Particle Filter and Differentiable Particle Filter

November 20, 2025

Abstract

This guide provides a practical, hands-on approach to understanding particle filters (PF) and differentiable particle filters (DPF). We follow the philosophy of "getting your hands dirty first" – building intuition through implementation before diving into theoretical proofs. The guide covers the mathematical foundations, algorithmic details, and complete Python implementations of both standard and differentiable particle filters.

Contents

1 Introduction

This guide provides a practical approach to understanding particle filters and their differentiable variants. The key insight is to start with implementation and visualization before exploring the underlying theory.

1.1 Learning Path

1. Start with basic particle filter implementation
2. Understand the mathematical framework
3. Transition to differentiable versions
4. Apply to learning problems with gradient descent

2 Prerequisites

2.1 Required Knowledge

- Basic probability theory (Bayes' rule, conditional distributions)
- Python programming
- Basic linear algebra
- Familiarity with NumPy and PyTorch

2.2 Software Requirements

The following Python packages are required:

```
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
```

3 Mathematical Foundation

3.1 State Space Model (SSM)

The generative model assumes a Hidden Markov Model (HMM) structure with the following components:

State Space Model

State Transition:

$$x_t = f(x_{t-1}) + q_t, \quad q_t \sim \mathcal{N}(0, Q) \tag{1}$$

Observation:

$$y_t = g(x_t) + r_t, \quad r_t \sim \mathcal{N}(0, R) \tag{2}$$

3.1.1 Toy Example Specification

In our illustrative example:

- **State transition:** $x_t = 0.9x_{t-1} + \epsilon_t$, where $\epsilon_t \sim \mathcal{N}(0, 0.5^2)$
- **Observation:** $y_t = x_t + \eta_t$, where $\eta_t \sim \mathcal{N}(0, 0.3^2)$

Goal: Estimate the posterior distribution $p(x_t \mid y_{1:t})$

3.2 Bayesian Recursion

The optimal Bayesian filtering solution follows a two-step recursion:

Bayesian Filter

Prediction Step:

$$p(x_t \mid y_{1:t-1}) = \int p(x_t \mid x_{t-1})p(x_{t-1} \mid y_{1:t-1})\, dx_{t-1} \tag{3}$$

Update Step:

$$p(x_t \mid y_{1:t}) \propto p(y_t \mid x_t)p(x_t \mid y_{1:t-1}) \tag{4}$$

Remark 3.1. This integral is intractable for nonlinear functions f and g. The particle filter provides a Monte Carlo approximation to this recursion.

4 The Kalman Filter: Optimality and Stability

While Particle Filters (PF) provide a generalized solution for non-linear non-Gaussian systems [cite: 152], the Kalman Filter (KF) remains the optimal Minimum Mean Square Error (MMSE) estimator for Linear-Gaussian State Space Models (LGSSM)[cite: 196]. However, in practical financial engineering applications involving high-dimensional data or single-precision arithmetic (e.g., GPU computing), standard KF implementations often suffer from numerical instability.

This section outlines the theoretical foundation of the LGSSM and derives the **Joseph stabilized form**, which guarantees the preservation of the covariance matrix's positive definiteness.

4.1 Linear-Gaussian State Space Model

We consider the classic Linear-Gaussian model as described in Example 2 of Doucet and Johansen (2011). The system is defined by the following stochastic difference equations:

LGSSM Definition

State Evolution:

$$x_t = Fx_{t-1} + w_t, \quad w_t \sim \mathcal{N}(0, Q) \tag{5}$$

Observation:

$$y_t = Hx_t + v_t, \quad v_t \sim \mathcal{N}(0, R) \tag{6}$$

where $x_t \in \mathbb{R}^{n_x}$ is the hidden state, $y_t \in \mathbb{R}^{n_y}$ is the observation, and F, H, Q, R are matrices of appropriate dimensions[cite: 193, 194]. The noise terms w_t and v_t are assumed to be uncorrelated Gaussian white noise sequences.

4.2 The Standard Kalman Recursion

The recursive solution for the posterior density $p(x_t|y_{1:t}) = \mathcal{N}(x_t; \hat{x}_{t|t}, P_{t|t})$ is given by the standard Kalman Filter equations[cite: 248].

1. Prediction (Time Update):

$$\hat{x}_{t|t-1} = F\hat{x}_{t-1|t-1} \tag{7}$$

$$P_{t|t-1} = FP_{t-1|t-1}F^T + Q \tag{8}$$

2. Correction (Measurement Update): The optimal Kalman Gain K_t minimizes the trace of the posterior covariance $P_{t|t}$:

$$K_t = P_{t|t-1}H^T S_t^{-1} \tag{9}$$

where $S_t = HP_{t|t-1}H^T + R$ is the innovation covariance. The state estimate is updated as:

$$\hat{x}_{t|t} = \hat{x}_{t|t-1} + K_t(y_t - H\hat{x}_{t|t-1}) \tag{10}$$

4.3 Numerical Instability and the Joseph Form

4.3.1 The Problem: Loss of Positive Definiteness

The standard update equation for the error covariance matrix is derived as:

$$P_{t|t} = (I - K_t H)P_{t|t-1} \tag{11}$$

Mathematically, Eq. (11) is correct. However, numerically, it is prone to instability.

1. **Asymmetry:** Due to floating-point round-off errors, the computation of $(I - K_t H)P_{t|t-1}$ may result in a non-symmetric matrix, violating the property that covariance matrices must be symmetric.

2. **Indefiniteness:** Since Eq. (11) involves subtraction (implicitly within the $I - KH$ term), numerical errors can lead to $P_{t|t}$ having negative eigenvalues, rendering it non-positive definite. This causes the filter to diverge immediately.

4.3.2 The Solution: Joseph Stabilized Update

To address this, we employ the **Joseph form** (also known as the symmetric update formula). It provides a numerically robust way to compute $P_{t|t}$:

Joseph Stabilized Update

$$P_{t|t} = (I - K_t H)P_{t|t-1}(I - K_t H)^T + K_t R K_t^T \tag{12}$$

Theoretical Justification: Observe that Eq. (12) is the sum of two terms:

- The first term $(I - K_t H) P_{t|t-1} (I - K_t H)^T$ is a quadratic form. Since $P_{t|t-1}$ is positive definite (PD), this term is guaranteed to be positive semi-definite (PSD).
- The second term $K_t R K_t^T$ is also a quadratic form involving the covariance R, guaranteeing it is PSD.

The sum of two PSD matrices is always PSD. Therefore, the Joseph form structurally guarantees the symmetry and positive semi-definiteness of the updated covariance, making it robust against round-off errors.

4.4 Stability Diagnostics: The Condition Number

To monitor the numerical health of the filter during execution, we analyze the **Condition Number** $\kappa(P_t)$ of the covariance matrix.

Definition 4.1 (Condition Number)**.** For a symmetric positive definite covariance matrix P, the condition number is the ratio of its largest eigenvalue to its smallest eigenvalue:

$$\kappa(P) = \frac{|\lambda_{\max}(P)|}{|\lambda_{\min}(P)|} \tag{13}$$

Interpretation:

- **Well-conditioned ($\kappa \approx 1$):** The error distribution is spherical. Matrix inversion (required for S_t^{-1}) is numerically stable.
- **Ill-conditioned ($\kappa \gg 1$):** The uncertainty ellipsoid is extremely elongated (high certainty in some directions, high uncertainty in others). This leads to a loss of precision during the inversion of S_t, potentially causing the filter to "explode."

In our TensorFlow implementation (Section 7), we explicitly log $\kappa(P_t)$ at each step. A diverging condition number (e.g., $> 10^{15}$ for float64) serves as an early warning signal for numerical instability.

5 Standard Particle Filter

5.1 Core Idea

Approximate the posterior distribution with weighted samples (particles):

$$p(x_t \mid y_{1:t}) \approx \sum_{i=1}^{N} w_t^{(i)} \delta(x_t - x_t^{(i)}) \tag{14}$$

where:

- $x_t^{(i)}$: particle i at time t
- $w_t^{(i)}$: normalized weight of particle i
- N: total number of particles
- $\delta(\cdot)$: Dirac delta function

5.2 Algorithm

Particle Filter Algorithm

Initialization: For $i = 1, \dots, N$

- Sample $x_0^{(i)} \sim p(x_0)$
- Set $w_0^{(i)} = 1/N$

For each time step $t = 1, 2, \dots, T$:

1. **Prediction:** For $i = 1, \dots, N$

$$x_t^{(i)} \sim p(x_t \mid x_{t-1}^{(i)}) = \mathcal{N}(f(x_{t-1}^{(i)}), Q) \tag{15}$$

2. **Weight Update:** For $i = 1, \dots, N$

$$\tilde{w}_t^{(i)} = w_{t-1}^{(i)} \cdot p(y_t \mid x_t^{(i)}) \tag{16}$$

where the likelihood is:

$$p(y_t \mid x_t^{(i)}) = \frac{1}{\sqrt{2\pi R}} \exp\left(-\frac{(y_t - g(x_t^{(i)}))^2}{2R}\right) \tag{17}$$

3. **Normalization:** For $i = 1, \dots, N$

$$w_t^{(i)} = \frac{\tilde{w}_t^{(i)}}{\sum_{j=1}^{N} \tilde{w}_t^{(j)}} \tag{18}$$

4. **State Estimation:**

$$\hat{x}_t = \sum_{i=1}^{N} w_t^{(i)} x_t^{(i)} \tag{19}$$

5. **Resampling:** For $i = 1, \dots, N$
   - Draw $a_i \sim \text{Categorical}(w_t^{(1)}, \dots, w_t^{(N)})$
   - Set $x_t^{(i)} \leftarrow x_t^{(a_i)}$
   - Reset $w_t^{(i)} \leftarrow 1/N$

5.3 Why Resampling?

Remark 5.1. Without resampling, most particles will have negligible weights after several iterations – a phenomenon known as **particle degeneracy**. Resampling concentrates computational resources (particles) in high-probability regions of the state space.

6 Differentiable Particle Filter

6.1 Motivation

Problem with Standard PF:

- The resampling step uses discrete sampling: $a_i \sim \text{Categorical}(w_t)$
- This operation is **non-differentiable**
- Cannot compute gradients $\frac{\partial \mathcal{L}}{\partial \theta}$
- Cannot use gradient descent to learn parameters $\theta = \{f_\theta, g_\theta, Q_\theta, R_\theta\}$

Solution: Replace discrete resampling with a **soft, differentiable approximation**.

6.2 Key Modification: Soft Resampling via Gumbel-Softmax

Replace categorical sampling with the **Gumbel-Softmax** trick:

$$\tilde{w}_t^{(i)} = \frac{\exp\left((\log w_t^{(i)} + g^{(i)})/\tau\right)}{\sum_{j=1}^{N} \exp\left((\log w_t^{(j)} + g^{(j)})/\tau\right)} \tag{20}$$

where:

- $g^{(i)} \sim \text{Gumbel}(0, 1)$ are i.i.d. Gumbel random variables
- $\tau > 0$ is the temperature parameter
- As $\tau \to 0$: recovers hard (categorical) resampling
- For $\tau > 0$: the operation becomes differentiable

Gumbel Sampling in Code:

```
        g = -torch.log(-torch.log(torch.rand_like(weights) +
           eps) + eps)
```

6.3 Gradient Flow and End-to-End Learning

Since all operations are now differentiable, we can compute:

$$\frac{\partial \mathcal{L}}{\partial \theta} \tag{21}$$

via backpropagation, where:

- θ: learnable parameters in f_θ, g_θ, Q_θ, R_θ
- $\mathcal{L} = \sum_{t=1}^{T} \|y_t - \hat{y}_t\|_2^2$ or other task-specific loss

This enables **end-to-end learning** of latent dynamics and observation models.

7 Implementation under TensorFlow Framework

We implement the filter using low-level tensor operations under the TensorFlow framework.

7.1 The KalmanFilterTF Class

```
import tensorflow as tf
import numpy as np

class KalmanFilterTF:
"""
TensorFlow implementation of Kalman Filter with Joseph Stabilized Update.
"""
def __init__(self, F, H, Q, R, x_init, P_init):
# Ensure all inputs are float32 tensors
self.F = tf.cast(F, dtype=tf.float32)
self.H = tf.cast(H, dtype=tf.float32)
self.Q = tf.cast(Q, dtype=tf.float32)
self.R = tf.cast(R, dtype=tf.float32)
self.x = tf.reshape(tf.cast(x_init, dtype=tf.float32), (-1, 1))
self.P = tf.cast(P_init, dtype=tf.float32)

def predict(self):
"""Time Update: x = Fx, P = FPF' + Q"""
self.x = tf.matmul(self.F, self.x)
fp = tf.matmul(self.F, self.P)
self.P = tf.matmul(fp, self.F, transpose_b=True) + self.Q
return self.x, self.P

def update(self, z_meas):
"""
Measurement Update using Joseph Form for Stability.
"""
z_meas = tf.reshape(tf.cast(z_meas, dtype=tf.float32), (-1, 1))

# 1. Innovation
z_pred = tf.matmul(self.H, self.x)
y_residual = z_meas - z_pred

# 2. Innovation Covariance S = HPH' + R
hp = tf.matmul(self.H, self.P)
S = tf.matmul(hp, self.H, transpose_b=True) + self.R

# 3. Kalman Gain K = PH'S^{-1}
# Using cholesky solve is preferred for stability if S is positive definite
pht = tf.matmul(self.P, self.H, transpose_b=True)
```

```python
    K = tf.matmul(pht, tf.linalg.inv(S))

    # 4. State Update
    self.x = self.x + tf.matmul(K, y_residual)

    # 5. Joseph Stabilized Covariance Update
    # P = (I-KH)P(I-KH)' + KRK'
    dim_x = tf.shape(self.P)[0]
    I = tf.eye(dim_x, dtype=tf.float32)
    I_KH = I - tf.matmul(K, self.H)

    p_term = tf.matmul(tf.matmul(I_KH, self.P), I_KH,
       transpose_b=True)
    r_term = tf.matmul(tf.matmul(K, self.R), K,
       transpose_b=True)
    self.P = p_term + r_term

    return self.x, self.P
```

Listing 1: Multidimensional Kalman Filter in TensorFlow

7.2 Standard Particle Filter in TensorFlow

Here we implement the standard Particle Filter. Note that resampling (indexing) is non-differentiable, which motivates the Differentiable PF in the next section.

```python
def standard_particle_filter_tf(observations,
   n_particles=1000):
    T = len(observations)
    # Initialize particles (TensorFlow)
    particles = tf.random.normal((n_particles,), stddev=1.0)
    weights = tf.ones((n_particles,)) / n_particles

    estimates = []

    for t in range(T):
    # 1. Prediction (Transition)
    # x_t = 0.9 * x_{t-1} + noise
    noise = tf.random.normal((n_particles,), stddev=0.5)
    particles = 0.9 * particles + noise

    # 2. Weight Update (Likelihood)
    # y_t = x_t + noise
    obs = observations[t]
    likelihood = tf.exp(-0.5 * ((obs - particles) / 0.3)**2)
    weights *= likelihood
    weights /= tf.reduce_sum(weights) + 1e-9

    # 3. Estimation
    est = tf.reduce_sum(particles * weights)
    estimates.append(est)
```

```
    # 4. Multinomial Resampling (Non-differentiable)
    # We use tf.random.categorical for resampling indices
    logits = tf.math.log(weights + 1e-9)
    indices = tf.random.categorical(tf.reshape(logits, (1,
        -1)), n_particles)
    indices = tf.reshape(indices, (-1,))

    particles = tf.gather(particles, indices)
    weights = tf.ones((n_particles,)) / n_particles

    return tf.stack(estimates)
```

Listing 2: Standard Particle Filter (TensorFlow)

7.3 Analysis: Numerical Stability via Condition Number

A key metric for the stability of the Kalman Filter is the **Condition Number** of the covariance matrix P. It is defined as the ratio of the largest to smallest eigenvalue:

$$\kappa(P) = \frac{|\lambda_{\max}(P)|}{|\lambda_{\min}(P)|} \tag{22}$$

In our implementation, we monitor $\kappa(P)$ at each step. A diverging condition number (e.g., $> 10^{15}$ for float64) indicates that the matrix is becoming singular, which leads to severe numerical errors in calculating the Kalman Gain. The Joseph form update helps maintain a healthy condition number by preserving symmetry.

8 Advanced Topics

8.1 Particle Degeneracy

Problem: After several iterations, most particle weights become negligible.

Solutions:

- Standard resampling (for classical PF)
- Soft resampling (for DPF)
- Adaptive number of particles
- Regularization techniques
- Monitoring effective sample size: $\text{ESS} = 1/\sum_{i=1}^{N}(w_t^{(i)})^2$

8.2 Nonlinear Dynamics

Extend to nonlinear systems:

```
# Example: Sine transition
def nonlinear_transition(x, noise_std=0.5):
return torch.sin(x) + torch.randn_like(x) * noise_std

# Use in forward pass
particles = nonlinear_transition(particles)
```

8.3 Comparison with Other Methods

Table 1: Comparison of Filtering Methods

Method	**Pros**	**Cons**
Extended Kalman Filter	Fast, analytical	Fails for strong nonlinearity
Particle Filter	Handles any nonlinearity	Not differentiable
Differentiable PF	Learnable, handles nonlinearity	Computationally expensive

9 Summary

Table 2: Standard vs Differentiable Particle Filter

Aspect	**Standard PF**	**Differentiable PF**
Transition	$x_t^{(i)} \sim p(x_t \mid x_{t-1}^{(i)})$	Same
Likelihood	$p(y_t \mid x_t^{(i)})$	Same
Weight Update	Hard normalization	Soft normalization
Resampling	Categorical sampling	Gumbel-Softmax
Gradient	Blocked	Flows via backprop
Purpose	State estimation	Learnable inference

A Gumbel Distribution

The Gumbel distribution enables differentiable sampling from categorical distributions.

A.1 Gumbel-Max Trick

If $g_i \sim \text{Gumbel}(0, 1)$ independently, then:

$$\arg\max_i(\log \pi_i + g_i) \sim \text{Categorical}(\pi) \tag{23}$$

A.2 Gumbel-Softmax Relaxation

Replace $\arg\max$ with softmax for differentiability:

$$\tilde{\pi}_i = \frac{\exp((\log \pi_i + g_i)/\tau)}{\sum_j \exp((\log \pi_j + g_j)/\tau)} \tag{24}$$

This provides a continuous, differentiable approximation to categorical sampling.