The Journal of Open Source Software

A Short Introduction to PF: A C++ Library for Particle Filtering

Taylor R. Brown[1]

1 Department of Statistics, University of Virginia, PO Box 400135, Charlottesville, VA 22904, USA

DOI: 10.21105/joss.02559

Software

- Review
- Repository
- Archive

Editor: Patrick Diehl

Submitted: 08 August 2020
Published: 10 August 2020

Summary

The `PF` library provides **class and function templates** that offer fast implementations for a variety of particles filtering algorithms. Each of these algorithms are useful for a wide range of time series models.

In this library, each available particle filtering algorithm is provided as an abstract base class template. Once the data analyst has a specific state-space or hidden Markov model in mind, she will pick which type(s) of particle filtering algorithm(s) to associate with that model by including the appropriate header file. For each model-particle filter pair, she will write a class template for his model that inherits from the particle filter's base class template.

The details of each algorithm are abstracted away, and each algorithm's class template's required functions are pure virtual methods, meaning that the data analyst will not be able to omit any function that is required by the algorithm.

This is by no means the first `C++` library to be offered that provides particle filter implementations. Other options include `LibBi` (Murray, 2013), `Biips` (Todeschini, Caron, Fuentes, Legrand, & Del Moral, 2014), `SMCTC` (Johansen, 2009) and `Nimble` (Valpine et al., 2017). The goals of these software packages are different, though–users of these libraries write their models in a scripting language, and that model file gets parsed into `C++` code. This library is designed for users that prefer to work in `C++` directly.

Statement of Need

It takes time and effort to implement different particle filters well, and this is true for two reasons. First, the mathematical notation used to describe them can be complex. The second reason is that, even if they are correctly implemented, they can be quite slow, limiting the number of tasks that they would be feasible for. This library attempts to provide speed and abstraction to mitigate these two difficulties.

Additionally, this software is designed in an object-oriented manner. This allows for individual particle filters to be used, and it facilitates the implementation of algorithms that update many particle filters, possibly in parallel. Some examples of opportunities include particle filters with parallelized resampling schemes (Bolic, Djuric, & Sangjin Hong, 2005, p. 1309.2918), particle Markov chain Monte Carlo algorithms (Andrieu, Doucet, & Holenstein, 2010), importance sampling "squared" (Tran, Scharth, Pitt, & Kohn, 2013), and the particle swarm algorithm (Brown, 2020).

Finally, this library is "header-only." As a result, building your `C++` project is as simple as possible. The only required steps are `#include`-ing relevant headers, and pointing the compiler at the `include/pf/` directory. This directory stores all necessary code, although there there are unit tests and examples provided as well.

Example

A fully-worked example is provided along with this software available at https://github.com/tbrown122387/pf. This example considers modeling a financial time series with a simple stochastic volatility model (Taylor, 1982) with three parameters: β, ϕ and σ. For this model, the observable rate of return y_t is normally distributed after conditioning on the contemporaneous state random variable x_t. The mean parameter of this normal distribution will remain fixed at 0. However, the scale of this distribution will vary with the evolving x_t. When x_t is relatively high, the returns will have a high conditional variance and be "volatile." When x_t is low, the returns will be much less volatile.

The observation equation is

$$y_t = \beta e^{x_t/2} z_t \tag{1}$$

where $\{z_t\}_{t=0}^T$ are independent and identically distributed (iid) normal random variates.

The state evolves randomly through time as an autoregressive process of order one:

$$x_t = \phi x_{t-1} + \sigma z_t'. \tag{2}$$

The collection $\{z_t'\}_{t=1}^T$ are also assumed to be iid normal random variates. At time 1, we assume the first state follows a mean zero normal distribution with $\sigma^2/(1-\phi^2)$. For simplicity, all of our proposal distributions are chosen to be the same as the state transitions.

The file `examples/svol_sisr.h` provides a fully-worked example of writing a class template called `svol_sisr` for this model-algorithm pair. Any model-algorithm pair will make use of a resampler type (found in `include/pf/resamplers.h`), sampler functions (found in `include/pf/rv_samp.h`), and density evaluator functions (found in `include/pf/rv_eval.h`). Because you are writing a class template instead of a class, the decision of what to pass in as template parameters will be pushed back to the instantiation site. These template parameters are often changed quite frequently, so this design allows them to be changed only in one location of the project. Instantiating an object after the class template has been written is much easier than writing the class template itself. Just provide the template parameters and the constructor parameters in the correct order. For example,

```
using Mod = svol_sisr<5000,1,1,mn_resampler<5000,1,double>,double>;
Mod sisrsvol(.91,.5,1.0);
```

instantiates the object called `sisrsvol`, which performs the SISR algorithm for the stochastic volatility model using multinomial sampling on $5,000$ particles. It sets $\phi = .95$, $\beta = .5$ and $\sigma = 1$.

As (possibly real-time, streaming) data becomes available, updating the model is accomplished with the `filter` method. The call at each time point would look something like

```
sisrsvol.filter(yt);
```

References

Andrieu, C., Doucet, A., & Holenstein, R. (2010). Particle markov chain monte carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, *72*(3), 269–342. doi:10.1111/j.1467-9868.2009.00736.x

Bolic, M., Djuric, P. M., & Sangjin Hong. (2005). Resampling algorithms and architectures for distributed particle filters. *IEEE Transactions on Signal Processing*, *53*(7), 2442–2450. doi:10.1109/TSP.2005.849185

Brown, T. R. (2020). Approximating posterior predictive distributions by averaging output from many particle filters.

Johansen, A. M. (2009). SMCTC: Sequential monte carlo in C++. *Journal of Statistical Software*, *30*(6), 1–41. Retrieved from http://www.jstatsoft.org/v30/i06

Murray, L. M. (2013). Bayesian state-space modelling on high-performance hardware using LibBi.

Taylor, S. (1982). Financial returns modelled by the product of two stochastic processes, a study of daily sugar prices 1961-79, *1*.

Todeschini, A., Caron, F., Fuentes, M., Legrand, P., & Del Moral, P. (2014). Biips: Software for Bayesian inference with interacting particle systems. *arXiv preprint arXiv:1412.3779*.

Tran, M.-N., Scharth, M., Pitt, M., & Kohn, R. (2013). Importance sampling squared for bayesian inference in latent variable models. *SSRN Electronic Journal*. doi:10.2139/ssrn.2386371

Valpine, P. de, Turek, D., Paciorek, C. J., Anderson-Bergman, C., Lang, D. T., & Bodik, R. (2017). Programming with models: Writing statistical algorithms for general model structures with NIMBLE. *Journal of Computational and Graphical Statistics*, *26*(2), 403–413. doi:10.1080/10618600.2016.1172487