc_mat[1, 1] = proc_var*dt
proc_mat_inv = np.linalg.pinv(proc_mat)
Lproc  = np.linalg.cholesky(proc_mat)

def proposal(current_state, data=None, dt=dt):
    """ Bootstrap Particle Filter the proposal is the dynamics!"""

    if current_state.ndim == 1:
        return pendulum_dyn(current_state, dt=dt) + np.dot(Lproc, np.random.
 ↪randn(2))
    else:
        nsamples = current_state.shape[0]
        return pendulum_dyn(current_state, dt=dt) + np.dot(Lproc, np.random.
 ↪randn(2, nsamples)).T

def proposal_logpdf(current, previous, data=None, proc_var=0.1):
    """ Bootstrap Particle Filter: the proposal is the dynamics"""
    nexts  = pendulum_dyn(previous, dt=dt)
    delta = nexts - current
    if current.ndim == 1:
        return -0.5 * np.dot(delta, np.dot(proc_mat_inv, delta))
    else:
        return -0.5 * np.sum(delta * np.dot(delta, proc_mat_inv.T), axis=1)


def likelihood(state, data, noise_var=noise_std**2):
    """Gaussian Likelihood through nonlinear model"""
    dpropose = observe(state)
    delta = dpropose - data
    if state.ndim == 1:
        return -0.5 * np.dot(delta, delta) / noise_var
    else:
        return -0.5 * np.sum(delta * delta, axis=1) / noise_var
```

[ ]:
```
timestep = 0.01
process_noise = 0.01 * np.array([[timestep ** 3 / 3, timestep ** 2 / 2],␣
 ↪[timestep ** 2 / 2, timestep]])
import numpy as np
from numpy.linalg import cholesky

def ukf_weights(alpha, beta, kappa, dim):
    lamda = alpha ** 2 * (dim + kappa) - dim
    wm = np.full(2 * dim + 1, 1 / (2 * (dim + lamda)))
    wc = np.full(2 * dim + 1, 1 / (2 * (dim + lamda)))
```

```
    wm[0] = lamda / (dim + lamda)
    wc[0] = lamda / (dim + lamda) + (1 - alpha ** 2 + beta)
    return wm, wc


def ukf_filter(init_mean, init_cov, measurements=None, gravity=9.81,␣
↪process_noise=process_noise, timestep=0.01, measure_noise=noise_std):
    dim = init_mean.shape[0]
    alpha = 1.0
    beta = 0.0
    kappa = 3 - dim
    wm, wc = ukf_weights(alpha, beta, kappa, dim)

    mean = init_mean.copy()
    covariance = init_cov.copy()
    #print(mean)


    # Compute the sigma points for the dynamics
    L = cholesky(covariance) * np.sqrt(dim + kappa)
    sigma_points = np.hstack([mean.reshape(-1, 1), (mean + L.T).T, (mean - L.T).
↪T])

    # Progagate through the dynamics
    sigma_points[0, :] += timestep * sigma_points[1, :]
    sigma_points[1, :] -= gravity * timestep * np.sin(sigma_points[0, :])

    # Predicted state distribution
    mean = np.dot(wm, sigma_points.T)
    covariance = np.dot(wc * (sigma_points - mean.reshape(-1, 1)),␣
↪(sigma_points - mean.reshape(-1, 1)).T) + process_noise

    # Update step if it's time to measure
    if measurements is not None:
        current_meas = measurements[0]

        # Compute the sigma points for the observation
        L = cholesky(covariance) * np.sqrt(dim + kappa)
        sigma_points = np.hstack([mean.reshape(-1, 1), (mean + L.T).T, (mean -␣
↪L.T).T])
        sigma_observations = np.sin(sigma_points[0, :])

        # Sigma points measurement mean and covariance
        predicted_mu = np.dot(wm, sigma_observations)
        predicted_cov = np.dot(wc * (sigma_observations - predicted_mu),␣
↪sigma_observations - predicted_mu) + measure_noise
```

```
        cross_cov = np.dot((sigma_points - mean.reshape(-1, 1)), wc *␣
 ↪(sigma_observations - predicted_mu))
        # Gain and update
        K = cross_cov / predicted_cov
        mean += K * (current_meas - predicted_mu)
        covariance -= predicted_cov * K.reshape(-1, 1) * K
    return mean, covariance

def ukf_proposal(current_samples, current_covs, data):
    new_samples = np.zeros(current_samples.shape)
    new_means = np.zeros(current_samples.shape)
    new_covs = np.zeros(current_covs.shape)
    for ii in range(current_samples.shape[0]):
        #print(current_samples[ii, :])
        mean, covariance = ukf_filter(current_samples[ii, :], current_covs[ii, :
 ↪, :], data)
        new_samples[ii, :] = np.random.multivariate_normal(mean, covariance)
        new_means[ii, :] = mean
        new_covs[ii, :, :] = covariance
    return new_samples, new_means, new_covs

from scipy.stats import multivariate_normal
def ukf_proposal_logpdf(current_samples, new_samples, new_means, new_covs):
    log_likelihoods = np.zeros(current_samples.shape[0])
    for ii in range(current_samples.shape[0]):
        mvn = multivariate_normal(new_means[ii, :], new_covs[ii, :, :])
        log_likelihoods = mvn.logpdf(current_samples[ii, :])
    return log_likelihoods
```

Next we setup the prior and run the algorithm

[ ]:
```
x0 = np.array([1.5, 0])
dt = 0.01
Nsteps = 500
observation_interval = 40
noise_std = 1  # Standard deviation of noise

true, times, data, data_times = simulate_pendulum(x0, dt, Nsteps,␣
 ↪observation_interval, noise_std=noise_std)
prior_mean = true[0, :] # start at the truth -- this is obviously not possible␣
 ↪in reality
prior_cov = np.eye(2) # identity covariance

samples, weights, eff =  particle_filter(data, prior_mean, prior_cov, Nsteps,
                                         proposal, proposal_logpdf,␣
 ↪likelihood, proposal_logpdf, observation_interval=observation_interval,
```

```
                                                nsamples=int(1e3),␣
 ↪resampling_threshold_frac=0.1)


means, stds = compute_mean_std(samples, weights)
fig, axs = plt.subplots(1, 2, figsize=(15, 5))
fig.suptitle('Interval: 40, R: 1, PF with Dynamics', fontsize=16,␣
 ↪fontweight='bold')

axs[0].plot(times, true[:, 0],'-r', label='True')
axs[0].plot(times, means[:, 0], '-', label='Filtered Mean')
axs[0].plot(data_times, data[:, 0], 'ko', ms=1, label='Data')
axs[0].fill_between(times, means[:, 0] - 2 * stds[:, 0], means[:, 0]+2*stds[:,␣
 ↪0], color='blue', alpha=0.1, label=r'$2\sigma$')
axs[0].set_xlabel('Time',fontsize=14)
axs[0].set_ylabel('Position', fontsize=14)
axs[0].legend(fontsize=14)

axs[1].plot(times, true[:, 1],'-r', label='True')
axs[1].plot(times, means[:, 1], '-', label='Filtered Mean')
axs[1].fill_between(times, means[:, 1] - 2 * stds[:, 1], means[:, 1]+2*stds[:,␣
 ↪1], color='blue', alpha=0.1, label=r'$2\sigma$')
axs[1].set_xlabel('Time',fontsize=14)
axs[1].set_ylabel('Velocity', fontsize=14)
```

[ ]:
```
nsamples = 1000
init_covs = np.tile(np.eye(2), (nsamples, 1, 1))


samples, weights, eff =  particle_filter(data, prior_mean, prior_cov, Nsteps,
                                         ukf_proposal, ukf_proposal_logpdf,␣
 ↪likelihood, proposal_logpdf, observation_interval=observation_interval,
                                         nsamples=nsamples,␣
 ↪resampling_threshold_frac=0.1, current_covs=init_covs)


means, stds = compute_mean_std(samples, weights)
fig, axs = plt.subplots(1, 2, figsize=(15, 5))

axs[0].plot(times, true[:, 0],'-r', label='True')
axs[0].plot(times, means[:, 0], '-', label='Filtered Mean')
axs[0].plot(data_times, data[:, 0], 'ko', ms=1, label='Data')
axs[0].fill_between(times, means[:, 0] - 2 * stds[:, 0], means[:, 0]+2*stds[:,␣
 ↪0], color='blue', alpha=0.1, label=r'$2\sigma$')
axs[0].set_xlabel('Time',fontsize=14)
axs[0].set_ylabel('Position', fontsize=14)
```

```
axs[0].legend(fontsize=14)

axs[1].plot(times, true[:, 1],'-r', label='True')
axs[1].plot(times, means[:, 1], '-', label='Filtered Mean')
axs[1].fill_between(times, means[:, 1] - 2 * stds[:, 1], means[:, 1]+2*stds[:,␣
 ↪1], color='blue', alpha=0.1, label=r'$2\sigma$')
axs[1].set_xlabel('Time',fontsize=14)
axs[1].set_ylabel('Velocity', fontsize=14)
```

[ ]:
```
x0 = np.array([1.5, 0])
dt = 0.01
Nsteps = 500
observation_interval = 10
noise_std = 0.1  # Standard deviation of noise

true, times, data, data_times = simulate_pendulum(x0, dt, Nsteps,␣
 ↪observation_interval, noise_std=noise_std)
prior_mean = true[0, :] # start at the truth -- this is obviously not possible␣
 ↪in reality
prior_cov = np.eye(2) # identity covariance

samples, weights, eff =  particle_filter(data, prior_mean, prior_cov, Nsteps,
                                         proposal, proposal_logpdf,␣
 ↪likelihood, proposal_logpdf, observation_interval=observation_interval,
                                         nsamples=int(1e3),␣
 ↪resampling_threshold_frac=0.1)


means, stds = compute_mean_std(samples, weights)
fig, axs = plt.subplots(1, 2, figsize=(15, 5))

fig.suptitle('Interval: 10, R: 0.1, PF with Dynamics', fontsize=16,␣
 ↪fontweight='bold')
axs[0].plot(times, true[:, 0],'-r', label='True')
axs[0].plot(times, means[:, 0], '-', label='Filtered Mean')
axs[0].plot(data_times, data[:, 0], 'ko', ms=1, label='Data')
axs[0].fill_between(times, means[:, 0] - 2 * stds[:, 0], means[:, 0]+2*stds[:,␣
 ↪0], color='blue', alpha=0.1, label=r'$2\sigma$')
axs[0].set_xlabel('Time',fontsize=14)
axs[0].set_ylabel('Position', fontsize=14)
axs[0].legend(fontsize=14)

axs[1].plot(times, true[:, 1],'-r', label='True')
axs[1].plot(times, means[:, 1], '-', label='Filtered Mean')
axs[1].fill_between(times, means[:, 1] - 2 * stds[:, 1], means[:, 1]+2*stds[:,␣
 ↪1], color='blue', alpha=0.1, label=r'$2\sigma$')
```

```
axs[1].set_xlabel('Time',fontsize=14)
axs[1].set_ylabel('Velocity', fontsize=14)
```

[ ]:
```
nsamples = 1000
init_covs = np.tile(np.eye(2), (nsamples, 1, 1))


samples, weights, eff =  particle_filter(data, prior_mean, prior_cov, Nsteps,
                                         ukf_proposal, ukf_proposal_logpdf,␣
 ↪likelihood, proposal_logpdf, observation_interval=observation_interval,
                                         nsamples=nsamples,␣
 ↪resampling_threshold_frac=0.1, current_covs=init_covs)


means, stds = compute_mean_std(samples, weights)
fig, axs = plt.subplots(1, 2, figsize=(15, 5))

axs[0].plot(times, true[:, 0],'-r', label='True')
axs[0].plot(times, means[:, 0], '-', label='Filtered Mean')
axs[0].plot(data_times, data[:, 0], 'ko', ms=1, label='Data')
axs[0].fill_between(times, means[:, 0] - 2 * stds[:, 0], means[:, 0]+2*stds[:,␣
 ↪0], color='blue', alpha=0.1, label=r'$2\sigma$')
axs[0].set_xlabel('Time',fontsize=14)
axs[0].set_ylabel('Position', fontsize=14)
axs[0].legend(fontsize=14)

axs[1].plot(times, true[:, 1],'-r', label='True')
axs[1].plot(times, means[:, 1], '-', label='Filtered Mean')
axs[1].fill_between(times, means[:, 1] - 2 * stds[:, 1], means[:, 1]+2*stds[:,␣
 ↪1], color='blue', alpha=0.1, label=r'$2\sigma$')
axs[1].set_xlabel('Time',fontsize=14)
axs[1].set_ylabel('Velocity', fontsize=14)
```

[ ]:
```
x0 = np.array([1.5, 0])
dt = 0.01
Nsteps = 500
observation_interval = 5
noise_std = 0.001  # Standard deviation of noise

true, times, data, data_times = simulate_pendulum(x0, dt, Nsteps,␣
 ↪observation_interval, noise_std=noise_std)
prior_mean = true[0, :] # start at the truth -- this is obviously not possible␣
 ↪in reality
prior_cov = np.eye(2) # identity covariance

samples, weights, eff =  particle_filter(data, prior_mean, prior_cov, Nsteps,
```

```
                                                proposal, proposal_logpdf,␣
 ↪likelihood, proposal_logpdf, observation_interval=observation_interval,
                                                nsamples=int(1e3),␣
 ↪resampling_threshold_frac=0.1)


means, stds = compute_mean_std(samples, weights)
fig, axs = plt.subplots(1, 2, figsize=(15, 5))

fig.suptitle('Interval: 5, R: 0.001, PF with Dynamics', fontsize=16,␣
 ↪fontweight='bold')
axs[0].plot(times, true[:, 0],'-r', label='True')
axs[0].plot(times, means[:, 0], '-', label='Filtered Mean')
axs[0].plot(data_times, data[:, 0], 'ko', ms=1, label='Data')
axs[0].fill_between(times, means[:, 0] - 2 * stds[:, 0], means[:, 0]+2*stds[:,␣
 ↪0], color='blue', alpha=0.1, label=r'$2\sigma$')
axs[0].set_xlabel('Time',fontsize=14)
axs[0].set_ylabel('Position', fontsize=14)
axs[0].legend(fontsize=14)

axs[1].plot(times, true[:, 1],'-r', label='True')
axs[1].plot(times, means[:, 1], '-', label='Filtered Mean')
axs[1].fill_between(times, means[:, 1] - 2 * stds[:, 1], means[:, 1]+2*stds[:,␣
 ↪1], color='blue', alpha=0.1, label=r'$2\sigma$')
axs[1].set_xlabel('Time',fontsize=14)
axs[1].set_ylabel('Velocity', fontsize=14)
```

[ ]:
```
nsamples = 1000
init_covs = np.tile(np.eye(2), (nsamples, 1, 1))


samples, weights, eff =  particle_filter(data, prior_mean, prior_cov, Nsteps,
                                                ukf_proposal, ukf_proposal_logpdf,␣
 ↪likelihood, proposal_logpdf, observation_interval=observation_interval,
                                                nsamples=nsamples,␣
 ↪resampling_threshold_frac=0.1, current_covs=init_covs)


means, stds = compute_mean_std(samples, weights)
fig, axs = plt.subplots(1, 2, figsize=(15, 5))

axs[0].plot(times, true[:, 0],'-r', label='True')
axs[0].plot(times, means[:, 0], '-', label='Filtered Mean')
axs[0].plot(data_times, data[:, 0], 'ko', ms=1, label='Data')
axs[0].fill_between(times, means[:, 0] - 2 * stds[:, 0], means[:, 0]+2*stds[:,␣
 ↪0], color='blue', alpha=0.1, label=r'$2\sigma$')
```

```
axs[0].set_xlabel('Time',fontsize=14)
axs[0].set_ylabel('Position', fontsize=14)
axs[0].legend(fontsize=14)

axs[1].plot(times, true[:, 1],'-r', label='True')
axs[1].plot(times, means[:, 1], '-', label='Filtered Mean')
axs[1].fill_between(times, means[:, 1] - 2 * stds[:, 1], means[:, 1]+2*stds[:,␣
 ↪1], color='blue', alpha=0.1, label=r'$2\sigma$')
axs[1].set_xlabel('Time',fontsize=14)
axs[1].set_ylabel('Velocity', fontsize=14)
```

[ ]:
```
def plot_2d(weights, samples, xlim, ylim, ax):
    """A function to plot an empirical distribution on given axes `ax`."""
    Nsamples = 3000
    s, w = resample(Nsamples, samples, weights)  # resample to obtain equal␣
 ↪weights

    xspace = np.linspace(xlim[0], xlim[1], 100)
    yspace = np.linspace(ylim[0], ylim[1], 100)
    XX, YY = np.meshgrid(xspace, yspace)

    positions = np.vstack([XX.ravel(), YY.ravel()])
    values = np.vstack([s[:, 0], s[:, 1]])
    kernel = stats.gaussian_kde(values)  # kernel density estimate to get␣
 ↪contours
    f = np.reshape(kernel(positions).T, XX.shape)
    ax.contour(XX, YY, f)
    ax.plot(s[:, 0], s[:, 1], 'x', color='grey', alpha=0.1)
    ax.set_xlabel("angle")
    ax.set_ylabel("angular rate")


from matplotlib.patches import Ellipse
import math

def plot_gaussian_ellipse(mean, cov, ax, n_std=2, **kwargs):
    """ Plot a Gaussian ellipse with n_std standard deviation. """
    if cov.shape[0] == 2:
        # Calculate the eigenvalues and eigenvectors of the covariance matrix
        vals, vecs = np.linalg.eigh(cov)
        order = vals.argsort()[::-1]
        vals, vecs = vals[order], vecs[:, order]

        # The angle to rotate our ellipse (counter-clockwise)
        theta = np.degrees(np.arctan2(*vecs[:, 0][::-1]))

        # Width and height are "full" widths, not radius
```

```
        width, height = 2 * n_std * np.sqrt(vals)
        ellipse = Ellipse(xy=mean, width=width, height=height, angle=theta,␣
 ↪**kwargs)

        ax.add_patch(ellipse)
    else:
        print("Error: Covariance matrix is not 2x2 dimensional.")
```

[ ]:
```
from numpy.linalg import cholesky
from dataclasses import dataclass
def ukf_weights(alpha, beta, kappa, dim):
    lamda = alpha ** 2 * (dim + kappa) - dim
    wm = np.full(2 * dim + 1, 1 / (2 * (dim + lamda)))
    wc = np.full(2 * dim + 1, 1 / (2 * (dim + lamda)))

    wm[0] = lamda / (dim + lamda)
    wc[0] = lamda / (dim + lamda) + (1 - alpha ** 2 + beta)
    return wm, wc


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

def ukf_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))

    alpha = 1.0
    beta = 0.0
    kappa = 3 - dim
    wm, wc = ukf_weights(alpha, beta, kappa, dim)

    mean = init_mean.copy()
    covariance = init_cov.copy()
    measurement_index = 0

    for step in range(steps):
        # Compute the sigma points for the dynamics
        L = cholesky(covariance) * np.sqrt(dim + kappa)
        sigma_points = np.hstack([mean.reshape(-1, 1), (mean + L.T).T, (mean -␣
 ↪L.T).T])
```

```
        # Progagate through the dynamics
        sigma_points[0, :] += timestep * sigma_points[1, :]
        sigma_points[1, :] -= gravity * timestep * np.sin(sigma_points[0, :])

        # Predicted state distribution
        mean = np.dot(wm, sigma_points.T)
        covariance = np.dot(wc * (sigma_points - mean.reshape(-1, 1)),␣
↪(sigma_points - mean.reshape(-1, 1)).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

            # Compute the sigma points for the observation
            L = cholesky(covariance) * np.sqrt(dim + kappa)
            sigma_points = np.hstack([mean.reshape(-1, 1), (mean + L.T).T,␣
↪(mean - L.T).T])
            sigma_observations = np.sin(sigma_points[0, :])

            # Sigma points measurement mean and covariance
            predicted_mu = np.dot(wm, sigma_observations)
            predicted_cov = np.dot(wc * (sigma_observations - predicted_mu),␣
↪sigma_observations - predicted_mu) + measure_noise
            cross_cov = np.dot((sigma_points - mean.reshape(-1, 1)), wc *␣
↪(sigma_observations - predicted_mu))

            # Gain and update
            K = cross_cov / predicted_cov
            mean += K * (current_meas - predicted_mu)
            covariance -= predicted_cov * K.reshape(-1, 1) * K

        # Store the predicted/updated mean and covariance
        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
```

[ ]:
```
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
```

[ ]:
```
from itertools import product

def gh_kalman_filter(init_mean, init_cov, gravity, process_noise, timestep,␣
↪measure_noise, measurements, steps, delta=10, order=5):
    dim = init_mean.shape[0]
    sp1D, weights1D = gh_oned(order)
    weights1D /= weights1D.sum()

    sigma_points = np.array(list(product(*(sp1D for _ in range(dim))))).T
```

```
    weights = np.array(list(product(*(weights1D for _ in range(dim))))).
↪prod(axis=1)

    num_steps = steps # Total number of steps to simulate based on measurement␣
↪frequency
    filtered_means = np.empty((num_steps, dim))
    filtered_covariances = np.empty((num_steps, dim, dim))
    filtered_stds = np.empty((num_steps, dim))

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

    for step in range(num_steps):
        if step % delta == 0 and measurement_index < len(measurements):
            current_meas = measurements[measurement_index]
            measurement_index += 1
        else:
            current_meas = None

        # Prediction step
        trans_sigma_points = rotate_points(sigma_points.T, mean.reshape(-1, 1).
↪squeeze(), covariance, alg="chol").T
        trans_sigma_points[0, :] += timestep * trans_sigma_points[1, :]
        trans_sigma_points[1, :] -= gravity * timestep * np.
↪sin(trans_sigma_points[0, :])

        mean = np.dot(trans_sigma_points, weights)
        covariance = np.dot(weights.reshape(1, -1) * (trans_sigma_points - mean.
↪reshape(-1, 1)),
                            (trans_sigma_points - mean.reshape(-1, 1)).T) +␣
↪process_noise

        # Update step if current_meas is available
        if current_meas is not None:
            trans_sigma_points = rotate_points(sigma_points.T, mean.reshape(-1,␣
↪1).squeeze(), covariance, alg="chol").T
            meas_predictions = np.sin(trans_sigma_points[0, :])
            predicted_mean = np.dot(meas_predictions, weights)
            predicted_cov = np.dot(weights * (meas_predictions -␣
↪predicted_mean),
                                   (meas_predictions - predicted_mean)) +␣
↪measure_noise
            cross_cov = np.dot((trans_sigma_points - mean.reshape(-1, 1)),␣
↪weights * (meas_predictions - predicted_mean))
```

```
            kalman_gain = cross_cov / predicted_cov
            mean += kalman_gain * (current_meas - predicted_mean)
            covariance -= predicted_cov * kalman_gain[:, np.newaxis] *␣
 ↪kalman_gain

        # Store the predicted/updated mean and covariance
        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
```

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt

x0 = np.array([1.5, 0])
dt = 0.01
Nsteps = 501
observation_interval = 5
noise_std = 0.001

true, times, data, data_times = simulate_pendulum(x0, dt, Nsteps,␣
 ↪observation_interval, noise_std=noise_std)
ukf = ukf_filter(x0, np.eye(2), 9.81, process_noise, timestep, noise_std, data,␣
 ↪Nsteps, observation_interval)

prior_mean = true[0, :]
prior_cov = np.eye(2)

samples, weights, eff = particle_filter(data, prior_mean, prior_cov, Nsteps,␣
 ↪proposal, proposal_logpdf, likelihood, proposal_logpdf,␣
 ↪observation_interval=observation_interval, nsamples=int(1e6),␣
 ↪resampling_threshold_frac=0.1)

means, stds = compute_mean_std(samples, weights)
xlim = [-3, 3]
ylim = [-6, 6]
N, d = means.shape

rows = 2
cols = 3
fig, axs = plt.subplots(rows, cols, figsize=(15, 10))

idx = 0
for ii in range(N):
    if ii % 100 == 0:
```

```
        ax = axs[idx // cols, idx % cols]
        plot_2d(weights[:, ii], samples[:, :, ii], xlim, ylim, ax)
        ax.set_title(f"Time = {times[ii]:.2f}")
        mean = ukf.means[ii]
        covariance = ukf.covs[ii]
        plot_gaussian_ellipse(mean, covariance, ax, n_std=2, edgecolor='red',␣
 ↪facecolor='none', linewidth=1.2, zorder=3)
        plot_gaussian_ellipse(mean, covariance, ax, n_std=10, edgecolor='red',␣
 ↪facecolor='none', linewidth=2, zorder=3)
        idx += 1


fig.suptitle('Particle Filter Snapshots at Various Times with Overlaid UKF␣
 ↪Gaussian in Red. Sampling Interval: 5, Noise R: 0.001', fontsize=16)
plt.tight_layout()
plt.show()
```

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt

x0 = np.array([1.5, 0])
dt = 0.01
Nsteps = 501
observation_interval = 10
noise_std = 0.1

true, times, data, data_times = simulate_pendulum(x0, dt, Nsteps,␣
 ↪observation_interval, noise_std=noise_std)
ukf = ukf_filter(x0, np.eye(2), 9.81, process_noise, timestep, noise_std, data,␣
 ↪Nsteps, observation_interval)

prior_mean = true[0, :]
prior_cov = np.eye(2)

samples, weights, eff = particle_filter(data, prior_mean, prior_cov, Nsteps,␣
 ↪proposal, proposal_logpdf, likelihood, proposal_logpdf,␣
 ↪observation_interval=observation_interval, nsamples=int(1e6),␣
 ↪resampling_threshold_frac=0.1)

means, stds = compute_mean_std(samples, weights)
xlim = [-3, 3]
ylim = [-6, 6]
N, d = means.shape

rows = 2
cols = 3
```

```
fig, axs = plt.subplots(rows, cols, figsize=(15, 10))

idx = 0
for ii in range(N):
    if ii % 100 == 0:
        ax = axs[idx // cols, idx % cols]
        plot_2d(weights[:, ii], samples[:, :, ii], xlim, ylim, ax)
        ax.set_title(f"Time = {times[ii]:.2f}")
        mean = ukf.means[ii]
        covariance = ukf.covs[ii]
        plot_gaussian_ellipse(mean, covariance, ax, n_std=2, edgecolor='red',␣
 ↪facecolor='none', linewidth=2, zorder=3)
        idx += 1


fig.suptitle('Particle Filter Snapshots at Various Times with Overlaid UKF␣
 ↪Gaussian in Red. Sampling Interval: 10, Noise R: 0.01', fontsize=16)
plt.tight_layout()
plt.show()
```

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt

x0 = np.array([1.5, 0])
dt = 0.01
Nsteps = 501
observation_interval = 40
noise_std = 1

true, times, data, data_times = simulate_pendulum(x0, dt, Nsteps,␣
 ↪observation_interval, noise_std=noise_std)
ukf = ukf_filter(x0, np.eye(2), 9.81, process_noise, timestep, noise_std, data,␣
 ↪Nsteps, observation_interval)
ekf = ekf_filter(x0, np.eye(2), 9.81, process_noise, timestep, noise_std, data,␣
 ↪Nsteps, observation_interval)
gkfthree = gh_kalman_filter(x0, np.eye(2), 9.81, process_noise, timestep,␣
 ↪noise_std, data, Nsteps, observation_interval, order=3)
gkffive = gh_kalman_filter(x0, np.eye(2), 9.81, process_noise, timestep,␣
 ↪noise_std, data, Nsteps, observation_interval, order=5)

prior_mean = true[0, :]
prior_cov = np.eye(2)
```

```
samples, weights, eff = particle_filter(data, prior_mean, prior_cov, Nsteps,␣
 ↪proposal, proposal_logpdf, likelihood, proposal_logpdf,␣
 ↪observation_interval=observation_interval, nsamples=int(1e6),␣
 ↪resampling_threshold_frac=0.1)

means, stds = compute_mean_std(samples, weights)
xlim = [-3, 3]
ylim = [-6, 6]
N, d = means.shape

rows = 2
cols = 3
fig, axs = plt.subplots(rows, cols, figsize=(15, 10))

idx = 0
for ii in range(N):
    if ii % 100 == 0:
        ax = axs[idx // cols, idx % cols]
        plot_2d(weights[:, ii], samples[:, :, ii], xlim, ylim, ax)
        ax.set_title(f"Time = {times[ii]:.2f}")
        mean = ukf.means[ii]
        covariance = ukf.covs[ii]
        plot_gaussian_ellipse(mean, covariance, ax, n_std=2, edgecolor='red',␣
 ↪facecolor='none', linewidth=2, zorder=3)
        idx += 1
    if ii == 200:
        save_weights = weights[:, ii]
        save_samples = samples[:, :, ii]
        save_mean = np.mean(save_samples, axis=0)
        save_covariance = np.cov(save_samples.T)
        gkftmean = gkfthree.means[ii]
        gkftcov = gkfthree.covs[ii]
        gkffmean = gkffive.means[ii]
        gkffcov = gkffive.covs[ii]
        ekfmean = ekf.means[ii]
        ekfcov = ekf.covs[ii]
        ukfmean = ukf.means[ii]
        ukfcov = ukf.covs[ii]


fig.suptitle('Particle Filter Snapshots at Various Times with Overlaid UKF␣
 ↪Gaussian in Red. Sampling Interval: 40, Noise R: 1', fontsize=16)
plt.tight_layout()
plt.show()
```

[ ]:
```
# Display the computed values
mean = save_mean
covariance = save_covariance
print("Computed Mean:", mean)
print("Computed Covariance:\n", covariance)
print("GKF (3-state) Mean:", gkftmean)
print("GKF (3-state) Covariance:\n", gkftcov)
print("GKF (5-state) Mean:", gkffmean)
print("GKF (5-state) Covariance:\n", gkffcov)
print("EKF Mean:", ekfmean)
print("EKF Covariance:\n", ekfcov)
print("UKF Mean:", ukfmean)
print("UKF Covariance:\n", ukfcov)

# Calculate the errors
filters = ["GKF (3-state)", "GKF (5-state)", "EKF", "UKF"]
means = [gkftmean, gkffmean, ekfmean, ukfmean]
covariances = [gkftcov, gkffcov, ekfcov, ukfcov]
mean_errors = [np.linalg.norm(mean - m) for m in means]
cov_errors = [np.linalg.norm(covariance - c, 'fro') for c in covariances]

# Find which filter comes closest
min_mean_error_idx = np.argmin(mean_errors)
min_cov_error_idx = np.argmin(cov_errors)

# Print the numerical rationale
print("\nClosest filters by mean error:")
for i, error in enumerate(mean_errors):
    print(f"{filters[i]} Mean Error: {error}")
print(f"Closest by mean: {filters[min_mean_error_idx]}")

print("\nClosest filters by covariance error:")
for i, error in enumerate(cov_errors):
    print(f"{filters[i]} Covariance Error: {error}")
print(f"Closest by covariance: {filters[min_cov_error_idx]}")

# Plotting all points and ellipses
fig, ax = plt.subplots(figsize=(10, 8))
plot_2d(save_weights, save_samples, xlim, ylim, ax)
plot_gaussian_ellipse(mean, covariance, ax, color='black', label='True␣
 ↪Posterior')
plot_gaussian_ellipse(gkftmean, gkftcov, ax, color='red', label='GKF (3-state)')
plot_gaussian_ellipse(gkffmean, gkffcov, ax, color='green', label='GKF␣
 ↪(5-state)')
plot_gaussian_ellipse(ekfmean, ekfcov, ax, color='blue', label='EKF')
plot_gaussian_ellipse(ukfmean, ukfcov, ax, color='purple', label='UKF')
ax.legend()
```

```
ax.set_title("Comparison of Gaussian Filter Estimates")
plt.show()
```

[ ]:
```
x0 = np.array([1.5, 0])
dt = 0.01
Nsteps = 500
observation_interval = 10
noise_std = 1  # Standard deviation of noise

# Simulate pendulum and get data
true, times, data, data_times = simulate_pendulum(x0, dt, Nsteps,␣
 ↪observation_interval, noise_std=noise_std)
prior_mean = true[0, :]  # Start at the truth -- this is not realistic
prior_cov = np.eye(2)  # Identity covariance

fig, axs = plt.subplots(1, 2, figsize=(15, 5))
fig.suptitle('Interval: 40, R: 1, PF with Dynamics and various sample nums',␣
 ↪fontsize=16, fontweight='bold')

nsamples_list = [2, 10, 1000]  # Different numbers of particles

for nsamples in nsamples_list:
    samples, weights, eff = particle_filter(data, prior_mean, prior_cov, Nsteps,
                                            proposal, proposal_logpdf,␣
 ↪likelihood, proposal_logpdf,
                                           ␣
 ↪observation_interval=observation_interval,
                                            nsamples=nsamples,␣
 ↪resampling_threshold_frac=0.1)
    means, stds = compute_mean_std(samples, weights)

    # Plot for Position
    axs[0].plot(times, means[:, 0], '-', label=f'{nsamples} Particles')
    axs[0].plot(data_times, data[:, 0], 'ko', ms=1)  # Plot data points

    # Plot for Velocity
    axs[1].plot(times, means[:, 1], '-', label=f'{nsamples} Particles')

# Set labels and titles for Position
axs[0].plot(times, true[:, 0], '-r', label='True')  # True trajectory for␣
 ↪reference
axs[0].set_xlabel('Time', fontsize=14)
axs[0].set_ylabel('Position', fontsize=14)

# Set labels and titles for Velocity
```

```
axs[1].plot(times, true[:, 1], '-r', label='True')  # True trajectory for␣
 ↪reference
axs[1].set_xlabel('Time', fontsize=14)
axs[1].set_ylabel('Velocity', fontsize=14)

plt.show()
```

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt

x0 = np.array([1.5, 0])
dt = 0.01
Nsteps = 500
observation_interval = 10
noise_std = 0.1  # Standard deviation of noise

# Simulate pendulum data
true, times, data, data_times = simulate_pendulum(x0, dt, Nsteps,␣
 ↪observation_interval, noise_std=noise_std)
prior_mean = true[0, :]
prior_cov = np.eye(2)

# Run particle filter with 1e5 samples for reference
reference_samples, reference_weights, reference_eff = particle_filter(data,␣
 ↪prior_mean, prior_cov, Nsteps,
                                                                        ␣
 ↪proposal, proposal_logpdf, likelihood, proposal_logpdf,
                                                                        ␣
 ↪observation_interval=observation_interval,
                                                                        ␣
 ↪nsamples=int(1e3), resampling_threshold_frac=0.1)
reference_means, reference_stds = compute_mean_std(reference_samples,␣
 ↪reference_weights)

# Array of sample numbers to test
nsamples_list = [2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30, 35, 40, 45, 50,␣
 ↪75, 100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600, 650, 700, 750,␣
 ↪800, 850, 900, 950, 1000, 2000, 5000, 10000]

mean_errors = []
std_means = []

for nsamples in nsamples_list:
    samples, weights, eff = particle_filter(data, prior_mean, prior_cov, Nsteps,
                                            proposal, proposal_logpdf,␣
 ↪likelihood, proposal_logpdf,
```

```
                                                    ␣
 ↪observation_interval=observation_interval,
                                                     nsamples=nsamples,␣
 ↪resampling_threshold_frac=0.1)
    means, stds = compute_mean_std(samples, weights)

    # Compute mean error for each state compared to reference
    mean_error = np.mean(np.abs(means - reference_means), axis=0)
    mean_errors.append(mean_error)

    # Compute mean standard deviations compared to reference
    std_mean = np.mean(stds, axis=0)
    print(std_mean)
    std_means.append(std_mean)

mean_errors = np.array(mean_errors)
std_means = np.array(std_means)

fig, axs = plt.subplots(1, 2, figsize=(15, 10))
fig.suptitle('Error and Standard Deviation Evolution with Sample Numbers -␣
 ↪Dynamics', fontsize=16, fontweight='bold')

axs[0].plot(nsamples_list, mean_errors[:, 0], label='Position Error')
axs[1].plot(nsamples_list, mean_errors[:, 1], label='Velocity Error')
axs[0].set_xlabel('Number of Samples')
axs[0].set_ylabel('Mean Error (Position)')
axs[1].set_xlabel('Number of Samples')
axs[1].set_ylabel('Mean Error (Velocity)')
axs[0].legend()
axs[1].legend()

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

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt

x0 = np.array([1.5, 0])
dt = 0.01
Nsteps = 500
observation_interval = 10
noise_std = 0.01  # Standard deviation of noise

# Simulate pendulum data
true, times, data, data_times = simulate_pendulum(x0, dt, Nsteps,␣
 ↪observation_interval, noise_std=noise_std)
prior_mean = true[0, :]
```

```
prior_cov = np.eye(2)

# Run particle filter with 1e5 samples for reference
reference_samples, reference_weights, reference_eff = particle_filter(data,␣
 ↪prior_mean, prior_cov, Nsteps,
                                                                        ␣
 ↪proposal, proposal_logpdf, likelihood, proposal_logpdf,
                                                                        ␣
 ↪observation_interval=observation_interval,
                                                                        ␣
 ↪nsamples=int(1e5), resampling_threshold_frac=0.1)
reference_means, reference_stds = compute_mean_std(reference_samples,␣
 ↪reference_weights)

# Array of sample numbers to test
nsamples_list = [2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30, 35, 40, 45, 50,␣
 ↪75, 100, 150, 200, 250, 300, 350, 400, 450, 500]

mean_errors = []
std_means = []

for nsamples in nsamples_list:
    samples, weights, eff = particle_filter(data, prior_mean, prior_cov, Nsteps,
                                            proposal, proposal_logpdf,␣
 ↪likelihood, proposal_logpdf,
                                           ␣
 ↪observation_interval=observation_interval,
                                            nsamples=nsamples,␣
 ↪resampling_threshold_frac=0.1)
    means, stds = compute_mean_std(samples, weights)

    # Compute mean error for each state compared to reference
    mean_error = np.mean(np.abs(means - reference_means), axis=0)
    mean_errors.append(mean_error)

    # Compute mean standard deviations compared to reference
    std_mean = np.mean(stds, axis=0)
    print(std_mean)
    std_means.append(std_mean)

mean_errors = np.array(mean_errors)
std_means = np.array(std_means)

fig, axs = plt.subplots(1, 2, figsize=(15, 10))
fig.suptitle('Error and Standard Deviation Evolution with Sample Numbers -␣
 ↪UKF', fontsize=16, fontweight='bold')
```

```
axs[0].plot(nsamples_list, mean_errors[:, 0], label='Position Error')
axs[1].plot(nsamples_list, mean_errors[:, 1], label='Velocity Error')
axs[0].set_xlabel('Number of Samples')
axs[0].set_ylabel('Mean Error (Position)')
axs[1].set_xlabel('Number of Samples')
axs[1].set_ylabel('Mean Error (Velocity)')
axs[0].legend()
axs[1].legend()

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