Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *http://www.jstatsoft.org/*

pyParticleEst: A Python Framework for Particle Based Estimation Methods

Jerker Nordh
Lund University

Abstract

Particle methods such as the Particle Filter and Particle Smoothers have proven very useful for solving challenging nonlinear estimation problems in a wide variety of fields during the last decade. However, there is still very few existing tools to support and assist researchers and engineers in applying the vast number of methods in this field to their own problems. This paper identifies the common operations between the methods and describes a software framework utilizing this information to provide a flexible and extensible foundation which can be used to solve a large variety of problems in this domain, thereby allowing code reuse to reduce the implementation burden and lowering the barrier of entry for applying this exciting field of methods. The software implementation presented in this paper is freely available and permissively licensed under the GNU Lesser General Public License, and runs on a large number of hardware and software platforms, making it usable for a large variety of scenarios.

Keywords: particle-filter, particle-smoother, expectation-maximization, system identification, rao-blackwellized, Python.

1. Introduction

During the last few years, particle based estimation methods such as particle filtering (Doucet, Godsill, and Andrieu 2000) and particle smoothing (Briers, Doucet, and Maskell 2010) have become increasingly popular and provide a powerful alternative for nonlinear/non-Gaussian and multi-modal estimation problems. Noteworthy applications of particle methods include multi-target tracking (Okuma, Taleghani, De Freitas, Little, and Lowe 2004), Simultanous Localization and Mapping (SLAM) (Montemerlo, Thrun, Koller, Wegbreit *et al.* 2002) and Radio Channel Estimation (Mannesson 2013). Popular alternatives to the particle filter are the Extended Kalman Filter (Julier and Uhlmann 2004) and Unscented Kalman Filter (Julier and Uhlmann 2004), but they can not always provide the performance needed, and neither

handles multimodal distributions well. The principles of the Particle Filter and Smoother are fairly straight forward, but there are still a few caveats when implementing them. There is a large part of the implementation effort that is not problem specific and thus could be reused, thereby reducing both the overall implementation effort and the risk of introducing errors. Currently there is very little existing software support for using these methods, and for most applications the code is simply written from scratch each time. This makes it harder for people new to the field to apply methods such as particle smoothing. It also increases the time needed for testing new methods and models for a given problem. This paper breaks a number of common algorithms down to a set of operations that need to be performed on the model for a specific problem and presents a software implementation using this structure. The implementation aims too exploit the code reuse opportunities by providing a flexible and extensible foundation to build upon where all the basic parts are already present. The model description is clearly separated from the algorithm implementations. This allows the end user to focus on the parts unique for their particular problem and to easily compare the performance of different algorithms. The goal of this article is not to be a manual for this framework, but to highlight the common parts of a number of commonly used algorithms from a software perspective. The software presented serves both as a proof of concept and as an invitation to those interested to study further, use and to improve upon.

The presented implementation currently supports a number of filtering and smoothing algorithms and has support code for the most common classes of models, including the special case of Mixed Linear/Nonlinear Gaussian State Space (MLNLG) models using Rao-Blackwellized algorithms described in Section 3, leaving only a minimum of implementation work for the end user to define the specific problem to be solved.

In addition to the filtering and smoothing algorithms the framework also contains a module that uses them for parameter estimation (grey-box identification) of nonlinear models. This is accomplished using an Expectation Maximization (EM)(Dempster, Laird, and Rubin 1977) algorithm combined with a Rao-Blackwellized Particle Smoother (RBPS) (Lindsten and Schön 2010).

The framework is implemented in Python and following the naming conventions typically used within the Python community it has been named **pyParticleEst**. For an introduction to Python and scientific computation see (Oliphant 2007). All the computations are handled by the **Numpy**/**Scipy** (Jones, Oliphant, Peterson *et al.* 2001–) libraries. The choice of Python is motivated by that it can run on a wide variety of hardware and software platforms, moreover since **pyParticleEst** is licensed under the LGPL (FSF 1999) it is freely usable for anyone without any licensing fees for either the software itself or any of its dependencies. The LGPL license allows it to be integrated into proprietary code only requiring any modifications to the actual library itself to be published as open source. All the code including the examples presented in this article can be downloaded from (Nordh 2013).

The remaining of this paper is organized as follows. Section 2 gives a short overview of other existing software within this field. Section 3 gives an introduction to the types of models used and a quick summary of notation, Section 4 presents the different estimation algorithms and isolates which operations each method requires from the model. Section 5 provides an overview of how the software implementation is structured and details of how the algorithms are implemented. Section 6 shows how to implement a number of different types of models in the framework. Section 7 presents some results that are compared with previously published data to show that the implementation is correct. Section 8 concludes the paper with a short

discussion of the benefits and drawbacks with the approach presented.

2. Related software

The only other software package within this domain to the authors knowledge is **LibBi** (Murray In review). **LibBi** takes a different approach and provides a domain-specific language for defining the model for the problem. It then generates high performance code for a Particle Filter for that specific model. In contrast, **pyParticleEst** is more focused on providing an easily extensible foundation where it is easy to introduce new algorithms and model types, a generality which comes at some expense of run-time performance making the two softwares suitable for different use cases. It also has more focus on different smoothing algorithms and filter variants.

There is also a lot of example code that can be found on the Internet, but nothing in the form of a complete library with a clear separation between model details and algorithm implementation. This separation is what gives the software presented in this article its usability as a general tool, not only as a simple template for writing a problem specific implementation. This also allows for easy comparison of different algorithms for the same problem.

3. Modelling

While the software framework supports more general models, this paper focuses on discrete time stace-space models of the form

$$x_{t+1} = f(x_t, v_t) \tag{1a}$$
$$y_t = h(x_t, e_t) \tag{1b}$$

where x_t are the state variables, v_t is the process noise and y_t is a measurement of the state affected by the measurement noise e_t. The subscript t is the time index. Both v and e are random variables according to some known distributions, f and h are both arbitrary functions.

If f, h are affine and v, e are Gaussian random variables the system is what is commonly referred to as a Linear Gaussian State Space system (LGSS) and the Kalman filter is both the best linear unbiased estimator (Arulampalam, Maskell, Gordon, and Clapp 2002) and the Maximum Likelihood estimator.

Due to the scaling properties of the Particle Filter and Smoother, which are discussed in more detail in Section 4.1, it is highly desirable to identify any parts of the models that conditioned on the other states would be linear Gaussian. The state-space can then be partitioned as $x^T = \left(\xi^T z^T\right)$, where z are the conditionally linear Gaussian states and ξ are the rest.

Extending the model above to explicitly indicate this gives

$$\begin{aligned}
\xi_{t+1} &= \begin{pmatrix} f_\xi^n(\xi_t, v_\xi^n) \\ f_\xi^l(\xi_t) \end{pmatrix} + \begin{pmatrix} 0 \\ A_\xi(\xi_t) \end{pmatrix} z_t + \begin{pmatrix} 0 \\ v_\xi^l \end{pmatrix} && \text{(2a)} \\
z_{t+1} &= f_z(\xi_t) + A_z(\xi_t) z_t + v_z && \text{(2b)} \\
y_t &= \begin{pmatrix} h_\xi(\xi_t, e^n) \\ h_z(\xi_t) \end{pmatrix} + \begin{pmatrix} 0 \\ C(\xi_t) \end{pmatrix} z_t + \begin{pmatrix} 0 \\ e^l \end{pmatrix} && \text{(2c)} \\
v_\xi^l &\sim N(0, Q_\xi(\xi_t)), \quad v_z \sim N(0, Q_z(\xi_t)), \quad e^l \sim N(0, R(\xi_t)) && \text{(2d)}
\end{aligned}$$

As can be seen all relations in (2) involving z are linear with additive Gaussian noise when conditioned on ξ. Here the process noise for the non-linear states v_ξ is split in two parts: v_ξ^l appears linearly and must be Gaussian whereas v_ξ^n can be from any distribution, similarly holds true for e^l and e^n. This is referred to as a Rao-Blackwellized model.

If we remove the coupling from z to ξ we get what is referred to as a hierarchical model

$$\begin{aligned}
\xi_{t+1} &= f_\xi(\xi_t, v_\xi) && \text{(3a)} \\
z_{t+1} &= f_z(\xi_t) + A(\xi_t) z_t + v_z && \text{(3b)} \\
y_t &= \begin{pmatrix} h_\xi(\xi_t, e^n) \\ h_z(\xi_t) \end{pmatrix} + \begin{pmatrix} 0 \\ C(\xi_t) \end{pmatrix} z_t + \begin{pmatrix} 0 \\ e^l \end{pmatrix} && \text{(3c)} \\
v_z &\sim N(0, Q_z(\xi_t)), \quad e^l \sim N(0, R(\xi_t)) && \text{(3d)}
\end{aligned}$$

Another interesting class are Mixed Linear/Nonlinear Gaussian (MLNLG) models

$$\begin{aligned}
\xi_{t+1} &= f_\xi(\xi_t) + A_\xi(\xi_t) z_k + v_\xi && \text{(4a)} \\
z_{t+1} &= f_z(\xi_t) + A_z(\xi_t) z_t + v_z && \text{(4b)} \\
y_t &= h(\xi_t) + C(\xi_t) z_t + e && \text{(4c)} \\
e &\sim N(0, R(\xi_t)) && \text{(4d)} \\
\begin{bmatrix} v_\xi \\ v_z \end{bmatrix} &\sim N\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} Q_\xi(\xi_t) & Q_{\xi z}(\xi_t) \\ Q_{\xi z}(\xi_t)^T & Q_z(\xi_t) \end{bmatrix} \right) && \text{(4e)} \\
& && \text{(4f)}
\end{aligned}$$

The MLNLG model class (4) allows for non-linear dynamics but with the restrictions that all noise must enter additively and be Gaussian.

4. Algorithms

This section gives an overview of some common particle based algorithms, they are subdivided into those used for filtering, smoothing and static parameter estimation. For each algorithm it is identified which operations need to be performed on the model.

4.1. Filtering

This subsection gives a quick summary of the principles of the Particle Filter, for a thorough introduction see for example Doucet *et al.* (2000).

The basic concept of a Particle Filter is to approximate the probability density function (pdf) for the states of the system by a number of point estimates

$$p(x_t|y_{1:t}) \approx \sum_{i=1}^{N} w^{(i)} \delta(x_t - x_t^{(i)}) \tag{5}$$

Each of the N particles in (5) consists of a state, $x_t^{(i)}$, and a corresponding weight, $w_t^{(i)}$, representing the likelihood of that particular particle. Each estimate is propagated forward in time using (1a) by sampling v_t from the corresponding noise distribution, providing an approximation of $p(x_{t+1}|y_t, ..., y_1)$. The measurement y_{t+1} is incorporated by updating the weights of each particle with respect to how well it predicted the new measurement, giving an approximation of $p(x_{t+1}|y_{t+1}, y_t, ..., y_1)$. This procedure is iterated forward in time providing a filtered estimate of the state x.

A drawback with this approach is that typically all but one of the weights, $w_t^{(i)}$, eventually go to zero resulting in a poor approximation of the true pdf. This is referred to as particle degeneracy and is commonly solved by a process called resampling (Arulampalam *et al.* 2002). The idea behind resampling is that at each time-step, or when some criteria is fulfilled, a new collection of particles with all weights equal ($w^{(i)} = \frac{1}{N}, \forall i$) is created by randomly drawing particles, with replacement, according to their weights. This focuses the particle approximation to the most likely regions of the pdf, not wasting samples in regions with low probability. This method is summarized in Algorithm 1.

Another issue with the standard Particle Filter is that the number of particles needed in the filter typically grows exponentially with the dimension of the state-space as discussed in Beskos, Crisan, and Jasra (2011) and Rebeschini and van Handel (2013), where they also present methods to avoid this issue. Another popular approach is to use Rao-Blackwellized methods when there exists a conditionally linear Gaussian substructure. Using the partitioning from model (2) this provides a better approximation of the underlying pdf for a given number of particles by storing the sufficient statistics for the z-states instead of sampling from the Gaussian distributions. For an introduction to the Rao-Blackwellized Particle Filter (RBPF) see Schön, Gustafsson, and Nordlund (2005).

A variant of the Particle Filter is the so called Auxiliary Particle Filter (APF), it attempts to focus the particles to regions of high interest by looking one step ahead by evaluating $p(y_{t+1}|x_t)$ and using this to resample the particles before the propagation stage. Since there is typically no analytic expression for this density it is often approximated by assuming that the next state will be the predicted mean; $p(y_{t+1}|x_{t+1} = \bar{x}_{t+1|t})$.

Table 1 summaries the methods needed for the two different filters.

Algorithm 1: Standard Particle Filter algorithm, this is typically improved by not performing the resampling step at every iteration, but only when some prespecified criteria on the weights is fulfilled

Draw $x_0^{(i)}$ from $p(x_0)$, $\quad i \in 1..N$
Set $w_0^{(i)} = \frac{1}{N}$, $\quad i \in 1..N$
for $t \leftarrow 0$ **to** $T-1$ **do**
  for $i \leftarrow 1$ **to** N **do**
    Sample $x_{t+1}^{(i)}$ from $p(x_{t+1}|x_t^{(i)})$
    Set $w_{t+1}^{(i)} = w_t^{(i)} p(y_{t+1}|x_{t+1}^{(i)})$;
  Normalize weights, $\hat{w}^{(i)} = w_{t+1}^{(i)} / \sum_j w_{t+1}^{(j)}$
  for $i \leftarrow 1$ **to** N **do**
    Sample $x_{t+1}^{(i)} \sim p(x_{t+1}|y_{t+1})$ by drawing new particles from the categorical distribution defined by $(x_{t+1}^{(k)}, \hat{w}^{(k)})$, $\quad k \in 1..N$
  Set $w_{t+1}^{(i)} = \frac{1}{N}$, $\quad i \in 1..N$

Operations	Methods
Sample from $p(x_1)$	PF, APF
Sample from $p(x_{t+1}\|x_t)$	PF, APF
Evaluate $p(y_t\|x_t)$	PF, APF
Evaluate* $p(y_{t+1}\|x_t)$	APF

Table 1: Operations that need to be performed on the model for the different filter algorithms. (*typically only approximately).

4.2. Smoothing

Conceptually the Particle Filter provides a smoothed estimate if the trajectory for each particle is saved and not just the estimate for the current time-step. The full trajectory weights are then given by the corresponding particle weights for the last time-step. In practice this doesn't work due to the resampling step which typically results in that all particles eventually share a common ancestor, thus providing a very poor approximation of the smoothed pdf for $t \ll T$. An example of this is shown in Fig. 1

*Forward Filter Backward Simulator*s (FFBSi) are a class of methods that reuse the point estimates for $x_{t|t}$ generated by the particle filter and attempt to improve the particle diversity by drawing backward trajectories that are not restricted to follow the same paths as those generated by the filter. This is accomplished by selecting the ancestor of each particle with probability $\omega_{t|T} \sim \omega_{t|t} p(x_{t+1}|x_t)$. Evaluating all the weights $\omega_{t|T}$ gives a time complexity $O(MN)$ where N is the number of forward particles and M the number of backward trajectories to be generated.

A number of improved algorithms have been proposed that improve this by removing the need to evaluate all the weights. One approach is to use rejection sampling (FFBSi-RS) (Lindsten and Schön 2013), this however does not guarantee a finite end-time for the algorithm, and

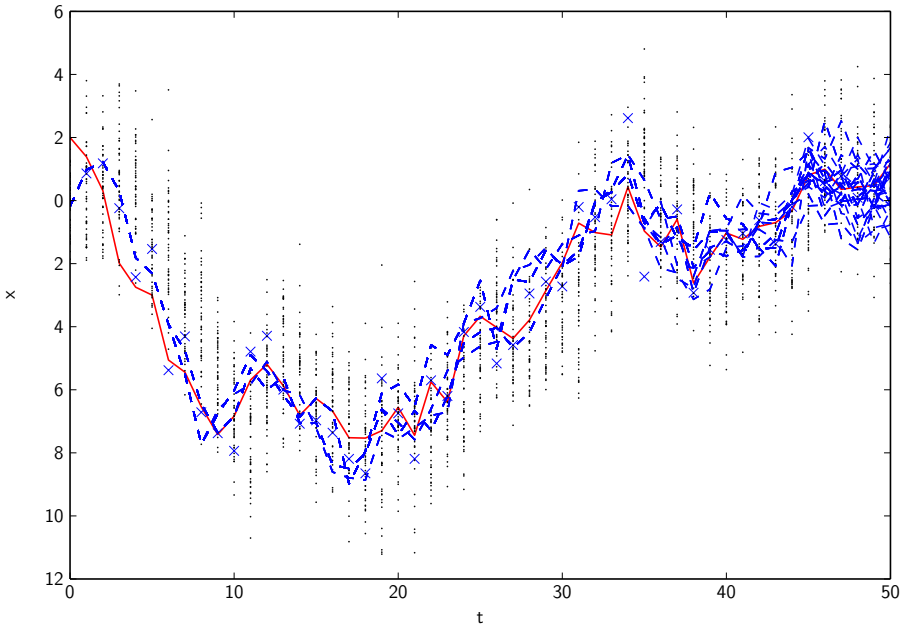

Figure 1: Example realization of a model of a simple integrator. The solid red line is the true trajectory. The black points are the filtered particle estimates forward in time, the blue dashed lines are the smoothed trajectories that result from using the particles ancestral paths. As can be seen this is severely degenerate for small values of t, whereas it works well for t close to the end of the dataset.

typically spends a lot of the time on just a few trajectories. This is handled by introducing *early stopping* (FFBSi-RSES) which falls back to evaluating the full weights for a given time step after a predetermined number of failed attempts at rejection sampling. Determining this number ahead of time can be difficult, and the method is further improved by introducing *adaptive stopping* (FFBSi-RSAS) (Taghavi, Lindsten, Svensson, and Schön 2013) which estimates the probability of successfully applying rejection sampling based on the previous successes and compares that with the cost of evaluating all the weights.

Another approach is to use Metropolis Hastings (MH-FFBsi) (Bunch and Godsill 2013) when sampling the backward trajectory, then instead of calculating N weights, R iterations of a Metropolis-Hastings sampler are used.

All the methods mentioned so far only reuse the point estimates from the forward filter, there also exists methods that attempt to create new samples to better approximate the true posterior. One such method is the Metropolis-Hastings Backward Proposer (MHBP) (Bunch and Godsill 2013), another is the Metropolis-Hastings improved particle smoother (MH-IPS) (Dubarry and Douc 2011).

MHBP starts with the degenerate trajectories from the filter and while traversing them backwards proposes new samples by running R iterations of a Metropolis-Hastings sampler targeting $p(x_t|x_{t-1}, x_{t+1}, y_t)$ for each time-step.

MH-IPS can be combined with the output from any of the other smoothers to give an improved estimate. It performs R iterations where each iteration traverses the full backward trajectory and for each time-step runs a single iteration of a Metropolis-Hastings sampler targeting $p(x_t|x_{t-1}, x_{t+1}, y_t)$.

Table 2 lists the operations needed for the different smoothing methods. For a more detailed introduction to Particle Smoothing see for example Briers *et al.* (2010), Lindsten and Schön (2013), and for an extension to the Rao-Blackwellized case see Lindsten and Schön (2011)

Operations	Methods
Evaluate $p(x_{t+1}\|x_t)$	FFBSi, FFBSi-RS, FFBSi-RSES, FFBSi-RSAS, MH-FFBSi, MH-IPS, MHBP
Evaluate $\operatorname{argmax}_{x_t+1} p(x_{t+1}\|x_t)$	FFBSi-RS, FFBSi-RSES, FFBSi-RSAS
Sample from $q(x_t\|x_{t-1}, x_{t+1}, y_t)$	MH-IPS, MHBP
Evaluate $q(x_t\|x_{t-1}, x_{t+1}, y_t)$	MH-IPS, MHBP

Table 2: Operations that need to be performed on the model for the different smoothing algorithms. They all to some extent rely on first running a forward filter, and thus in addition require the operations needed for the filter. Here q is a proposal density, a simple option is to choose $q = p(x_{t+1}|x_t)$, as this does not require any further operations. The ideal choice would be $q = p(x_t|x_{t+1}, x_{t-1}, y_t)$, but it is typically not possible to directly sample from this density.

4.3. Parameter estimation

Using a standard Particle Filter or Smoother it is not possible to estimate stationary parameters, θ, due to particle degeneracy. A common work-around for this is to include θ in the state vector and model the parameters as a random walk process with a small noise covariance. A drawback with this approach is that the parameter is no longer modeled as being constant, in addition it increases the dimension of the state-space, worsening the problems mentioned in Section 4.1.

PS+EM

Another way to do parameters estimation is to use an Expectation Maximization (EM) algorithm where the expectation part is calculated using a RBPS. For a detailed introduction to the EM-algorithm see Dempster *et al.* (1977) and for how to combine it with a RBPS for parameter estimates in model (4) see Lindsten and Schön (2010).

The EM-algorithm finds the maximum likelihood solution by alternating between estimating the Q-function for a given θ_k and finding the θ that maximizes the log-likelihood for a given estimate of $x_{1:T}$, where

$$Q(\theta, \theta_k) = \mathrm{E}_{X|\theta_k}[\mathrm{L}_\theta(X, Y|Y)] \tag{6a}$$

$$\theta_{k+1} = \operatorname*{argmax}_\theta Q(\theta, \theta_k) \tag{6b}$$

Here X is the complete state trajectory $(x_1, ..., x_N)$, Y is the collection of all measurements $(y_1, ..., y_N)$ and L_θ is the log-likelihood as a function of the parameters θ. In Lindsten and

Schön (2010) it is shown that the Q-function can be split into three parts as follows

$$
\begin{aligned}
Q(\theta, \theta_k) &= I_1(\theta, \theta_k) + I_2(\theta, \theta_k) + I_3(\theta, \theta_k) && \text{(7a)}\\
I_1(\theta, \theta_k) &= \mathrm{E}_{\theta_k}[\log p_\theta(x_1)|Y] && \text{(7b)}\\
I_2(\theta, \theta_k) &= \sum_{t=1}^{N-1} \mathrm{E}_{\theta_k}[\log p_\theta(x_{t+1}|x_t)|Y] && \text{(7c)}\\
I_3(\theta, \theta_k) &= \sum_{t=1}^{N} \mathrm{E}_{\theta_k}[\log p_\theta(y_t|x_t)|Y] && \text{(7d)}
\end{aligned}
$$

The expectations in (7b)-(7d) are approximated using a (Rao-Blackwellized) Particle Smoother, where the state estimates are calculated using the old parameter estimate θ_k. This procedure is iterated until the parameter estimates converge. The methods needed for PS+EM are listed in Table 3

Operations	Methods
Maximize $\mathrm{E}_{\theta_k}[\log p_\theta(x_1)\|Y]$	PS+EM
Maximize $\mathrm{E}_{\theta_k}[\log p_\theta(x_{t+1}\|x_t)\|Y]$	PS+EM
Maximize $\mathrm{E}_{\theta_k}[\log p_\theta(y_t\|x_t)\|Y]$	PS+EM
Evaluate $q(\theta'\|\theta)$	PMMH
Sample from $q(\theta'\|\theta)$	PMMH
Evaluate $\pi(\theta)$	PMMH

Table 3: Operations that need to be performed on the model for the presented parameter estimation methods. PS-EM relies on running a smoother, and thus in addition requires the operations needed for the smoother. The maximization is with respect to θ. Typically the maximization can not be performed analytically, and then depending on which type of numerical solver is used, gradients and Hessians might be needed as well. PMMH does not require a smoothed estimate, it only uses a filter, and thus puts fewer requirements on the types of models that can be used. Here q is the proposal density for the static parameters, π is the prior probability density function. PMMH does not need a smoothed trajectory estimate, it is sufficient with the filtered estimate.

PMMH

Another method which instead takes a Bayesian approach is Particle Marginal Metropolis-Hastings (PMMH)(Andrieu, Doucet, and Holenstein 2010) which is one method within the broader class known as Particle Markov Chain Monte Carlo (PMCMC) methods. It uses a particle filter as part of a Metropolis-Hastings sampler targeting the joint density of the state trajectory and the unknown parameters. This method is not discussed further in this paper. The methods needed for PMMH are listed in Table 3.

5. Implementation

5.1. Language

The framework is implemented in Python, for an introduction to the use of Python in scientific computing see Oliphant (2007). The numerical computations rely on **Numpy**/**Scipy** (Jones *et al.* 2001–) for a fast and efficient implementation. This choice was made as it provides a free environment, both in the sense that there is no need to pay any licensing fees to use it, but also that the code is open source and available for a large number of operating systems and hardware platforms. The **pyParticleEst** framework is licensed under the LGPL (FSF 1999), which means that it can be freely used and integrated into other products, but any modifications to the actual **pyParticleEst** code must be made available. The intent behind choosing this license is to make the code easily usable and integrable into other software packages, but still encourage sharing of any improvements made to the library itself. The software and examples used in this article can be found in Nordh (2013).

5.2. Overview

The fundamental idea in **pyParticleEst** is to provide algorithms operating on the methods identified in Section 4, thus effectively separating the algorithm implementation from the problem description. Additionally, the framework provides an implementation of these methods for a set of common model classes which can be used for solving a large set of problems. They can also be extended or specialized by the user by using the inheritance mechanism in Python. This allows new types of problems to be solved outside the scope of what is currently implemented, but it also allows creation of classes building on the foundations present but overriding specific methods for increased performance, without rewriting the whole algorithm from scratch. The author believes this provides a good trade-off between generality, extensibility and ease of use.

For each new type of problem to be solved the user defines a class extending the most suitable of the existing base classes, for example the one for MLNLG systems. In this case the user only has to specify how the matrices and functions in (4) depend on the current estimate of the nonlinear state. For a more esoteric problem class the end user might have to do more implementation work and instead derive from a class higher up in the hierarchy, for example the base class for models that can be partioned into a conditionally linear part, which is useful when performing Rao-Blackwellized filtering or smoothing. This structure is explained in more detail in sections 5.3.1-5.3.3.

The main interface to the framework is through the `Simulator` class, it is used to store the model used for the estimation toghether with the input signals and measurements, it also provides a mechanism for executing the different algorithms on the provided model and data. It is used by creating an object of the Simulator class with input parameters that specify the problem to be solved as follows

```
sim = Simulator(model, u, y)
```

Here `model` is an object defining all model specific operations, `u` is an array of all the input signals and `y` is an array of all measurements. Once the object has been created it serves as the interface to the actual algorithm, an example of how it could be used is shown below

```
sim.simulate(num, nums, res=0.67, filter='PF', smoother='mcmc')
```

Here `num` is the number of particles used in the forward filter, `nums` are the number of smoothed trajectories generated by the smoother, `res` is the resampling threshhold (expressed as the ratio of effective particles compared to total number of particles), `filter` is the filtering method to be used and finally `smoother` is the smoothing algorithm to be used.

After calling the method above the results can be access by using some of the following methods

```
(est_filt, w_filt) = sim.get_filtered_estimates()
mean_filt = sim.get_filtered_mean()
est_smooth = sim.get_smoothed_estimates()
smean = sim.get_smoothed_mean()
```

where `(est_filt, w_filt)` will contain the forward particles for each time step with the corresponding weights, `mean_filt` is the weighted mean of all the forward particles for each time step. `est_smooth` is an array of all the smoothed trajectories and `smean` the mean value for each time step of the smoothed trajectories.

5.3. Software design

The software consists of a number of supporting classes that store the objects and their relations, the most important of these are shown in Figure 2 and are summarized below.

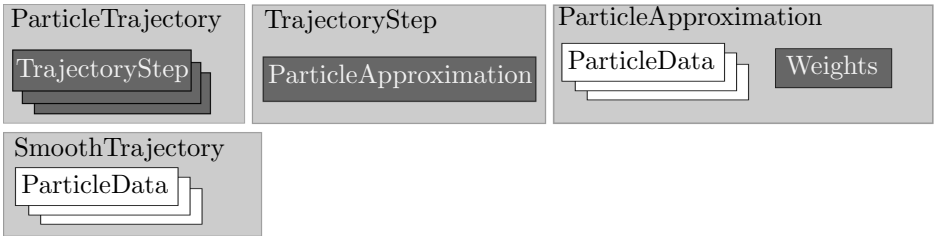

Figure 2: Overview of the classes used for representing particle estimates and their relation. The grey boxes are classes that are part of the framework, the white boxes represent objects of problem specific data-types. A box encapsulating another box shows that objects from that class contains objects from the other class. The illustration is not complete, but serves as an overview of the overall layout.

The particles are stored as raw data, where each model class is responsible for determining how it is best represented. This data is then sent as one of the parameters to each method the model class defines. This allows the model to choose an efficient representation allowing for e.g., parallell processing of all the particles for each time-step. The details of the class hierarchy and the models for some common cases are explored further in sections 5.3.1-5.3.3.

The particle data is stored using the `ParticleApproximation` class, which in addition to the raw data also stores the corresponding weights according to (5). The class `TrajectoryStep` stores the approximation for a given time instant combined with other related data such as input signals and measurement data. The `ParticleTrajectory` class represents the filtered

estimates of the entire trajectory by storing a collection of `TrajectorySteps`, it also provides the methods for interfacing with the chosen filtering algorithm.

The `SmoothTrajectory` class takes a `ParticleTrajectory` as input and using a Particle Smoother creates a collection of point estimates representing the smoothed trajectory estimate. In the same manner as for the `ParticleApproximation` class the point estimates here are of the problem specific data type defined by the model class, but not necessarily of the same structure as the estimates created by the forward filter. This allows for example methods where the forward filter is Rao-Blackwellized but the backward smoother samples the full state vector.

Model class hierarchy

The software utilizes the **Python ABC** package to create a set of abstract base-classes that define all the needed operations for the algorithms. Figure 3 shows the complete class hierarchy for the algorithm interfaces and model types currently implemented.

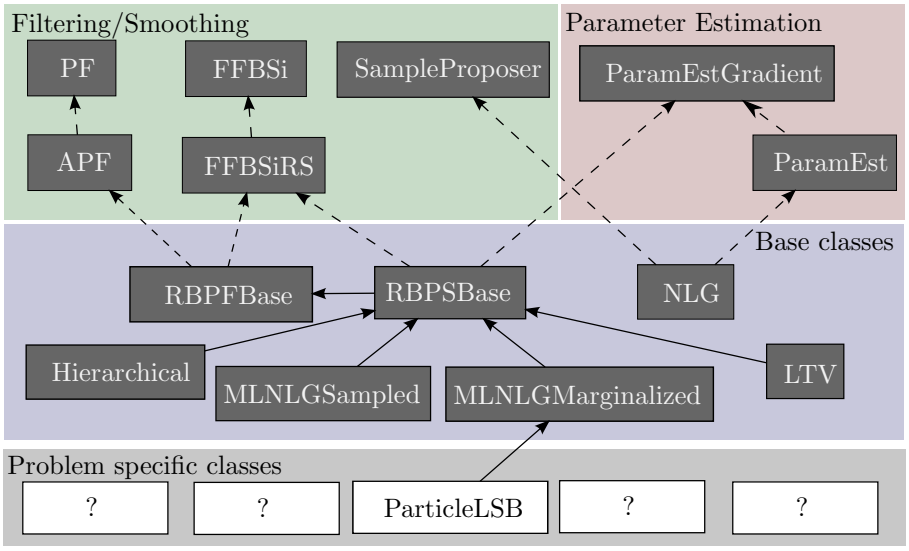

Figure 3: Class hierarchy for models that are used in the framework. The `ParticleLSB` class is presented in Section 6.3 and is an implementation of Example B from Lindsten and Schön (2011).

- `PF` defines the basic operations needed for performing particle filtering:
  - `create_initial_estimate`: Create particle estimate of initial state.
  - `sample_process_noise`: Sample v_t from the process noise distribution.
  - `update`: Calculate x_{t+1} given x_t using the supplied noise v_t.
  - `measure`: Evaluate $\log p(y_t|x_{t|t-1})$ and for the RBPF case update the sufficient statistics for the z-states.
- `APF` extends `PF` with extra methods needed for the Auxiliary Particle Filter:
  - `eval_1st_stage_weights`: Evaluate (approximately) the so called first stage weights, $p(y_{t+1}|x_t)$.

- `FFBSi` defines the basic operations needed for performing particle smoothing:
  - `logp_xnext_full`: Evaluate $\log p(x_{t+1:T}|x_{1:t}, y_{1:T})$. This method normally just calls `logp_xnext`, but the distinction is needed for non-Markovian models.
  - `logp_xnext`: Evaluate $\log p(x_{t+1}|x_t)$.
  - `sample_smooth`: For normal models the default implementation can be used which just copies the estimate from the filter, but for e.g., Rao-Blackwellized models additional computations are made in this method.
- `FFBSiRS` extends `FFBSi`:
  - `next_pdf_max`: Calculate maximum of $\log p(x_{t+1}|x_t)$.
- `SampleProposer` defines the basic operations needed for proposing new samples, used in the MHBP and MH-IPS algorithms:
  - `propose_smooth`: Propose new sample from $q(x_t|x_{t+1}, x_{t-1}, y_t)$.
  - `logp_proposal`: Evaluate $\log q(x_t|x_{t+1}, x_{t-1}, y_t)$.
- `ParamEstInterface` defines the basic operations needed for performing parameter estimation using the EM-algorithm presented in Section 4.3:
  - `set_params`: Set θ_k estimate.
  - `eval_logp_x0`: Evaluate $\log p(x_1)$.
  - `eval_logp_xnext`: Evaluate $\log p(x_{t+1}|x_t)$.
  - `eval_logp_y`: Evaluate $\log p(y_t|x_t)$.
- `ParamEstInterface_GradientSearch` extends the operations from the ParamEstInterface to include those needed when using analytic derivatives in the maximization step:
  - `eval_logp_x0_val_grad`: Evaluate $\log p(x_1)$ and its gradient.
  - `eval_logp_xnext_val_grad`: Evaluate $\log p(x_{t+1}|x_t)$ and its gradient.
  - `eval_logp_y_val_grad`: Evaluate $\log p(y_t|x_t)$ and its gradient.

Base classes

To complement the abstract base classes from Section 5.3.1 the software includes a number of base classes to help implement the required functions.

- `RBPFBase` Provides an implementation handling the Rao-Blackwellized case automatically by defining a new set of simpler functions that are required from the derived class.
- `RBPSBase` Extends `RBPFBase` to provide smoothing for Rao-Blackwellized models.

Model classes

These classes further specialize those from sections 5.3.1-5.3.2.

- `LTV` Handles Linear Time-Varying systems, the derived class only needs to provide callbacks for how the system matrices depend on time.

- `NLG` Nonlinear dynamics with additive Gaussian noise.

- `MixedNLGaussianSampled` Provides support for models of type (4) using an algorithm which samples the linear states in the backward simulation step. The sufficient statistics for the linear states are later recovered in a post processing step. See Lindsten and Schön (2011) for details. The derived class needs to specify how the linear and non-linear dynamics depend on time and the current estimate of ξ.

- `MixedNLGaussianMarginalized` Provides an implementation for models of type (4) that fully marginalizes the linear Gaussian states, resulting in a non-Markovian smoothing problem. See Lindsten, Bunch, Godsill, and Schon (2013) for details. The derived class needs to specify how the linear and non-linear dynamics depend on time and the current estimate of ξ. This implementation requires that $Q_{\xi z} = 0$.

- `Hierarchial` Provides a structure useful for implementing models of type (3) using sampling of the linear states in the backward simulation step. The sufficient statistics for the linear states are later recovered in a post processing step.

For the LTV and MLNLG classes the parameters estimation interfaces, `ParamEstInterface` and `ParamEstInterface_GradientSearch`, are implemented so that the end-user can specify the element-wise derivative for the matrices instead of directly calculating gradients of (7b)-(7d). Typically there is some additional structure to the problem, and it is then beneficial to override this generic implementation with a specialized one to reduce the computational effort by utilizing that structure.

5.4. Algorithms

RBPF

The Particle Filter implemented is summarized with pseudo-code in Algorithm 2. The predict step is detailed in Algorithm 3 and the measurement step in Algorithm 4. N_{eff} is the effective number of particles as defined in Arulampalam *et al.* (2002) and is used to trigger the resampling step when a certain predefined threshold is crossed.

Algorithm 2: (Rao-Blackwellized) Particle Filter

for $t \leftarrow 0$ **to** $T-1$ **do**
  for $i \leftarrow 1$ **to** N **do**
    Predict $x_{t+1|t} \leftarrow x_{t|t}$ using Alg. 3
    Update $x_{t+1|t+1} \leftarrow x_{t+1|t}, y_{t+1}$ using Alg. 4
    if $N_{eff} < N_{treshold}$ **then**
      Resample using Alg. 5

Algorithm 3: RBPF Predict, step 4-6 is only needed for the Rao-Blackwellized case

1. *Update system dynamics,* $f(x_t, v_t)$*, based on* ξ_t
2. *Sample process noise,* v_t
3. *Calculate* $\xi_{t+1|t}$ *using sampled* v_t
4. *Use knowledge of* $\xi_{t+1|t}$ *to update estimate of* z_t
5. *Update linear part of system dynamics with knowledge of* $\xi_{t+1|t}$
6. *Predict* $z_{t+1|t}$ *(conditioned on* $\xi_{t+1|t}$*)*

Algorithm 4: RBPF Measure, step 2 is only needed for the Rao-Blackwellized case

1. *Update system dynamics,* $h(x_t, e_t)$*, based on* $\xi_{t+1|t}$
2. *Calculate* $z_{t+1|t+1}$ *using* y_{t+1}
3. *Update weights* $w^{(i)}_{t+1|t+1} = w^{(i)}_{t+1|t} p(y_{t+1}|x_{t+1|t})$

RBPS

The main RBPS algorithm implemented in **pyParticleEst** is of the type JBS-RBPS with constrained RTS-smoothing from Lindsten *et al.* (2013). It simulates M backward trajectories using filtered estimates. During the backward simulation step it samples the linear/Gaussian states, but later recovers the sufficient statistics, i.e., the mean and covariance, by running a constrained RTS-smoother (Rauch, Striebel, and Tung 1965) conditioned on the nonlinear part of the trajectory. It can be combined with any of the backward simulation methods, for example FFBSi-RS or MH-FFBSi. The method is summarized in Algorithm 6.

Additionally for MLNLG models (4) there is another RBPS algorithm implemented which fully marginalizes the linear states, it is an implementation of the method described in Lindsten *et al.* (2013). This is the statistically correct way to solve the problem, and it gives better accuracy, but it also requires more computations resulting in a longer execution time. Due to the difficulty in evaluating $\text{argmax}_{x_{t+1:T}} p(x_{t+1:T}, y_{t+1:T}|x_{t|t}, z_{t|t}, P_{t|t})$ rejection sampling is not implemented, it is anyhow unlikely to perform well due the large dimension of target variables (since they are full trajectories, and no longer single time instances). The implementation is also limited to cases where the cross covariance ($Q_{\xi z}$) between the nonlinear and linear states is zero. This method is summarized in Algorithm 7.

Algorithm 5: The resampling algorithm used in the framework. Different resampling algorithms have been proposed in the literature, this one has the property that a particle, $x^{(i)}$, with $w^{(i)} \geq \frac{1}{N}$ is guaranteed to survive the resampling step.

$\tilde{\omega}_c = \text{cumsum}(\omega)$
$\omega_c = \tilde{\omega}_c / \sum \tilde{\omega}_c$
$u = ([0 : N-1] + U(0,1))/N$
for $k \leftarrow 1$ **to** N **do**
  $x^{(k)} = x^{(i)}, i = \underset{j}{\text{argmin}}\, \omega_c^{(j)} > u^{(k)}$

Algorithm 6: (Rao-Blackwellized) Particle Smoother

(0. Run RBPF generating filtered estimates)
Sample index i with probability $w_{T|T}^{(i)}$
Add $x_{T|T}^{(i)}$ to the backward trajectory
for $t \leftarrow T-1$ **to** 0 **do**
  Sample $\tilde{z}_{t+1}$ from $z_{t+1|T}^{(i)}$
  Sample index k with probability $w_{t|t}^{(k)} p(\xi_{t+1}^{(i)}, \tilde{z}_{t+1} | x_{t|t}^{(k)})$
  Update sufficent statistics of $z_t^{(k)}$ conditioned on $(\xi_{t+1}, \tilde{z}_{t+1})$
  Append $x_{t|T}^{(k)}$ to trajectory
  $i \leftarrow k$
Calculate dynamics for Rao-Blackwellized states conditioned on the non-linear trajectory $\xi_{0:T}$
Run constrained RTS-smoothing to recover sufficient statistics for $z_{0:T}$

Algorithm 7: Fully Marginalized Particle Smoother for MLNLG (4)

(0. Run RBPF generating filtered estimates)
Sample index i with probability $w_{T|T}^{(i)}$
Add $\xi_{T|T}^{(i)}$ to the backward trajectory
for $t \leftarrow T-1$ **to** 0 **do**
  Sample index k with probability $w_{t|t}^{(k)} p(\xi_{t+1:T}^{(i)}, y_{t+1:T} | \xi_{t|t}^{(k)}, z_{t|t}^{(k)}, P_{z\,t|t}^{(k)})$
  Append $\xi_{t|T}^{(k)}$ to trajectory
  $i \leftarrow k$
Calculate dynamics for Rao-Blackwellized states conditioned on the non-linear trajectory $\xi_{0:T}$
Run constrained RTS-smoothing to recover sufficient statistics for $z_{0:T}$

Parameter estimation

Parameter estimation is accomplished using an EM-algorithm as presented in Section 4.3. It requires that the derived particle class implements `ParamEstInterface`. The method is summarized in Algorithm 8. The maximization step in (6b) is performed using `scipy.optimize.minimize` with the `l-bfgs-b` method (Zhu, Byrd, Lu, and Nocedal 1997), which utilizes the analytic Jacobian when present.

Algorithm 8: RBPS+EM algorithm

(0. Initial parameter guess, θ_0)
for $i \leftarrow 1$ **to** *max_iter* **do**
  Estimate $p(x_{1:T} | y_{1:T}, \theta_{i-1})$ using Alg. 6 (RBPS)
  Approximate $Q(\theta, \theta_{i-1})$ using estimated trajetory
  Calculate $\theta_i = \underset{\theta}{\operatorname{argmax}}\, Q(\theta, \theta_{i-1})$

6. Example models

6.1. Integrator

A trivial example consisting of a linear Gaussian system

$$x_{t+1} = x_t + w_t \tag{8a}$$
$$y_t = x_t + e_t, \quad x_1 \sim N(0,1) \tag{8b}$$
$$w_t \sim N(0,1), \quad e_t \sim N(0,1) \tag{8c}$$

This model could be implemented using either the `LTV` or `NLG` model classes, but for this example it was decided to directly implement the required top level interfaces to illustrate how they work. In this example only the methods needed for filtering are implemented. To use smoothing the `logp_xnext` method would be needed as well. An example realization using this model was shown in Fig. 1

```
1  class Integrator(interfaces.ParticleFiltering):
       def __init__(self, P0, Q, R):
           self.P0 = numpy.copy(P0)
           self.Q = numpy.copy(Q)
           self.R = numpy.copy(R)
6
       def create_initial_estimate(self, N):
           return numpy.random.normal(0.0, self.P0, (N,)
                                      ).reshape((-1, 1))

11     def sample_process_noise(self, particles, u, t):
           N = len(particles)
           return numpy.random.normal(0.0, self.Q, (N,)
                                      ).reshape((-1, 1))

16     def update(self, particles, u, t, noise):
           particles += noise

       def measure(self, particles, y, t):
           logyprob = numpy.empty(len(particles))
21         for k in range(len(particles)):
               logyprob[k] = kalman.lognormpdf(particles[k, 0] - y,
                                               self.R)
           return logyprob
```

Line `08` Samples the initial particles from a zero-mean Gaussian distribution with variance `P0`.

Line `13` This samples the process noise at time `t`.

Line `16` Propagates the estimates forward in time using the noise previously sampled.

Line `19` Calculates the log-probability for the measurement y_t for particles $x_t^{(i)}$.

6.2. Standard nonlinear model

This is a model that is commonly used as an example when demonstrating new algorithms, see e.g., Lindsten and Schön (2013), Arulampalam *et al.* (2002) and Briers *et al.* (2010)

$$x_{t+1} = 0.5x_t + 25\frac{x_t}{1+x_t^2} + 8\cos 1.2t + w_t \tag{9a}$$
$$y_t = 0.05x_t^2 + e_t, \quad x_1 \sim N(0,5) \tag{9b}$$
$$w_t \sim N(0,10), \quad e_t \sim N(0,1) \tag{9c}$$

For the chosen noise covariances the filtering distribution is typically multi-modal whereas the smoothing distribution is mostly unimodal. Figure 4 shows an example realization from this model, the smoothed estimates have been calculated using backward simulation with rejection sampling using adapative stopping (FFBSi-RSAS).

The corresponding model definition exploits that this is a model of the type Nonlinear Gaussian, and thus inherits the base class for that model type.

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

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

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

Line 03 In this example the covariance matrices are time-invariant and can thus be set in the constructor. This also allows the base class to later perform optimization where the fact that the matrices are identical for all particles can be exploited.

Line 05 `calc_g` utilizes that all the particles are stored in an array to effectively evaluate $g_t(x_t^{(i)})$ for all particles in a single method call.

Line 08 `calc_f` evaluates $f_t(x_t^{(i)})$ in a similar fashion as above.

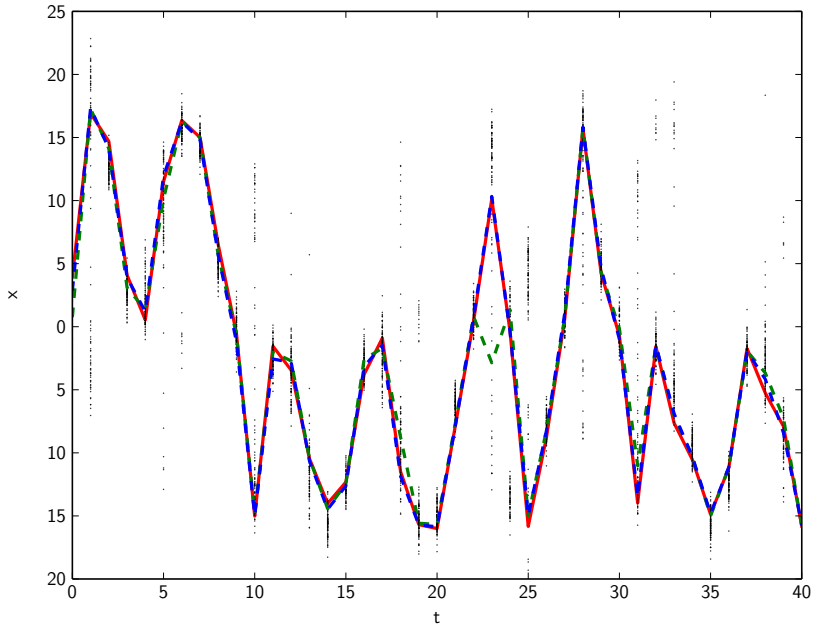

Figure 4: Example realization using the standard nonlinear model. The solid red line is the true trajectory. The black points are the filtered particle estimates forward in time, the green dashed line is the mean value of the filtered estimates, the blue dashed line is the mean value of the smoothed trajectories. The smoothing was performed using the BSi RSAS algorithm. Notice that the filtered mean does not follow the true state trajectory due to the multi-modality of the distribution, whereas the smoothed estimate does not suffer from this problem.

6.3. Lindsten and Schön, Model B

This model was introduced in Lindsten and Schön (2011) as an extension to the standard nonlinear model from Section 6.2. It replaces the constant 25 by the output of a fourth order linear system.

$$\xi_{t+1} = 0.5\xi_t + \theta_t \frac{\xi_t}{1+\xi_t^2} + 8\cos 1.2t + v_{\xi,t} \tag{10a}$$

$$z_{t+1} = \begin{pmatrix} 3 & -1.691 & 0.849 & -0.3201 \\ 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \end{pmatrix} z_t + v_{z,t} \tag{10b}$$

$$y_t = 0.05\xi_t^2 + e_t \tag{10c}$$

$$\theta_t = 25 + \begin{pmatrix} 0 & 0.04 & 0.044 & 0.008 \end{pmatrix} z_t \tag{10d}$$

$$\xi_0 = 0, \quad z_0 = \begin{pmatrix} 0 & 0 & 0 & 0 \end{pmatrix}^T \tag{10e}$$

Since this model conforms to the class from (4) it was implemented using the `MLNLG` base class. Doing so it only requires the user to define the functions and matrices as a function of the current state. The corresponding source code is listed below.

```
class ParticleLSB(mlnlg.MixedNLGaussianMarginalizedInitialGaussian):
    def __init__(self):
        xi0 = numpy.zeros((1, 1))
        z0 = numpy.zeros((4, 1))
        P0 = numpy.zeros((4, 4))

        Az = numpy.array([[3.0, -1.691, 0.849, -0.3201],
                          [2.0, 0.0, 0.0, 0.0],
                          [0.0, 1.0, 0.0, 0.0],
                          [0.0, 0.0, 0.5, 0.0]])

        Qxi = numpy.diag([ 0.005])
        Qz = numpy.diag([ 0.01, 0.01, 0.01, 0.01])
        R = numpy.diag([0.1, ])

        super(ParticleLSB, self).__init__(xi0=xi0, z0=z0,
                                          Pz0=P0, Az=Az,
                                          R=R, Qxi=Qxi,
                                          Qz=Qz,)

    def get_nonlin_pred_dynamics(self, particles, u, t):
        tmp = numpy.vstack(particles)[:, numpy.newaxis, :]
        xi = tmp[:, :, 0]

        Axi = (xi / (1 + xi ** 2)).dot(C_theta)
        Axi = Axi[:, numpy.newaxis, :]

        fxi = (0.5 * xi +
               25 * xi / (1 + xi ** 2) +
               8 * math.cos(1.2 * t))
        fxi = fxi[:, numpy.newaxis, :]

        return (Axi, fxi, None)

    def get_meas_dynamics(self, particles, y, t):
        if (y != None):
            y = numpy.asarray(y).reshape((-1, 1)),
        tmp = 0.05 * particles[:, 0] ** 2
        h = tmp[:, numpy.newaxis, numpy.newaxis]

        return (y, None, h, None)
```

Line 02 In the constructor all the time-invariant parts of the model are set

Line 21 This function calculates $A_\xi(\xi_t^{(i)})$, $f_\xi(\xi_t^{(i)})$ and $Q_\xi(\xi_t^{(i)})$

Line 26 The array is resized to match the expected format (The first dimension indices the particles, each entry being a two-dimensional matrix)

Line 33 Return a tuple containing A_ξ, f_ξ and Q_ξ arrays. Returning `None` for any element in the tuple indicates that the time-invariant values set in the constructor should be used

Line 36 This function works in the same way as above, but instead calculates $h(\xi_t)$, $C(\xi_t)$ and $R(\xi_t)$. The first value in the returned tuple should be the (potentially preprocessed) measurement.

7. Results

The aim of this section is to demonstrate that the implementation in **pyParticleEst** is correct by reproducing results previously published elsewhere.

7.1. Rao-Blackwellized particle filtering / smoothing

Here Example B from Lindsten and Schön (2011) is reproduced, it uses the model definition from Section 6.3 the marginalized base class for MLNLG models. The results are shown in Table 4 which also contains the corresponding values from Lindsten and Schön (2011). The values were calculated by running the RBPS-agorithm on 1000 random realizations of model (10) using 300 particles and 50 smoothed trajectories. The smoothed trajectories where averaged to give a point estimate for each time step. The average was used to calculated the RMSE for a single realization. The values in this article were computed using the marginalized `MLNLG` base class, which uses the smoothing algorithm presented in Lindsten *et al.* (2013). This is a later improvement to the algorithm used in the original article, which explains why the values presented here are better than those in Lindsten and Schön (2011). The mean RMSE is also highly dependent on the particular realizations, 89.8% of the realizations have a lower RMSE than the average, whereas 3.3% have an RMSE greater than 1.0. This also makes a direct comparison of the values problematic since the exact amount of outliers in the dataset will have a significant impact om the average RMSE.

RMSE	ξ	θ
pyParticleEst	0.275	0.545
Lindsten & Schön	0.317	0.585

Table 4: Root mean squared error (RMSE) values for ξ and θ from model 10, compared with those presented in Lindsten and Schön (2011).

7.2. Parameter estimation in MLNLG

In Lindsten and Schön (2010) the following model is introduced

$$\xi_{t+1} = \theta_1 \arctan \xi_t + \begin{pmatrix} \theta_2 & 0 & 0 \end{pmatrix} z_t + v_{\xi,t} \tag{11a}$$

$$z_{t+1} = \begin{pmatrix} 1 & \theta_3 & 0 \\ 0 & \theta_4 \cos\theta_5 & -\theta_4 \sin\theta_5 \\ 0 & \theta_4 \sin\theta_5 & \theta_4 \cos\theta_5 \end{pmatrix} z_t + v_{z,t} \tag{11b}$$

$$y_t = \begin{pmatrix} 0.1\xi_t^2 \operatorname{sgn}(\xi_t) \\ 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 1 & -1 & 1 \end{pmatrix} z_t + e_t \tag{11c}$$

The task presented is to identify the unknown parameters, θ_i. Duplicating the conditions as presented in the original article, but running the algorithm on 160 random data realizations instead of 70, gives the results presented in Table 5. The authors of Lindsten and Schön (2010) do not present the number of smoothed trajectories used in their implementation, for the results in this article 5 smoothed trajectories were used.

	True value	Lindsten & Schön	pyParticleEst	pyParticleEst*
θ_1	1	0.966 ± 0.163	0.981 ± 0.254	1.006 ± 0.091
θ_2	1	1.053 ± 0.163	0.947 ± 0.158	0.984 ± 0.079
θ_3	0.3	0.295 ± 0.094	0.338 ± 0.308	0.271 ± 0.112
θ_4	0.968	0.967 ± 0.015	0.969 ± 0.032	0.962 ± 0.017
θ_5	0.315	0.309 ± 0.057	0.263 ± 0.134	0.312 ± 0.019

Table 5: Results presented by Lindsten and Schön in Lindsten and Schön (2010) compared to results calculated using *pyParticleEst*. The column marked with * are the statistics when excluding those realizations where the EM-algorithm was stuck in local maxima for θ_5.

Looking at the histogram of the estimate of θ_5 shown in Figure 5 it is clear that there are several local maxima. Of the 160 realizations 21 converged to a local maximum for θ_5 thus giving an incorrect solution. This is typically handled by solving the optimization problem using several different initial conditions and choosing the one with the maximum likelihood. However since that does not appear to have been performed in Lindsten and Schön (2010) it is problematic to compare the values obtained, since they will be highly dependent on how many of the realizations that converged to local maxima. Therefore Table 5 contains a second column named pyParticleEst* which presents the same statistics but excluding those realizations where θ_5 converged to a local maxima.

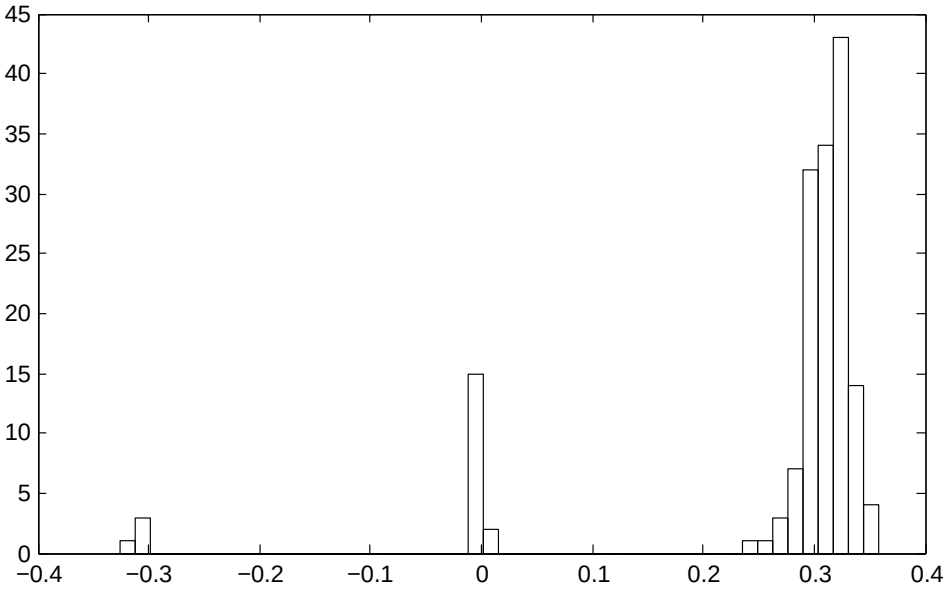

Figure 5: Histogram for θ_5. The peaks around -0.3 and 0 are likely due to the EM-algorithm converging to local maxima. Since θ_5 enters the model through $\sin\theta_5$ and $\cos\theta_5$, with cos being a symmetric function the peak around -0.3 could intuitively be expected.

8. Conclusion

pyParticleEst lowers the barrier of entry to the field of particle methods, allowing many problems to be solved with significantly less implementation effort compared to starting from scratch. This was exemplified by the models presented in Section 6, demonstrating the significant reduction in the amount of code needed to be produced by the end-user. Its use for grey-box identification was demonstrated in Section 7.2. The software and examples used in this article can be found at Nordh (2013).

There is an overhead due to the generic design which by necessity gives lower performance compared to a specialized implementation in a low-level language. For example a hand optimized C-implementation that fully exploits the structure of a specific problem will always be faster, but also requires significantly more time and knowledge from the developer. Therefore the main use-case for this software when it comes to performance critical applications is likely to be prototyping different models and algorithms that will later be re-implemented in a low-level language. That implementation can then be validated against the results provided by the generic algorithms. In many circumstances the execution time might be of little concern and the performance provided using **pyParticleEst** will be sufficient. There are projects such as **Numba** (Continuum Analytics 2014), **Cython** (Behnel, Bradshaw, and Seljebotn 2009) and **PyPy** (Rigo 2004) that aim to increase the efficiency of Python-code. **Cython** is already used for some of the heaviest parts in the framework. By selectively moving more of the computationally heavy parts of the model base classes to **Cython** it should be possible to use the framework directly for many real-time applications.

For the future the plan is to extend the framework to contain more algorithms, for example the interesting field of PMCMC methods (Del Moral, Doucet, and Jasra 2006). Another interesting direction is smoothing of non-Markovian models as examplified by the marginalized smoother for MLNLG models. This type of smoother could also be combined with Gaussian processes as shown by Lindsten and Schön (2013). The direction taken by e.g., Murray (In review) with a high level language is interesting, and something that might be worthwhile to implement for automatically generating the Python code describing the model, providing a further level of abstraction for the end user.

Acknowledgments

The author is a member of the LCCC Linnaeus Center and the eLLIIT Excellence Center at Lund University. The author would like to thank professor Bo Bernhardsson, Lund University, and professor Thomas Schön, Uppsala University, for the feedback provided on this work.

References

Andrieu C, Doucet A, Holenstein R (2010). "Particle Markov Chain Monte Carlo Methods." *Journal of the Royal Statistical Society B (Statistical Methodology)*, **72**(3), 269–342.

Arulampalam M, Maskell S, Gordon N, Clapp T (2002). "A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesian Tracking." *IEEE Trans. Signal Process.*, **50**(2), 174–188. ISSN 1053-587X.

Behnel S, Bradshaw RW, Seljebotn DS (2009). "Cython Tutorial." In G Varoquaux, S van der Walt, J Millman (eds.), *Proceedings of the 8th Python in Science Conference*, pp. 4 – 14. Pasadena, CA USA.

Beskos A, Crisan D, Jasra A (2011). "On the Stability of Sequential Monte Carlo Methods in Hgh Fimensions." *arXiv preprint arXiv:1103.3965*.

Briers M, Doucet A, Maskell S (2010). "Smoothing Algorithms for State-Space Models." *Annals of the Institute of Statistical Mathematics*, **62**(1), 61–89. ISSN 0020-3157. doi:10.1007/s10463-009-0236-2. URL http://dx.doi.org/10.1007/s10463-009-0236-2.

Bunch P, Godsill S (2013). "Improved Particle Approximations to the Joint Smoothing Distribution using Markov Chain Monte Carlo." *Signal Processing, IEEE Transactions on*, **61**(4), 956–963.

Continuum Analytics (2014). *Numba, Version 0.14*. URL http://numba.pydata.org/numba-doc/0.14/index.html.

Del Moral P, Doucet A, Jasra A (2006). "Sequential Monte Carlo Samplers." *Journal of the Royal Statistical Society B (Statistical Methodology)*, **68**(3), 411–436.

Dempster AP, Laird NM, Rubin DB (1977). "Maximum Likelihood from Incomplete Data via the EM Algorithm." *Journal of the Royal Statistical Society B*, **39**(1), 1–38.

Doucet A, Godsill S, Andrieu C (2000). "On Sequential Monte Carlo Sampling Methods for Bayesian Filtering." *Statistics and Computing*, **10**(3), 197–208. ISSN 0960-3174. doi:10.1023/A:1008935410038. URL http://dx.doi.org/10.1023/A%3A1008935410038.

Dubarry C, Douc R (2011). "Particle Approximation Improvement of the Joint Smoothing Distribution with On-the-Fly Variance Estimation." *arXiv preprint arXiv:1107.5524*.

FSF (1999). "The GNU Lesser General Public License." See http://www.gnu.org/copyleft/lesser.html.

Jones E, Oliphant T, Peterson P, *et al.* (2001–). "SciPy: Open Source Scientific Tools for Python." URL http://www.scipy.org/.

Julier S, Uhlmann J (2004). "Unscented Filtering and Nonlinear Estimation." *Proceedings of the IEEE*, **92**(3), 401–422. ISSN 0018-9219. doi:10.1109/JPROC.2003.823141.

Lindsten F, Bunch P, Godsill SJ, Schon TB (2013). "Rao-Blackwellized Particle Smoothers for Mixed Linear/Nonlinear State-Space Models." In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 6288–6292. IEEE.

Lindsten F, Schön T (2010). "Identification of Mixed Linear/Nonlinear State-Space Models." In *Decision and Control (CDC), 2010 49th IEEE Conference on*, pp. 6377–6382. ISSN 0743-1546. doi:10.1109/CDC.2010.5717191.

Lindsten F, Schön T (2011). "Rao-Blackwellized Particle Smoothers for Mixed Linear/Nonlinear State-Space Models." *Technical report*. URL http://user.it.uu.se/~thosc112/pubpdf/lindstens2011.pdf.

Lindsten F, Schön TB (2013). "Backward Simulation Methods for Monte Carlo Statistical Inference." *Foundations and Trends® in Machine Learning*, **6**(1), 1–143. ISSN 1935-8237. doi:10.1561/2200000045. URL http://dx.doi.org/10.1561/2200000045.

Mannesson A (2013). "Joint Pose and Radio Channel Estimation." *Licentiate thesis*, Dept. of Automatic Control, Lund University, Sweden.

Montemerlo M, Thrun S, Koller D, Wegbreit B, *et al.* (2002). "FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem." In *AAAI/IAAI*, pp. 593–598.

Murray LM (In review). "Bayesian State-Space Modelling on High-Performance Hardware Using LibBi." URL http://arxiv.org/abs/1306.3277.

Nordh J (2013). "pyParticleEst." URL http://www.control.lth.se/Staff/JerkerNordh/pyparticleest.html.

Okuma K, Taleghani A, De Freitas N, Little JJ, Lowe DG (2004). "A Boosted Particle Filter: Multitarget Detection and Tracking." In *Computer Vision-ECCV 2004*, pp. 28–39. Springer-Verlag.

Oliphant TE (2007). "Python for Scientific Computing." *Computing in Science Engineering*, **9**(3), 10–20. ISSN 1521-9615. doi:10.1109/MCSE.2007.58.

Rauch HE, Striebel CT, Tung F (1965). "Maximum Likelihood Estimates of Linear Dynamic Systems." *Journal of the American Institute of Aeronautics and Astronautics*, **3**(8), 1445–1450.

Rebeschini P, van Handel R (2013). "Can Local Particle Filters Beat the Curse of Dimensionality?" *arXiv preprint arXiv:1301.6585*.

Rigo A (2004). "Representation-Based Just-in-Time Specialization and the Psyco Prototype for Python." In *Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 15–26. ACM, Verona, Italy. ISBN 1-58113-835-0. doi:10.1145/1014007.1014010. URL http://portal.acm.org/citation.cfm?id=1014010.

Schön T, Gustafsson F, Nordlund PJ (2005). "Marginalized Particle Filters for Mixed Linear/Nonlinear State-Space Models." *Signal Processing, IEEE Transactions on*, **53**(7), 2279–2289. ISSN 1053-587X. doi:10.1109/TSP.2005.849151.

Taghavi E, Lindsten F, Svensson L, Schön T (2013). "Adaptive Stopping for Fast Particle Smoothing." In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 6293–6297. ISSN 1520-6149. doi:10.1109/ICASSP.2013.6638876.

Zhu C, Byrd RH, Lu P, Nocedal J (1997). "Algorithm 778: L-BFGS-B: Fortran Subroutines for Large-Scale Bound-Constrained Optimization." *ACM Transactions on Mathematical Software (TOMS)*, **23**(4), 550–560.

Affiliation:

Jerker Nordh
Department of Automatic Control
Lund University
Box 118, SE-221 00 Lund, Sweden
E-mail: jerker.nordh@control.lth.se
URL: http://www.control.lth.se/Staff/JerkerNordh/

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume VV, Issue II *Submitted:* yyyy-mm-dd
MMMMMM YYYY *Accepted:* yyyy-mm-dd