pyParticleEst

A Software Framework for Nonlinear Estimation

Jerker Nordh

Department of Automatic Control, Lund University

jerker.nordh@control.lth.se

LUND UNIVERSITY

github.com/jerkern/pyParticleEst

Introduction

Particle filters and smoothers have been successfully applied to a large number of challenging nonlinear estimation problems, such as target tracking, indoor navigation, simultaneous localization and mapping (SLAM) and radio channel estimation.

The particle based methods allow estimation of both nonlinear systems and non-Gaussian noise distributions. In contrast to e.g. the Extended and Unscented Kalman filters (EKF/UKF) they can also handle multi-modal distributions.

While the usefulness of these methods is widely recognized there is very little software support for applying them to new problems. Typically a particle filter is written from scratch for each new application, although of course individual researchers reuse code they've written before. For the typical particle filter this is not a big issue, since the amount of code required is rather small, there is however a few caveats to avoid to achieve the best possible performance.

The complexity of the implementation quickly increases when more advanced methods are to be used, eg. different variants of backward simulators or Rao-Blackwellized filters/smoothers. Most of this complexity is not problem specific and the implementation could be reused to reduce the effort needed when solving new problems and reduce the risk of software bugs. Providing such a framework to facilitate reuse is the focus of the work presented here.

Capabilities

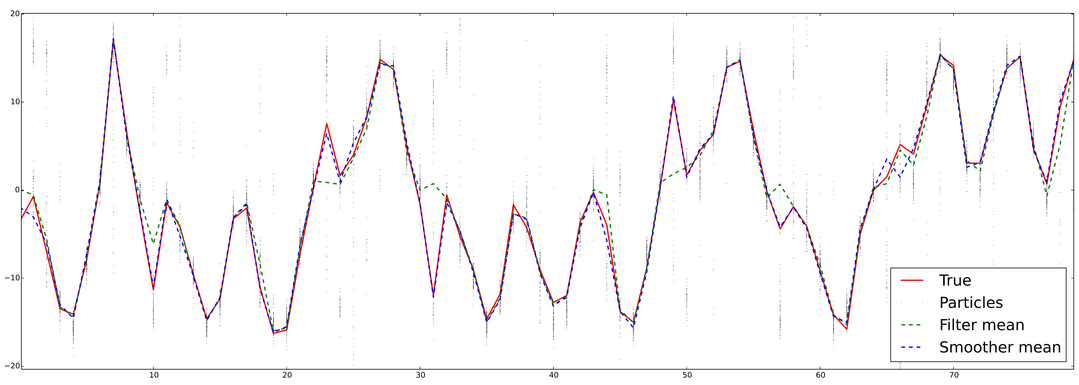
Filters: PF, APF

Smoothers: FFBSi, FFBSi-RS, FFBSi-RSAS, MH-FFBSi, MHBP, MH-IPS

Parameter estimation: PS+EM

Model classes: LTV, NLG, MLNLG (Rao-Blackwellized), Hierarchical (Rao-Blackwellized)

Example



$$x_{t+1} = f(x_t) + v_t = 0.5x_t + 25\frac{x_t}{1+x_t^2} + 8\cos 1.2t + v_t, \quad v_t \sim N(0, Q)$$

$$y_t = g(x_t) + e_t = 0.05x_t^2 + e_k, \quad e_k \sim N(0, R), \quad x_0 \sim N(0, P_0)$$

```
class Model(nlg.NonlinearGaussianInitialGaussian):
    def __init__(self, P0, Q, R):
        super(Model, self).__init__(Px0=P0, Q=Q, R=R)

    def get_g(self, particles, t):
        return 0.05*particles**2

    def get_f(self, particles, u, t):
        return (0.5*particles +
                25.0*particles/(1+particles**2) +
                8*math.cos(1.2*t))
```

Future work

The framework will continue to be expanded with interesting new methods and model classes, outside contributions are also welcomed.

There is also an interest in collaborations and to provide assistance with using the software to solve new and challenging problems.

Acknowledgements: The author is a member of the LCCC Linnaeus Center and the eLLIIT Excellence Center at Lund University.

Implementation

Overview

- Licensed as LGPL, free for commercial use and can be integrated into proprietary software. Changes to the library itself have to be published.
- Implemented in Python, provides a completely free environment that supports a large number operating systems and hardware platforms.
- Uses Numpy/Scipy for efficient numerical computations

Design

- Object-oriented, models are defined by specializing base-classes and reusing common characteristics
- Algorithms define a set of interfaces that are required for that algorithm
- Provides a set of base-classes reducing the implementation burden for typical model classes such as LTV, NLG, MLNLG
- Easy to incorporate new algorithms and model classes

Particle Filter

The particle filter (PF) is an application of sequential importance sampling. It proposes N samples from the dynamics model, these samples are then reweighted using the information obtained from the measurements.

1. Draw $x_0^{(i)}$ from $p(x_0), \quad i \in 1..N$
2. Set $w_0^{(i)} = \frac{1}{N}, \quad i \in 1..N$
3. For $t = 1..T-1$
   (a) For $i = 1..N$
      i. Sample $x_{t+1}^{(i)}$ from $p(x_{t+1}|x_t^{(i)})$
      ii. Set $w_{t+1}^{(i)} = w_t^{(i)} p(y_{t+1}|x_{t+1}^{(i)})$
   (b) Normalize weights, $\hat{w}^{(i)} = w_{t+1}^{(i)} / \sum_j w_{t+1}^{(j)}$
   (c) For $i = 1..N$
      i. Sample $x_{t+1}^{(i)} \sim p(x_{t+1}|y_{t+1})$ by drawing from the categorical distribution defined by $(x_{t+1}^{(k)}, \hat{w}^{(k)}), \quad k \in 1..N$
      ii. Set $w_{t+1}^{(i)} = \frac{1}{N}, \quad i \in 1..N$

This algorithm is typically improved by not performing the resampling step (3c) at every iteration, but only when some prespecified criteria on the weights is fulfilled.

A variant of the Particle Filter is the Auxiliary Particle Filter (APF) which incorporates the information of the measurement already in the proposal step (3ai) by only drawing samples from particles where the measurement is predicted to be likely.

Particle Smoother

Most particle smoothers work through the concept of backward simulation (FFBSi). In FFBSi the particle estimates generated from the filter are reused when creating backward trajectories by randomly choosing the ancestor according to $\omega_{t|T}^{(i)}$, which for normal statespace models can be calculated as $\omega_{t|T}^{(i)} = \omega_{t|t}^{(i)} p(x_{t+1}|x_t^{(i)})$. This reduces the degeneracy of the estimate of $x_{t|T}$ that is typical for the particle filter when $t \ll T$.

Evaluating the weights $\omega_{t|T}^{(i)}$ for the categorical distribution in the backward step is computationally expensive, therefore a number of methods have been proposed to reduce the time complexity, one is to use rejection sampling (FFBSi-RS) another is to use a Metropolis-Hastings sampler (MH-FFBSi)

Additionally there exists methods which not only reuse the point estimates from the forward filter but also propose new values, two such methods are the Metropolis-Hastings Backward Proposer (MHBP) and the Metropolis-Hastings Improved Particle Smoother (MH-IPS)