Journal of Statistical Software

November 2021, Volume 100, Issue 3. *doi: 10.18637/jss.v100.i03*

Sequential Monte Carlo Methods in the nimble and nimbleSMC R Packages

Nicholas Michaud
University of California,
Berkeley

Perry de Valpine
University of California,
Berkeley

Daniel Turek
Williams College

Christopher J. Paciorek
University of California,
Berkeley

Dao Nguyen
University of Mississippi

Abstract

nimble is an R package for constructing algorithms and conducting inference on hierarchical models. The **nimble** package provides a unique combination of flexible model specification and the ability to program model-generic algorithms. Specifically, the package allows users to code models in the BUGS language, and it allows users to write algorithms that can be applied to any appropriate model. In this paper, we introduce the **nimbleSMC** R package. **nimbleSMC** contains algorithms for state-space model analysis using sequential Monte Carlo (SMC) techniques that are built using **nimble**. We first provide an overview of state-space models and commonly-used SMC algorithms. We then describe how to build a state-space model in **nimble** and conduct inference using existing SMC algorithms within **nimbleSMC**. SMC algorithms within **nimbleSMC** currently include the bootstrap filter, auxiliary particle filter, ensemble Kalman filter, IF2 method of iterated filtering, and a particle Markov chain Monte Carlo (MCMC) sampler. These algorithms can be run in R or compiled into C++ for more efficient execution. Examples of applying SMC algorithms to linear autoregressive models and a stochastic volatility model are provided. Finally, we give an overview of how model-generic algorithms are coded within **nimble** by providing code for a simple SMC algorithm. This illustrates how users can easily extend nimble's SMC methods in high-level code.

Keywords: particle filtering, sequential Monte Carlo, auxiliary particle filter, IF2 iterated filtering, ensemble Kalman filter, particle MCMC, R, **nimbleSMC**, **nimble**.

1. Introduction

State-space models provide a framework for analyzing time-series data, where observations are assumed to be noisy measurements of unobserved latent states that evolve over time (Durbin and Koopman 2012). State-space models have been used in such diverse fields as population ecology (Knape and de Valpine 2012), epidemiology (Andersson and Britton 2000), economics (Fernández-Villaverde and Rubio-Ramírez 2007), and meteorology (Wikle, Milliff, Herbei, and Leeds 2013). With the broad applicability of state-space models has come a variety of methods for conducting inference. One common goal is to determine the filtering distribution of the model, that is, the distribution of the most recent latent states given data. A second goal lies in estimating the likelihood of the model by integrating over the latent state dimensions, which can in turn be used to obtain maximum likelihood or Bayesian estimates of top-level parameters. State-space models are also frequently used for forecasting and for estimating the smoothing distribution, which is the distribution of all previous latent states given the data.

For linear, Gaussian state-space models, the Kalman filter gives an exact solution to the above problems. However, for more general state-space models, analytical solutions are usually unavailable. Modifications such as the extended Kalman filter (Anderson and Moore 1979), which uses linearization to approximate the filtering distribution, have been applied to non-linear, non-Gaussian models but can yield inaccurate results. More recently, it has become common to use a set of flexible computational algorithms known as sequential Monte Carlo (SMC) methods (Doucet, de Freitas, and Gordon 2001).

SMC methods, or variations thereof, are attractive as they provide a general framework for conducting inference on any state-space model. SMC algorithms estimate the filtering distribution using sequential importance sampling (SIR, Doucet *et al.* 2001), although different versions differ in their details. Generally speaking, SMC algorithms proceed by tracking a set of particles through time, where each particle represents a sample from the filtering distribution of the latent state. When a new data point is received, the particles are updated via SIR to represent the filtering distribution given the most current data. In this manner, SMC methods can be used to perform "on-line" inference. "On-line" inference is inference on filtering and forecasting distributions that can be updated iteratively as more data are received, without the need to re-process previously-received data. In contrast, "off-line" methods such as MCMC must be re-run with the entire set of data each time a new data point is obtained. SMC methods are nevertheless also of interest in "off-line" problems when MCMC or other approaches present difficulties.

A variety of SMC methods currently exist, including the bootstrap filter (Gordon, Salmond, and Smith 1993), auxiliary particle filter (Pitt and Shephard 1999), Liu-West filter (Liu and West 2001), Storvik filter (Storvik 2002), particle learning algorithm (Carvalho, Johannes, Lopes, and Polson 2010), ensemble Kalman filter (Evensen 2003), and others. In addition, algorithms such as particle MCMC (PMCMC, Andrieu, Doucet, and Holenstein 2010) have been developed that place SMC methods within a broader MCMC framework. SMC algorithms are also a foundation for maximum likelihood estimation such as by the IF2 version of iterated filtering (Ionides, Nguyen, Atchadé, Stoev, and King 2015). SMC algorithms have also been used to conduct approximate Bayesian computation (Del Moral, Doucet, and Jasra 2012).

The generality of SMC methods makes them well-suited for implementation within the **nim-**

bleSMC and **nimble** (de Valpine, Turek, Paciorek, Anderson-Bergman, Lang, and Bodik 2017) software packages for R (R Core Team 2021). Package **nimble** (de Valpine *et al.* 2021) is available from the Comprehensive R Archive Network (CRAN) at https://CRAN.R-project.org/package=nimble. **nimble** adopts and extends a dialect of the hierarchical modeling language used by **WinBUGS**, **OpenBUGS**, and **JAGS** (Lunn, Thomas, Best, and Spiegelhalter 2000; Lunn, Jackson, Best, Thomas, and Spiegelhalter 2012; Plummer 2003). These models can then be analyzed using **nimble**'s library of model-generic algorithms. Additionally, **nimble** provides a programming system embedded in R for users to write their own model-generic algorithms. These algorithms can be run in R or compiled via automatically generated C++ (Stroustrup 2013) for more efficient execution.

nimbleSMC is an R package built using **nimble**'s programming system that implements SMC algorithms. Package **nimbleSMC** (Michaud, de Valpine, Paciorek, and Turek 2020) is available from CRAN at https://CRAN.R-project.org/package=nimbleSMC. In this paper, **nimbleSMC**'s SMC algorithms are described in detail. Because they are all written in **nimble**'s algorithm programming system, they can be readily inspected and modified by users, much the way many base R functions are written in R. **nimble** also has a variety of MCMC algorithms for more general Bayesian inference, as well as an MCEM (Monte Carlo expectation-maximization) algorithm.

In Section 2, we provide a more detailed comparison between **nimble**, **nimbleSMC**, and other packages that can be used for state-space and more general hierarchical modeling. In Section 3, we introduce state-space models and the idea of filtering distributions. We then describe a variety of algorithms that can be used for inference on filtering distributions. In Section 4, we provide examples of writing state-space models within **nimble**. Section 5 demonstrates inference for a linear autoregressive state-space model with fixed parameters. Section 6 gives examples of inference on top-level parameters using PMCMC for a stochastic volatility model and using IF2 for a linear random-walk state-space model. In both cases, the examples enable verification of correct behavior as well as exploration of sensitivity to key tuning parameters of each method. In Section 7, we demonstrate **nimble**'s programmability by coding an SMC algorithm within **nimble**'s algorithm programming system.

2. Comparison to other software packages

nimbleSMC can be used to conduct inference on hierarchical models via built-in SMC methods. Additionally, **nimble** can conduct inference via MCMC methods and other applicable built-in algorithms such as ascent-based Monte Carlo expectation-maximization (Caffo, Jank, and Jones 2005). **nimble** can also be used as a tool that allows users to write functions in an R-like domain-specific language for model-generic algorithms, which we refer to as its algorithm programming system. These can then be compiled and run in C++ for increased speed. In Section 7 we demonstrate creating a user-defined SMC algorithm written in **nimble**'s algorithm programming system.

nimbleSMC offers a suite of sequential Monte Carlo algorithms that can be applied to state-space models written in the BUGS language (Lunn *et al.* 2012). **Biips**, which has an R interface **rbiips** (Todeschini, Caron, Fuentes, Legrand, and Del Moral 2014), similarly allows inference via SMC to be performed on BUGS models. Compared to **nimbleSMC**, **Biips** acts as a "black-box" tool for SMC inference that automatically chooses an SMC algorithm for a given BUGS model. **nimbleSMC** allows users to choose and customize the SMC method they would like to use. Other current software packages that implement SMC algorithms include the **pomp** R package (King, Nguyen, and Ionides 2016), the **LibBi** software (Murray 2015) with interface to R via the **rbi** package (Jacob and Funk 2021), and the **vSMC** C++ template library (Zhou 2015). Of these, both **pomp** and **LibBi** offer algorithms not found in **nimbleSMC**, with **pomp** allowing for approximate Bayesian computation (Toni, Welch, Strelkowa, Ipsen, and Stumpf 2009), and **LibBi** providing an SMC^2 algorithm (Chopin, Jacob, and Papaspiliopoulos 2013). **nimble**, on the other hand, allows users to write their own SMC algorithms using **nimble**'s algorithm programming system. Additionally, state-space models in **nimble** can benefit from **nimble**'s other algorithms. For example, state-space models created in **nimble** can be compared to each other using **nimble**'s WAIC or cross-validation metrics.

For linear, Gaussian state-space models, SMC algorithms are unnecessary, as exact inference can be conducted via the Kalman filter (Kalman 1960). R packages that implement the Kalman filter include **dlm** (Petris 2010) and **dse** (Gilbert 2020). The **KFAS** R package (Helske 2017) allows for variations of Kalman filter inference to be conducted on models with non-Gaussian observation equations. See Petris and Petrone (2011) and Tusell (2011) for a detailed comparison of these packages.

The **TMB** R package enables maximum likelihood estimation of state-space models via Laplace approximation (Kristensen, Nielsen, Berg, Skaug, and Bell 2016), using the **CppAD** C++ package to enable automatic differentiation of likelihood functions (Bell 2005). To conduct inference in **TMB**, a user must write a C++ function that returns the log-likelihood of the state-space model. In contrast, **nimble** currently does not support Laplace approximation, although maximum likelihood estimation for state-space models can be achieved through the IF2 algorithm described in Section 3.5.

nimble also allows MCMC algorithms to be be flexibly composed and controlled by users. The ability to try such different methods on the same models is distinct from other packages, such as **WinBUGS** and **OpenBUGS** (Lunn *et al.* 2000, 2012), **JAGS** (Plummer 2003), and Stan (Carpenter *et al.* 2017). It means, for example, that particle MCMC algorithms in **nimble** can harness **nimble**'s MCMC implementations.

3. Sequential Monte Carlo methods for state-space models

3.1. State-space models

State-space models, also known as hidden Markov models, are used to model time-series data or any sequential data. The vector of data at each time t, labeled Y_t, is assumed to be related to a latent, unobserved state X_t through an observation distribution $Y_t \sim g_t(y_t \mid x_t, \theta)$. Here, θ is a vector of top-level parameters that are assumed not to change with time. X_t depends on X_{t-1} through a transition distribution $X_t \sim f_t(x_t \mid x_{t-1}, \theta)$. Frequently, the observation and transition distributions remain constant over all time points, in which case the t subscripts on f_t and g_t are dropped. (Below, we drop the subscripts to simplify notation, but neither the algorithms nor **nimbleSMC**'s implementations require the distributions to remain constant.) State-space models have the following Markov property: $f(x_t \mid x_{1:t-1}, \theta) = f(x_t \mid x_{t-1}, \theta)$, where $x_{1:t-1} = (x_1, \ldots, x_{t-1})$. Note that we assume no observation exists for $t = 0$, and that x_0 comes from a known prior distribution $f(x_0 \mid \theta)$.

One goal is to determine the distribution $p(x_t \mid y_{1:t}, \theta)$, known as the filtering distribution for X_t. Consider a situation where new data are received sequentially in time, and data are currently available up to time $t-1$, that is, $y_{1:t-1}$ are known. Upon receiving y_t, the filtering distribution $p(x_t \mid y_{1:t}, \theta)$ provides information about the latent state at the most recent time point, given all data. Similarly, the smoothing distribution $p(x_{1:t} \mid y_{1:t}, \theta)$ provides information about latent states from all time points given the most recent data. The forecast distribution $p(x_{t+\tau} \mid y_{1:t}, \theta)$ for $\tau > 0$ is also frequently of interest. Another common goal is to calculate the likelihood $p(y_{1:t} \mid \theta)$. This likelihood can in turn be used to obtain maximum likelihood estimates of θ, or in an MCMC framework to obtain samples from the posterior $p(\theta \mid y_{1:t})$.

SMC methods, at each time t, keep track of M weighted samples $\left\{x_t^{(m)}\right\}_{m=1}^{M}$ from the filtering distribution $p(x_t \mid y_{1:t}, \theta)$, along with associated weights $\left\{\pi_t^{(m)}\right\}_{m=1}^{M}$. These samples are known as "particles". When a new data point y_{t+1} is considered, a combination of propagating particles forward, re-weighting and/or re-sampling them is done to obtain an updated sample $p(x_{t+1} \mid y_{1:t+1}, \theta)$ with updated weights $\left\{\pi_{t+1}^{(m)}\right\}_{m=1}^{M}$. The steps involved also provide an approximation of $p(y_{t+1} \mid y_{1:t}, \theta)$. The methods differ in how the propagation, re-weighting and/or re-sampling are done.

As compared to MCMC methods, SMC methods generally perform well at generating samples, given fixed θ, from the filtering, smoothing, and forecasting distributions, and at estimating model likelihoods. SMC methods are also able to perform "on-line" inference. On the other hand, SMC methods alone do not provide an easy way to achieve inference on top-level parameters. Recently, algorithms such as PMCMC (described in Section 3.6) and IF2 (described in Section 3.5) have been proposed that use SMC methods in conjunction with MCMC and maximum likelihood algorithms, respectively, to achieve top-level parameter inference.

MCMC algorithms can also be used to draw samples from the filtering, smoothing, or forecasting distributions, as well as samples of top-level parameters conditioned on the current data. However, MCMC algorithms can encounter difficulty in producing valid proposals for $x_{1:t}$ due to the often-high degree of correlation among the latent states at each time point. Attempts have been made at creating proposal distributions that account for these correlations, as in Pooley, Bishop, and Marion (2015) and Newman, Fernández, Thomas, and Buckland (2009), but applying generic MCMC algorithms to state-space models does not always work efficiently.

3.2. Filtering algorithms

In Section 3.3 and Section 3.4, two types of SMC methods (the bootstrap filter and auxiliary particle filter) are described, each of which can be used to generate samples from the filtering distribution $p(x_t \mid y_{1:t}, \theta)$ or the smoothing distribution $p(x_{1:t} \mid y_{1:t}, \theta)$, and to obtain likelihood estimates. In Section 3.5, we describe the IF2 algorithm, which uses SMC algorithms to produce maximum-likelihood estimates of θ. In Section 3.6, a PMCMC algorithm is detailed that uses SMC methods to estimate likelihoods within a Metropolis-Hastings MCMC sampling scheme for θ.

The Ensemble Kalman filter, or EnKF (Evensen 2003), can also be used to estimate the filtering distribution. Similar to other SMC techniques, the EnKF approximates the filtering distribution via a collection of particles that are propagated forwards in time. However, whereas other SMC methods use importance sampling to select particles at each time point, the EnKF instead shifts particles towards the filtering distribution using an approximation to the Kalman gain matrix. The EnKF is described in Section 3.7.

The algorithm summaries provided in this paper are not intended to be a comprehensive overview of the current state of filtering methods. Several papers describe the landscape of filtering algorithms and related inference methods for state-space models, including Arulampalam, Maskell, Gordon, and Clapp (2002), Doucet and Johansen (2011), and Kantas, Doucet, Singh, Maciejowski, and Chopin (2015).

Note that in the following discussions of the bootstrap, auxiliary particle and ensemble Kalman filters, θ is treated as fixed, so we have omitted it to reduce notational overhead.

3.3. Bootstrap filter

The bootstrap filter of Gordon *et al.* (1993) uses importance sampling to sequentially generate samples from $p(x_t \mid y_{1:t})$ and approximate $p(y_t \mid y_{1:t-1})$ at each time t. The bootstrap filter first propagates each particle from time $t-1$ forward according to a proposal distribution $\tilde{x}_t^{(m)} \sim q(x_t \mid x_{t-1}^{(m)}, y_t)$. Importance weights $\pi_t^{(m)}$ are then calculated for each particle. In the basic version of the algorithm, the propagated particles are resampled according to these weights at each step. This results in an equally weighted sample $\left\{x_t^{(m)}\right\}_{m=1}^{M}$ from $p(x_t \mid y_{1:t})$. (See below for extensions that do not resample at each step.)

nimbleSMC uses a simple choice for $q(\cdot \mid \cdot)$, namely the forward simulation density, $f(x_t^{(m)} \mid x_{t-1}^{(m)})$. With this choice, Step 8 simplifies to $w_t^{(m)} = g(y_t \mid \tilde{x}_t^{(m)})\pi_{t-1}^{(m)}$. Then one does not need to calculate $f(\tilde{x}_t^{(m)} \mid x_{t-1}^{(m)})$, which is important because these methods are sometimes used when it is feasible to simulate forward in time but not actually calculate the forward density. Note that resampling (Steps 14–16 in Algorithm 1) creates an equally weighted sample from the target distribution of latent states at time t. Additionally, an estimate of the likelihood $p(y_{1:T})$ can be obtained by $\tilde{p}(y_{1:T}) = \prod_{t=1}^{T} \tilde{p}(y_{t|1:t-1})$, where $\tilde{p}(y_{t|1:t-1})$ is given in Line 18 of the algorithm.

The resampling step of Algorithm 1 is performed to reduce particle degeneracy. Particle degeneracy is a common problem in filtering algorithms, where a small number of particles have most of the weight placed on them, leaving most particles with low weights (Doucet, Godsill, and Andrieu 2000). Particle degeneracy corresponds to high Monte Carlo variance of approximations made using the filtered particles. It causes the filter to spend computational effort in propagating and weighting particles that contribute little to our knowledge of the target distribution. Resampling ensures that mostly highly-weighted particles will be propagated forwards, increasing algorithm efficiency by providing a better estimate of the target distribution and likelihood.

However, resampling particles at each time point can also lead to a loss of particle "diversity" (Doucet *et al.* 2000), as many of the resampled particles at each time point will have the same value. Thus it has been proposed (Smith, Doucet, de Freitas, and Gordon 2001) that resampling should take place only if particle degeneracy becomes too significant. An estimate of particle degeneracy is the effective sample size, calculated at each time t as $ESS = 1/\sum_{m=1}^{M}\left(\pi_t^{(m)}\right)^2$. Smith *et al.* (2001) recommend that resampling should be con-

Algorithm 1 Bootstrap filter.

```
 1: for $m$ in $1 : M$ do
 2:     Generate $x_0^{(m)} \sim f(x_0)$
 3:     Set $\pi_0^{(m)} = \frac{1}{M}$
 4: end for
 5: for $t$ in $1 : T$ do
 6:     for $m$ in $1 : M$ do
 7:         Generate $\tilde{x}_t^{(m)} \sim q(x_t \mid x_{t-1}^{(m)}, y_t)$
 8:         Calculate unnormalized weight $w_t^{(m)} = \frac{f(\tilde{x}_t^{(m)}|x_{t-1}^{(m)})g(y_t|\tilde{x}_t^{(m)})}{q(\tilde{x}_t^{(m)}|x_{t-1}^{(m)},y_t)}\pi_{t-1}^{(m)}$
 9:     end for
10:     for $m$ in $1 : M$ do
11:         Normalize $w_t^{(m)}$ as $\pi_t^{(m)} = \frac{w_t^{(m)}}{\sum_{i=1}^M w_t^{(i)}}$.
12:     end for
13:     for $m$ in $1 : M$ do
14:         Sample an index $j_m$ from the set 1,. . . ,M with probabilities $\{\pi_t^{(m)}\}_{m=1}^M$.
15:         Set $x_t^{(m)} = \tilde{x}_t^{(j_m)}$.
16:         Set $\pi_t^{(m)} = \frac{1}{M}$
17:     end for
18:     Calculate $\tilde{p}(y_{t|1:t-1}) = \frac{1}{M}\sum_{m=1}^M w_t^{(m)}$
19: end for
```

ducted only if the effective sample size becomes too low, indicating many particles with low weights. As a criterion for when a resampling step should take place, a threshold τ is chosen with $0 \leq \tau \leq 1$, such that the algorithm will resample particles whenever $\frac{ESS}{M} < \tau$. Note that choosing $\tau = 0$ will mean that the resampling step is never performed, and choosing $\tau = 1$ will ensure that sampling is performed at each time point. To perform the above algorithm without resampling, simply remove Steps 14–16. If the resampling step is not performed, the set $\left\{\tilde{x}_t^{(m)}, \pi_t^{(m)}\right\}_{m=1}^M$ will constitute an unequally weighted sample from the target distribution.

Various methods for resampling particles have been employed, including systematic resampling, stratified resampling, residual resampling, and multinomial resampling (Doucet and Johansen 2011). Generally, systematic resampling, stratified resampling, and residual resampling have been found to perform similarly, and to outperform multinomial resampling due to its higher variance (Douc and Cappe 2005). **nimbleSMC** allows users to choose from any of these four methods. Systematic resampling is used by default.

Additionally, the above filter can be used to produce samples from the smoothing distribution $p(x_{1:t} \mid y_{1:t})$ using methods described in Doucet and Johansen (2011). **nimbleSMC** currently uses the naive approach of storing the ancestors of the set of particles at time t. Specifically, using the notation of Andrieu *et al.* (2010), **nimbleSMC**'s smoothing method keeps track of the set $x_{1:T}^{(m)} = \left(x_1^{(B_1^{(m)})}, x_2^{(B_2^{(m)})}, \dots, x_T^{(B_T^{(m)})}\right)$ for $m = 1, \dots, M$, where $B_t^{(m)}$ is the index of the ancestor particle at time t that gave rise to particle m at time T. We note that this approach to estimating the smoothing distribution can produce poor estimates of $p(x_t \mid y_{1:T}, \theta)$ when T is much greater than t, as the number of unique ancestors at time t will decrease as T increases.

More accurate methods for estimating the smoothing distribution, such as forward-backward smoothing, can be found in Kantas *et al.* (2015).

Though the bootstrap filtering algorithm was first put forth in 1993, it remains commonly used in a wide variety of applications, such as inferring the distribution of radii of planets (Silburt, Gaidos, and Wu 2015) and estimating neurological activity (Croce, Zappasodi, Merla, and Chiarelli 2017). Yang and Eisenstein (2013) use a bootstrap filter to help to normalize text obtained from social media. Oladyshkin, Class, and Nowak (2013) employ a bootstrap filter to model CO_2 storage in geological formations.

3.4. Auxiliary particle filter

The auxiliary particle filter algorithm (APF) of Pitt and Shephard (1999) uses importance sampling similarly to the bootstrap filter but includes an additional "look-ahead step". At each time t, the APF calculates first-stage weights $w_{t|t-1}^{(m)}$ for particles from time $t-1$. These weights are calculated using a rough estimate of the likelihood of the current data given each particle from the previous time point, labeled $\hat{p}(y_t \mid x_{t-1}^{(m)})$. Particles with high first-stage weights correspond to values of the latent state at time $t-1$ that are likely to generate the observed data at time t. To estimate $\hat{p}(y_t \mid x_{t-1}^{(m)})$, Pitt and Shephard (1999) recommend choosing an auxiliary variable $\tilde{x}_{t|t-1}^{(m)}$ and then setting $\hat{p}(y_t \mid x_{t-1}^{(m)}) = p(y_t \mid \tilde{x}_{t|t-1}^{(m)})$. Possible methods for choosing $\tilde{x}_{t|t-1}^{(m)}$ include simulating a value from $f(x_t \mid x_{t-1}^{(m)})$, or taking $\tilde{x}_{t|t-1}^{(m)} = E(x_t \mid x_{t-1}^{(m)})$.

The first-stage weights are used to sample M particles from time $t-1$, labeled $\tilde{x}_{t-1}^{(m)}$ for $m = 1, \ldots, M$. The sampled particles are then propagated forwards by a proposal distribution $q(x_t^{(m)} \mid \tilde{x}_{t-1}^{(m)}, y_t)$ and reweighted using second-stage weights $w_t^{(m)}$, providing a weighted sample from $p(x_t \mid y_{1:t})$. The APF as shown in Pitt and Shephard (1999) optionally includes a second resampling step using the second-stage weights. However, the algorithm using a single resampling step has been shown to be more efficient (Carpenter, Clifford, and Fearnhead 1999).

nimbleSMC again uses the basic choice $f(x_t \mid \tilde{x}_{t-1}^{(m)})$ for $q(\cdot \mid \cdot)$, which simplifies the weight in Step 18 to $w_t^{(m)} = g(y_t \mid x_t^{(m)})/\hat{p}(y_t \mid x_{t-1}^{(m)})$. In a manner similar to the bootstrap filter, the APF can be used to obtain an estimate of the likelihood $p(y_{1:T})$ as $\tilde{p}(y_{1:T}) = \prod_{t=1}^{T} \tilde{p}(y_{t|1:t-1})$, where $\tilde{p}(y_{t|1:t-1})$ is given in Line 23 of the APF algorithm. **nimbleSMC** provides the two choices for generating the first stage weights (Steps 7–9) described above: either via forward simulation of the latent state or, when available for the specific model, via the predicted latent state mean from each particle at time $t-1$.

Similar to the bootstrap filter, the auxiliary particle filter was developed some time ago, but still sees frequent use. Recently, APF algorithms have been applied to problems in fields such as pedestrian navigation (Yu, El-Sheimy, Lan, and Liu 2017) and battery life prediction (Liu, Wang, and Ma 2011).

3.5. IF2 algorithm

Unlike the bootstrap and auxiliary particle filters, the IF2 algorithm (Ionides *et al.* 2015) is designed for maximum likelihood estimation. The IF2 algorithm is a variant of the basic form of iterated filtering, proposed by Ionides, Bretó, and King (2006) and Ionides, Bhadra, Atchadé, and King (2011). The original iterated filtering algorithm was found to have favorable per-

Algorithm 2 Auxiliary particle filter.

```
for $m$ in $1:M$ do
    Generate $x_0^{(m)} \sim f(x_0)$
    Set $\pi_0^{(m)} = \frac{1}{M}$
end for
for $t$ in $1:T$ do
    for $m$ in $1:M$ do
        Generate $\tilde{x}_{t|t-1}^{(m)}$ from either $\tilde{x}_{t|t-1}^{(m)} \sim f(x_t \mid x_{t-1}^{(m)})$ or $\tilde{x}_{t|t-1}^{(m)} = E(x_t \mid x_{t-1}^{(m)})$
        Calculate $\hat{p}(y_t \mid x_{t-1}^{(m)}) = p(y_t \mid \tilde{x}_{t|t-1}^{(m)})$
        Calculate unnormalized weight $w_{t|t-1}^{(m)} = \pi_{t-1}^{(m)} \hat{p}(y_t \mid x_{t-1}^{(m)})$
    end for
    for $m$ in $1:M$ do
        Normalize $w_{t|t-1}^{(m)}$ as $\pi_{t|t-1}^{(m)} = \frac{w_{t|t-1}^{(m)}}{\sum_{i=1}^{M} w_{t|t-1}^{(i)}}$
    end for
    for $m$ in $1:M$ do
        Sample an index $j_m$ from the set 1,. . . ,M with probabilities $\{\pi_{t|t-1}^{(m)}\}_{m=1}^{M}$.
        Set $\tilde{x}_{t-1}^{(m)} = x_{t-1}^{(j_k)}$
        Generate $x_t^{(m)} \sim q(x_t \mid \tilde{x}_{t-1}^{(m)}, y_t)$
        Calculate unnormalized weight $w_t^{(m)} = \frac{f(x_t^{(m)}|\tilde{x}_{t-1}^{(m)}) g(y_t|x_t^{(m)})}{\hat{p}(y_t|x_{t-1}^{(m)}) q(x_t^{(m)}|\tilde{x}_{t-1}^{(m)}, y_t)}$
    end for
    for $m$ in $1:M$ do
        Normalize $w_t^{(m)}$ as $\pi_t^{(m)} = \frac{w_t^{(m)}}{\sum_{i=1}^{M} w_t^{(i)}}$
    end for
    Calculate $\tilde{p}(y_{t|1:t-1}) = \left(\frac{1}{M}\sum_{m=1}^{M} w_t^{(m)}\right)\left(\sum_{m=1}^{M} w_{t|t-1}^{(m)}\right)$
end for
```

formance and theoretical properties as compared to the Liu-West method of estimating the posterior distribution $p(\theta \mid y_{1:T})$ (Liu and West 2001). The innovation of Ionides *et al.* (2006) is to perturb θ by random walk noise and to use particle filter (SMC) likelihood estimates to approximate derivatives of the log-likelihood function in order to produce maximum likelihood estimates.

Ionides *et al.* (2015) modified the theory developed by Ionides *et al.* (2006) by using an iterated Bayes maximum a posteriori (MAP) estimate in place of gradient estimates of the perturbed parameters θ to optimize the MLE. Iterating in this way has some benefits:

1. A theoretical foundation for this method can be obtained by convergence of the iterated Bayes map.

2. Methods that are not based on local polynomial approximations to the likelihood surface can be advantageous when the likelihood surface has nonlinear ridges.

3. Computational expense is reduced by removing the need for the computationally demanding gradient of the log-likelihood.

The IF2 algorithm proceeds by running a modified particle filter for I iterations. As in the Liu-West filter, parameters θ are also represented by particles that are weighted and resampled along with their associated latent states. The perturbations to the parameters are generated at each time step of each particle filter iteration. The perturbations follow a schedule of decreasing magnitude to yield convergence to the maximum likelihood parameters. In what follows, our notation differs somewhat from that of Ionides *et al.* (2015) to be both more specific and more consistent with the other algorithms described here.

For iteration i, let $h_t(\theta \mid \Sigma_i)$ for $t = 1, \ldots, T$ be a sequence of user-chosen densities that represent perturbations of θ. Indexing the density by t (the particle filter time index) and its parameters by i (the IF2 iteration index) is how Ionides *et al.* (2015) present the method, but the associated implementation in **pomp** (King *et al.* 2016) uses a scheme where the $h(\cdot)$ does not change – it is a normal distribution – but rather its parameters change in both t and i. Hence from here on we write this as $h(\theta \mid \Sigma_{i,t})$ for clarity. Formally, this can be viewed as a special case of $h_t(\theta \mid \Sigma_i)$. Choice of h and $\Sigma_{i,t}$ is described below.

IF2 iteration i begins with a parameter swarm $\left\{\theta_{i,0}^{(m)}\right\}_{m=1}^{M}$ where $\theta_{i,0}^{(m)} \sim h(\theta \mid \theta_{i-1,T}^{(m)}, \Sigma_{i,0})$. A bootstrap particle filter is then run, with a key modification: at each time step t, before simulating, weighting and resampling, we perturb parameters by drawing $\tilde{\theta}_{i,t}^{(m)} \sim h(\theta \mid \theta_{i,t-1}^{(m)}, \Sigma_{i,t})$. These parameter values are then included in weighting and resampling along with their corresponding latent state values.

Once again, for the choice for forward proposals, $q(\cdot \mid \cdot)$ in Step 12, **nimbleSMC** uses the basic choice of simulating from the model's latent state dynamics, $\tilde{x}_t^{(m)} \sim f(x_t \mid x_{t-1}^{(m)}, \tilde{\theta}_{i,t}^{(m)})$. The weight in Step 13 then simplifies to $w_t^{(m)} = g(y_t \mid \tilde{x}_t^{(m)}, \tilde{\theta}_{i,t}^{(m)})$.

After the I-th iteration, the parameter samples $\left\{\theta_{I,T}^{(m)}\right\}_{m=1}^{M}$ are averaged to produce an estimate of the MLE of θ. In the IF2 algorithm, by analogy with simulated annealing, it has been shown that, with an appropriate schedule for decreasing perturbation magnitudes, in the limit as the perturbations $h(\theta \mid \Sigma_{i,t})$ go to zero variance with mean θ as $i \to \infty$, the algorithm should reach the maximum likelihood solution. Algorithm 3 presents the IF2 algorithm as it is implemented in **nimbleSMC**.

nimbleSMC's implementation of IF2, and specifically the schedule of decreasing (or cooling) perturbation magnitudes, follows that of **pomp** (King *et al.* 2016). $h(\theta \mid \theta_{i,t-1}^{(m)}, \Sigma_{i,t})$ is a product of independent normals, i.e., a multivariate normal with mean $\theta_{i,t-1}^{(m)}$ and diagonal covariance matrix $\Sigma_{i,t}$. The d-th diagonal element of $\Sigma_{i,t}$ is $c_{i,t}^2 \sigma_d^2$, where $c_{i,t} = \alpha^{\frac{t-1+(i-1)T}{50T}}$. The user must provide a choice of σ_d for each dimension θ_d of θ as well as the parameter α that determines the cooling schedule for all dimensions together. The formula for $c_{i,t}$ implies that after 50 IF2 iterations ($i = 51$, $t = 0$), the perturbation standard deviation for parameter θ_d will be α times the original perturbation standard deviation, σ_d.

The initial set of parameter particles is important for IF2's performance. To generate these, the user must provide the mean, θ_{init}, and standard deviations, i.e., a vector of square roots of diagonal elements of Σ_{init}. Note that the time step "T" for the initial particles $\theta_{0,T}^{(m)}$ is artificial so that generation of $\theta_{1,0}^{(m)}$ from $h(\theta \mid \theta_{i-1,T}^{(m)})$ conforms to notation suitable for later iterations.

IF2 has been used for estimating parameters of stochastic differential equation models of

Algorithm 3 IF2.

```
for $m$ in $1:M$ do
    Generate $\theta_{0,T}^{(m)} \sim h(\theta \mid \theta_{\text{init}}, \Sigma_{\text{init}})$
end for
for $i$ in $1:I$ do
    for $m$ in $1:M$ do
        Generate $\theta_{i,0}^{(m)} \sim h(\theta \mid \theta_{i-1,T}^{(m)}, \Sigma_{i,0})$
        Generate $x_0^{(m)} \sim f(x_0 \mid \theta_{i,0}^{(m)})$
    end for
    for $t$ in $1:T$ do
        for $m$ in $1:M$ do
            Generate $\tilde{\theta}_{i,t}^{(m)} \sim h(\theta \mid \theta_{i,t-1}^{(m)}, \Sigma_{i,t})$
            Generate $\tilde{x}_t^{(m)} \sim q(x_t \mid x_{t-1}^{(m)}, y_t, \tilde{\theta}_{i,t}^{(m)})$
            Calculate unnormalized weight $w_t^{(m)} = \frac{f(\tilde{x}_t^{(m)}|x_{t-1}^{(m)},\tilde{\theta}_{i,t}^{(m)})g(y_t|\tilde{x}_t^{(m)},\tilde{\theta}_{i,t}^{(m)})}{q(\tilde{x}_t^{(m)}|x_{t-1}^{(m)},y_t,\tilde{\theta}_{i,t}^{(m)})}$
        end for
        for $m$ in $1:M$ do
            Normalize $w_t^{(m)}$ as $\pi_t^{(m)} = \frac{w_t^{(m)}}{\sum_{i=1}^{M} w_t^{(i)}}$
        end for
        for $m$ in $1:M$ do
            Sample an index $j_m$ from the set 1,. . . ,M with probabilities $\{\pi_t^{(m)}\}_{m=1}^{M}$.
            Set $x_t^{(m)} = \tilde{x}_t^{(j_m)}$ and $\theta_{i,t}^{(m)} = \tilde{\theta}_{i,t}^{(j_m)}$
        end for
    end for
end for
Return $\bar{\theta} = \frac{1}{M} \sum_{m=1}^{M} \theta_{I,T}^{(m)}$
```

disease dynamics (King, Domenech de Cellès, Magpantay, and Rohani 2015; Martinez-Bakker, King, and Rohani 2015; Azman, Luquero, Ciglenecki, Grais, Sack, and Lessler 2015).

3.6. Particle MCMC methods

Particle MCMC methods (PMCMC, Andrieu *et al.* 2010) allow joint sampling from the posterior distribution of the states and the top-level parameters. PMCMC takes advantage of the ability of certain particle filters to provide unbiased estimates of the (marginal) likelihood, that is, $\tilde{p}(y_{1:T} \mid \theta) \approx \int_X p(y_{1:T} \mid x_{1:T}, \theta) p(x_{1:T} \mid \theta) dx_{1:T}$. For example, the bootstrap filter and auxiliary filter (Pitt 2002) can both be used to provide unbiased estimates of the marginal likelihood, as detailed in Sections 3.3 and 3.4. Building upon the framework of Andrieu and Roberts (2009), Andrieu *et al.* (2010) showed that using this approximation when calculating a Metropolis-Hastings acceptance ratio yields an "exact approximate" algorithm. Although it seems to rely on an approximation, formally it samples from an auxiliary space of all states and indices sampled during the SMC run. The marginal distribution of this enhanced space matches the desired target distribution, so it samples from exactly the correct target distribution based upon approximate likelihood calculations. See Dahlin and Schön (2019) for a gentle introduction. Below, we detail the particle marginal Metropolis Hastings (PMMH)

Algorithm 4 PMMH algorithm.

```
Choose an initial value $\theta^{(0)}$
Run a particle filter to obtain the marginal likelihood estimate $\tilde{p}^{(0)} = \tilde{p}(y_{1:T} \mid \theta^{(0)})$.
After the last time step of the particle filter, draw $x_{1:T}^{(0)} \sim p(x_{1:T} \mid y_{1:T}, \theta^{0})$ by sampling a
particle with its full state history.
for $i$ in $1 : I$ do
    Generate $\theta^* \sim q(\theta \mid \theta^{(i-1)})$
    Run a particle filter to obtain the marginal likelihood estimate $\tilde{p}(y_{1:T} \mid \theta^*)$.
    After the last time step of the particle filter, draw $x^*_{1:T} \sim p(x_{1:T} \mid y_{1:T}, \theta^*)$ by sampling
    a particle with its full state history.
    Compute $a^* = 1 \wedge \frac{\tilde{p}(y_{1:T} \mid \theta^*) p(\theta^*)}{\tilde{p}^{(i-1)} p(\theta^{(i-1)})} \frac{q(\theta^{(i-1)} \mid \theta^*)}{q(\theta^* \mid \theta^{(i-1)})}$
    Generate $r \sim unif(0, 1)$
    if $a^* > r$ then
        Set $\theta^{(i)} = \theta^*$, $x_{1:T}^{(i)} = x^*_{1:T}$, and $\tilde{p}^{(i)} = \tilde{p}(y_{1:T} \mid \theta^*)$
    else
        Set $\theta^{(i)} = \theta^{(i-1)}$, $x_{1:T}^{(i)} = x_{1:T}^{(i-1)}$, and $\tilde{p}^{(i)} = \tilde{p}^{(i-1)}$.
    end if
end for
```

algorithm, one of three algorithms provided in Andrieu *et al.* (2010). The PMMH algorithm is available as a sampler to be used in **nimble**'s general MCMC framework.

At each iteration i, the PMMH algorithm first proposes a value θ^* for the model parameters θ from a proposal distribution $q(\theta^* \mid \theta^{(i-1)})$. Using this proposed value for θ, a particle filter is then run, which provides an estimate $\tilde{p}(y_{1:T} \mid \theta^*)$ of the likelihood given the proposed parameters. This marginal likelihood estimate is used to calculate the Metropolis-Hastings acceptance probability for θ^*. By default, the algorithm saves the chain of θ samples. If latent states are needed, an index j is sampled from $\{1, \ldots, M\}$ using the final particle filter weights $\{\pi_T^{(m)}\}_{m=1}^M$ in PMMH iteration i, and the latent states $x_{1:T}^{(i)}$ for the i-th PMMH iteration are set to the j-th latent state sequence, $x_{1:T}^{(j)}$, from the i-th run of the particle filter. Importantly, one does not need to record all states and indices used in the particle filter run, but one does need to retain $\tilde{p}(y_{1:T} \mid \theta^*)$ if θ^* is accepted. If the latent states, $x_{1:T}$, are not needed as part of the PMMH output, the steps to track and record them can simply be skipped.

PMCMC methods have been adopted in a wide variety of fields since their introduction. Examples of research conducted using PMCMC include hydrology models (Vrugt, Ter Braak, Diks, and Schoups 2013) and epidemiological models that incorporate genealogy (Rasmussen, Volz, and Koelle 2014).

Like all MCMC algorithms, PMCMC methods can suffer from poor mixing. In particular for PMCMC methods, such mixing problems can arise as a result of high variance of the likelihood estimates. For example, an erroneously high likelihood estimate from the particle filter can make it difficult for subsequent proposals to be accepted even for nearby parameters. These problems were studied theoretically by Sherlock, Thiery, Roberts, and Rosenthal (2015) and Doucet, Pitt, Deligiannidis, and Kohn (2015) and are illustrated in the simulation exercise below. Additionally, since high variances for the estimated likelihood occur especially for areas of low posterior density (Murray, Jones, and Parslow 2013), it is important to provide reasonable initial values for the parameters, θ. One idea for obtaining reasonable initial values, similar to an idea from Bernton, Jacob, Gerber, and Robert (2017), is to use maximum a posteriori (MAP) estimates, if available, as starting values for PMCMC.

3.7. Ensemble Kalman filter

In Section 3.2, the Kalman filter was mentioned as providing an analytic solution to the filtering problem when working with a linear, Gaussian state-space model. When using a model with non-linear transition equations or observation equations, however, the Kalman filter is no longer applicable. One approximation to the filtering problem for Gaussian state-space models with non-linear transition or observation equations is the ensemble Kalman filter (EnKF), which uses a particle representation of the latent states at each time point.

Although the EnKF uses particles to represent filtering distributions, as do the particle filters described in Sections 3.3 and 3.4, it updates the latent state particles using a fundamentally different approach than those methods. Whereas bootstrap and auxiliary particle filters update particles via a re-weighting and re-sampling framework, the EnKF first propagates particles forward using the transition equation, and then shifts particles towards the filtering distribution using an approximation to the Kalman gain matrix from the original Kalman Filter (Mandel 2009).

The EnKF assumes the following forms for the transition and observation equations:

$$x_t = M(x_{t-1}) + w_t \tag{1}$$

$$y_t = D(x_t) + v_t \tag{2}$$

where w_t and v_t are independent, normally distributed error terms with covariance matrices Q_t and R_t respectively. At time $t-1$, assume that we have a sample $\{x_{t-1}^{(m)}\}_{m=1}^{M}$ of particles from $p(x_{t-1} \mid y_{1:t-1})$. Each particle is propagated forward according to Equation 1, with random draws for w_t, giving $\tilde{x}_t^{(m)}$. In addition, the mean observation from each $\tilde{x}_t^{(m)}$ is calculated as $\tilde{y}_t^{(m)} = D(\tilde{x}_t^{(m)})$. The idea behind the EnKF is to use these samples to approximate the covariance between latent states and observations, as well as the covariance among observation dimensions, at time t. From these, one can approximate the Kalman gain matrix, $\tilde{G}_t$, and apply a simulated version of the Kalman update step as follows:

$$x_t^{(m)} = \tilde{x}_t^{(m)} + \tilde{G}_t(y_t + v_t^{(m)} - \tilde{y}_t^{(m)}),$$

where $v_t^{(m)} \sim N(0, R_t)$ are simulated observation errors. The $x_t^{(m)}$ values form a sample of the approximated filtering distribution at time t, i.e., $p(x_t \mid y_{1:t})$.

The covariance between latent states and observations is calculated as the covariance between $e_t^x = (\tilde{x}_t^{(1)} - \bar{\tilde{x}}_t, \ldots, \tilde{x}_t^{(M)} - \bar{\tilde{x}}_t)$ and $e_t^y = (\tilde{y}_t^{(1)} - \bar{\tilde{y}}_t, \ldots, \tilde{y}_t^{(M)} - \bar{\tilde{y}}_t))$. This, along with a similarly approximated covariance of observations, is used to construct the approximate Kalman gain matrix $\tilde{G}_t$. A more detailed overview of the EnKF, and how it relates to the Kalman Filter, can be found in Gillijns, Mendoza, Chandrasekar, De Moor, Bernstein, and Ridley (2006) and Katzfuss, Stroud, and Wikle (2016).

Note that the multiplication of e_t terms in Steps 11 and 12 are matrix multiplications.

We remark that although the EnKF assumes normally distributed error terms for Equations 1 and 2, the filter has been shown to be robust to each of those assumptions (Katzfuss *et al.*

Algorithm 5 Ensemble Kalman filter.

```
for $m$ in $1 : M$ do
    Generate $x_0^{(m)} \sim f(x_0)$
end for
for $t$ in $1 : T$ do
    for $m$ in $1 : M$ do
        Generate $\tilde{x}_t^{(m)} \sim f(x_t \mid x_{t-1}^{(m)})$
        Calculate $\tilde{y}_t^{(m)} = D(\tilde{x}_t^{(m)})$
    end for
    Calculate $e_t^x = (\tilde{x}_t^{(1)} - \bar{\tilde{x}}_t, \ldots, \tilde{x}_t^{(M)} - \bar{\tilde{x}}_t)$
    Calculate $e_t^y = (\tilde{y}_t^{(1)} - \bar{\tilde{y}}_t, \ldots, \tilde{y}_t^{(M)} - \bar{\tilde{y}}_t)$
    Calculate $\tilde{P}_t^{xy} = \frac{1}{M-1} e_t^x (e_t^y)'$
    Calculate $\tilde{P}_t^{yy} = \frac{1}{M-1} e_t^y (e_t^y)' + R_t$
    Calculate $\tilde{G}_t = \tilde{P}_t^{xy} (\tilde{P}_t^{yy})^{-1}$
    for $m$ in $1 : M$ do
        Generate $v_t^{(m)} \sim N(0, R_t)$
        Calculate $x_t^{(m)} = \tilde{x}_t^{(m)} + \tilde{G}_t(y_t + v_t^{(m)} - \tilde{y}_t^{(m)})$
    end for
end for
```

2016). Also, although newer variations of the EnKF exist that may perform better for some models, such as the Ensemble Adjusted Kalman Filter (Anderson 2001), we have provided only a basic version the EnKF algorithm in **nimbleSMC**. Users are welcome to modify this basic EnKF if a different version is desired - see Section 7 for an overview of creating and modifying particle filters in **nimble**. Recently, the EnKF has been used extensively for atmospheric data modeling (Houtekamer and Zhang 2016), as well as for problems in petroleum modeling (Heidari, Gervais, Ravalec, and Wackernagel 2013) and geophysics (Bocher, Fournier, and Coltice 2018).

4. Creating and manipulating models in nimble

The workflow of modeling and conducting inference in **nimble** is somewhat unique as compared to other statistical modeling software. Thus, before demonstrating **nimbleSMC**'s SMC features, we will first show in this section how to create and work with **nimble**'s model objects. Section 5 describes **nimbleSMC**'s SMC methods for latent state inference in state-space models. Section 6 describes **nimbleSMC**'s SMC methods for top-level parameter inference. A supplement to the paper includes a full R script of all code shown below.

The **nimble** package uses the BUGS language to specify hierarchical statistical models. We will not describe the BUGS language here – interested readers can find a brief overview in the **nimble** user manual (**nimble** Development Team 2020), or a more detailed guide in Lunn *et al.* (2012). Unlike other packages that use dialects of the BUGS language (**WinBUGS** and **JAGS**), **nimble** creates model objects, via the `nimbleModel` function, which can be queried and manipulated by the user. To introduce **nimble**'s modeling framework, we will use a linear Gaussian state-space model in which all parameters are fixed.

Let y_t be the observed data at time t, let x_t be the latent state at time t, and suppose we have 10 times. The model is:

$$\begin{aligned} x_0 &\sim N(0,1) \\ x_t &\sim N(0.8x_{t-1},1) \quad \text{for} \quad t = 1,\ldots,10 \\ y_t &\sim N(x_t, 0.5) \quad \text{for} \quad t = 1,\ldots,10 \end{aligned}$$

where $N(\mu, \sigma^2)$ denotes the normal distribution with mean μ and variance σ^2. Although this example model is relatively simple, it will serve to demonstrate the process of building and working with BUGS models in **nimble**.

Code written in the BUGS language is entered via the `nimbleCode` function. For example, BUGS code for the linear Gaussian model is entered like this:

```
R> library("nimble")
R> exampleCode <- nimbleCode({
+    x0 ~ dnorm(0, var = 1)
+    x[1] ~ dnorm(0.8 * x0, var = 1)
+    y[1] ~ dnorm(x[1], var = 0.5)
+    for (t in 2:10) {
+      x[t] ~ dnorm(0.8 * x[t - 1], var = 1)
+      y[t] ~ dnorm(x[t], var = 0.5)
+    }
+  })
```

Each line of BUGS code declares one or more nodes in the model. For example, this model has the nodes `x0`, `x[1]`, ..., `x[10]`, `y[1]`, ..., `y[10]`.

Once the model code has been written, a model object in R can be created using the `nimbleModel` function. Data and initial values can optionally be provided at this step. For example:

```
R> simulatedData <- c(-0.9, 1.6, 0.6, 1.3, 1.5, 0.3, -0.8, -1.3, 0.5, 1.1)
R> exampleModel <- nimbleModel(code = exampleCode,
+    data = list(y = simulatedData), inits = list(x0 = 0))
```

The `exampleModel` object is an R reference class object. `exampleModel` has a field for each of the variables in our BUGS model. For example, we can look at the value of the `y[3]` node by calling

```
R> exampleModel$y[3]
```

```
[1] 0.6
```

Additionally, `exampleModel` provides `simulate` and `calculate` methods. The `simulate` method takes a vector of node names as an argument, simulates values for these nodes conditioned upon the values of other nodes in the model that are parents of the nodes being simulated, and stores the simulated values in the corresponding model variables. The following code simulates values for `x0`, `x[1]`, ..., `x[10]`. We note that `x[1]` is simulated conditional upon the current value of `x0`, `x[2]` is simulated conditional upon the simulated value of `x[1]`, and so on. Note that the call to the `getNodeNames` method, using `includeData = FALSE`, will return the names of all non-data nodes in our model in an order that is valid for calculation or simulation.

```
R> exampleModel$simulate(nodes =
+    exampleModel$getNodeNames(includeData = FALSE))
R> exampleModel$x

 [1] -0.31751972 -1.08964439  0.72356529  0.90836000 -0.09378038
 [6]  0.41240475  1.06824850  1.43038015  0.83891574  2.18291376
```

The `calculate` method takes a vector of node names as an argument, and returns the summed log probability of all of the given stochastic nodes. The following line demonstrates calculating $\sum_{t=1}^{10} \log(g(y_t \mid x_t))$.

```
R> exampleModel$calculate(nodes = "y")

[1] -28.25143
```

nimble model objects contain additional methods that can provide details about the model and allow for more fine-grained manipulation. These are described in Chapter 6 of the **nimble** user manual (**nimble** Development Team 2020).

Although models and algorithms in **nimble** and **nimbleSMC** can run entirely within R, they are generally compiled into C++ for greatly increased efficiency. Compilation is achieved through the `compileNimble` function, which takes as arguments one or more model objects (or objects created by calls to `nimbleFunction`, described in Section 7). `compileNimble` generates model-specific C++, compiles and loads the result, and returns a version of the model object with the same fields and methods, but where the methods are now executed in compiled C++ instead of R.

```
R> C_exampleModel <- compileNimble(exampleModel)
```

`compileNimble` can also be used to compile algorithms in **nimble**, which will use compiled versions of the model.

Generally, analysis in **nimble** and **nimbleSMC** will start with writing model code and creating a model object. If available, initial values and data values should be provided to the model. A user then has a choice of inference methods: in addition to the SMC methods documented in this paper, **nimble** has a flexible MCMC system for posterior inference, an MCEM algorithm, and cross-validation methods. A user may also wish to write their own method using **nimble**'s programming system. Once an algorithm has been chosen, the user can compile both the model and the algorithm to C++ using the `compileNimble` function. Finally, the user can run the algorithm and analyze the results.

In Section 5, we introduce **nimbleSMC**'s bootstrap filter, auxiliary filter, and ensemble Kalman filter by applying them to the state-space model created above.

5. Filtering given fixed parameters

Now that we have built a `model` object for the example model, we can use algorithms from **nimbleSMC**. These algorithms are all written as `nimbleFunctions` using **nimble**'s programming system embedded in R. We begin by demonstrating the use of the bootstrap filter (Section 3.3) to estimate the filtering distribution $p(x_t \mid y_{1:t})$.

```
R> exampleBootstrapFilter <- buildBootstrapFilter(exampleModel, nodes = "x",
+    control = list(saveAll = TRUE, thresh = 0.9))
```

The `buildBootstrapFilter` function builds a bootstrap filter for the model given in the first argument. The `nodes` argument gives the name (or names) of the latent states to be filtered. Importantly, the latent states must have the same dimension at each time point. The algorithm parameters, packaged in the control list, include `saveAll` (should filtered state estimates be saved from all time points, or from just the last one) and `thresh` (a threshold for resampling, labeled τ in Section 3.4). Additional arguments to the control list can be found by calling `help(buildBootstrapFilter)`. One control list parameter of note is `smoothing`, which defaults to `FALSE`. If `smoothing = TRUE`, the particles returned from the algorithm will be samples from the smoothing distribution $p(x_{1:T} \mid y_{1:T})$. Along with similar functions that appear below, `buildBootstrapFilter` is actually a `nimbleFunction`, meaning it is written in **nimble**'s algorithm programming system.

The bootstrap filter in **nimbleSMC** sets the proposal distribution $q(x_t \mid x_{t-1}^{(m)}, y_t)$ to be the transition equation $f(x_t \mid x_{t-1}, \theta)$. A user wishing to use a different proposal distribution would need to copy and modify `buildBootstrapFilter` (see Section 7).

After the bootstrap filter has been built, it can be run in R by calling the `run` method of the filter, taking the number of particles to use as an argument, and returning an estimate of the log-likelihood of the data.

```
R> exampleBootstrapFilter$run(100)

[1] -15.16996
```

Running an algorithm uncompiled allows for easy testing and debugging of algorithm logic using, for example, R's `browser()` and `trace()` functions. Once an algorithm has been successfully constructed in R, it can be compiled into C++ for efficient execution. Below, we compile the bootstrap filter algorithm using the `compileNimble` function, as seen in Section 4, and run the compiled filter using 10,000 particles. Note that the model must be compiled before or in the same step as the algorithm. The `exampleModel` here was compiled in Section 4.

```
R> CexampleBootstrapFilter <- compileNimble(exampleBootstrapFilter,
+    project = exampleModel)
R> CexampleBootstrapFilter$run(10000)
R> bootstrapFilterSamples <- as.matrix(CexampleBootstrapFilter$mvEWSamples)
```

The bootstrap filter, like most particle filters in **nimbleSMC**, saves two arrays with samples from the filtering distribution. One array, named `mvEWSamples`, contains equally weighted samples from the filtering distribution. The second array, `mvWSamples`, contains non-equally weighted samples from the filtering distribution along with log weights for each sample. These arrays can be easily converted to R matrices via the `as.matrix` function as shown above.

Next, we demonstrate **nimbleSMC**'s auxiliary particle filter algorithm (Section 3.4). The auxiliary particle filter is constructed by the `buildAuxiliaryFilter` function. Users can choose between two lookahead functions: one that uses a simulation from the transition equation $\tilde{x}_{t|t-1}^{(m)} \sim f(x_t \mid x_{t-1}^{(m)})$, and one that uses the expected value of the transition equation $\tilde{x}_{t|t-1}^{(m)} = E(x_t \mid x_{t-1}^{(m)})$, via the `lookahead` control list argument. The expected value lookahead function can only be used for state-space models with normal or multivariate normal transition equations.

```
R> exampleAuxiliaryFilter <- buildAuxiliaryFilter(exampleModel, nodes = "x",
+    control = list(saveAll = TRUE, lookahead = "mean"))
R> CexampleAuxiliaryFilter <- compileNimble(exampleAuxiliaryFilter,
+    project = exampleModel, resetFunctions = TRUE)
R> CexampleAuxiliaryFilter$run(10000)
R> auxiliaryFilterSamples <- as.matrix(CexampleAuxiliaryFilter$mvEWSamples)
```

The `resetFunctions = TRUE` option is helpful when compiling a second algorithm for a model. See the **nimble** user manual for more details.

The final method we demonstrate for models with fixed parameters is the ensemble Kalman filter, which can be built via a call to `buildEnsembleKF`. Note that the ensemble Kalman filter, as described in Section 3.7, does not produce weights with its particle estimates. Thus there is only one output array, named `mvSamples`. Additionally, we note that the ensemble Kalman filter in **nimbleSMC** will only work with Gaussian noise in the process and observations, although the mean state dynamics or observations can be non-linear.

```
R> exampleEnsembleKF <- buildEnsembleKF(exampleModel, nodes = "x",
+    control = list(saveAll = TRUE))
R> CexampleEnsembleKF <- compileNimble(exampleEnsembleKF,
+    project = exampleModel, resetFunctions = TRUE)
R> CexampleEnsembleKF$run(10000)
R> EnKFSamples <- as.matrix(CexampleEnsembleKF$mvSamples)
```

Since our example model has normal transition and observation equations, the filtering distribution can also be calculated exactly using the Kalman filter (Kalman 1960). Below, we use the **dlm** package (Petris 2010) to apply a Kalman filter to our model and compare the exact filtering distribution provided by the Kalman filter to the approximate filtering distributions given by the bootstrap filter, auxiliary particle filter, and EnKF. Note that the quantiles in Figure 1 align almost exactly for all filters.

6. Inference on models with unknown parameters

The example model in the previous section had no unknown parameters – an uncommon scenario with real data. We next demonstrate **nimbleSMC**'s PMCMC and IF2 algorithms, which can be used to estimate the unknown top-level parameters in a Bayesian and frequentist framework, respectively. For PMCMC, we use a stochastic volatility example following package **stochvol** (Kastner 2016; Hosszejni and Kastner 2021). For IF2, we use a linear random

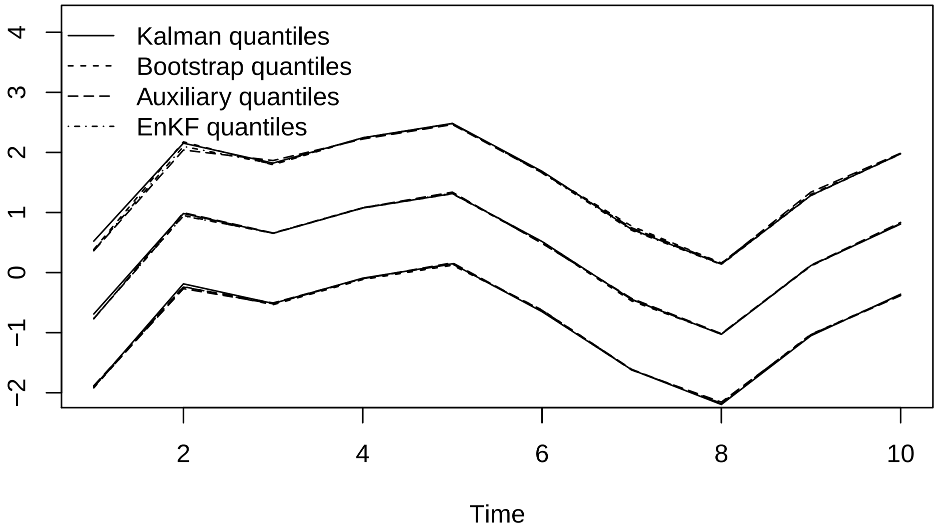

Figure 1: 2.5%, 50% and 97.5% quantiles of the filtering distribution for the Kalman filter and nimble's particle filters.

walk model of Nile river flows with a single changepoint, following the use of this example by Durbin and Koopman (2012) and in packages **dlm** (Petris 2010) and **FKF** (Luethi, Erb, Otziger, McDonald, and Smith 2021). For both methods, we present some computational experiments illustrating the roles of some tuning parameters: proposal scale for PMCMC, cooling parameter for IF2, and number of particles for both. While IF2 can handle nonlinear and/or non-Gaussian models, use of the linear, Gaussian example facilitates our computational experiment by allowing easy calculation of the correct likelihood. Both examples are also chosen to facilitate verification of correct results, by comparison to **stochvol**'s own MCMC for our PMCMC example, and by exact maximum likelihood estimation for our IF2 example.

6.1. Particle marginal Metropolis-Hastings

nimbleSMC's PMCMC uses the particle marginal Metropolis-Hastings (PMMH) method. This can take advantage of **nimble**'s existing MCMC framework, in which the user can assign different samplers to different nodes in a model. For a full description of **nimble**'s MCMC design, see Chapter 7 of the **nimble** user manual (**nimble** Development Team 2020). The PMMH sampler in **nimbleSMC** uses a normal proposal distribution, and can sample either scalar parameters (using the `RW_PF` sampler) or vectors of parameters (using the `RW_PF_block` sampler). Multiple such samplers can also be combined. Before setting up PMMH, we introduce the stochastic volatility example.

Stochastic volatility example

Stochastic volatility models are widely used for time-series of the log returns of financial assets such as stocks. The main idea is to model volatility (standard deviation of log returns) as an unobserved autoregressive process. Our example is drawn from the R package **stochvol**, described in detail in Kastner (2016). Pitt and Shephard (1999) use a related auxiliary particle filter example. We reparameterize relative to the **stochvol** example as noted below.

Let r_t be the exchange rate at time t, and define y_t as the daily log return, that is, $y_t = (\log(r_t) - \log(r_{t-1}))$ for $t = 2, \dots, T$. The stochastic volatility model is:

$$\begin{aligned} x_t &\sim N(\phi x_{t-1}, \sigma^2). \\ y_t &\sim N(0, (\beta \exp(x_t/2))^2) \end{aligned}$$

In this model, β can be interpreted as the constant volatility, while x_t is the latent, evolving log of volatility squared. For the distribution of the initial state, we use the stationary distribution:

$$x_1 \sim N(0, \sigma^2/(1-\phi^2)).$$

Prior distributions are placed on the parameters β, ϕ, and σ as follows, using the same choices as the **stochvol** vignette, specifically:

$$\begin{aligned} \phi^* &\sim B(20, 1.1), \\ \phi &= 2\phi^* - 1, \\ \Omega &\sim \frac{r^s e^{s\Omega} e^{-re^{\Omega}}}{\Gamma(s)}, s = 0.5, r = 5, \\ \sigma^2 &= \exp(\Omega), \\ \mu &\sim N(-10, 1), \\ \beta &= \exp(\mu/2), \end{aligned}$$

where the density function given for Ω is such that $\exp(\Omega)$ follows a gamma distribution with shape s and rate r.

A normal prior is placed on $\mu = 2\log\beta$. This is moderately informative, based on knowledge of typical mean volatility. The choice of -10 for the mean would be different if we used volatility on a percent scale ($100 \times y_t$ instead of y_t). An informative beta prior is placed on $\phi^* = (\phi+1)/2$. According to **stochvol**, informative priors are common because the data are only weakly informative for ϕ, yet there is domain knowledge of typical values. Finally, a moderately-informative gamma prior is placed on σ^2, as done in **stochvol**, but we parameterize that prior in terms of $\Omega = \log\sigma^2$. The latter choice is made because it facilitates better mixing when block-sampling all three parameters. In order to define Ω to follow a distribution corresponding to a gamma on σ^2, we use **nimble**'s extensibility for writing new distributions. While these choices for priors are somewhat arbitrary, sticking to the choices of the **stochvol** vignette while reparameterizing provides an illustration of the usefulness of **nimble**'s extensibility.

Code for the stochastic volatility model is:

```
R> stochVCode <- nimbleCode({
+    x[1] ~ dnorm(0, sd = sigma / sqrt(1 - phi * phi))
+    y[1] ~ dnorm(0, sd = beta * exp(0.5 * x[1]))
+    for (t in 2:T) {
+      x[t] ~ dnorm(phi * x[t - 1], sd = sigma)
+      y[t] ~ dnorm(0, sd = beta * exp(0.5 * x[t]))
+    }
+    phi <- 2 * phiStar - 1
+    phiStar ~ dbeta(20, 1.1)
+    logsigma2 ~ dgammalog(shape = 0.5, rate = 1 / (2 * 0.1))
+    sigma <- exp(0.5 * logsigma2)
+    mu ~ dnorm(-10, sd = 1)
+    beta <- exp(0.5 * mu)
+ })
```

Note that `dgammalog` is the opposite of a log-gamma distribution. That is, $\exp(\Omega)$ (rather than $\log(\Omega)$) follows a gamma distribution. Code for the density and random simulation functions for `dgammalog` is:

```
R> dgammalog <- nimbleFunction(
+    run = function(x = double(), shape = double(), rate = double(),
+                   log = integer(0, default = 0)) {
+      logProb <- shape * log(rate) + shape * x - rate * exp(x) - lgamma(shape)
+      if (log) return(logProb)
+      else return(exp(logProb))
+      returnType(double())
+    }
+ )
R> rgammalog <- nimbleFunction(
+    run = function(n = integer(), shape = double(), rate = double()) {
+      xg <- rgamma(1, shape = shape, rate = rate)
+      return(log(xg))
+      returnType(double())
+   }
+ )
```

We use as data exchange rates for the Euro (EUR) quoted in US Dollars (USD) starting after January 1st, 2010, and continuing until the end of the time-series, 582 days after that. This data set can be found in **stochvol** (Kastner 2016), along with the function `logret` to calculate log returns.

```
R> library("stochvol")
R> data("exrates", package = "stochvol")
R> y <- logret(exrates$USD[exrates$date > "2010-01-01"], demean = TRUE)
R> T <- length(y)
```

We next create and compile a `nimbleModel` object for the above BUGS code, using as starting values $\mu = -10$, $\phi^* = 0.99$, and $\log \sigma^2 = -5.52$, and providing T as a constant.

```
R> stochVolModel <- nimbleModel(code = stochVCode,
+    constants = list(T = T), data = list(y = y),
+    inits = list(mu = -10, phiStar = 0.99, logsigma2 = log(0.004)))
R> CstochVolModel <- compileNimble(stochVolModel)
```

Building and running PMMH

To implement the PMMH algorithm, we first set up an MCMC configuration for our stochastic volatility model using the `configureMCMC` function. We use the `monitors` argument to set the list of nodes for which we want **nimble** to return posterior samples.

The PMMH sampler can be added to the MCMC configuration with a call to the `addSampler` method. Additional options to customize the sampler can be specified within the `control` list.

```
R> stochVolMCMCConf <- configureMCMC(stochVolModel, nodes = NULL,
+    monitors = c("mu", "beta", "phiStar", "phi", "logsigma2", "sigma"))
R> auxpf <- buildAuxiliaryFilter(stochVolModel, "x",
+    control = list(saveAll = FALSE, smoothing = FALSE, initModel = FALSE))
R> h <- 1
R> propSD <- h * c(0.089, 0.039, 1.45)
R> m <- 100
R> stochVolMCMCConf$addSampler(target = c("mu", "phiStar", "logsigma2"),
+    type = "RW_PF_block", control = list(propCov = diag(propSD^2),
+      pf = auxpf, adaptive = FALSE, pfNparticles = m, latents = "x"))
```

In the last step above, we add a block random walk sampler with a multivariate normal proposal distribution by specifying `type = "RW_PF_block"`. This sampler is used to obtain posterior samples of the `target` parameters: `mu`, `phiStar`, and `logsigma2`. Although these are the parameters with priors, we give results below for the transformed parameters of interest, namely μ, ϕ, and σ. In this example, we chose to build the auxiliary particle filter first and provide it as a `control` argument to the sampler. Doing it this way can allow multiple samplers to share the same particle filter. If there is only one sampler (as here), one can alternatively use `pf = "auxiliary"` to have the sampler itself create the particle filter.

Once the PMMH sampler is added to the MCMC configuration, the algorithm can be built using the `buildMCMC` function and then compiled. Posterior samples are stored in `cMCMC$mvSamples`. Below we demonstrate running **nimbleSMC**'s PMMH algorithm for 50,000 iterations.

```
R> stochVolMCMC <- buildMCMC(stochVolMCMCConf)
R> cMCMC <- compileNimble(stochVolMCMC, project = stochVolModel,
+    resetFunctions = TRUE)
R> cMCMC$run(50000)
R> samples <- as.matrix(cMCMC$mvSamples)
```

Figure 2 shows traceplots of 10,000 iterations from this MCMC (right panel, M = 100). For comparisons below, traceplots with fewer particles (left panel, M = 25) are shown. These display more pronounced "stickiness", where the MCMC gets stuck for many subsequent iterations because the likelihood was overestimated when accepting a proposal in a given iteration, as discussed in Section 6.2 In harder applications, larger numbers of particles will typically be needed.

As mentioned in Section 5, **nimbleSMC** offers two choices for the lookahead function of the auxiliary particle filter, which can be specified for use in a PMMH sampler via the `pfLookahead` control list argument or specified when building the particle filter directly.

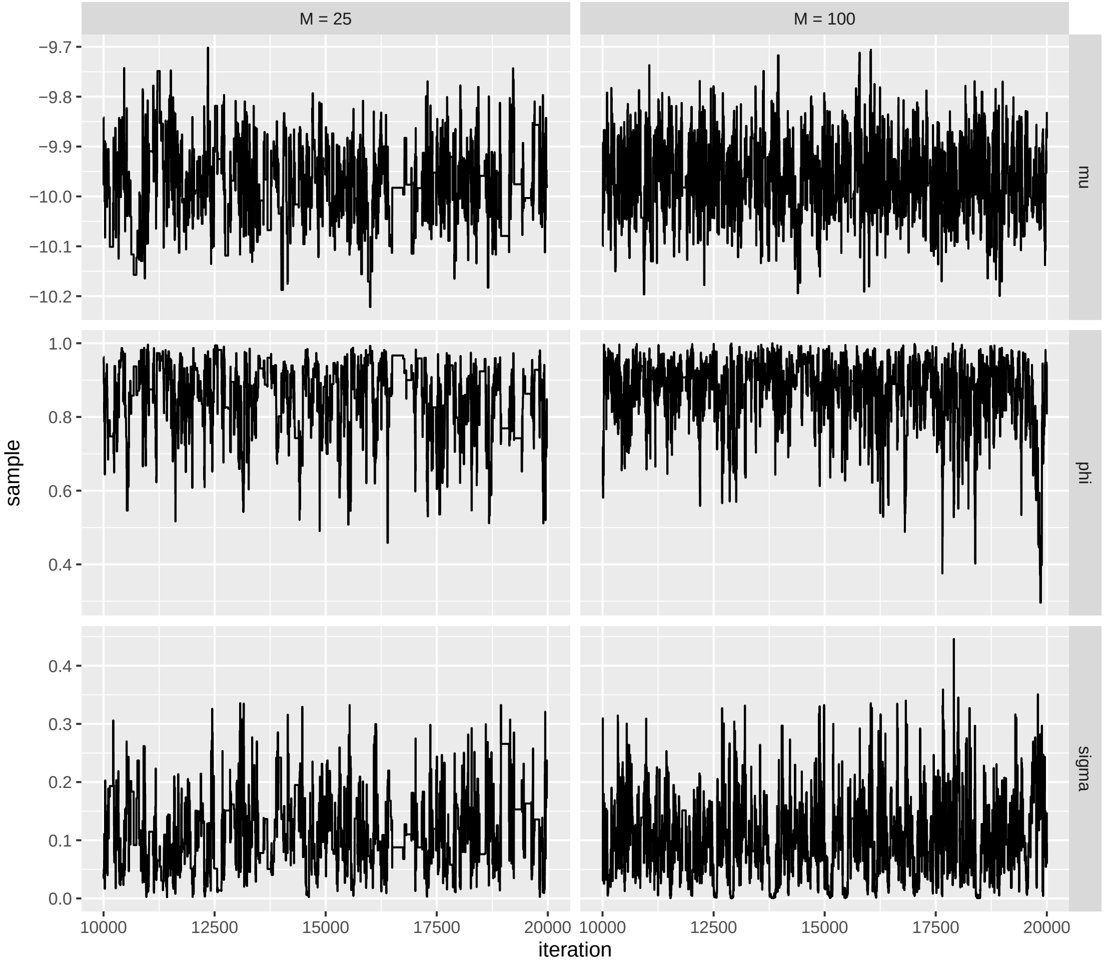

Figure 2: Traceplots of PMCMC with $M = 25$ (left) or 100 (right) particles in each run of the particle filter. 10,000 of 50,000 iterations are shown.

However, there may be cases where another choice of lookahead function is desired, or even where a user would like to use a filtering algorithm other than the bootstrap and auxiliary filters. To accommodate these scenarios, **nimbleSMC** allows for user-defined filtering algorithms within its PMCMC sampler. Thus a user could, for example, modify **nimbleSMC**'s auxiliary particle filter code to have a customized lookahead function and use this modified sampler to produce likelihood estimates in PMCMC. More information on creating user-defined filtering algorithms can be found in Section 7, and information on using such algorithms in PMCMC can be found in Section 8.1.2 of **nimble**'s User Manual (**nimble** Development Team 2020). Pitt and Shephard (1999) show that a filter that uses a good look-ahead method will tend to have a lower mean squared error than either the bootstrap or other auxiliary particle filters. Such an adapted filter would be a prime candidate for a user-defined filter in **nimbleSMC**'s particle MCMC samplers.

Similarly, **nimbleSMC**'s `RW_PF` and `RW_PF_block` samplers use normal proposal distributions, but **nimble** provides a a straightforward system for writing new samplers. See Section 13.5 of the user manual for information on user-defined MCMC samplers.

6.2. Tuning PMCMC

Next we illustrate tuning considerations for PMMH, which has aspects distinct from other MCMC methods. An intuitive interpretation of PMMH is that it uses SMC to approximate the likelihood in the Metropolis-Hastings acceptance probability. However, its theoretical justification is actually different. PMMH is valid because the entire set of simulated states and indices (during resampling) represent auxiliary variables in a complicated enhanced space for the MCMC. The marginal distribution of the variables of interest matches the target distribution, which justifies sampling in the enhanced space. This is discussed in Andrieu and Roberts (2009), Andrieu *et al.* (2010), Doucet *et al.* (2015), Sherlock *et al.* (2015), and Dahlin and Schön (2019).

This insight leads to what might be surprising performance considerations. From the intuitive interpretation, one might be tempted to increase the number of SMC particles in order to make the approximation in the acceptance probability more accurate. However, the justification of the method is valid even for small numbers of SMC particles, leading to the label "exact approximation" (Andrieu *et al.* 2010). The risk with a small number of particles is not that the algorithm will be formally invalid but rather that it will get stuck due to high variance in the SMC likelihood approximation. If a single run of the SMC yields an extremely large likelihood by chance, it will typically be accepted, after which many proposals may be rejected because the acceptance ratio will be low. Increasing SMC sample size decreases the variance of likelihood approximations, which reduces the chance of getting stuck. However, computational cost scales approximately linearly with SMC sample size, so it may be better to use fewer SMC samples and be able to run more iterations. Theory about these considerations is developed by Sherlock *et al.* (2015) and Doucet *et al.* (2015). In summary, whereas sample size is chosen for accuracy of the likelihood and state approximations in vanilla SMC, it is chosen to balance mixing and computational cost when the SMC is used in PMMH.

To illustrate these tradeoffs, we ran a set of computational experiments (Figure 3). We compared different values of SMC sample size, M, and scale of the multivariate normal random-walk proposals, h (see below). Values of M were $25, 50, \ldots, 125$. Values of h were $0.5, 0.75, \ldots, 1.5$. For each case (values of M and h), results were compared using effective sample size (ESS) and efficiency, defined as ESS per computation time (in seconds). Efficiency is the number of effectively independent samples generated per second. To estimate ESS, we used the overlapping batch means (`obm`) method of package **mcmcse** (Flegal, Hughes, Vats, and Dai 2021), with a batch size of 500. While this is larger than necessary in most cases, there are some cases where the autocorrelation can be non-zero for lags of several hundred iterations, so the choice of 500 is conservative. Ten replicates of each case were run. Figure 3 shows mean $+/-$ one standard error for ESS and efficiency for each case.

Proposals were multivariate normal with mean equal to the current values of (μ, ϕ^*, Ω) and diagonal covariance matrix with standard deviations $(0.89h,\ 0.39h,\ 1.45h)$. These standard deviations are initial estimates (from a preliminary PMCMC run) of the posterior standard deviations multiplied by h. This scheme provides a simply way to explore the role of proposal scale by using a single variable, h. One could also experiment with using univariate proposals, which typically allow larger moves in one direction at a time but would require more SMC evaluations. The setup of the present experiments provided a pragmatic way to gain insight on the impacts of proposal scale and SMC sample size on MCMC efficiency and to illustrate the tradeoffs involved.

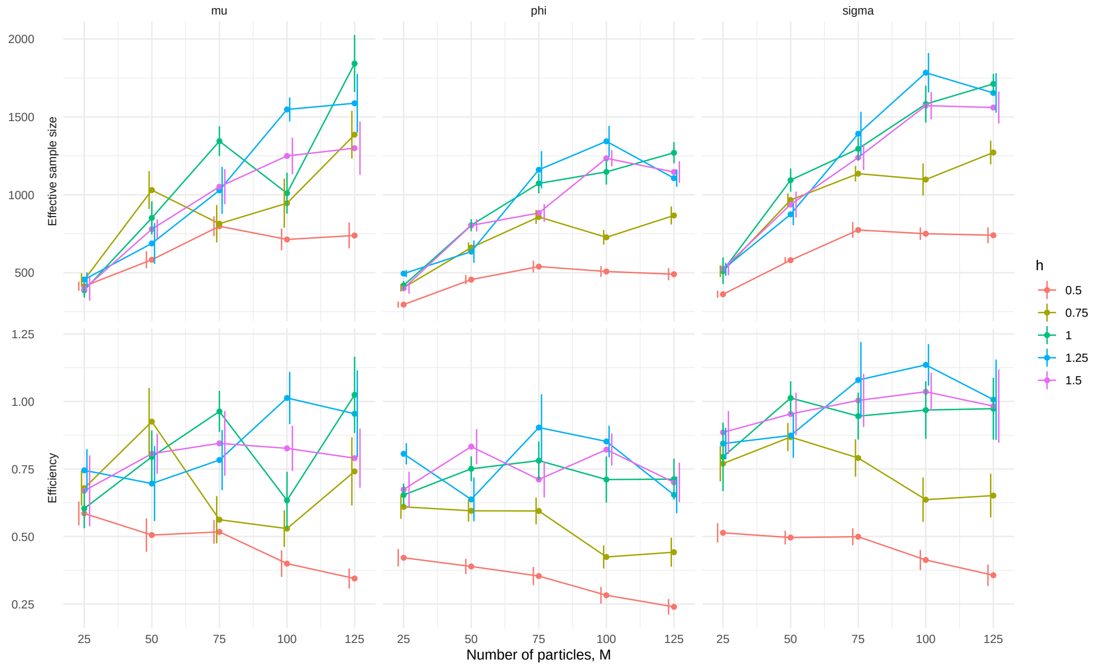

Figure 3: PMCMC effective sample size (ESS) and efficiency (ESS / computation time) for the stochastic volatility example for different numbers of particles (M) and proposal scales (h).

While the results illustrate that estimates of ESS are noisy, clear patterns nevertheless emerge. Increasing the number of particles improves mixing (ESS, Figure 3, top row) but is often not worth its computational cost (Figure 3, bottom row). A doubling in the number of particles will roughly double the computation time, so it must at least double ESS to be worth the cost, and generally this is not the case in this example. However, small numbers of particles demonstrate a pattern of "stickiness" that may not be satisfactory (Figure 2, left panel). Therefore, a good choice would be use to sufficient particles to avoid excessive stickiness while not imposing too great a computational cost.

In typical problems, one will want to use a larger number of particles. We were surprised at how small we could choose M in this example and still obtain good results, but this is an example where the model fits the data reasonably. In a case where the model cannot fit the data well, the variance in particle filter likelihood approximations will be higher, potentially causing stickiness and calling for more particles.

6.3. IF2

The steps to create an IF2 algorithm are similar to those for PMCMC: build a model, build the algorithm, compile both, and run. For IF2, the example is a Gaussian random-walk with Gaussian measurement error of Nile river flows and a changepoint at a known time, following Durbin and Koopman (2012), **dlm** and **FKF**.

Nile river flow example

Define x_t to be the state of flow in year t and y_t to be measured flow. The model is:

$$\begin{aligned} x_t &\sim N(I_{29}(t)c + x_{t-1}, \sigma^2) \\ y_t &\sim N(x_t, \sigma_M^2) \end{aligned}$$

where $I_{29}(t)$ is 1 when $t = 29$ and 0 otherwise, c is the changepoint shift in mean flow, and σ and σ_M are standard deviations of the innovations and measurements, respectively. ("Innovation" refers to process stochasticity in state-space models.) Year 29 corresponds to 1899, when the Aswan Dam apparently changed Nile flows. Package **dlm** implements this changepoint with an inflated variance of state dynamics. We instead model this concept with a shift in the mean. By use of maximum likelihood estimation, we found that the two approaches yield similar models and maximum likelihoods (not shown). Maximum likelihood estimates obtained using package **FKF** are $\hat{\sigma} = 0.01$, $\hat{\sigma}_M = 127$, and $\hat{c} = -267$. Thus, the model with MLE parameters has essentially no stochasticity in the latent states. The maximum log-likelihood is -626.4. BUGS code for this model in **nimble** is:

```
R> nileCode <- nimbleCode({
+    for (t in 1:n)
+      y[t] ~ dnorm(x[t], sd = sigmaMeasurements)
+    x[1] ~ dnorm(x0, sd = sigmaInnovations)
+    for (t in 2:n)
+      x[t] ~ dnorm((t - 1 == 28) * meanShift1899 + x[t - 1],
+                   sd = sigmaInnovations)
+    logSigmaInnovations ~ dnorm(0, sd = 100)
+    logSigmaMeasurements ~ dnorm(0, sd = 100)
+    sigmaInnovations <- exp(logSigmaInnovations)
+    sigmaMeasurements <- exp(logSigmaMeasurements)
+    x0 ~ dnorm(1120, var = 100)
+    meanShift1899 ~ dnorm(0, sd = 100)
+ })
```

The prior distributions are not used by IF2, except possibly to obtain boundaries of valid parameter values, but that is not the case here.

The next step is to build the model and algorithm and compile both:

```
R> y <- Nile
R> nileModel <- nimbleModel(nileCode, data = list(y = y),
+    constants = list(n = length(y)),
+    inits = list(logSigmaInnovations = log(sd(y)),
+      logSigmaMeasurements = log(sd(y)), meanShift1899 = -100))
R> perturbThetaSD <- c(0.1, 0.1, 5)
R> initParamSigma <- c(0.1, 0.1, 5)
R> ff <- buildIteratedFilter2(model = nileModel, nodes = "x",
+    params = c("logSigmaInnovations", "logSigmaMeasurements",
+      "meanShift1899"), baselineNode = "x0",
+    control = list(sigma = perturbThetaSD, initParamSigma = initParamSigma))
```

```
R> cNileModel <- compileNimble(nileModel)
R> cff <- compileNimble(ff, project = nileModel)
```

In `buildIteratedFilter2`, `nodes` gives the vector of latent state nodes, `params` gives the parameters to be optimized over, and `baselineNode` gives the node for the initial time point of the latent state (which should not have any data dependent on it), if applicable. The `control` list elements include `sigma`, the vector of initial perturbation standard deviations (σ_d values in Section 3.5, in the order of `params`) to be multiplied by a cooling factor, and `initParamSigma`, the vector of standard deviations of the initial particle swarm of parameters (also in the order of `params`).

```
R> numParticles <- 1000
R> numPFruns <- 100
R> alpha <- 0.2
R> est <- cff$run(m = numParticles, niter = numPFruns, alpha = alpha)
```

Here `m` is the number of particles, `niter` is the number of iterations of the IF2 algorithm, and `alpha` is the α in the equation for $c_{i,t}$ used to specify the cooling schedule via diagonal elements of $\Sigma_{i,t}$.

6.4. Tuning IF2

IF2 presents a user with numerous tuning parameters. These include the number of particles, number of particle filter runs, cooling parameter α, initial perturbation standard deviation for each parameter, and initial standard deviation for each parameter. To explore and illustrate how choices of these parameters can affect performance, we ran a computational experiment with different choices for the number of particles and the cooling parameter. Specifically, we ran the Nile flow example with `alpha` values of 0.1, 0.2, 0.4 and 0.6 and M (number of particles) values of 100, 200, 500, 1000, and 2000. Initial values were $\log(\mathrm{sd}(y_{1:T}))$ for both log sigmas and -100 for the changepoint shift, c. Perturbation standard deviations were 0.1, 0.1, and 5, and standard deviations of the initial particle swarm were the same.

For each combination of α and M, we ran IF2 for a number of iterations such that there were 10^5 total particles simulated through the entire time-series, or 10^7 particle time-steps (since the length of the data is $T = 100$). For example, with $M = 100$, we used $I = 1000$ iterations, while with $M = 500$, we used $I = 200$ iterations. Since computational cost is closely related to number of particle time steps, this arrangement means that each case was run for a similar total computational effort. The horizontal axis in Figure 4 shows iterations but can be easily interpreted in terms of computational cost. For example, half-way across the x-axis represents a different number of iterations but similar computational cost for each case.

To examine the path to convergence in each case, we calculated the correct likelihood using the Kalman Filter at each iteration of IF2. The ability to calculate the correct likelihood is the reason for using a linear, Gaussian example.

Figure 4 reveals several useful insights about IF2. With too few particles, IF2 can move too noisily to find the MLE before the cooling schedule effectively stops further exploration, as seen in the $M = 100$ and $M = 200$ cases. This problem is exacerbated by smaller choices of α (faster cooling schedule). On the other hand, choosing a larger α will incur more computational cost by requiring more iterations to converge, as seen in the $M = 500$ case,

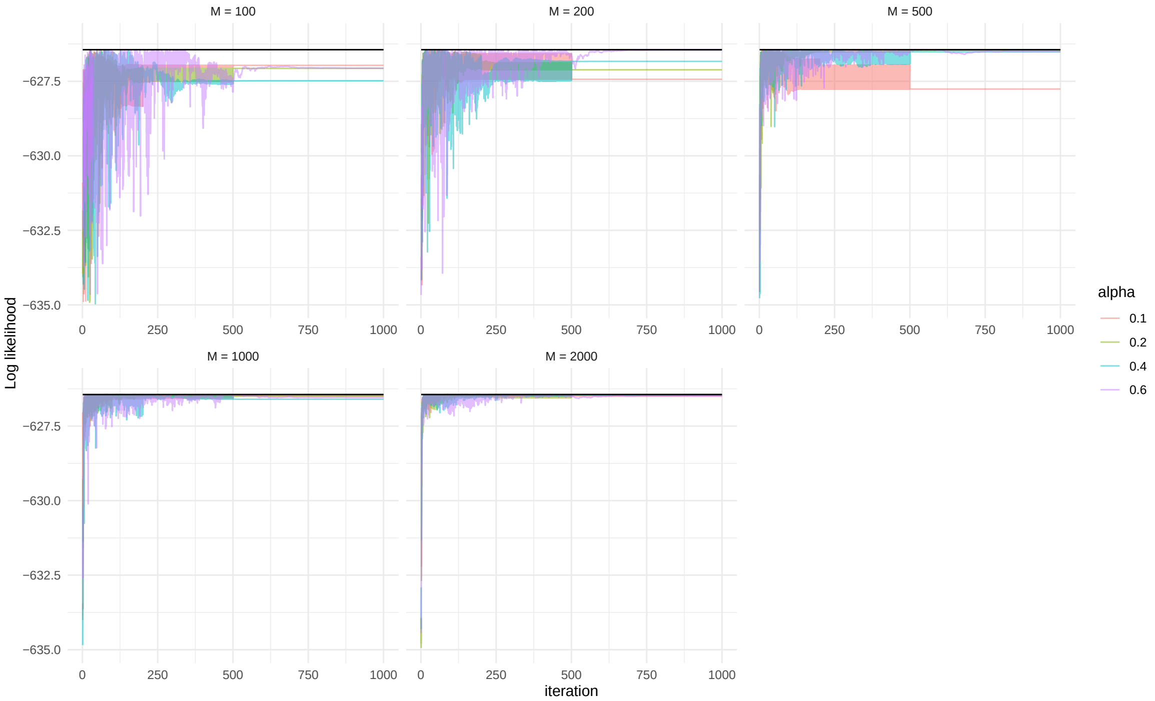

Figure 4: IF2 log-likelihood vs. iteration for the Nile river flow example for different numbers of particles (M) and cooling parameters (α).

where all values of α appear to be converging to the MLE but larger ones are doing so more slowly. Finally, at large values of M, further increases in M will yield little additional benefit. What we have not explored here are choices for initial perturbation standard deviations (σ_d). In general, relatively small values for these can work well because perturbations are applied at every time step of every particle filter run.

7. Programming SMC algorithms in nimble

In this section, we show how **nimbleSMC**'s SMC algorithms are implemented as `nimbleFunctions`, a high-level system for programming model-generic algorithms. We will not cover every detail of programming in **nimble**, but rather we wish to show how algorithms are expressed compactly in high-level code that gets compiled via C++. This will allow other programmers to quickly adapt our functions for their own needs. For a more detailed discussion of `nimbleFunction` programming, see de Valpine *et al.* (2017) or Section IV of the **nimble** user manual (**nimble** Development Team 2020).

We demonstrate programming in **nimble** by providing code for a bootstrap filtering algorithm. Note that the code shown below is simpler than the actual implementation of the bootstrap filter available in **nimbleSMC** through the `buildBootstrapFilter` function. However, this demonstration code is indeed a fully functional bootstrap filter. The bootstrap filter included in the **nimbleSMC** package simply has more customization options, and can handle somewhat more complicated models, than the demonstration algorithm provided here.

nimble programming uses two-stage evaluation defined by a `nimbleFunction`, which has two different types of code: `setup` code and `run` code. When a `nimbleFunction` is called, the `setup` code is evaluated first. `setup` code is written in R and is primarily used to extract model information for later use in the `run` code. The purpose of extracting model information is to specialize the algorithm to a particular model. After `setup` code has been run once, the `run` code can be executed many times. `run` code is written in the **nimble** domain-specific language (DSL) for hierarchical model algorithms, which allows the code to be compiled into C++, in turn providing efficient execution of an algorithm's computations. `run` code can make use of objects created in the `setup` code.

The first `nimbleFunction` shown below, named `bootstrapFilter`, includes `setup` and `run` code that manage the time-iterations of the bootstrap filter. It uses a list of other `nimbleFunctions`, each of which is responsible for one time step. To set up this list, it iterates through time and calls the `nimbleFunction` `bootstrapStep` for each time. The `setup` code of `bootstrapStep` takes information about the latent state at time t and determines how to use the model to conduct Steps 6 through 18 of the bootstrap filter algorithm given in Section 3.3 in its `run` code. As the filtering algorithm progresses through each time point, samples from the filtering distribution at that time will be saved in **nimble** `modelValues` objects. `modelValues` objects provide generic containers for storing sets of values of model nodes.

Below is the call to `nimbleFunction` that defines `setup` and `run` code for the `bootstrapFilter`. The `setup` code takes an argument called `model`, which must be a **nimble** model object created by a call to `nimbleModel` (described in Section 4). It also takes the names of the latent states as argument `latentNodes`. The `setup` code first specializes a `nimbleFunction` that will initialize the model and then obtains the names and dimensions of the latent states in the model in time order. Two `modelValues` objects are created to store samples from the latent states. The `mvWSamples` object will store a non-equally weighted sample, while `mvEWSamples` will store an equally weighted sample. Finally, the `setup` code creates a list of `bootstrapStep` functions (called a `nimbleFunctionList`). For each time point $t = 1, \ldots, T$, the list contains one `bootstrapStep` function, which will conduct one time-step of the bootstrap filter. We note that the creation of a separate `bootstrapStep` function for each time t is necessary to allow the latent state x_t at each time to have arbitrary observation dependencies y_t, which may even have been declared with different model code for different times.

```
R> bootstrapFilter <- nimbleFunction(
+    setup = function(model, latentNodes) {
+      my_initializeModel <- initializeModel(model)
+      latentNodes <- model$expandNodeNames(latentNodes, sort = TRUE)
+      dims <- lapply(latentNodes, function(n) nimDim(model[[n]]))
+      mvWSpec <- modelValuesConf(vars = c("x", "wts"),
+        types = c("double", "double"),
+        sizes = list(x = dims[[1]], wts = 1))
+      mvWSamples <- modelValues(mvWSpec)
+      mvEWSpec <- modelValuesConf(vars = c("x"), types = c("double"),
+        sizes = list(x = dims[[1]]))
+      mvEWSamples <- modelValues(mvEWSpec)
+      bootStepFunctions <- nimbleFunctionList(bootstrapStepVirtual)
+      timePoints <- length(latentNodes)
+       for (t in 1:timePoints)
+         bootStepFunctions[[t]] <- bootstrapStep(model, mvWSamples,
+           mvEWSamples, latentNodes, t)
+     },
+     run = function(M = integer()) {
+       my_initializeModel$run()
+       resize(mvWSamples, M)
+       resize(mvEWSamples, M)
+       for (t in 1:timePoints)
+         bootStepFunctions[[t]]$run(M)
+     }
+ )
```

The `run` code for the `bootstrapFilter` function takes as its only argument the number of particles (M) to use for estimation. `run` code requires explicit declaration of the type of any arguments, so here M is specified as a scalar integer. In general, the type of a return object must also be declared, although this function does not return anything so no declaration is necessary. The `run` function first initializes the model (conducting Steps 2 and 3 of the bootstrap filter algorithm), and then re-sizes the `modelValues` objects so that they can store M particles. After that, the `run` function iterates through each time point, running the `bootstrapStep` function that was defined for that time point in the `setup` code. To keep the example simple, this version does not provide an estimate of the likelihood $\tilde{p}(y_{1:T})$.

Creating a `nimbleFunctionList`, such as the one used in the `setup` code above, requires an additional piece of code that informs **nimble** about the input arguments and return objects of each function in that list. Specifically, the `nimbleFunctionVirtual` function is used to define the methods and their argument and return types that each element in the `nimbleFunctionList` will have. Below, we specify that each element of our `nimbleFunctionList` will have a run function with a single integer input and no return object.

```
R> bootstrapStepVirtual <- nimbleFunctionVirtual(
+     run = function(M = integer()) {}
+ )
```

`setup` and `run` code for the `bootstrapStep` function are given below. At each time point t, the `setup` function gets the names and deterministic dependencies of the previous and current latent states. The `run` code first declares a length M vector of integers (to store particle indices) and a length M vector of doubles (to store particle weights). The `run` code then iterates through the particles. For each particle, the code takes the value of the latent state at $t-1$ from the equally weighted `modelValues` object, uses that value to propagate a value for the latent state at time t, and calculates a weight. The particles and corresponding weights are stored in the non-equally weighted `modelValues` object. Finally, particles are resampled proportional to their weights and the resampled particles are stored in the equally weighted `modelValues` object.

In this algorithm particles are propagated using the proposal distribution $q(x_t \mid x_{t-1}^{(m)}, y_t) = f(x_t \mid x_{t-1}^{(m)})$, which simplifies the weight calculation in Step 8 of Algorithm 1. Additionally, since resampling is performed at each time point, weights from time $t-1$ do not need to be used when calculating weights at time t. This results in a weight calculation of $w_t^{(m)} = g(y_t \mid \tilde{x}_t^{(m)})$.

```
R> bootstrapStep <- nimbleFunction(
+    contains = bootstrapStepVirtual,
+    setup = function(model, mvWSamples, mvEWSamples, latentNodes,
+      timePoint) {
+      notFirst <- timePoint != 1
+      prevNode <- latentNodes[if (notFirst) timePoint - 1 else timePoint]
+      thisNode <- latentNodes[timePoint]
+      prevDeterm <- model$getDependencies(prevNode, determOnly = TRUE)
+      thisDeterm <- model$getDependencies(thisNode, determOnly = TRUE)
+      thisData   <- model$getDependencies(thisNode, dataOnly = TRUE)
+    },
+    run = function(M = integer()) {
+      ids <- integer(M, 0)
+      wts <- numeric(M, 0)
+      for (m in 1:M) {
+        if (notFirst) {
+          copy(from = mvEWSamples, to = model, nodes = "x",
+            nodesTo = prevNode, row = m)
+          model$calculate(prevDeterm)
+        }
+        model$simulate(thisNode)
+        copy(from = model, to = mvWSamples, nodes = thisNode,
+          nodesTo = "x", row = m)
+        model$calculate(thisDeterm)
+        wts[m] <- exp(model$calculate(thisData))
+        mvWSamples["wts", m][1] <<- wts[m]
+      }
+      rankSample(wts, M, ids)
+      for (m in 1:M) {
+        copy(from = mvWSamples, to = mvEWSamples, nodes = "x",
+          nodesTo = "x", row = ids[m], rowTo = m)
+      }
+    })
```

The calls to `calculate` within the above `run` code serve two purposes. The first two `calculate` calls are used to calculate the values of any deterministic dependencies of the latent state, as these dependencies must be recalculated any time the latent state takes on a new value. The third call to `calculate` is used to calculate the log-likelihood of the data given the current latent state value, which is then used as a particle weight. The `rankSample` function fills the elements of the `ids` vector with the indices of the particles that have been chosen in the resampling procedure.

Once the `nimbleFunction`s have been defined, we can build, compile, and run the bootstrap filter. The code below runs the example filter on the `exampleModel` of Section 5 and creates a histogram of samples from the filtering distribution of x at the last time point.

```
R> myBootstrap <- bootstrapFilter(exampleModel, "x")
R> cmyBootstrap <- compileNimble(myBootstrap, project = exampleModel,
+    resetFunctions = TRUE)
```

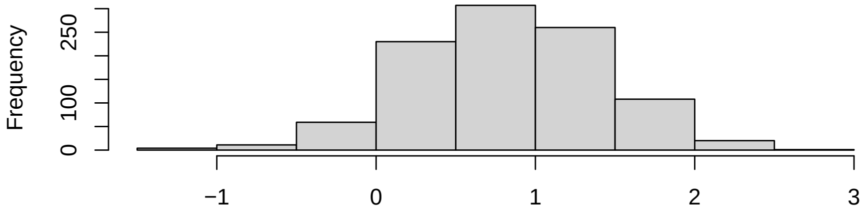

Figure 5: A histogram of the filtering distribution of x.

```
R> cmyBootstrap$run(1000)
R> filterSamps <- as.matrix(cmyBootstrap$mvEWSamples, "x")
R> hist(filterSamps, main = "", xlab = "")
```

The bootstrap filter code provided above demonstrates **nimble**'s ability to program model-generic algorithms. The filter could be used to conduct filtering on any correctly specified state-space model. In addition to the generality of the algorithm, it would be relatively straightforward to modify the filter, changing it to an auxiliary particle filter, an IF2 algorithm, or a filter type not currently included in **nimbleSMC**. The ease with which existing algorithms can be modified, along with the generality with which they are written, promotes the development of user-written filters.

8. Conclusion

This paper has described **nimbleSMC**'s suite of SMC algorithms, which provide straightforward methods of conducting inference on state-space models. In addition, **nimble**'s model-generic programmability make it well-suited for implementing new SMC algorithms, an example of which was given in Section 7. **nimble**'s flexible model specification also enables the application of existing algorithms to more general models or new settings. For example, a model could be written where a number of state-space models are set within a larger hierarchical structure. Using **nimble** and **nimbleSMC**, SMC algorithms could be used to estimate the individual state-space models, while an MCMC algorithm could conduct inference on higher-level parameters. **nimble** also provides easily accessible model comparison tools that can be used in conjunction with its state-space modeling algorithms, which we hope will allow users to answer previously difficult questions about their time-series data.

Additional examples of modeling and inference using **nimble** can be found at https://R-nimble.org/.

Acknowledgments

This work was supported by the NSF under grants DBI-1147230 and ACI-1550488, and by the Google Summer of Code.

References

Anderson BDO, Moore JB (1979). *Optimal Filtering*. Prentice-Hall.

Anderson JL (2001). "An Ensemble Adjustment Kalman Filter for Data Assimilation." *Monthly Weather Review*, **129**(12), 2884–2903. doi:10.1175/1520-0493(2001)129<2884:aeakff>2.0.co;2.

Andersson H, Britton T (2000). *Stochastic Epidemic Models and Their Statistical Analysis*, volume 151. Springer-Verlag. doi:10.1007/978-1-4612-1158-7.

Andrieu C, Doucet A, Holenstein R (2010). "Particle Markov Chain Monte Carlo Methods." *Journal of the Royal Statistical Society B*, **72**(3), 269–342. doi:10.1111/j.1467-9868.2009.00736.x.

Andrieu C, Roberts GO (2009). "The Pseudo-Marginal Approach for Efficient Monte Carlo Computations." *The Annals of Statistics*, **37**(2), 697–725. doi:10.1214/07-aos574.

Arulampalam MS, Maskell S, Gordon N, Clapp T (2002). "A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesian Tracking." *IEEE Transactions on Signal Processing*, **50**(2), 174–188. doi:10.1109/78.978374.

Azman AS, Luquero FJ, Ciglenecki I, Grais RF, Sack DA, Lessler J (2015). "The Impact of a One-Dose Versus Two-Dose Oral Cholera Vaccine Regimen in Outbreak Settings: A Modeling Study." *PLOS Medicine*, **12**(8), 1–18. doi:10.1371/journal.pmed.1001867.

Bell BM (2005). "**CppAD**: A Package for C++ Algorithmic Differentiation." URL http://www.coin-or.org/CppAD.

Bernton E, Jacob PE, Gerber M, Robert CP (2017). "Inference in Generative Models Using the Wasserstein Distance." arXiv:1701.05146v2 [stat.ME], URL https://arxiv.org/abs/1701.05146.

Bocher M, Fournier A, Coltice N (2018). "Ensemble Kalman Filter for the Reconstruction of the Earth's Mantle Circulation." *Nonlinear Processes in Geophysics*, **25**(1), 99–123. doi:10.5194/npg-25-99-2018.

Caffo BS, Jank W, Jones GL (2005). "Ascent-Based Monte Carlo Expectation-Maximization." *Journal of the Royal Statistical Society B*, **67**(2), 235–251. doi:10.1111/j.1467-9868.2005.00499.x.

Carpenter B, Gelman A, Hoffman M, Lee D, Goodrich B, Betancourt M, Brubaker M, Guo J, Li P, Riddell A (2017). "Stan: A Probabilistic Programming Language." *Journal of Statistical Software*, **76**(1), 1–32. ISSN 1548-7660. doi:10.18637/jss.v076.i01.

Carpenter J, Clifford P, Fearnhead P (1999). "Improved Particle Filter for Nonlinear Problems." *IEEE Proceedings-Radar, Sonar and Navigation*, **146**(1), 2–7. doi:10.1049/ip-rsn:19990255.

Carvalho C, Johannes MS, Lopes HF, Polson N (2010). "Particle Learning and Smoothing." *Statistical Science*, **25**(1), 88–106. doi:10.1214/10-sts325.

Chopin N, Jacob PE, Papaspiliopoulos O (2013). "SMC2: An Efficient Algorithm for Sequential Analysis of State Space Models." *Journal of the Royal Statistical Society B*, **75**(3), 397–426. doi:10.1111/j.1467-9868.2012.01046.x.

Croce P, Zappasodi F, Merla A, Chiarelli AM (2017). "Exploiting Neurovascular Coupling: A Bayesian Sequential Monte Carlo Approach Applied to Simulated EEG fNIRS Data." *Journal of Neural Engineering*, **14**(4), 046029. doi:10.1088/1741-2552/aa7321.

Dahlin J, Schön TB (2019). "Getting Started with Particle Metropolis-Hastings for Inference in Nonlinear Dynamical Models." *Journal of Statistical Software*, **88**(1), 1–41. doi:10.18637/jss.v088.c02.

de Valpine P, Paciorek C, Turek D, Michaud N, Anderson-Bergman C, Obermeyer F, Wehrhahn Cortes C, Rodríguez A, Temple Lang D, Paganin S (2021). ***nimble**: MCMC, Particle Filtering, and Programmable Hierarchical Modeling.* R package version 0.12.1, URL https://CRAN.R-project.org/package=nimble.

de Valpine P, Turek D, Paciorek CJ, Anderson-Bergman C, Lang DT, 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.

Del Moral P, Doucet A, Jasra A (2012). "An Adaptive Sequential Monte Carlo Method for Approximate Bayesian Computation." *Statistics and Computing*, **22**(5), 1009–1020. doi:10.1007/s11222-011-9271-y.

Douc R, Cappe O (2005). "Comparison of Resampling Schemes for Particle Filtering." In *ISPA 2005 – Proceedings of the 4th International Symposium on Image and Signal Processing and Analysis, 2005*, pp. 64–69. doi:10.1109/ispa.2005.195385.

Doucet A, de Freitas N, Gordon N (2001). "An Introduction to Sequential Monte Carlo Methods." In *Sequential Monte Carlo Methods in Practice*, pp. 3–14. Springer-Verlag.

Doucet A, Godsill S, Andrieu C (2000). "On Sequential Monte Carlo Sampling Methods for Bayesian Filtering." *Statistics and Computing*, **10**(3), 197–208. doi:10.1023/a:1008935410038.

Doucet A, Johansen AM (2011). "A Tutorial on Particle Filtering and Smoothing: Fifteen Years Later." In *The Oxford Handbook of Nonlinear Filtering*, pp. 656–704. Oxford University Press.

Doucet A, Pitt MK, Deligiannidis G, Kohn R (2015). "Efficient Implementation of Markov Chain Monte Carlo When Using an Unbiased Likelihood Estimator." *Biometrika*, **102**(2), 295–313. doi:10.1093/biomet/asu075.

Durbin J, Koopman SJ (2012). *Time Series Analysis by State Space Methods*. 2nd edition. Oxford University Press, Oxford. doi:10.1093/acprof:oso/9780199641178.001.0001.

Evensen G (2003). "The Ensemble Kalman Filter: Theoretical Formulation and Practical Implementation." *Ocean Dynamics*, **53**, 343–367. doi:10.1007/s10236-003-0036-9.

Fernández-Villaverde J, Rubio-Ramírez JF (2007). "Estimating Macroeconomic Models: A Likelihood Approach." *The Review of Economic Studies*, **74**(4), 1059–1087. doi:10.1111/j.1467-937x.2007.00437.x.

Flegal JM, Hughes J, Vats D, Dai N (2021). ***mcmcse**: Monte Carlo Standard Errors for MCMC.* R package version 1.5-0, URL `https://CRAN.R-project.org/package=mcmcse`.

Gilbert P (2020). ***dse**: Dynamic Systems Estimation (Time Series Package).* R package version 2020.2-1, URL `https://CRAN.R-project.org/package=dse`.

Gillijns S, Mendoza OB, Chandrasekar J, De Moor B, Bernstein DS, Ridley A (2006). "What Is the Ensemble Kalman Filter and How Well Does It Work?" In *American Control Conference, 2006*. IEEE. `doi:10.1109/acc.2006.1657419`.

Gordon NJ, Salmond DJ, Smith AFM (1993). "Novel Approach to Nonlinear/Non-Gaussian Bayesian State Estimation." *IEEE Proceedings F – Radar and Signal Processing*, **140**(2), 107–113. `doi:10.1049/ip-f-2.1993.0015`.

Heidari L, Gervais V, Ravalec ML, Wackernagel H (2013). "History Matching of Petroleum Reservoir Models by the Ensemble Kalman Filter and Parameterization Methods." *Computers & Geosciences*, **55**, 84 – 95. `doi:10.1016/j.cageo.2012.06.006`.

Helske J (2017). "**KFAS**: Exponential Family State Space Models in R." *Journal of Statistical Software*, **78**(10), 1–39. `doi:10.18637/jss.v078.i10`.

Hosszejni D, Kastner G (2021). "Modeling Univariate and Multivariate Stochastic Volatility in R with **stochvol** and **factorstochvol**." *Journal of Statistical Software*, **100**(12), 1–34. `doi:10.18637/jss.v100.i12`.

Houtekamer PL, Zhang F (2016). "Review of the Ensemble Kalman Filter for Atmospheric Data Assimilation." *Monthly Weather Review*, **144**(12), 4489–4532. `doi:10.1175/mwr-d-15-0440.1`.

Ionides EL, Bhadra A, Atchadé Y, King A (2011). "Iterated Filtering." *The Annals of Statistics*, **39**(3), 1776–1802. `doi:10.1214/11-aos886`.

Ionides EL, Bretó C, King AA (2006). "Inference for Nonlinear Dynamical Systems." *Proceedings of the National Academy of Sciences of the United States of America*, **103**(49), 18438–18443. `doi:10.1073/pnas.0603181103`.

Ionides EL, Nguyen D, Atchadé Y, Stoev S, King AA (2015). "Inference for Dynamic and Latent Variable Models via Iterated, Perturbed Bayes Maps." *Proceedings of the National Academy of Sciences of the United States of America*, **112**(3), 719–724. `doi:10.1073/pnas.1410597112`.

Jacob PE, Funk S (2021). ***rbi**: R Interface to **LibBi**.* R package version 0.10.4, URL `https://CRAN.R-project.org/package=rbi`.

Kalman RE (1960). "A New Approach to Linear Filtering and Prediction Problems." *Transactions of the ASME, Journal of Basic Engineering, Series D*, **82**, 35–45. `doi:10.1115/1.3662552`.

Kantas N, Doucet A, Singh SS, Maciejowski J, Chopin N (2015). "On Particle Methods for Parameter Estimation in State-Space Models." *Statistical Science*, **30**(3), 328–351. `doi:10.1214/14-sts511`.

Kastner G (2016). "Dealing with Stochastic Volatility in Time Series Using the R Package **stochvol**." *Journal of Statistical Software*, **69**(1), 1–30. doi:10.18637/jss.v069.i05.

Katzfuss M, Stroud JR, Wikle CK (2016). "Understanding the Ensemble Kalman Filter." *The American Statistician*, **70**(4), 350–357. doi:10.1080/00031305.2016.1141709.

King A, Nguyen D, Ionides E (2016). "Statistical Inference for Partially Observed Markov Processes via the R Package **pomp**." *Journal of Statistical Software*, **69**(1), 1–43. doi:10.18637/jss.v069.i12.

King AA, Domenech de Cellès M, Magpantay FMG, Rohani P (2015). "Avoidable Errors in the Modelling of Outbreaks of Emerging Pathogens, with Special Reference to Ebola." *Proceedings of the Royal Society of London B: Biological Sciences*, **282**(1806). doi:10.1098/rspb.2015.0347.

Knape J, de Valpine P (2012). "Fitting Complex Population Models by Combining Particle Filters with Markov Chain Monte Carlo." *Ecology*, **93**(2), 256–263. doi:10.1890/11-0797.1.

Kristensen K, Nielsen A, Berg C, Skaug H, Bell B (2016). "**TMB**: Automatic Differentiation and Laplace Approximation." *Journal of Statistical Software*, **70**(5), 1–21. doi:10.18637/jss.v070.i05.

Liu J, Wang W, Ma F (2011). "A Regularized Auxiliary Particle Filtering Approach for System State Estimation and Battery Life Prediction." *Smart Materials and Structures*, **20**(7), 075021. doi:10.1088/0964-1726/20/7/075021.

Liu J, West M (2001). "Combined Parameter and State Estimation in Simulation-Based Filtering." In *Sequential Monte Carlo Methods in Practice*, pp. 197–223. Springer-Verlag. doi:10.1007/978-1-4757-3437-9_10.

Luethi D, Erb P, Otziger S, McDonald D, Smith P (2021). ***FKF**: Fast Kalman Filter.* R package version 0.2.2, URL https://CRAN.R-project.org/package=FKF.

Lunn D, Jackson C, Best N, Thomas A, Spiegelhalter D (2012). *The BUGS Book: A Practical Introduction to Bayesian Analysis.* Texts in Statistical Science. Chapman & Hall/CRC. doi:10.1201/b13613.

Lunn DJ, Thomas A, Best N, Spiegelhalter D (2000). "**WinBUGS** – A Bayesian Modelling Framework: Concepts, Structure, and Extensibility." *Statistics and Computing*, **10**(4), 325–337. doi:10.1023/a:1008929526011.

Mandel J (2009). "A Brief Tutorial on the Ensemble Kalman Filter." arXiv:0901.3725 [physics.ao-ph], URL https://arxiv.org/abs/0901.3725.

Martinez-Bakker M, King AA, Rohani P (2015). "Unraveling the Transmission Ecology of Polio." *PLOS Biology*, **13**(6), 1–21. doi:10.1371/journal.pbio.1002172.

Michaud N, de Valpine P, Paciorek C, Turek D (2020). ***nimbleSMC**: Sequential Monte Carlo Methods for **nimble**.* R package version 0.10.0, URL https://CRAN.R-project.org/package=nimbleSMC.

Murray L (2015). "Bayesian State-Space Modelling on High-Performance Hardware Using **LibBi**." *Journal of Statistical Software*, **67**(1), 1–36. doi:10.18637/jss.v067.i10.

Murray LM, Jones EM, Parslow J (2013). "On Disturbance State-Space Models and the Particle Marginal Metropolis-Hastings Sampler." *SIAM/ASA Journal of Uncertainty Quantification*, **1**(1), 494–521. doi:10.1137/130915376.

Newman KB, Fernández C, Thomas L, Buckland ST (2009). "Monte Carlo Inference for State-Space Models of Wild Animal Populations." *Biometrics*, **65**(2), 572–583. doi:10.1111/j.1541-0420.2008.01073.x.

nimble Development Team (2020). ***nimble** User Manual.* doi:10.5281/zenodo.4091503.

Oladyshkin S, Class H, Nowak W (2013). "Bayesian Updating via Bootstrap Filtering Combined with Data-Driven Polynomial Chaos Expansions: Methodology and Application to History Matching for Carbon Dioxide Storage in Geological Formations." *Computational Geosciences*, **17**(4), 671–687. doi:10.1007/s10596-013-9350-6.

Petris G (2010). "An R Package for Dynamic Linear Models." *Journal of Statistical Software*, **36**(1), 1–16. doi:10.18637/jss.v036.i12.

Petris G, Petrone S (2011). "State Space Models in R." *Journal of Statistical Software*, **41**(4), 1–25. doi:10.18637/jss.v041.i04.

Pitt MK (2002). "Smooth Particle Filters for Likelihood Evaluation and Maximisation." *The Warwick Economics Research Paper Series (TWERPS) 651*, University of Warwick, Department of Economics. URL http://wrap.warwick.ac.uk/1536/1/WRAP_Pitt_twerp651.pdf.

Pitt MK, Shephard N (1999). "Filtering via Simulation: Auxiliary Particle Filters." *Journal of the American Statistical Association*, **94**(446), 590–599. doi:10.1080/01621459.1999.10474153.

Plummer M (2003). "**JAGS**: A Program for Analysis of Bayesian Graphical Models Using Gibbs Sampling." In K Hornik, F Leisch, A Zeileis (eds.), *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003).* Technische Universität Wien, Vienna, Austria. URL https://www.R-project.org/conferences/DSC-2003/Proceedings/Plummer.pdf.

Pooley CM, Bishop SC, Marion G (2015). "Using Model-Based Proposals for Fast Parameter Inference on Discrete State Space, Continuous-Time Markov Processes." *Journal of The Royal Society Interface*, **12**(107). doi:10.1098/rsif.2015.0225.

Rasmussen DA, Volz EM, Koelle K (2014). "Phylodynamic Inference for Structured Epidemiological Models." *PLOS Computational Biology*, **10**(4), 1–16. doi:10.1371/journal.pcbi.1003570.

R Core Team (2021). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Sherlock C, Thiery AH, Roberts GO, Rosenthal JS (2015). "On the Efficiency of Pseudo-Marginal Random Walk Metropolis Algorithms." *The Annals of Statistics*, **43**(1), 238–275. doi:10.1214/14-aos1278.

Silburt A, Gaidos E, Wu Y (2015). "A Statistical Reconstruction of the Planet Population around Kepler Solar-Type Stars." *The Astrophysical Journal*, **799**(2), 180. doi:10.1088/0004-637x/799/2/180.

Smith A, Doucet A, de Freitas N, Gordon N (2001). *Sequential Monte Carlo Methods in Practice*. Springer-Verlag, New York, NY. doi:10.1007/978-1-4757-3437-9.

Storvik G (2002). "Particle Filters for State-Space Models with the Presence of Unknown Static Parameters." *IEEE Transactions on Signal Processing*, **50**(2), 281–289. doi:10.1109/78.978383.

Stroustrup B (2013). *The C++ Programming Language*. 4th edition. Addison-Wesley.

Todeschini A, Caron F, Fuentes M, Legrand P, Del Moral P (2014). "**Biips**: Software for Bayesian Inference with Interacting Particle Systems." arXiv:1412.3779 [stat.CO], URL https://arxiv.org/abs/1412.3779.

Toni T, Welch D, Strelkowa N, Ipsen A, Stumpf MPH (2009). "Approximate Bayesian Computation Scheme for Parameter Inference and Model Selection in Dynamical Systems." *Journal of The Royal Society Interface*, **6**(31), 187–202. doi:10.1098/rsif.2008.0172.

Tusell F (2011). "Kalman Filtering in R." *Journal of Statistical Software*, **39**(2), 1–27. doi:10.18637/jss.v039.i02.

Vrugt JA, Ter Braak CJF, Diks CGH, Schoups G (2013). "Hydrologic Data Assimilation Using Particle Markov Chain Monte Carlo Simulation: Theory, Concepts and Applications." *Advances in Water Resources*, **51**, 457 – 478. doi:10.1016/j.advwatres.2012.04.002.

Wikle CK, Milliff RF, Herbei R, Leeds WB (2013). "Modern Statistical Methods in Oceanography: A Hierarchical Perspective." *Statistical Science*, **28**(4), 466–486. doi:10.1214/13-sts436.

Yang Y, Eisenstein J (2013). "A Log-Linear Model for Unsupervised Text Normalization." In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pp. 61–72.

Yu C, El-Sheimy N, Lan H, Liu Z (2017). "Map-Based Indoor Pedestrian Navigation Using an Auxiliary Particle Filter." *Micromachines*, **8**(7). doi:10.3390/mi8070225.

Zhou Y (2015). "**vSMC**: Parallel Sequential Monte Carlo in C++." *Journal of Statistical Software*, **62**(1), 1–49. doi:10.18637/jss.v062.i09.

Affiliation:

Nicholas Michaud, Perry de Valpine
Department of Environmental Science, Policy, and Management
University of California, Berkeley
Berkeley, California, 94720, United States of America
E-mail: nicholas.michaud@gmail.com, pdevalpine@berkeley.edu

Daniel Turek
Department of Mathematics and Statistics
Williams College
Williamstown, Massachusetts, 01267, United States of America
E-mail: dbt1@williams.edu

Christopher J. Paciorek
Department of Statistics
University of California, Berkeley
Berkeley, California, 94720, United States of America
E-mail: paciorek@stat.berkeley.edu

Dao Nguyen
Department of Mathematics
University of Mississippi
University, Mississippi, 38677, United States of America
E-mail: dxnguyen@olemiss.edu

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
November 2021, Volume 100, Issue 3 *Submitted:* 2016-08-26
doi:10.18637/jss.v100.i03 *Accepted:* 2020-02-26