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.02599

Software

- Review
- Repository
- Archive

Editor: Patrick Diehl

Reviewers:

- @ziotom78
- @andremrsantos

Submitted: 08 August 2020
Published: 09 October 2020

Summary

The `PF` library provides **class and function templates** that offer fast implementations for a variety of particle filtering algorithms. Each of these algorithms is 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 her 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), `Nimble` (Valpine et al., 2017) and `SMCTC` (Johansen, 2009). The goals of most of these software packages are different, though–users of these libraries typically write their models in a scripting language, and that model file gets parsed into `C++` code. This library, on the other hand, is designed for users that prefer to work in `C++` directly.

Statement of Need

State-space models describe a partially-observed Markov chain $\{(x_t, y_t)\}_{t \geq 1}$ that possesses a hidden/latent variable at each time point (denoted by x_t), as well as an observed variable (denoted by y_t). "Filtering" is defined as obtaining the distribution of each unobserved state/code random variable, conditioning on all of the observed information up to that point in time. In other words, it is the task of finding

$$p(x_t \mid y_t, y_{t-1}, \ldots, y_1). \tag{1}$$

"Particle filters" are a class of algorithms that approximate this sequence of distributions with weighted samples (termed particles). Filtering is a useful tool for a variety of applications in a variety of fields, and it should also be mentioned that they can be used for real-time forecasting, and they are critical component of more advanced parameter estimation algorithms.

Unfortunately, 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 in isolation, and it also facilitates the implementation of more complicated algorithms that update many particle filters through many iterations in a stateful way, possibly in parallel. For this second class of algorithms, there are usually two necessary loops: the "outer" loop that loops over time, and the "inner" loop that iterates over each distinct particle filter. Without an object-oriented design, there would be a third loop, which loops over all particle samples in each particle filter. Some examples of algorithms that run many particle filters within the overall algorithm are 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." This will allow some users to build their own C++ project by `#include`-ing relevant headers from this library and pointing the compiler at the `include/pf/` directory (this is "Option 2" described in the `README.md` file). On the other hand, other uses will prefer to use the `CMake` build procedure (this is "Option 1" in the `README.md` file), as this will help build the included unit tests and more closely follows the style of the provided example code projects.

Examples

A fully-worked example is provided with this software and is made available in the `examples/` directory in the Github repository. This example considers modeling a financial time series with a simple stochastic volatility model (Taylor, 2018) 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

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

where $\{z_t\}_{t=1}^T$ are independent and identically distributed (iid) normal random variates.

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

$$x_t = \phi x_{t-1} + \sigma z_t'. \tag{3}$$

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: $x_1 = sigma/\sqrt{1-\phi^2} z_1'$. For simplicity, all of our proposal distributions are chosen to be the same as the state transitions.

The file `examples/svol_sisr.h` provides an 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 $\phi = .91$, $\beta = .5$ 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);
```

The repository also provides an implementation of the "almost constant velocity model" that is comparable to the one described in section 5.1 of (Johansen, 2009). This can be found in the `smctc_comparison_example/` directory. Comparing the two implementations, ours makes use of more modern and higher-level C++ types and features and is slightly faster at first glance. Our implementation is written to match as many aspects of the one provided in SMCTC sample code as possible: both use a bootstrap filter proposal distribution, both only retain the most recent particles instead of the entire particle trajectory, both use the same parameters, both iterate across the same data set, and both use $100,000$ particles. Both programs were run on Ubuntu 18.04 with an Intel(R) Xeon(R) CPU E3-1241 v3 @ 3.50GHz chip. Ours ran in 2.883 seconds, while the SMCTC-provided implementation ran in 5.605 seconds. However, this test is far from conclusive: these packages make use of different random number generators, different data-reading functions, and different build procedures.

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. *arXiv preprint arXiv:2006.15396*.

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. *arXiv preprint arXiv:1306.3277*.

Taylor, S. (2018). Financial Returns Modelled by the Product of Two Stochastic Processes, A Study of Daily Sugar Prices. In T. Andersen & T. Bollerslev (Eds.), *Volatility*, The international library of critical writings in economics (pp. 423–446). Edward Elgar. ISBN: 9781788110617

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