Journal of Statistical Software

December 2014, Volume 62, Issue 9. http://www.jstatsoft.org/

vSMC: Parallel Sequential Monte Carlo in C++

Yan Zhou
National University of Singapore

Abstract

Sequential Monte Carlo is a family of algorithms for sampling from a sequence of distributions. Some of these algorithms, such as particle filters, are widely used in physics and signal processing research. More recent developments have established their application in more general inference problems such as Bayesian modeling.

These algorithms have attracted considerable attention in recent years not only because that they have desired statistical properties, but also because they admit natural and scalable parallelization. However, they are perceived to be difficult to implement. In addition, parallel programming is often unfamiliar to many researchers though conceptually appealing.

A C++ template library is presented for the purpose of implementing generic sequential Monte Carlo algorithms on parallel hardware. Two examples are presented: a simple particle filter and a classic Bayesian modeling problem.

Keywords: sequential Monte Carlo, particle filter, C++ template metaprogramming.

1. Introduction

Sequential Monte Carlo (SMC) is a class of sampling algorithms that combine importance sampling and resampling. They have been used as "particle filters" to solve optimal filtering problems; see, for example, Cappé, Godsill, and Moulines (2007) and Doucet and Johansen (2011) for recent reviews. They are also used in a static setting where a target distribution is of interest, for example, a posterior distribution in Bayesian modeling. This was proposed by Del Moral, Doucet, and Jasra (2006b) and developed by Peters (2005) and Del Moral, Doucet, and Jasra (2006a). This framework involves the construction of a sequence of artificial distributions on spaces of increasing dimensions which admit the distributions of interest as particular marginals.

SMC algorithms are perceived as being difficult to implement while general tools were not available until the development of **SMCTC** (Johansen 2009). SMC algorithms admit natu-

ral and scalable parallelization. However, there are only parallel implementations of SMC algorithms for many problem specific applications available, usually associated with specific research. Lee, Yau, Giles, Doucet, and Holmes (2010) studied the parallelization of SMC algorithms on GPUs with some generality. There are few tools to implement generic SMC algorithms on parallel hardware though multicore CPUs are very common today and computing on specialized hardware such as GPUs is attracting considerable interests.

The purpose of the current work is to provide a framework for implementing generic SMC algorithms on both sequential and parallel hardware. There are two main goals of the presented framework. The first is reusability. It will be demonstrated that the same implementation source can be used to build a serialized sampler, or using different programming models (for example OpenMP, OpenMP Architecture Review Board 2011, and Intel **TBB**, Intel Cooperation 2013c) to build parallelized samplers for multicore CPUs. They can be scaled for clusters using MPI (Message Passing Interface Forum 2012) with few modifications. And with a little effort they can also be used to build parallelized samplers on specialized massive parallel hardware such as GPUs using OpenCL (Kronos OpenCL Working Group 2012). The second is extensibility. It is possible to write a backend for **vSMC** to use new parallel programming models while reusing existing implementations. It is also possible to enhance the library to improve performance for specific applications. Almost all components of the library can be reimplemented by users and thus if the default implementation is not suitable for a specific application, they can be replaced while being integrated with other components seamlessly.

Though **vSMC** is a new library designed from the ground up, it has some similarity to the aforementioned **SMCTC** package, which is by intention. For example, they both use user defined callbacks to perform application specific operations while the library only implements tasks that are common to all samplers. However, there are also significant differences. First of all, **vSMC** allows easy implementation of parallelized algorithms. The library is also more flexible and enjoys better performance. The reusability and extensibility of **vSMC** lead to better algorithm implementation design, such that less duplicate work needs to be done for different algorithms. Both flexibility and better performance are made possible through the use of template metaprogramming, a technique pioneered by **Blitz++** (Veldhuizen 2006). Some differences of the two libraries and the advantage of the approach of **vSMC** will be discussed when features are introduced in Section 4. A more detailed comparison can be found in Appendix A. We will not elaborate this topic further before the SMC algorithms and the library are properly introduced in the following sections.

2. Sequential Monte Carlo

2.1. Sequential importance sampling and resampling

Importance sampling is a technique which allows the calculation of the expectation of a function φ with respect to a distribution π using samples from some other distribution η with respect to which π is absolutely continuous, based on the identity,

$$\mathsf{E}_\pi[\varphi(X)] = \int \varphi(x)\pi(x)\ \mathrm{d}\,x = \int \frac{\varphi(x)\pi(x)}{\eta(x)}\eta(x)\ \mathrm{d}\,x = \mathsf{E}_\eta\Big[\frac{\varphi(X)\pi(X)}{\eta(X)}\Big]. \tag{1}$$

Let $\{X^{(i)}\}_{i=1}^N$ be samples from η, then $\mathsf{E}_\pi[\varphi(X)]$ can be approximated by

$$\hat{\varphi}_1 = \frac{1}{N}\sum_{i=1}^{N}\frac{\varphi(X^{(i)})\pi(X^{(i)})}{\eta(X^{(i)})}. \tag{2}$$

In practice, π and η are often only known up to some normalizing constants, which can be estimated using the same samples. Let $w^{(i)} = \pi(X^{(i)})/\eta(X^{(i)})$, then we have

$$\hat{\varphi}_2 = \frac{\sum_{i=1}^N w^{(i)}\varphi(X^{(i)})}{\sum_{i=1}^N w^{(i)}}, \tag{3}$$

or

$$\hat{\varphi}_3 = \sum_{i=1}^{N} W^{(i)}\varphi(X^{(i)}), \tag{4}$$

where $W^{(i)} \propto w^{(i)}$ and are normalized such that $\sum_{i=1}^N W^{(i)} = 1$.

Sequential importance sampling (SIS) generalizes the importance sampling technique for a sequence of distributions $\{\pi_t\}_{t\geq 0}$ defined on spaces $\{\prod_{k=0}^t E_k\}_{t\geq 0}$. At time $t = 0$, the samples $\{X_0^{(i)}\}_{i=1}^N$ are generated from η_0 and the weights are calculated as $W_0^{(i)} \propto \pi_0(X_0^{(i)})/\eta_0(X_0^{(i)})$. At time $t \geq 1$, each sample $X_{0:t-1}^{(i)}$, usually termed *particles* in the literature, is extended to $X_{0:t}^{(i)}$ by a proposal distribution $q_t(\cdot|X_{0:t-1}^{(i)})$. And the weights are recalculated as $W_t^{(i)} \propto \pi_t(X_{0:t}^{(i)})/\eta_t(X_{0:t}^{(i)})$ where

$$\eta_t(X_{0:t}^{(i)}) = \eta_{t-1}(X_{0:t-1}^{(i)})q_t(X_{0:t}^{(i)}|X_{0:t-1}^{(i)}), \tag{5}$$

and

$$\begin{aligned} W_t^{(i)} \propto \frac{\pi_t(X_{0:t}^{(i)})}{\eta_t(X_{0:t}^{(i)})} &= \frac{\pi_t(X_{0:t}^{(i)})\pi_{t-1}(X_{0:t-1}^{(i)})}{\eta_{t-1}(X_{0:t-1}^{(i)})q_t(X_{0:t}^{(i)}|X_{0:t-1}^{(i)})\pi_{t-1}(X_{0:t-1}^{(i)})} \\ &= \frac{\pi_t(X_{0:t}^{(i)})}{q_t(X_{0:t}^{(i)}|X_{0:t-1}^{(i)})\pi_{t-1}(X_{0:t-1}^{(i)})}W_{t-1}^{(i)}. \end{aligned} \tag{6}$$

The importance sampling estimate of $\mathsf{E}_{\pi_t}[\varphi_t(X_{0:t})]$ can be obtained using $\{W_t^{(i)}, X_{0:t}^{(i)}\}_{i=1}^N$. However this approach fails as t becomes large. The weights tend to become concentrated on a few particles as the discrepancy between η_t and π_t becomes larger. Resampling techniques are applied such that, a new particle system $\{\bar{W}_t^{(i)}, \bar{X}_{0:t}^{(i)}\}_{i=1}^M$ with the following property is generated,

$$\mathsf{E}\Big[\sum_{i=1}^{M}\bar{W}_t^{(i)}\varphi_t(\bar{X}_{0:t}^{(i)})\Big] = \mathsf{E}\Big[\sum_{i=1}^{N}W_t^{(i)}\varphi_t(X_{0:t}^{(i)})\Big]. \tag{7}$$

In practice, the resampling algorithm is usually chosen such that $M = N$ and $\bar{W}^{(i)} = 1/N$ for $i = 1, \ldots, N$. Resampling can be performed at each time t or adaptively based on some criteria of the discrepancy. One popular quantity used to monitor the discrepancy is the *effective sample size* (ESS), introduced by Liu and Chen (1998), defined as

$$\mathrm{ESS}_t = \frac{1}{\sum_{i=1}^N (W_t^{(i)})^2}, \tag{8}$$

where $\{W_t^{(i)}\}_{i=1}^N$ are normalized weights. The value of ESS_t tells us the accumulated mismatch between the sample distribution and the target distribution. Intuitively, when the sample distribution is close to the target distribution, most weights will be close to $1/N$ and ESS_t will be close to N. In the special case of the samples being generated directly from the target distribution, the weights are all exactly $1/N$ and ESS_t is exactly N. On the other hand, an extreme small value of ESS_t suggests that most weights are close to zero while a few are close to one. In this situation, those few particles can have overwhelming effects on the sampler. Therefore it is desired to control the sampler such that ESS_t does not fall below a certain threshold by performing resampling regularly. However, since resampling can be a computational intensive operation, and a bottleneck to parallel performance, it is also desired to avoid resampling unless it is necessary. It is a common practice to perform resampling only when $\mathrm{ESS}_t \leq \alpha N$ where $\alpha \in [0, 1]$. There are little theoretical guidelines on the choosing of the threshold α. However, $\alpha = 0.5$ is often seen in practice, for example, see Jasra, Stephens, Doucet, and Tsagaris (2010).

A common practice of resampling is to replicate particles with large weights and discard those with small weights. In other words, instead of generating random samples $\{\bar{X}_{0:t}^{(i)}\}_{i=1}^N$ directly, random samples of integers $\{R^{(i)}\}_{i=1}^N$ are generated, such that $R^{(i)} \geq 0$ for $i = 1, \ldots, N$ and $\sum_{i=1}^N R^{(i)} = N$. And each particle $X_{0:t}^{(i)}$ is replicated $R^{(i)}$ times in the new particle system. The distribution of $\{R^{(i)}\}_{i=1}^N$ shall fulfill the requirement of Equation 7. One such distribution is a multinomial distribution of size N and weights $(W_t^{(i)}, \ldots, W_t^{(N)})$. See Douc, Cappé, and Moulines (2005) for some commonly used resampling algorithms.

2.2. SMC samplers

SMC samplers allow us to obtain, iteratively, collections of weighted samples from a sequence of distributions $\{\pi_t\}_{t\geq 0}$ over essentially any random variables on some spaces $\{E_t\}_{t\geq 0}$, by constructing a sequence of auxiliary distributions $\{\tilde{\pi}_t\}_{t\geq 0}$ on spaces of increasing dimensions, $\tilde{\pi}_t(x_{0:t}) = \pi_t(x_t) \prod_{s=0}^{t-1} L_s(x_{s+1}, x_s)$, where the sequence of Markov kernels $\{L_s\}_{s=0}^{t-1}$, termed the *backward kernels*, is formally arbitrary but critically influences the estimator variance. See Del Moral *et al.* (2006b) for further details and guidance on the selection of these kernels.

Standard sequential importance sampling and resampling algorithms can then be applied to the sequence of synthetic distributions, $\{\tilde{\pi}_t\}_{t\geq 0}$. At time $t-1$, assume that a set of weighted particles $\{W_{t-1}^{(i)}, X_{0:t-1}^{(i)}\}_{i=1}^N$ approximating $\tilde{\pi}_{t-1}$ is available, then at time t, the path of each particle is extended with a Markov kernel say, $K_t(x_{t-1}, x_t)$ and the set of particles $\{X_{0:t}^{(i)}\}_{i=1}^N$ reaches the distribution $\eta_t(X_{0:t}^{(i)}) = \eta_0(X_0^{(i)}) \prod_{k=1}^t K_k(X_{k-1}^{(i)}, X_k^{(i)})$, where η_0 is the initial distribution of the particles. To correct the discrepancy between η_t and $\tilde{\pi}_t$, Equation 6 is applied and in this case,

$$W_t^{(i)} \propto \frac{\tilde{\pi}_t(X_{0:t}^{(i)})}{\eta_t(X_{0:t}^{(i)})} = \frac{\pi_t(X_t^{(i)}) \prod_{s=0}^{t-1} L_s(X_{s+1}^{(i)}, X_s^{(i)})}{\eta_0(X_0^{(i)}) \prod_{k=1}^t K_k(X_{k-1}^{(i)}, X_k^{(i)})} \propto \tilde{w}_t(X_{t-1}^{(i)}, X_t^{(i)}) W_{t-1}^{(i)}, \tag{9}$$

where the $\tilde{w}_t$, termed the *incremental weights*, are calculated as,

$$\tilde{w}_t(X_{t-1}^{(i)}, X_t^{(i)}) = \frac{\pi_t(X_t^{(i)}) L_{t-1}(X_t^{(i)}, X_{t-1}^{(i)})}{\pi_{t-1}(X_{t-1}^{(i)}) K_t(X_{t-1}^{(i)}, X_t^{(i)})}. \tag{10}$$

If π_t is only known up to a normalizing constant, say $\pi_t(x_t) = \gamma_t(x_t)/Z_t$, then we can use the *unnormalized* incremental weights,

$$w_t(X_{t-1}^{(i)}, X_t^{(i)}) = \frac{\gamma_t(X_t^{(i)})L_{t-1}(X_t^{(i)}, X_{t-1}^{(i)})}{\gamma_{t-1}(X_{t-1}^{(i)})K_t(X_{t-1}^{(i)}, X_t^{(i)})}, \tag{11}$$

for importance sampling. Further, with the *normalized* weights from the last iterations, $\{W_{t-1}^{(i)}\}_{i=1}^N$, we can estimate the ratio of normalizing constant Z_t/Z_{t-1} by,

$$\frac{\hat{Z}_t}{Z_{t-1}} = \sum_{i=1}^N W_{t-1}^{(i)} w_t(X_{t-1}^{(i)}, X_t^{(i)}). \tag{12}$$

Sequentially, the normalizing constant between initial distribution π_0 and some target π_T, $T \geq 1$ can be estimated. See Del Moral *et al.* (2006b) for details on calculating the incremental weights. In particular, when K_t is invariant to π_t, and an approximated suboptimal backward kernel

$$L_{t-1}(x_t, x_{t-1}) = \frac{\pi_t(x_{t-1})K_t(x_{t-1}, x_t)}{\pi_t(x_t)} \tag{13}$$

is used, the unnormalized incremental weights are

$$w_t(X_{t-1}^{(i)}, X_t^{(i)}) = \frac{\gamma_t(X_{t-1}^{(i)})}{\gamma_{t-1}(X_{t-1}^{(i)})}. \tag{14}$$

2.3. Parallel implementations of SMC algorithms

Parallel computing is a form of computation in which many calculations are carried out simultaneously. It operates on the principle that large problems can be divided into *independent* smaller ones and can be solved *concurrently* ("in parallel"). There are many benefits of parallel computing, ranging from power efficiency to improved usage of memory. However, the most relevant one here is perhaps that one can perform the same amount of computational work with less time. In the ideal (and much simplified) situation, assuming that there are K identical computing units that can perform computational tasks independently at the same time, and the total amount of work can be evenly divided into K independent smaller problems, then it is possible to finish the same amount of work with time T/K, where T is the time required to complete the work sequentially using only one of the computing units. The key point here is that those computational tasks are *independent* of each other. If one task requires the finishing of another, then those two tasks have to be performed in a particular order, and essentially cannot be performed concurrently.

In the particular context of SMC algorithms. The updating of the particles according to a transition kernel can be carried out independently. That is, it is not important that the particles have to be updated in a particular order. Therefore, in the above ideal scenario, each computing unit can perform the computation of updating N/K particles, where N is the total number of particles. Other steps of the algorithm, such as calculation of the weights can also be performed in parallel. It is often the case, though unfortunately not always, that these tasks consume the most of the computational time of a given SMC algorithm. By parallelizing these tasks, the algorithm may require considerably less time to complete. The independence

of operations that update each particle makes parallel computing of particular interest in the context of SMC related researches.

However, implementing parallelized algorithms is more difficult than sequential ones for at least two important reasons. First, users often have to learn new programming models. These can be compiler or language specific constructs or new libraries. Second, and more importantly, as mentioned before, dependencies between computations prevent parallelization. And if ignored, such issues result in incorrect implementations of algorithms. Without going into technical details, sequential implementations often implicitly comply with these dependency requirements, while parallel implementations usually do not. When two tasks are assigned to two computing units to be carried out concurrently, there is no guarantee that they will be performed in any particular order without proper programming constructs. Some of such dependencies are easy to see. For example, in some SMC algorithms, the calculation of weights need to be performed after the particles being updated. And users are less likely to attempt to perform these two tasks in parallel. Others might be less obvious. For example, consider reading weight $W_t^{(i)}$ and updating weight $W_t^{(j)}$ with $i \neq j$. It might first appear that these two tasks are independent. However, when the value of $W_t^{(i)}$ is needed, the value of $\sum_{i=1}^N W_t^{(i)}$ is often also needed or otherwise $W_t^{(i)}$ need to be the *normalized* weight. And changing the value of $W_t^{(j)}$ changes the value of $\sum_{i=1}^N W_t^{(i)}$ or the normalized weights. Therefore, in general there needs to be a well defined order of these two tasks.

The **vSMC** library attempts to help users implement parallelized SMC algorithms while avoiding these two difficulties. First, it abstracts many parallel programming models such that users do not have to know the technical details of parallel constructs. Good understanding of C++ programming is sufficient for using the library. Second, it also imposes certain restrictions, such that users are less likely to encounter some dependency issues. For example, through the interface of the library, it is impossible to update a single weight $W_t^{(j)}$ while reading another. Of course, as usual with programming, there are ways to work around such restrictions. And the library does provide facilities such that more advanced users can easily implement algorithms when these restrictions are getting in the way.

It should be also noted that not all tasks of a given SMC algorithm can be parallelized, at least not easily, for example, some resampling algorithms. Moreover, some operations, such as computing the ESS or the normalized weights $\{W_t^{(i)}\}_{i=1}^N$, which involve simple summations, can be parallelized, but not necessarily in an efficient way. In reality, assigning tasks to different computing units has its own computational cost. And when such cost is non-trivial when compared to the tasks themselves, a parallelized implementation can cost more time than a sequential one. The **vSMC** library does not attempt to parallelize these operations. Instead, it provides default sequential implementations and interfaces for users to provide specialized implementations when they are desired. Limited by the scope, in this paper only the most common usage scenarios of the library are introduced.

2.4. Other sequential Monte Carlo algorithms

Some other commonly used sequential Monte Carlo algorithms can be viewed as special cases of algorithms introduced above. The annealed importance sampling (AIS; Neal 2001) can be viewed as SMC sampler without resampling.

Particle filters as seen in the physics and signal processing literature, can also be interpreted

as sequential importance sampling and resampling algorithms. See Doucet and Johansen (2011) for a review of this topic. A simple particle filter example is used in Section 5.1 to demonstrate basic features of the **vSMC** library.

3. Using the vSMC library

3.1. Overview

The **vSMC** library uses templates of C++ code to implement generic SMC algorithms. This library is formed by a few major modules listed below. Some features not included in these modules are introduced later in context.

Core The highest level of abstraction of SMC samplers. Users interact with classes defined within this module to create and manipulate SMC samplers. Classes in this module include `Sampler`, `Particle` and others. These classes use user defined callback functions or callable objects, such as functors, to perform problem specific operations, such as updating particle values and weights. This module is documented in Section 4.1.

Symmetric multiprocessing (SMP) This is the form of computing most people use everyday, including multiprocessor workstations, multicore desktops and laptops. Classes within this module make it possible to write operations that manipulate a single particle but can be applied either sequentially or in parallel through various parallel programming models. A method defined through classes of this module can be used by `Sampler` as callback objects. This module is documented in Section 4.2.

Message passing interface MPI is the *de facto* standard for parallel programming on distributed memory architectures. This module enables users to adapt implementations of algorithms using the SMP module such that the same sampler can be parallelized using MPI. In addition, when used with the SMP module, it allows easy implementation of hybrid parallelization such as MPI/OpenMP. In Section 5.2, an example is shown how to extend existing SMP parallelized samplers using this module.

OpenCL This module is similar to the two above except it eases the parallelization through OpenCL, such as for the purpose of general purpose GPU programming (GPGPU). OpenCL is a framework for writing programs that can be executed across heterogeneous platforms. OpenCL programs can run on either CPUs or GPUs. It is beyond the scope of this paper to give a proper introduction to GPGPU, OpenCL and their use in **vSMC**. However, later we will demonstrate the relative performance of this programming model on both CPUs and GPUs in Section 5.2.

3.2. Obtaining, installation and documentation

vSMC is a header only library. There is practically no installation step. The library can be downloaded as supplementary material from the journal web page and from `http://zhouyan.github.io/vSMC/`. After downloading and unpacking, one can start using **vSMC** by ensuring that the compiler can find the headers inside the `include` directory. To permanently install

the library in a system directory, one can simply copy the contents of the `include` directory to an appropriate place.

A reference manual can be found as supplementary material on the journal web page and at http://zhouyan.github.io/vSMCDoc/. It is beyond the scope of this paper to document every feature of the library. In many places we will refer to this reference manual for further information.

The source of examples are in the `example` directory. To build the examples distributed with the library, one needs **CMake** (Martin and Hoffman 2010), version 2.8.3 or later.

```
$ cd path_to_vSMC_source
$ mkdir build
$ cd build
$ cmake .. -DCMAKE_BUILD_TYPE=Release
$ make example
```

To build only the two examples documented in this paper, change the target of `make` from `example` to `paper`. The source of these two examples are in the `paper` subdirectory of the `example` directory.

The library is under active development. See the homepage of the library, http://zhouyan.github.io/vSMC/, for latest information. In particular, one can use **git** (Torvalds and others 2013) to obtain the latest source of the library and the examples,

```
$ git clone https://github.com/zhouyan/vSMC.git
$ cd vSMC
```

One can proceed as before using the **CMake** script to build the examples. In addition, given **Doxygen** (van Heesch 2013), version 1.8.3 or later, one can build the reference manual by invoking `make docs`. The latest reference manual can also be found on http://zhouyan.github.io/vSMCDoc/master/.

3.3. Third-party dependencies

The library does not have mandatory third-party dependencies. However, there are a few libraries that might be of interest to users.

vSMC uses **Random123** (Salmon, Moraes, Dror, and Shaw 2011), a collection of counter-based random number generators (RNG), for random number generation. For an SMC sampler with N particles, **vSMC** constructs N (statistically) independent RNG streams. It is possible to use millions of such streams without a huge memory footprint or other performance penalties. Since each particle has its own independent RNG stream, it frees users from many thread-safety and statistical independence considerations. **vSMC** provides its own implementations of the same algorithms in **Random123**. It is not necessary for users to install this library. However, the performance of the implementations in **vSMC** can be either faster or slightly slower than **Random123**, depending on the platform. And therefore, using **Random123** instead of the **vSMC** implementation can be desirable for some users. We will show how to replace the RNG used by **vSMC** later in Section 4.1. Within **vSMC**, these RNG streams are wrapped under C++11 RNG engines, and can be replaced by other compatible RNG engines seamlessly. Users only need to be familiar with classes defined in C++11 `<random>` or their

Boost (Dawes and others 2013) equivalents to use these RNG streams. See the documentation of the corresponding libraries for details, as well as examples in Section 5.

The library makes use of some C++11 libraries, in particular the `<functional>` and `<random>` headers. However, if C++11 features are not available, one can use the **Boost** library instead. Version 1.49 or later is required. In most cases, the library is able to check the availability of the C++11 features automatically. When it is not able to determine the usability of these C++11 headers with certainty, it falls back to using their **Boost** equivalents. In these cases, to instruct the library to use the standard library headers instead of falling back to the **Boost** library, one needs to define configuration macros before including any **vSMC** headers. For example,

```
$ clang++ -std=c++11 -stdlib=libc++ -DVSMC_HAS_CXX11LIB_FUNCTIONAL=1 \
>   -DVSMC_HAS_CXX11LIB_RANDOM=1 -o prog prog.cpp
```

tells the library to use C++11 `<functional>` and `<random>`. Defining these macros to `0` will force **vSMC** to use the **Boost** library.

3.4. Compiler support

vSMC has been tested with recent versions of **clang** (The LLVM Developer Group 2013a), **GCC** (GNU Project 2013), Intel C++ Compiler (Intel Cooperation 2013a) and Microsoft Visual C++ (Microsoft Cooperation 2012). **vSMC** can optionally use some C++11 features to improve performance and usability. In particular, as mentioned before, **vSMC** can use C++11 standard libraries instead of the **Boost** library. At the time of writing, **clang** with **libc++** (The LLVM Developer Group 2013b) has the most comprehensive support of C++11 with respect to standard compliant and feature completion. **GCC** 4.8, Microsoft Visual C++ 2012 and Intel C++ Compiler 2013 also have very good C++11 support. Note that, provided the **Boost** library is available, all C++11 language and library features are optional. **vSMC** can be used with any C++98 conforming compilers. However, due to the heavy usage of templates, older compilers may require considerable longer time and more memory to compile the library. More recent compilers can perform significantly better. From the author's experience, **clang** 3.3, **GCC** 4.7, Intel C++ Compiler 2010 and Microsoft Visual C++ 2010, or later versions of these compilers are recommended. They are also regularly tested during the development of the library.

4. The vSMC library

4.1. Core module

The core module abstracts general SMC samplers. SMC samplers can be viewed as formed by a few concepts regardless of the specific problems. The following is a list of the most commonly seen components of SMC samplers and their corresponding **vSMC** abstractions. All public classes, namespaces and free functions, are declared in the namespace `vsmc`.

- A collection of all particle state values, namely $\{X_t^{(i)}\}_{i=1}^N$. In **vSMC**, users need to define a class, say `T`, to abstract this concept. We refer to this as the *value collection*. We

will slightly abuse the generic name `T` in this paper. Whenever a template parameter is mentioned with the name `T`, it always refers to such a value collection type unless stated otherwise.

- A collection of all particle state values and their associated weights. This is abstracted by a `Particle<T>` object. We refer to this as the *particle collection*. A `Particle<T>` object has three primary sub-objects. One is the above type `T` value collection object. Another is an object that abstracts weights $\{W_t^{(i)}\}_{i=1}^N$. By default this is a `WeightSet` object. The last is a collection of RNG streams, one for each particle. By default this is an `RngSet<Threefry4x32, Vector>` object.
- Operations that perform tasks common to all samplers to these particles, such as resampling. These are the member functions of `Particle<T>`.
- A sampler that updates the particles (state values and weights) using user defined callbacks. This is a `Sampler<T>` object.
- Monitors that record the importance sampling estimates of certain functions defined for the values when the sampler iterates. These are `Monitor<T>` objects. A monitor for the estimates of $\mathsf{E}[h(X_t)]$ computes $h(X_t^{(i)})$ for each $i = 1, \ldots, N$. The function value $h(X_t)$ is allowed to be a vector.

Note that within the core module, all operations are applied to `Particle<T>` objects, that is $\{W_t^{(i)}, X_t^{(i)}\}_{i=1}^N$, instead of a single particle. This is one of the major differences between **vSMC** and **SMCTC**. In the later library, users define classes that abstract $X_t^{(i)}$ and user defined callbacks operate on $(W_t^{(i)}, X_t^{(i)})$ for each $i = 1, \ldots, N$. As we will see very soon, the **vSMC** approach provides greater flexibility. And later we will see also how to write operations that can be applied to individual particles and can be parallelized easily.

Program structures

A **vSMC** program usually consists of the following tasks.

- Define a value collection type `T`.
- Constructing a `Sampler<T>` object.
- Configure the behavior of initialization and updating by adding callable objects to the sampler object.
- Optionally add monitors.
- Initialize and iterate the sampler.
- Retrieve the outputs, estimates and other informations.

In this section we document how to implement each of these tasks.

The value collection

The template parameter `T` is a user defined type that abstracts the value collection. **vSMC** does not restrict how the values shall be actually stored. They can be stored in memory,

spread among nodes of a cluster, in GPU memory or even in a database. However this kind of flexibility comes at a small price. The value collection does need to fulfill two requirements. We will see later that for most common usage, **vSMC** provides readily usable implementations, on top of which users can create problem specific classes.

First, the value collection class `T` has to provide a constructor of the form

```
T (SizeType N)
```

where `SizeType` is some integer type. Optionally, one can also provide the following definition,

```
typedef SizeType size_type;
```

If `size_type` is found inside a value collection type `T`, **vSMC** classes such as `Particle<T>` will define their own `size_type` to exactly the same type. Since **vSMC** allows one to allocate the states in any way suitable, one needs to provide this constructor which `Particle<T>` can use to allocate them.

Second, the class has to provide a member function named `copy` that copies each particle according to replication numbers given by a resampling algorithm. For the same reason as above, **vSMC** has no way of knowing how it can extract and copy a single particle when it is doing the resampling. The signature of this member function may look like

```
template <typename SizeType>
void copy (std::size_t N, const SizeType *copy_from);
```

The pointer `copy_from` points to an array that has N elements, where N is the number of particles. After calling this member function, the value of particle `i` shall be copied from the particle `j = copy_from[i]`. In other words, particle `i` is a child of particle `copy_from[i]`, or `copy_from[i]` is the parent of particle `i`. If a particle `j` shall remain itself, then `copy_from[j] == j`. How the values are actually copied is user defined. Note that, the member function can take other forms, as usual when using C++ template programming. The actual type of the pointer `copy_from`, `SizeType`, is `Particle<T>::size_type`, which depends on the type `T`. For example, defining the member function as the following is also allowed,

```
void copy (int N, const std::size_t *copy_from);
```

provided that `Particle<T>::size_type` is indeed `std::size_t`. However, writing it as a member function template releases users from finding the actual type of pointer `copy_from` and the sometimes troubling forward declaration issues. We will not elaborate such more technical issues further.

The value collection type `T` can also be used to provide additional informations to **vSMC** of how to construct the sampler. For example, one can change the RNG used by `Particle<T>` by providing a `typedef` within the type `T`. For instance, the following will instruct **vSMC** to use an advanced encryption standard new instructions (AES-NI) based RNG engine instead of the default one.

```
typedef RngSet<AES128_32, Vector> rng_set_type;
```

Constructing a sampler

Once the value collection class `T` is defined, one can start constructing SMC samplers. For example, the following line creates a sampler with N particles

```
Sampler<T> sampler(N);
```

The number of particles is the only mandatory argument of the constructor. By default, no resampling steps will be performed. There are two other constructors,

```
Sampler<T>::Sampler (Sampler<T>::size_type N, ResampleScheme scheme);
Sampler<T>::Sampler (Sampler<T>::size_type N, ResampleScheme scheme,
  double threshold);
```

The `scheme` parameter is self-explanatory. **vSMC** provides a few built-in resampling schemes; see the reference manual for a list of them. User defined resampling algorithms can also be used; see the reference manual for details. The `threshold` parameter is the threshold of ESS/N below which resampling will be performed. It is obvious that if `threshold` ≥ 1, then resampling will be performed at every iteration. If `threshold` ≤ 0, then resampling will never be performed. Both parameters can be changed later. However, the size of the sampler can never be changed after the construction. If the `threshold` parameter is omitted but not the `scheme` parameter, the default behavior is that the sampler will resample at every iteration.

Initialization and updating

All the callable objects that initialize, move and weight the particle collection can be added to a sampler through a set of member functions. All these objects operate on the `Particle<T>` object. Because **vSMC** allows one to manipulate the particle collection as a whole, in principle many kinds of parallelization are possible.

To set an initialization method, one needs to implement a function with the following signature,

```
std::size_t init_func (Particle<T> &particle, void *param);
```

or a class with `operator()` properly defined, such as

```
struct init_class
{ std::size_t operator() (Particle<T> &particle, void *param); };
```

They can be added to the sampler through

```
sampler.init(init_func);
```

or

```
sampler.init(init_class());
```

respectively. C++11 `std::function` or its **Boost** equivalent `boost::function` can also be used. For example,

```
std::function<std::size_t (Particle<T> &, void *)> init_obj(init_func);
sampler.init(init_obj);
```

In principle, when **vSMC** requires a user defined callback, any object that supports function calling syntax can be used. Unlike **SMCTC**, only function pointers are accepted. This design of **vSMC** allows users to structure the program with a more object-oriented design, if it is desired.

The addition of updating methods is more flexible. There are two kinds of updating methods. One is simply called `move` in **vSMC**, and is performed before the possible resampling at each iteration. These moves usually perform the updating of the weights among other tasks. The other is called `mcmc`, and is performed after the possible resampling. They are often MCMC type moves. Multiple `move`'s or `mcmc`'s are also allowed. In fact, a **vSMC** sampler consists of a queue of `move`'s and a queue of `mcmc`'s. The `move`'s in the queue can be changed through `Sampler<T>::move`,

```
Sampler<T> &move (const Sampler<T>::move_type &new_move, bool append);
```

If `append == true` then `new_move` is appended to the existing (possibly empty) queue. Otherwise, the existing queue is cleared and `new_move` is added. The member function returns a reference to the updated sampler. For example, the following move,

```
std::size_t move_func (std::size_t iter, Particle<T> &particle);
```

can be added to a sampler by

```
sampler.move(move_func, false);
```

This will clear the (possibly empty) existing queue of `move`'s and set a new one. To add multiple moves into the queue,

```
sampler.move(move1, true).move(move2, true).move(move3, true);
```

Objects of class types with `operator()` properly defined can also be used, similar to the initialization method. The queue of `mcmc`'s can be used similarly. See the reference manual for other methods that can be used to manipulate these two queues.

In principle, one can combine all moves into a single move. However, sometimes it is more natural to think of a queue of moves. For instance, if a multi-block Metropolis random walk consists of kernels K_1 and K_2, then one can implement each of them as functions, say `mcmc_k1` and `mcmc_k2`, and add them to the sampler sequentially,

```
sampler.mcmc(mcmc_k1, true).mcmc(mcmc_k2, true);
```

At each iteration, they will be applied to the particle collection sequentially in the order in which they are added.

Running the algorithm, monitoring and outputs

Having set all the operations, one can initialize and iterate the sampler by calling

```
sampler.initialize((void *)param);
sampler.iterate(iter_num);
```

The `param` argument to `initialize` is optional, with `NULL` as its default value. This parameter is passed to user defined `init_func`. The `iter_num` argument to `iterate` is also optional; the default is 1. The `param` parameter can be used to pass informations to the `init_func` function. This is one of very few places where **vSMC** allows type unsafe operations. Whenever possible, it is recommended that a class type of initialization method is used, to which informations can be passed through member functions or constructors. However, passing information to functions through a `void` pointer is a common idiom used by many researchers, and thus here we also support this kind of usage.

Before initializing the sampler or after a certain time point, one can add monitors to the sampler. The concept is similar to BUGS (Spiegelhalter, Thomas, and Best 1996; Lunn, Spiegelhalter, Thomas, and Best 2009)'s `monitor` statement, except that it does not monitor the individual values but rather the importance sampling estimates. Consider approximating $\mathsf{E}[h(X_t)]$, where $h(X_t) = (h_1(X_t), \ldots, h_m(X_t))$ is an m-vector function. The importance sampling estimate can be obtained by AW where A is the N by m matrix, $A(i,j) = h_j(X_t^{(i)})$ and $W = (W_t^{(i)}, \ldots, W_t^{(N)})^\top$ is the N-vector of normalized weights. To compute this importance sampling estimate, one needs to define the following evaluation function (or a class with `operator()` properly defined),

```
void monitor_eval (std::size_t iter, std::size_t m,
  const Particle<T> &particle, double *res)
```

and add it to the sampler by calling,

```
sampler.monitor("some.user.chosen.variable.name", m, monitor_eval);
```

When the function `monitor_eval` is called at the end of each iteration, `iter` has the value of the iteration number of the sampler with the initialization step counting as zero; `m` has the same value as the one passed to `Sampler<T>::monitor`. The output pointer `res` points to an $N \times m$ output array of row major order. That is, after the calling of the function, `res[i * dim + j]` should be $h_j(X_t^{(i)})$.

After each iteration of the sampler, the importance sampling estimate will be computed automatically. See the reference manual for various ways of retrieving the results. Usually it is sufficient to output the sampler by

```
std::ofstream output("file.name");
output << sampler << std::endl;
```

where the output file will contain the importance sampling estimates among other things. Alternatively, one can use the `Monitor<T>::record` member function to access specific historical results.

A reference to the value collection `T` object can be retrieved through the `Particle<T>::value` member function. The `Particle<T>` object manages the weights through a weight set object, which by default is of type `WeightSet`. The `Particle<T>::weight_set` member function returns a reference to this weight set object. A user defined weight set class that abstracts

$\{W_t^{(i)}\}_{i=1}^N$ can also be used. The details involve some more advanced C++ template techniques and are documented in the reference manual. One possible reason for replacing the default is to provide special memory management of the weight set. For example, the MPI module provides a special weight set class that manages weights across multiple nodes and perform proper normalization, computation of ESS, and other tasks.

The default `WeightSet` object provides some ways of retrieving weights. Here we document some of the most commonly used. See the reference manual for details of others. The weights can be accessed one by one, for example,

```
Particle<T> &particle = sampler.particle();
double w_i = particle.weight_set().weight(i);
double log_w_i = particle.weight_set().log_weight(i);
```

One can also read all weights into a container, for example,

```
std::vector<double> w(particle.size());
particle.weight_set().read_weight(w.begin());
double *lw = new double[particle.size()];
particle.weight_set().read_log_weight(lw);
```

Note that these member functions accept general output iterators.

Implementing initialization and updating

So far we have only discussed how to add initialization and updating objects to a sampler. To actually implement them, one writes callable objects that operate on the `Particle<T>` object. For example, a move can be implemented through the following function as mentioned before,

```
std::size_t move_func (std::size_t iter, Particle<T> &particle);
```

Inside the body of this function, one can change the value by manipulating the object through the reference returned by `particle.value()`.

When using the default weight set class, the weights can be updated through a set of member functions of `WeightSet` objects. For example,

```
std::vector<double> weight(particle.size());
particle.weight_set().set_equal_weight();
particle.weight_set().set_weight(weight.begin());
particle.weight_set().mul_weight(weight.begin());
particle.weight_set().set_log_weight(weight.begin());
particle.weight_set().add_log_weight(weight.begin());
```

The `set_equal_weight` member function sets all weights to be equal. The `set_weight` and `set_log_weight` member functions set the values of weights and logarithm weights ($\log W_t^i$), respectively. The `mul_weight` and `add_log_weight` member functions multiply the weights or add to the logarithm weights by the given values, respectively. All these member functions accept general input iterators as their arguments.

Generating random numbers

The `Particle<T>` object has a sub-object, a collection of RNG engines that can be used with C++11 `<random>` or **Boost** distributions. For each particle `i`, one can obtain an engine that produces an RNG stream independent of others by

```
particle.rng(i);
```

To generate distribution random variates, one can use the C++11 `<random>` library. For example,

```
std::normal_distribution<double> rnorm(mean, sd);
double r = rnorm(particle.rng(i));
```

or the **Boost** library,

```
boost::random::normal_distribution<double> rnorm(mean, sd);
double r = rnorm(particle.rng(i));
```

vSMC itself also uses C++11 `<random>` or **Boost** depending on the value of the configuration macro `VSMC_HAS_CXX11LIB_RANDOM`. Though users are free to choose which one to use in their own code, there is a convenient alternative. For each class defined in C++11 `<random>`, it is imported to the `cxx11` namespace. Therefore one can use

```
cxx11::normal_distribution<double> rnorm(mean, sd);
```

while the underlying implementation of `normal_distribution` can be either the C++11 standard library or **Boost**. Note that, **vSMC** defines classes that are available in C++11 `<random>` but not in **Boost** inside the `cxx11` namespace if **Boost** is used, but not *vice versa*. The benefit is that if one needs to develop on multiple platforms, and only some of them support C++11 and some of them have the **Boost** library, then only the configuration macro `VSMC_HAS_CXX11LIB_RANDOM` needs to be changed. This can be configured through **CMake** and other build systems. Of course, one can also use an entirely different RNG system than those provided by **vSMC**.

4.2. SMP module

The value collection

Many typical problems' value collections can be viewed as a matrix of certain types. For example, a simple particle filter whose state is a vector of length `Dim` and type `double` can be viewed as an N by `Dim` matrix where N is the number of particles. A trans-dimensional problem can use an N by 1 matrix whose type is a user defined class, say `StateType`. For this kind of problems, **vSMC** provides a class template

```
template <MatrixOrder Order, std::size_t Dim, typename StateType>
class StateMatrix;
```

which provides the constructor and the `copy` member function required by the core module interface, as well as methods for accessing individual values. The first template parameter (possible values `RowMajor` or `ColMajor`) specifies how the values are ordered in memory. Usually one shall choose `RowMajor` to optimize data access. The second template parameter is the number of variables, an integer value no less than 1 or the special value `Dynamic`, in which case `StateMatrix` provides a member function `resize_dim` such that the number of variables can be changed at runtime. The third template parameter is the type of the state values.

Each particle's state is thus a vector of length `Dim`, indexed from `0` to `Dim - 1`. To obtain the value at position `pos` of the vector of particle `i`, one can use one of the following member functions,

```
StateBase<RowMajor, Dim, StateType> value(N);
StateType val = value.state(i, pos);
StateType val = value.state(i, Position<Pos>());
StateType val = value.state<Pos>(i);
```

where `Pos` is a compile time constant expression whose value is the same as `pos`, assuming the position is known at compile time. One can also read all values. To read the variable at position `pos`,

```
std::vector<StateType> vec(value.size());
value.read_state(pos, vec.begin());
```

Or one can read all values through an iterator,

```
std::vector<StateType> mat(Dim * value.size());
value.read_state_matrix(ColMajor, mat.first());
```

Alternatively, one can also read all values through an iterator which points to iterators,

```
std::vector<std::vector<StateType> > mat(Dim);
for (std::size_t i = 0; i != Dim; ++i)
  mat[i].resize(value.size());
std::vector<std::vector<StateType>::iterator> iter(Dim);
for (std::size_t i = 0; i != Dim; ++i)
  iter[i] = mat[i].begin();
value.read_state_matrix(iter.first());
```

If the compiler supports C++11 `<tuple>`, **vSMC** also provides a `StateTuple` class template, which is similar to `StateMatrix` except that the types of values do not have to be the same for each variable. This is similar to R (R Core Team 2014)'s `data.frame`. For example, suppose each particle's state is formed by two `double`'s, an `int` and a user defined type `StateType`, then the following constructs a value collection using `StateTuple`,

```
StateTuple<ColMajor, double, double, int, StateType> value(N);
```

And there are a few ways to access the state values, similar to `StateMatrix`,

```
double x0 = value.state(i, Position<0>());
int x2 = value.state<2>(i);
std::vector<StateType> vec(value.size());
state.read_state(Position<3>(), vec.begin());
```

See the reference manual for details.

A single particle

For a `Particle<T>` object, one can construct a `SingleParticle<T>` object that abstracts one of the particles from the collection. For example,

```
Particle<T> particle(N);
SingleParticle<T> sp(i, &particle);
```

creates a `SingleParticle<T>` object corresponding to the particle at position `i`. There are a few member functions of `SingleParticle<T>` that make access to individual particles easier than through the interface of `Particle<T>`. First `sp.id()` returns the value of the argument `i` in the above code that created this `SingleParticle<T>` object. In addition, `sp.rng()` is equivalent to `particle.rng(i)`. Also `sp.particle()` returns a constant reference to the `Particle<T>` object. And `sp.particle_ptr()` returns a pointer to such a constant `Particle<T>` object. Note that, one cannot get write access to a `Particle<T>` object through the interface of `SingleParticle<T>`. Instead, one can only get write access to a single particle. For example, if `T` is a `StateMatrix` instantiation or its derived class, then `sp.state(pos)` is equivalent to `particle.value().state(i, pos)` and the reference it returns is mutable. See the reference manual for more information on the interface of the `SingleParticle` class template. `SingleParticle<T>` objects are usually not constructed by users, but rather by other classes in the SMP module, and passed to user defined functions, as we will see very soon.

The default implementation of `SingleParticle` has only a handfull features. The users can provide application specific functionalities by defining a `single_particle_type` class template inside the value collection type. For example,

```
class UserValueType : public StateMatrix<RowMajor, 2, double>
{
  public :

  typedef StateMatrix<RowMajor, 2, double> base_type;

  template <typename T>
  class single_particle_type :
    public typename base_type::single_particle_type<T>
  {
    double mean () const { return this->state(0); }
    double sd   () const { return this->state(1); }
  };
};
```

Now one can use `sp.mean()` and `sp.sd()` to access the two parameters instead of using `sp.state(0)` and `sp.state(1)`, which might introduce difficult to spot errors if used in many places. Alternative solutions such as defining global constants, say

```
const Mean = 0;
const SD = 1;
```

and using `sp.state(Mean)` and `sp.state(SD)` can also introduce other undesired side effects, even though one can partially overcome the issue of using "magical numbers".

Sequential and parallel implementations

Once we have the `SingleParticle<T>` concept, we are ready to introduce how to write implementations of SMC algorithms that manipulate a single particle and can be applied to all particles in parallel or sequentially. For sequential implementations, this can be done through five base classes,

```
template <typename BaseState> class StateSEQ;
template <typename T, typename D = Virtual> class InitializeSEQ;
template <typename T, typename D = Virtual> class MoveSEQ;
template <typename T, typename D = Virtual> class MonitorEvalSEQ;
template <typename T, typename D = Virtual> class PathEvalSEQ;
```

The template parameter `BaseState` needs to satisfy the general value collection requirements in addition to a `copy_particle` member function, for example, `StateMatrix`. Other base classes expect `T` to satisfy general value collection requirements. Note that, it is not required that `InitializeSEQ`, etc., are used with `StateSEQ`. The details of all these class templates can be found in the reference manual. Here we use the `MoveSEQ<T>` class as an example to illustrate their usage. Recall that `Sampler<T>` expects a callable object which has the following signature as a move,

```
std::size_t move_func (std::size_t iter, Particle<T> &particle);
```

For the purpose of illustration, the type `T` is defined as,

```
typedef StateMatrix<RowMajor, 1, double> T;
```

Here is a typical example of implementation of such a function,

```
std::size_t move_func (std::size_t iter, Particle<T> &particle)
{
  std::vector<double> inc_weight(particle.size());
  for (Particle<T>::size_type i = 0; i != particle.size(); ++i) {
    cxx11::normal_distribution<double> rnorm(0, 1);
    particle.value().state(i, 0) = rnorm(particle.rng(i));
    inc_weight[i] = cal_inc_weight(particle.value().state(i, 0);
  }
  particle.weight_set().add_log_weight(inc_weight.begin());
}
```

where `cal_inc_weight` is some function that calculates the logarithm incremental weights. As we see, there are three main parts of a typical move. First, we allocate a vector `inc_weight`. Second, we generate normal random variates for each particle and calculate the incremental weights. This is done through a `for` loop. Third, we add the logarithm incremental weights. The first and the third are *global* operations while the second is *local*. The first and the third are often optional and absent. The local operation is also usually the most computational intensive part of SMC algorithms and can benefit the most from parallelization.

With `MoveSEQ<T>`, one can derive from this class, which defines the `operator()` required by the core module interface,

```
std::size_t operator() (std::size_t iter, Particle<T> &particle);
```

and customize what this operator does by defining one or more of the following three member functions, corresponding to the three parts, respectively,

```
void pre_processor (std::size_t iter, Particle<T> &particle);
std::size_t move_state (std::size_t iter, SingleParticle<T> sp);
void post_processor (std::size_t iter, Particle<T> &particle);
```

For example,

```
#include <vsmc/smp/backend_seq.hpp>

class move : public MoveSEQ<T>
{
  public :

  void pre_processor (std::size_t iter, Particle<T> &particle)
  {inc_weight_.resize(particle.size());}

  std::size_t move_state (std::size_t iter, SingleParticle<T> sp)
  {
    cxx11::normal_distribution<double> rnorm(0, 1);
    sp.state(0) = rnorm(sp.rng());
    inc_weight_[sp.id()] = cal_inc_weight(sp.state(0));
  }

  void post_processor (std::size_t iter, Particle<T> &particle)
  {particle.weight_set().add_log_weight(inc_weight_.begin());}

  private :

  std::vector<double> inc_weight_;
};
```

The `operator()` of `MoveSEQ<T>` is equivalent to the single function implementation as shown before.

In the simplest case, `MoveSEQ` only takes away the loop around the local operations. However if one implements the move in such a way, and then replace `MoveSEQ` with `MoveOMP`, the changing of the base class name causes **vSMC** to use OpenMP to parallelize the loop. For example, one can declare the class as

```
#include <vsmc/smp/backend_omp.hpp>
class move : public MoveOMP<T>;
```

and use exactly the same implementation as before. Now when `move::operator()` is called, it will be parallelized by OpenMP. Other backends are also supported in case that OpenMP is not available. Among them there are Cilk Plus (Intel Cooperation 2011) and Intel **TBB**. In addition to these well known parallelization programming models, **vSMC** also has its own implementation using C++11 `<thread>`. There are other backends documented in the reference manual. To use any of them, all one needs to do is to change a few base class names. In practice, one can use conditional compilation. For example, to use a sequential implementation or an OpenMP parallelized one, we can write,

```
#ifdef USE_SEQ
#include <vsmc/smp/backend_seq.hpp>
#define BASE_MOVE MoveSEQ
#endif

#ifdef USE_OMP
#include <vsmc/smp/backend_omp.hpp>
#define BASE_MOVE MoveOMP
#endif

class move : public BASE_MOVE<T>;
```

And we can compile the same source into different samplers with `Makefile` rules such as the following,

```
prog-seq : prog.cpp
        $(CXX) $(CXXFLAGS) -DUSE_SEQ -o prog-seq prog.cpp
prog-omp : prog.cpp
        $(CXX) $(CXXFLAGS) -DUSE_OMP -o prog-omp prog.cpp
```

Or one can configure the source file through a build system such as **CMake**, which can also determine which programming model is available on the system.

Adapter

Sometimes, the cumbersome task of writing a class to implement a move and other operations can outweigh the power we gain through object-oriented programming. For example, a simple `move_state`-like function is all that we need to get the work done. In this case, one can create a move through the `MoveAdapter`. For example, say we have implemented

```
std::size_t move_state (std::size_t iter, SingleParticle<T> sp);
```

as a function. Then one can create a callable object acceptable to `Sampler<T>::move` through

```
MoveAdapter<T, MoveSEQ>  move_obj(move_state);
MoveAdapter<T, MoveSTD>  move_obj(move_state);
MoveAdapter<T, MoveTBB>  move_obj(move_state);
MoveAdapter<T, MoveCILK> move_obj(move_state);
MoveAdapter<T, MoveOMP>  move_obj(move_state);
sampler.move(move_obj, false);
```

These are respectively, sequential, C++11 `<thread>`, Intel **TBB**, Cilk Plus, and OpenMP implementations. The first template parameter is the type of value collection and the second is the name of the base class template. Actually, the `MoveAdapter`'s constructor accepts two more optional arguments, the `pre_processor` and the `post_processor`, corresponding to the other two aforementioned member functions. Similar adapters for the other three base classes also exist.

Another scenario where an adapter is desired is the situation where which backend to use needs to be decided at runtime. The above simple adapters can already be used for this purpose. In addition, another form of the adapter is given by the following,

```
class move;
MoveAdapter<T, MoveTBB, move> move_obj;
sampler.move(move_obj, false);
```

where the class `move` has the same definition as before but it no longer derives from any base class. The class `move` is required to have a default constructor, a copy constructor and a copy assignment operator.

4.3. Thread-safety and scalability considerations

When implementing parallelized SMC algorithms, issues such as thread-safety cannot be avoided even though the **vSMC** library hides most parallel constructs from users.

Classes in the **vSMC** library usually guarantee that their member functions are thread-safe in the sense that calling the same member function on different objects at the same time from different threads is safe. However, calling mutable member functions on the same object from different threads is usually not safe. Calling immutable member functions is generally safe. There are a few exceptions.

- The constructors of `Particle` and `Sampler` are not thread-safe. Therefore if one needs to construct multiple `Sampler`s from different threads, a mutex protection is needed. However, subsequent member function calls on the constructed objects are thread-safe according to the above rules.
- Member functions that concern a single particle are generally thread-safe in the sense that one can call them on the same object from different threads as long as they are called for different particles. For example, `Particle::rng` and `StateMatrix::state` are thread-safe.

In general, one can safely manipulate different individual particles from different threads, which is the minimal requirement for scalable parallelization. But one cannot manipulate the

whole particle collection from different threads. For example, `WeightSet::set_log_weight` cannot be called from multiple threads on the same object.

User defined callbacks shall generally follow the same rules. For example, for a `MoveOMP` subclass, `pre_processor` and `post_processor` do not have to be thread-safe, but `move_state` needs to be. In general, avoid write access to memory locations shared by all particles from `move_state` and other similar member functions. One needs to take some extra care when using third-party libraries. For example, in our example implementation of the `move` class, the `rnorm` object, which is used to generate normal random variates, is defined within `move_state` instead of being class member data even though it is created with the same parameters for all particles. This is because the call `rnorm(sp.rng())` is not thread-safe in some implementations, for example, when using the **Boost** library.

For scalable performance, certain practices should be avoided when implementing member functions such as `move_state`. For example, dynamic memory allocation is usually lock-based and thus should be avoided. Alternatively one can use a scalable memory allocator such as the one provided by Intel **TBB**. In general, in functions such as `move_state`, one should avoid using locks to guarantee thread-safety, which can be a bottleneck to parallel performance.

5. Examples

5.1. A simple particle filter

The model and algorithm

This is an example used in **SMCTC** (Johansen 2009). Through this example, we will show how to re-implement a simple particle filter in **vSMC**. In this way one is walked through the basic features of the library. **SMCTC** is the first general framework for implementing SMC algorithms in C++ and serves as one of the most important inspirations for the **vSMC** library. It is widely used by many researchers. One of the goals of the current work is that users familiar with **SMCTC** shall find little difficulty in using the new library.

The state space model, known as the almost constant velocity model in the tracking literature, provides a simple scenario. The state vector X_t contains the position and velocity of an object moving in a plane. That is, $X_t = (x^t_{\text{pos}}, y^t_{\text{pos}}, x^t_{\text{vel}}, y^t_{\text{vel}})^\top$. Imperfect observations $Y_t = (x^t_{\text{obs}}, y^t_{\text{obs}})^\top$ of the positions are possible at each time instance. The state and observation equations are linear with additive noises,

$$X_t = AX_{t-1} + V_t,$$
$$Y_t = BX_t + \alpha W_t,$$

where

$$A = \begin{pmatrix} 1 & 0 & \Delta & 0 \\ 0 & 1 & 0 & \Delta \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \qquad B = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}, \qquad \Delta = 0.1, \qquad \alpha = 0.1.$$

Initialization

Set $t \leftarrow 0$.

Sample $x_{\text{pos}}^{(0,i)}, y_{\text{pos}}^{(0,i)} \sim \mathcal{N}(0, 4)$ and $x_{\text{vel}}^{(0,i)}, y_{\text{vel}}^{(0,i)} \sim \mathcal{N}(0, 1)$.

Weight $W_0^{(i)} \propto L(X_0^{(i)}|Y_0)$ where L is the likelihood function.

Iteration

Set $t \leftarrow t + 1$.

Sample

$$x_{\text{pos}}^{(t,i)} \sim \mathcal{N}(x_{\text{pos}}^{(t-1,i)} + \Delta x_{\text{vel}}^{(t-1,i)}, 0.02) \qquad x_{\text{vel}}^{(t,i)} \sim \mathcal{N}(x_{\text{vel}}^{(t-1,i)}, 0.001)$$
$$y_{\text{pos}}^{(t,i)} \sim \mathcal{N}(y_{\text{pos}}^{(t-1,i)} + \Delta y_{\text{vel}}^{(t-1,i)}, 0.02) \qquad y_{\text{vel}}^{(t,i)} \sim \mathcal{N}(y_{\text{vel}}^{(t-1,i)}, 0.001)$$

Weight $W_t^{(i)} \propto W_{t-1}^{(i)} L(X_t^{(i)}|Y_t)$.

Repeat the Iteration *step until all data are processed.*

Algorithm 1: Particle filter algorithm for the almost constant velocity model.

The constant Δ is the time interval between observations. We assume that the elements of the noise vector V_t are independent normal with variance 0.02 and 0.001 for position and velocity, respectively. The observation noise, W_t comprises independent, identically distributed t-distributed random variables with degree of freedom $\nu = 10$. The prior at time 0 corresponds to an axis-aligned Gaussian with variance 4 for the position coordinates and 1 for the velocity coordinates. The particle filter algorithm is shown in Algorithm 1.

Implementations

The full implementation can be found in `example/paper/src/paper_pf.cpp` within the downloaded source of the library. We first introduce the body of the `main` function, showing the typical work flow of a **vSMC** program.

```
#include <cassert>
#include <cmath>
#include <fstream>
#include <vsmc/core/sampler.hpp>
#include <vsmc/core/state_matrix.hpp>
#include <vsmc/smp/adapter.hpp>
#include <vsmc/smp/backend_seq.hpp>

const std::size_t DataNum = 100;
const std::size_t ParticleNum = 1000;
const std::size_t Dim = 4;

int main ()
{
  Sampler<cv> sampler(ParticleNum, Stratified, 0.5);
  sampler.init(cv_init);
  sampler.move(cv_move(), false);
```

```
  MonitorEvalAdapter<cv, MonitorEvalSEQ> cv_est(cv_monitor_state);
  sampler.monitor("pos", 2, cv_est);

  sampler.initialize((void *) "paper_pf.data");
  sampler.iterate(DataNum - 1);

  std::ofstream est("paper_pf.est");
  est << sampler << std::endl;
  est.close();
  est.clear();

  return 0;
}
```

In the `main` function, we constructed a sampler with `ParticleNum` particles. We added the initialization function `cv_init` and the move object of type `cv_move`. Then we added a monitor that will record the importance sampling estimates of the two position parameters. Next, we initialized the sampler with data file `paper_pf.data` and iterate the sampler until all data are processed. The last step is that we output the results into a file called `paper_pf.est`.

The class `cv` will be our value collection which is a subclass of `StateMatrix<RowMajor, Dim, double>`. To illustrate both the core module and the SMP module, the initialization `cv_init` will be implemented as a standalone function. The move `cv_move` will be implemented as a derived class of `MoveSEQ<cv>`. To monitor the importance sampling estimates of the two position parameters, we will implement a simple function `cv_monitor_state` and use the adapter `MonitorEvalAdapter<cv, MonitorEvalSEQ>`.

The value collection is an N by `Dim` (in this case `Dim` $= 4$) matrix of type `double`. We can simply use `StateMatrix<RowMajor, Dim, double>` as our value collection. However, we would like to enhance its functionality through inheritance. First, since the data is shared by all particles, it is natural to bind it with the value collection. Second, both the initialization and move will need to calculate the log-likelihood. We can implement it as a standalone function. But since the log-likelihood function will need to access the data, it is convenient to implement it as a member function of the value collection `cv`. Here is the full implementation of this simple value collection class.

```
class cv : public StateMatrix<RowMajor, Dim, double>
{
  public :

  cv (size_type N) :
    StateMatrix<RowMajor, Dim, double>(N),
    x_obs_(DataNum), y_obs_(DataNum) {}

  double log_likelihood (std::size_t iter, size_type id) const
  {
    const double scale = 10;
    const double nu = 10;
    double llh_x = scale * (state(id, 0) - x_obs_[iter]);
```

```
    double llh_y = scale * (state(id, 1) - y_obs_[iter]);
    llh_x = std::log(1 + llh_x * llh_x / nu);
    llh_y = std::log(1 + llh_y * llh_y / nu);

    return -0.5 * (nu + 1) * (llh_x + llh_y);
  }

  void read_data (const char *filename)
  {
    std::ifstream data(filename);
    for (std::size_t i = 0; i != DataNum; ++i)
      data >> x_obs_[i] >> y_obs_[i];
    data.close();
    data.clear();
  }

  private :

  std::vector<double> x_obs_;
  std::vector<double> y_obs_;
};
```

The `log_likelihood` member function accepts the iteration number and the particle `id` as arguments. It returns the log-likelihood of the `id`'th particle at iteration `iter`. Inside the function, the constant `scale` is the scale of the t-distribution, i.e., $1/\alpha$. The `read_data` member function simply reads the data from a file.

The initialization is implemented through the `cv_init` function,

```
std::size_t cv_init (Particle<cv> &particle, void *filename);
```

such that, it first checks if `filename` is `NULL`. If not, then we use it to read the data. So the first initialization may look like

```
sampler.initialize((void *)filename);
```

And after that, if we want to re-initialize the sampler, we can simply call,

```
sampler.initialize();
```

This will reset the sampler and initialize it again but without reading the data. If the data set is large, repeated input and output (IO) can be very expensive. After reading the data, we will initialize each particle value by normal random variates, and calculate its log-likelihood. The last step is to set the logarithm weights of the particle collection. Since this is not a accept-reject type algorithm, the returned acceptance count bares no meaning. Here is the complete implementation,

```
std::size_t cv_init (Particle<cv> &particle, void *filename)
{
```

```
  if (filename)
    particle.value().read_data(static_cast<const char *>(filename));

  const double sd_pos0 = 2;
  const double sd_vel0 = 1;
  cxx11::normal_distribution<double> norm_pos(0, sd_pos0);
  cxx11::normal_distribution<double> norm_vel(0, sd_vel0);
  std::vector<double> log_weight(particle.size());

  for (Particle<cv>::size_type i = 0; i != particle.size(); ++i) {
    particle.value().state(i, 0) = norm_pos(particle.rng(i));
    particle.value().state(i, 1) = norm_pos(particle.rng(i));
    particle.value().state(i, 2) = norm_vel(particle.rng(i));
    particle.value().state(i, 3) = norm_vel(particle.rng(i));
    log_weight[i] = particle.value().log_likelihood(0, i);
  }
  particle.weight_set().set_log_weight(log_weight.begin());

  return 0;
}
```

In this example, we read all data from a single file for simplicity. In a realistic application, the data is often processed online – the filter is applied when new data becomes available. In this case, users can use the optional argument of `Sampler<T>::initialize` to pass necessary information to open a data connection instead of a file name.

The updating method `cv_move` is similar to `cv_init`. It will update the values of particles by adding normal random variates. However, as we see above, each call to `cv_init` causes a `log_weight` vector being allocated. Its size does not change between iterations. So it can be viewed as some resource of `cv_move` and it is natural to use a class object to manage it. Here is the implementation of `cv_move`,

```
class cv_move : public MoveSEQ<cv>
{
  public :

  void pre_processor (std::size_t, Particle<cv> &particle)
  {incw_.resize(particle.size());}

  std::size_t move_state (std::size_t iter, SingleParticle<cv> sp)
  {
    const double sd_pos = std::sqrt(0.02);
    const double sd_vel = std::sqrt(0.001);
    const double delta = 0.1;
    cxx11::normal_distribution<double> norm_pos(0, sd_pos);
    cxx11::normal_distribution<double> norm_vel(0, sd_vel);

    sp.state(0) += norm_pos(sp.rng()) + delta * sp.state(2);
```

```
    sp.state(1) += norm_pos(sp.rng()) + delta * sp.state(3);
    sp.state(2) += norm_vel(sp.rng());
    sp.state(3) += norm_vel(sp.rng());
    incw_[sp.id()] = sp.particle().value().log_likelihood(iter, sp.id());

    return 0;
  }

  void post_processor (std::size_t, Particle<cv> &particle)
  {particle.weight_set().add_log_weight(incw_.begin());}

  private :

  std::vector<double> incw_;
};
```

First, before calling any `move_state`, the `pre_processor` will be called, as described in Section 4.2. At this step, we will resize the vector used for storing incremental weights. After the first resizing, subsequent calls to `resize` will only cause reallocation if the size changed. In our example, the size of the particle system is fixed, so we do not need to worry about excessive dynamic memory allocations. The `move_state` member function moves each particle according to our model. And after `move_state` is called for each particle, the `post_processor` will be called and we simply add the logarithm incremental weights.

For each particle, we want to monitor the x_{pos} and y_{pos} parameters and get the importance sampling estimates. To extract the two values from a particle, we can implement the following function

```
void cv_monitor_state (std::size_t iter, std::size_t dim,
  ConstSingleParticle<cv> csp, double *res)
{
  assert(dim <= Dim);
  for (std::size_t d = 0; d != dim; ++d)
    res[d] = csp.state(d);
}
```

and in the `main` function we construct a monitor by

```
MonitorEvalAdapter<cv, MonitorEvalSEQ> cv_est(cv_monitor_state);
```

and add it to the sampler through

```
sampler.monitor("pos", 2, cv_est);
```

If later we decided to monitor all states, we only need to change the `2` in the above line to `Dim`.

After we implemented all the above, compiled and ran the program, a file called `paper_pf.est` was written by the following statement in the `main` function,

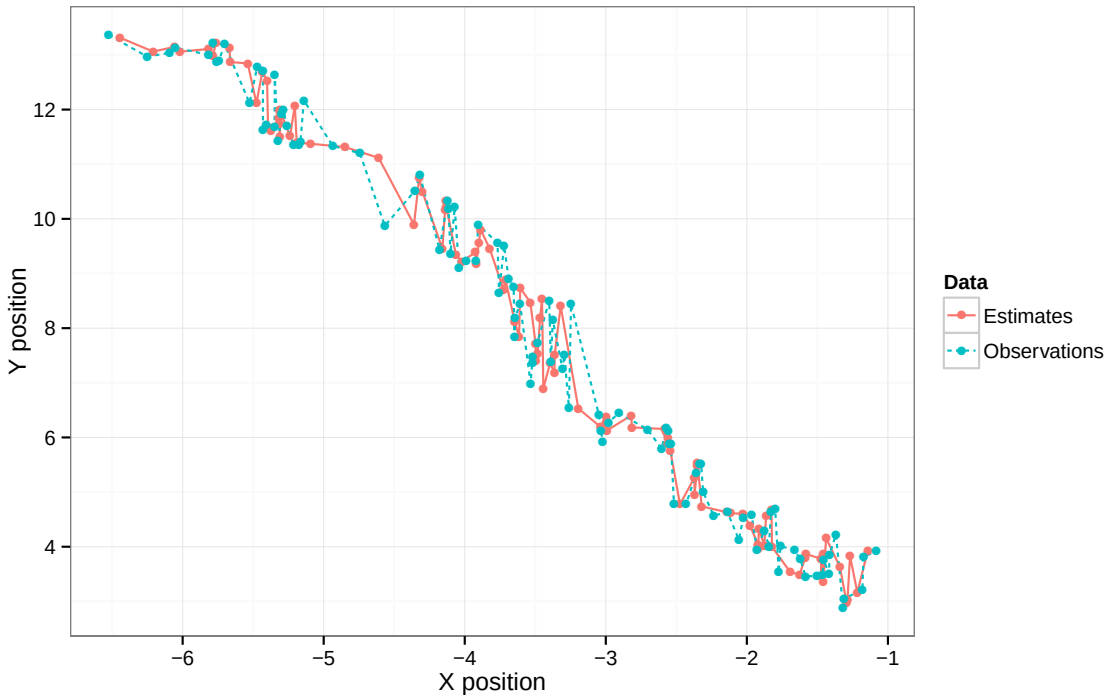

Figure 1: Observations and estimates of a simple particle filter.

```
est << sampler << std::endl;
```

The output file contains the ESS, resampling, importance sampling estimates and other informations in a table. This file can be read by most statistical software packages for further analysis. For instance, we can process this file with R, and get the plot of the estimates of positions against the observations, as shown in Figure 1.

For people familiar with the **SMCTC** library, the implementation demonstrated here is not very different. Users still implement the same functions that calculate the log-likelihood etc. In fact, the complete implementation in **vSMC** requires users to write roughly the same amount of code as an implementation in **SMCTC**. However, now we can benefit from parallelization. In addition, even the sequential version runs about twice as fast than that of **SMCTC**. The object-oriented design of **vSMC** also allows better and safer implementations. For example, in **SMCTC**, to pass informations to a callback, one can only do that through a `void` pointer. And in some cases, it is simply impossible to do so. In **vSMC**, this can be easily done through constructors or member functions of a class that defines `operator()`.

5.2. Bayesian modeling of a Gaussian mixture model

The model and algorithm

Since Richardson and Green (1997), the Gaussian mixture model (GMM) has provided a canonical example of a model-order-determination problem. We use the same model as in Del Moral *et al.* (2006b) to illustrate the implementation of this classical example in the

For each model $M_r \in \mathcal{M}$ perform the following algorithm.
Initialization
Set $t \leftarrow 0$.
Sample $\theta_r^{(i,t)} \sim \pi(\theta_r^{(i,t)}|M_r)$.
Weight $W_0^{(i)} \propto 1$.
Iteration
Set $t \leftarrow t + 1$.
Weight $W_t^{(i)} \propto W_{t-1}^{(i)} p(\boldsymbol{y}|\theta_r^{(i,t-1)}, M_r)^{\alpha(t/T_r)-\alpha([t-1]/T_r)}$.
Apply resampling if necessary.
Sample $\theta_r^{(i,t)} \sim K_t(\cdot|\theta_r^{(i,t-1)})$, a π_t-invariant MCMC kernel.
Repeat the Iteration *step up to* $t = T_r$.

Algorithm 2: SMC algorithm for Bayesian modeling of a GMM.

Monte Carlo literature. This model is also used in Zhou, Johansen, and Aston (2013) for demonstration of the use of SMC in the context of Bayesian model comparison, which provides more details of the following setting. The model is as the following, data $\boldsymbol{y} = (y_1, \dots, y_n)$ are independently and identically distributed as

$$y_i|\theta_r \sim \sum_{j=1}^{r} \omega_j \mathcal{N}(\mu_j, \lambda_j^{-1}),$$

where $\mathcal{N}(\mu_j, \lambda_j^{-1})$ denotes the normal distribution with mean μ_j and precision λ_j; $\theta_r = (\mu_{1:r}, \lambda_{1:r}, \omega_{1:r})$ and r is the number of components in each model. The parameter space is thus $R^r \times R^{+r} \times S_r$ where $S_r = \{\omega_{1:r} : 0 \le \omega_j \le 1; \sum_{j=1}^r \omega_j = 1\}$ is the standard $(r-1)$-simplex. The priors which are the same for each component are taken to be $\mu_j \sim \mathcal{N}(\xi, \kappa^{-1})$, $\lambda_j \sim \mathcal{G}(\nu, \chi)$ and $\omega_{1:r} \sim \mathcal{D}(\rho)$ where $\mathcal{D}(\rho)$ is the symmetric Dirichlet distribution with parameter ρ and $\mathcal{G}(\nu, \chi)$ is the Gamma distribution with shape ν and scale χ. The prior parameters are set in the same manner as in Richardson and Green (1997); see also Zhou *et al.* (2013) for details. The data is simulated from a four components model with $\mu_{1:4} = (-3, 0, 3, 6)$, and $\lambda_j = 2$, $\omega_j = 0.25$, $j = 1, \dots, 4$. Our interest is to simulate the posterior distribution of models with r components, denoted by M_r and obtaining the normalizing constant for the purpose of Bayesian model comparison (Robert 2007, Chap. 7).

Numerous strategies are possible to construct a sequence of distributions for the purpose of SMC sampling. One option is to use for each model M_r, $r \in \{1, 2, \dots\}$, the sequence $\{\pi_t\}_{t=0}^{T_r}$, defined by

$$\pi_t(\theta_r^t) \propto \pi(\theta_r^t|M_r) p(\boldsymbol{y}|\theta_r^t, M_r)^{\alpha(t/T_r)}, \tag{15}$$

where the number of distributions, T_r, and the annealing schedule, $\alpha : [0, 1] \to [0, 1]$, may be different for each model. This leads to Algorithm 2.

The MCMC kernel K_t in Algorithm 2 is constructed as a three-blocks Metropolis random walk,

1. Update $\mu_{1:r}$ through a normal random walk.
2. Update $\lambda_{1:r}$ through a normal random walk on logarithm scale, that is, on $\log \lambda_j$, $j = 1, \dots, r$.

3. Update $\omega_{1:r}$ through a normal random walk on logit scale, that is, on $\log(\omega_j/\omega_r)$, $j = 1, \ldots, r-1$.

The standard direct estimate of the normalizing constants (Del Moral *et al.* 2006b) can be obtained from the output of this SMC algorithm as,

$$\hat{\lambda}_{\mathrm{DS}}^{T_r,N} = \sum_{i=1}^{N} \frac{\pi(\theta_r^{(i,0)}|M_r)}{\nu(\theta_0^{(i,0)})} \times \prod_{t=1}^{T_r} \sum_{i=1}^{N} W_{t-1}^{(i)} p(\boldsymbol{y}|\theta_r^{(i,t)}, M_r)^{\alpha(t/T_r)-\alpha([t-1]/T_r)}, \tag{16}$$

where $W_{t-1}^{(i)}$ is the importance weight of sample $\theta_{t-1}^{(i)}$ and ν is the initial distribution of θ.

Path sampling for estimation of normalizing constants

As shown in Zhou *et al.* (2013) the estimation of the normalizing constant associated with our sequence of distributions can also be achieved by a Monte Carlo approximation to the *path sampling* formulation given by Gelman and Meng (1998), also known as thermodynamic integration or Ogata's method. Given a parameter α which defines a family of distributions, $\{p_\alpha = q_\alpha/Z_\alpha\}_{\alpha\in[0,1]}$ that moves smoothly from $p_0 = q_0/Z_0$ to $p_1 = q_1/Z_1$ as α increases from zero to one, one can estimate the logarithm of the ratio of their normalizing constants via a simple integral relationship,

$$\log\left(\frac{Z_1}{Z_0}\right) = \int_0^1 \mathsf{E}_\alpha\left[\frac{\mathrm{d}\log q_\alpha(\cdot)}{\mathrm{d}\,\alpha}\right] \mathrm{d}\,\alpha, \tag{17}$$

where E_α denotes expectation under p_α. The sequence of distributions in the SMC algorithm for this example can be interpreted as belonging to such a family of distributions, with $\alpha = \alpha(t/T_r)$.

The SMC sampler provides us with a set of weighted samples obtained from a sequence of distributions suitable for approximating this integral. At each time t we can obtain an estimate of the expectation within the integral via the usual importance sampling estimator, and this integral can then be approximated via a trapezoidal integration. In summary, the path sampling estimator of the ratio of normalizing constants $\lambda^{T_r} = \log(Z_1/Z_0)$ can be approximated by,

$$\hat{\lambda}_{\mathrm{PS}}^{T_r,N} = \sum_{t=1}^{T_r} \frac{1}{2}(\alpha_t - \alpha_{t-1})(U_t^N + U_{t-1}^N), \tag{18}$$

where

$$U_t^N = \sum_{i=1}^{N} W_t^{(i)} \frac{\mathrm{d}\log q_\alpha(X_t^{(i)})}{\mathrm{d}\,\alpha}\Big|_{\alpha=\alpha_t}. \tag{19}$$

Implementations

In this example we will implement the following classes.

- `gmm_param` is a class that abstracts the parameters of the model, $\theta_r = (\mu_{1:r}, \lambda_{1:r}, \omega_{1:r})$.
- `gmm_state` is the value collection class.

- `gmm_init` is a class that implements operations used to initialize the sampler.
- `gmm_move_smc` is a class that implements operations used to update the weights as well as selecting the random walk proposal scales and the distribution parameter $\alpha(t/T_r)$.
- `gmm_move_mu`, `gmm_move_lambda` and `gmm_move_weight` are classes that implement the random walks, each for one of the three blocks.
- `gmm_path` is a class that implements monitors for the path sampling estimator. This class is similar to the importance sampling monitor introduced before. It is to be used with `Sampler<gmm_state>::path_sampling`. Its interface requirement will be documented later.
- `gmm_alpha_linear` and `gmm_alpha_prior` are classes that implement two of the many possible annealing schemes, $\alpha(t/T_r) = t/T_r$ (linear) and $\alpha(t/T_r) = (t/T_r)^p$, $p > 1$ (prior).
- And last, the `main` function, which configures, initializes and iterates the sampler.

This example is considerably more complicated than the last one. Instead of documenting all the implementation details, for many classes we will only show the interfaces. In most cases, the implementations are straightforward as they are either data member accessors or simple translations of mathematical formulations. For member functions with more complex structures, detailed explanation will be given. Interested readers can see the source (`example/paper/src/paper_gmm.cpp`) for more details.

Later we will build both sequential and parallelized samplers. A few configuration macros will be defined at compile time. For example, the sequential sampler is compiled with the following header and macros,

```
#include <vsmc/smp/backend_seq.hpp>
#define BASE_SATE StateSEQ
#define BASE_INIT InitializeSEQ
#define BASE_MOVE MoveSEQ
#define BASE_PATH PathEvalSEQ
```

The definitions of these macros will be changed at compile time to build parallelized samplers. For example, when using OpenMP parallelization, the header `backend_omp.hpp` will be used instead of `backend_seq.hpp`; and `StateSEQ` will be changed to `StateOMP` along with similar changes to the other macros. In the distributed source, this is configured by the **CMake** build system.

Again, we first introduce the `main` function. The required headers are the same as the last particle filter example in addition to the SMP backend headers as described above. The following variables used in the `main` function will be set by user input.

```
int ParticleNum;
int AnnealingScheme;
int PriorPower;
int CompNum;
std::string DataFile;
```

In the `main` function, we will create objects that set the distribution parameter $\alpha(t/T_r)$ at each iteration according to the user input of `AnnealingScheme`. Below is the `main` function. Note that some code of IO operations which set the parameters above is omitted.

```
int main ()
{
  gmm_move_smc::alpha_setter_type alpha_setter;
  if (AnnealingScheme == 1)
    alpha_setter = gmm_alpha_linear(IterNum);
  if (AnnealingScheme == 2)
    alpha_setter = gmm_alpha_prior(IterNum, PriorPower);

  Sampler<gmm_state> sampler(ParticleNum, Stratified, 0.5);
  sampler.particle().value().comp_num(CompNum);
  sampler
    .init(gmm_init());
    .move(gmm_move_smc(alpha_setter), false);
    .mcmc(gmm_move_mu(), false);
    .mcmc(gmm_move_lambda(), true);
    .mcmc(gmm_move_weight(), true);
    .path_sampling(gmm_path());
    .initialize((void *) DataFile.c_str());
    .iterate(IterNum);

  double ds = sampler.particle().value().nc().log_zconst();
  double ps = sampler.path().log_zconst();
  std::cout << "Standard estimate :      " << ds << std::endl;
  std::cout << "Path sampling estimate : " << ps << std::endl;

  return 0;
}
```

The sampler first sets the number of components and allocates memory through member function `comp_num` of `gmm_state`. Then it sets the initialization and updating methods. Before possible resampling, a `gmm_move_smc` object is added. After that, three Metropolis random walks are appended. In addition, we add a `gmm_path` object to calculate the path sampling integration. Then we initialize and iterate the sampler and get the normalizing constant estimates.

It is obvious that the parameter class `gmm_param` needs to store the parameters $(\mu_{1:r}, \lambda_{1:r}, \omega_{1:r})$. We also associate with each particle its log-likelihood and log-prior. Here is the definition of the `gmm_param` class. We omitted definitions of some data access member functions.

```
class gmm_param
{
  public :

  void comp_num (std::size_t num);
```

```
  void save_old ();

  double log_prior () const {return log_prior_;}
  double &log_prior () {return log_prior_;}

  double log_likelihood () const {return log_likelihood_;}
  double &log_likelihood () {return log_likelihood_;}

  int mh_reject_mu (double p, double u);
  int mh_reject_lambda (double p, double u);
  int mh_reject_weight (double p, double u);
  int mh_reject_common (double p, double u);

  double log_lambda_diff () const;
  double logit_weight_diff () const;

  void update_log_lambda ();

  private :

  std::size_t comp_num_;
  double log_prior_, log_prior_old_, log_likelihood_, log_likelihood_old_;
  std::vector<double> mu_, mu_old_;
  std::vector<double> lambda_, lambda_old_;
  std::vector<double> weight_, weight_old_;
  std::vector<double> log_lambda_;
};
```

The `comp_num` member function allocates the memory for a given number of components. The `save_old` member function saves the current particle states. It is used before the states are updated with the random walk proposals, as we will see later when we implement the `gmm_move_mu` class. The `mh_reject_mu` member function accepts the Metropolis acceptance probability p and a uniform $(0, 1]$ random variate, say u; it rejects the proposed change if $p < u$, and restores the particle state of the parameters $\mu_{1:r}$ with those saved by `save_old`. The member functions `mh_reject_lambda` and `mh_reject_weight` do the same for the other two sets of parameters. All these three also call the function `mh_reject_common` which restores the stored log-likelihood and log-prior values. The use of these member functions will be seen in the implementation of `gmm_move_mu`, in the context of which their own implementation becomes obvious. Other member functions provide some useful computations such as the logarithm of the $\lambda_{1:r}$. They are used when computing the log-likelihood.

The class `gmm_state` contains some properties common to all particles, such as the data and the distribution parameter $\alpha(t/T_r)$. Also, we will use it to record the logarithm of the ratio of normalizing constants, using the `NormalizingConstant` class. We will see how to update this variable at each iteration in the implementation of `gmm_move_smc`. The prior parameters are also stored in the value collection. Here is the definition of this value collection class. Again, we omitted some data access member functions,

```
class gmm_state : public BASE_STATE<StateMatrix<RowMajor, 1, gmm_param> >
{
  public :

  NormalizingConstant &nc () {return nc_;}
  const NormalizingConstant &nc () const {return nc_;}

  void alpha (double a)
  {
    a = a < 1 ? a : 1;
    a = a > 0 ? a : 0;
    if (a == 0) {
      alpha_inc_ = 0;
      alpha_ = 0;
    } else {
      alpha_inc_ = a - alpha_;
      alpha_ = a;
    }
  }

  void comp_num (std::size_t num)
  {
    comp_num_ = num;
    for (size_type i = 0; i != this->size(); ++i)
      this->state(i, 0).comp_num(num);
  }

  double update_log_prior (gmm_param &param) const;

  double update_log_likelihood (gmm_param &param) const
  {
    const double log2pi = 1.8378770664093455;
    double ll = -0.5 * obs_.size() * log2pi;
    param.update_log_lambda();
    for (std::size_t k = 0; k != obs_.size(); ++k) {
      double lli = 0;
      for (std::size_t i = 0; i != param.comp_num(); ++i) {
        double resid = obs_[k] - param.mu(i);
        lli += param.weight(i) * std::exp(
          0.5 * param.log_lambda(i) -
          0.5 * param.lambda(i) * resid * resid);
      }
      ll += std::log(lli);
    }

    return param.log_likelihood() = ll;
  }
```

```
  void read_data (const char *filename);

  private :

  NormalizingConstant nc_;
  std::size_t comp_num_;
  double alpha_, alpha_inc_;
  double mu0_, sd0_, shape0_, scale0_;
  double mu_sd_, lambda_sd_, weight_sd_;
  std::vector<double> obs_;
};
```

The variable `alpha_inc_` is $\Delta\alpha(t/T_r) = \alpha(t/T_r) - \alpha((t-1)/T_r)$, which will be used when we update the weights. The variable `nc_` of type `NormalizingConstant` will be updated when the weights are changed by `gmm_move_smc` and it will compute the standard normalizing constant estimate $\hat{\lambda}_{\mathrm{DS}}^{T_r,N}$. The variables `mu0_` and `sd0_` are the prior parameters of the means $\mu_{1:r}$. The variables `shape0_` and `scale0_` are the prior parameters of the precisions $\lambda_{1:r}$. The variables `mu_sd_`, `lambda_sd_`, and `weight_sd_` are the proposal scales of the three random walks, respectively. The data access member functions of these variables are omitted in the above code snippet.

In the `update_log_likelihood` member function, the calculation is a straightforward translation of the mathematical formulation. The `gmm_param::update_log_lambda` member function is used before the loop, which simply calculates $\log \lambda_j$ for $j = 1, \ldots, r$, and stores their values. The purpose is to avoid repeated computation of these quantities inside the loop. When the function returns, it uses the mutable version of the `gmm_param::log_likelihood` member function to update the log-likelihood stored in the `param` object. This is the reason that the function is named with an `update` prefix. As we will see later, whenever the parameter values are updated, it will be followed by a call to `update_log_likelihood` and `update_log_prior`, which are implemented in a similar fashion. Therefore the value we get by calling `gmm_param::log_likelihood` will always be "up-to-date" while no repeated computation is involved. Surely there are other and possibly better design choices. However, for this simple example, this design serves our purpose well.

The initialization is implemented using the `gmm_init` class,

```
class gmm_init : public BASE_INIT<gmm_state, gmm_init>
{
  public :

  std::size_t initialize_state (SingleParticle<gmm_state> sp)
  {
    const gmm_state &state = sp.particle().value();
    gmm_param &param = sp.state(0);

    cxx11::normal_distribution<> rmu(state.mu0(), state.sd0());
    cxx11::gamma_distribution<> rlambda(state.shape0(), state.scale0());
    cxx11::gamma_distribution<> rweight(1, 1);
```

```
    double sum = 0;
    for (std::size_t i = 0; i != param.comp_num(); ++i) {
      param.mu(i) = rmu(sp.rng());
      param.lambda(i) = rlambda(sp.rng());
      param.weight(i) = rweight(sp.rng());
      sum += param.weight(i);
    }
    for (std::size_t i = 0; i != param.comp_num(); ++i)
      param.weight(i) /= sum;

    state.update_log_prior(param);
    state.update_log_likelihood(param);

    return 1;
  }

  void initialize_param (Particle<gmm_state> &particle,
    void *filename)
  {
    if (filename)
      particle.value().read_data(static_cast<const char *>(filename));
    particle.value().alpha(0);
    particle.set_equal_weight();
    particle.value().nc().initialize();
  }
};
```

The `initialize_param` member function is called before the `pre_processor`, which is absent in this case and has a default implementation which does nothing. It processes the optional parameter of `Sampler::initialize`, the file name of the data. The `initialize_state` member function initializes the state values according to the prior and updates the log-prior and log-likelihood.

After initialization, at each iteration, the `gmm_move_smc` class will implement the updating of weights as well as the selecting of the proposal scales and the distribution parameter. For example, when using the linear annealing scheme, we can implement a `gmm_alpha_linear` class by the following,

```
class gmm_alpha_linear
{
  public :

  gmm_alpha_linear (const std::size_t iter_num) : iter_num_(iter_num) {}

  void operator() (std::size_t iter, Particle<gmm_state> &particle)
  {particle.value().alpha(static_cast<double>(iter) / iter_num_);}
```

```
  private :

  std::size_t iter_num_;
};
```

It accepts the total number of iterations T_r as an argument to its constructor. And it implements an `operator()` that updates the distribution parameter $\alpha(t/T_r)$. The prior annealing scheme can be implemented similarly. For simplicity and demonstration purpose, we only allow `gmm_move_smc` to be configured with different annealing schemes, and hard code the proposal scales. An industry strength design may make this class a template with annealing scheme and proposal scales as policy template parameters.

```
class gmm_move_smc
{
  public :

  typedef cxx11::function<void (std::size_t, Particle<gmm_state> &)>
    alpha_setter_type;

  gmm_move_smc (const alpha_setter_type &alpha_setter) :
    alpha_setter_(alpha_setter) {}

  std::size_t operator() (std::size_t iter, Particle<gmm_state> &particle)
  {
    alpha_setter_(iter, particle);

    double alpha = particle.value().alpha();
    alpha = alpha < 0.02 ? 0.02 : alpha;
    particle.value().mu_sd(0.15 / alpha);
    particle.value().lambda_sd((1 + std::sqrt(1 / alpha)) * 0.15);
    particle.value().weight_sd((1 + std::sqrt(1 / alpha)) * 0.2);

    incw_.resize(particle.size());
    weight_.resize(particle.size());
    particle.read_weight(weight_.begin());
    double coeff = particle.value().alpha_inc();
    for (Particle<gmm_state>::size_type i = 0; i != particle.size(); ++i)
    {
      incw_[i] = coeff * particle.value().state(i, 0).log_likelihood();
    }
    particle.value().nc().add_log_weight(&incw_[0], particle.weight_set());
    particle.weight_set().add_log_weight(&incw_[0]);

    return 0;
  }

  private :
```

```
  alpha_setter_type alpha_setter_;
  std::vector<double> incw_;
  std::vector<double> weight_;
};
```

Note that `cxx11::function` is an alias to either `std::function` or `boost::function`, depending on the value of the configuration macro `VSMC_HAS_CXX11LIB_FUNCTIONAL`. Objects of this class type can be added to a sampler as a move. The `operator()` satisfies the interface requirement of the core module. First it uses `alpha_setter_` to set the distribution parameter $\alpha(t/T_r)$. Second, it sets the proposal scales for the three Metropolis random walks according to the current value of α. Then it computes the *unnormalized* incremental weights. The `NormalizingConstant` class has member function `add_log_weight`, which is not unlike the one with the same name in `WeightSet`. It accepts the logarithm of the incremental weights and a `WeightSet` object. The standard normalizing constant estimates will be computed using these values. Last, we also modify the `WeightSet` type object itself by adding the logarithm of the incremental weights.

At each iteration, random walks are also performed. The implementations of the random walks are straightforward. Below is the implementation of the random walk on the mean parameters. The random walks on the other parameters are similar.

```
class gmm_move_mu : public BASE_MOVE<gmm_state, gmm_move_mu>
{
  public :

  std::size_t move_state (std::size_t iter, SingleParticle<gmm_state> sp)
  {
    const gmm_state &state = sp.particle().value();
    gmm_param &param = sp.state(0);

    cxx11::normal_distribution<> rmu(0, state.mu_sd());
    cxx11::uniform_real_distribution<> runif(0, 1);

    double p =
      param.log_prior() + state.alpha() * param.log_likelihood();
    param.save_old();
    for (std::size_t i = 0; i != param.comp_num(); ++i)
      param.mu(i) += rmu(sp.rng());
    p = state.update_log_prior(param) +
      state.alpha() * state.update_log_likelihood(param) - p;
    double u = std::log(runif(sp.rng()));

    return param.mh_reject_mu(p, u);
  }
};
```

First we save the logarithm of the value of the target density computed using the old values in `p`, the acceptance probability. And then we call `gmm_param::save_old` to save the old values.

Next we update each parameter with a proposed normal random variates and compute the new log-prior and the log-likelihood as well as the new value of the target density. Then we reject it according to the Metropolis algorithm, as implemented in `gmm_param`, which manages both the current states as well as the backup.

Last we need to monitor certain quantities for inference purpose. Recall that, in the `main` function we used `sampler.path_sampling(gmm_path())` to set the monitoring of path sampling integrands. The `path_sampling` member function requires a callable object with the following signature,

```
double path_eval (std::size_t iter, const Particle<T> &, double *res);
```

The input parameter `iter` is the iteration number, the value of t in Equation 18. The return value shall be the value of α_t. The output parameter `res` shall store the array of values

$$\frac{\mathrm{d} \log q_\alpha(X_t^{(i)})}{\mathrm{d}\, \alpha} \Big|_{\alpha=\alpha_t}.$$

Our implementation of `gmm_path` is a subclass of an SMP module base class, which provides an `operator()` that satisfies the above interface requirement. Its usage is similar to the `MoveSEQ` template introduced in Section 4.2.

The path sampling integrands under this geometry annealing scheme are simply the log-likelihoods. Therefore the implementation of `gmm_path` class is rather simple.

```
class gmm_path : public BASE_PATH<gmm_state, gmm_path>
{
  public :

  double path_state (std::size_t, ConstSingleParticle<gmm_state> sp)
  {
    return sp.state(0).log_likelihood();
  }

  double path_grid (std::size_t, const Particle<gmm_state> &particle)
  {
    return particle.value().alpha();
  }
};
```

Results

After compiling and running the algorithm, the results were consistent with those reported in Del Moral *et al.* (2006b). For a more in depth analysis of the methodologies, extensions and the results see Zhou *et al.* (2013).

Extending the implementation using MPI

vSMC's MPI module assumes that identical samplers are constructed on each node, with possibly different number of particles to accommodate the difference in capacities among

nodes. To extend the above SMP implementation for use with MPI, first at the beginning of the `main` function, we add the following,

```
MPIEnvironment env(argc, argv);
```

to initialize the MPI environment. When the object `env` is destroyed at the exit of the `main` function, the MPI environment is finalized. Second, we need to replace the base value collection class template with `StateMPI`. So now `gmm_state` is declared as the following.

```
class gmm_state :
  public StateMPI<BASE_STATE<StateMatrix<RowMajor, 1, gmm_param> > >;
```

The implementation is exactly the same as before. Third, the `gmm_param` class now needs to be transferable using MPI. Unlike the SMP situations, a simple copy constructor is not enough. **vSMC** uses the **Boost** MPI library, and thus one only needs to write a `serialize` member function for `gmm_param` such that the data can be serialized into bytes. See the documentation of the **Boost** MPI and serialization libraries for details. In summary, the following member function accepts an `Archive` object as input, and it can perform a store or a load operation based on the `Archive` type. In a load operation, the `Archive` object is like an input stream and in a store operation, it is like an output stream.

```
template <typename Archive>
void serialize (Archive &ar, const unsigned)
{
  int num = comp_num_;
  ar & num;
  comp_num(num);

  ar & log_prior_;
  ar & log_likelihood_;
  for (std::size_t i = 0; i != comp_num_; ++i) {
    ar & mu_[i];
    ar & lambda_[i];
    ar & weight_[i];
    ar & log_lambda_[i];
  }
}
```

Fourth, after user input of the sampler parameters, we need to sync them with all nodes. For example, for the `ParticleNum` parameter,

```
boost::mpi::communicator World;
boost::mpi::broadcast(World, ParticleNum, 0);
```

Last, any importance sampling estimates that are computed on each node, need to be combined into final results. For example, the path sampling results are now obtained through adding the results from each node together,

```
double ps_sum = 0;
boost::mpi::reduce(World, ps, ps_sum, std::plus<double>(), 0);
ps = ps_sum;
```

For the standard normalizing constant ratio estimator, we will replace `NormalizingConstant` with `NormalizingConstantMPI`, which will perform these and other tasks.

After these few lines of modification, the sampler is now parallelized using MPI and can be deployed to clusters and other distributed memory architecture. On each node, the selected SMP parallelization is used to perform multi-threading parallelization locally. **vSMC**'s MPI module will take care of normalizing weights and other tasks.

Parallelization performance

One of the main motivations behind the creation of **vSMC** is to ease the parallelization with different programming models. The same implementation can be used to build different samplers based on what kind of parallel programming model is supported on users' platforms. In this section we compare the performance of various SMP parallel programming models and OpenCL parallelization.

We consider five different implementations supported by the Intel C++ Compiler 2013: sequential, Intel **TBB**, Cilk Plus, OpenMP and C++11 `<thread>`. The samplers are compiled with

```
CXX=icpc -std=c++11 -gcc-name=gcc-4.7 -gxx-name=g++-4.7
CXXFLAGS=-O3 -xHost -fp-model precise  \
  -DVSMC_HAS_CXX11LIB_FUNCTIONAL=1 -DVSMC_HAS_CXX11LIB_RANDOM=1
```

on a Ubuntu 12.10 workstation with a Xeon W3550 (3.06GHz, 4 cores, 8 hardware threads through hyper-threading) CPU. A four components model and 100 iterations with a prior annealing scheme is used for all implementations. A range of numbers of particles are tested, from 2^3 to 2^{17}.

For different number of particles, the wall clock time and speedup are shown in Figure 2. For 10^4 or more particles, the differences are minimal among all the programming models. They all have roughly 550% speedup. With smaller number of particles, **vSMC**'s C++11 parallelization is less efficient than other industry strength programming models. However, with 1000 or more particles, which is less than those used in typical applications, the difference is not very significant.

OpenCL implementations are also compared on the same workstation, which also has an NVIDIA Quadro 2000 graphic card. OpenCL programs can be compiled to run on both CPUs and GPUs. For CPU implementation, there are Intel OpenCL (Intel Cooperation 2013b) and AMD **APP** OpenCL (Advanced Micro Devices Inc. 2012) platforms. We use the Intel **TBB** implementation as a baseline for comparison. The same OpenCL implementations are used for all the CPU and GPU runtimes. Therefore they are not particularly optimized for any of them. For the GPU implementation, in addition to double precision, we also tested a single precision configuration. Unlike modern CPUs, which have the same performance for double and single precision floating point operations (unless SIMD), GPUs penalize double precision performance heavily.

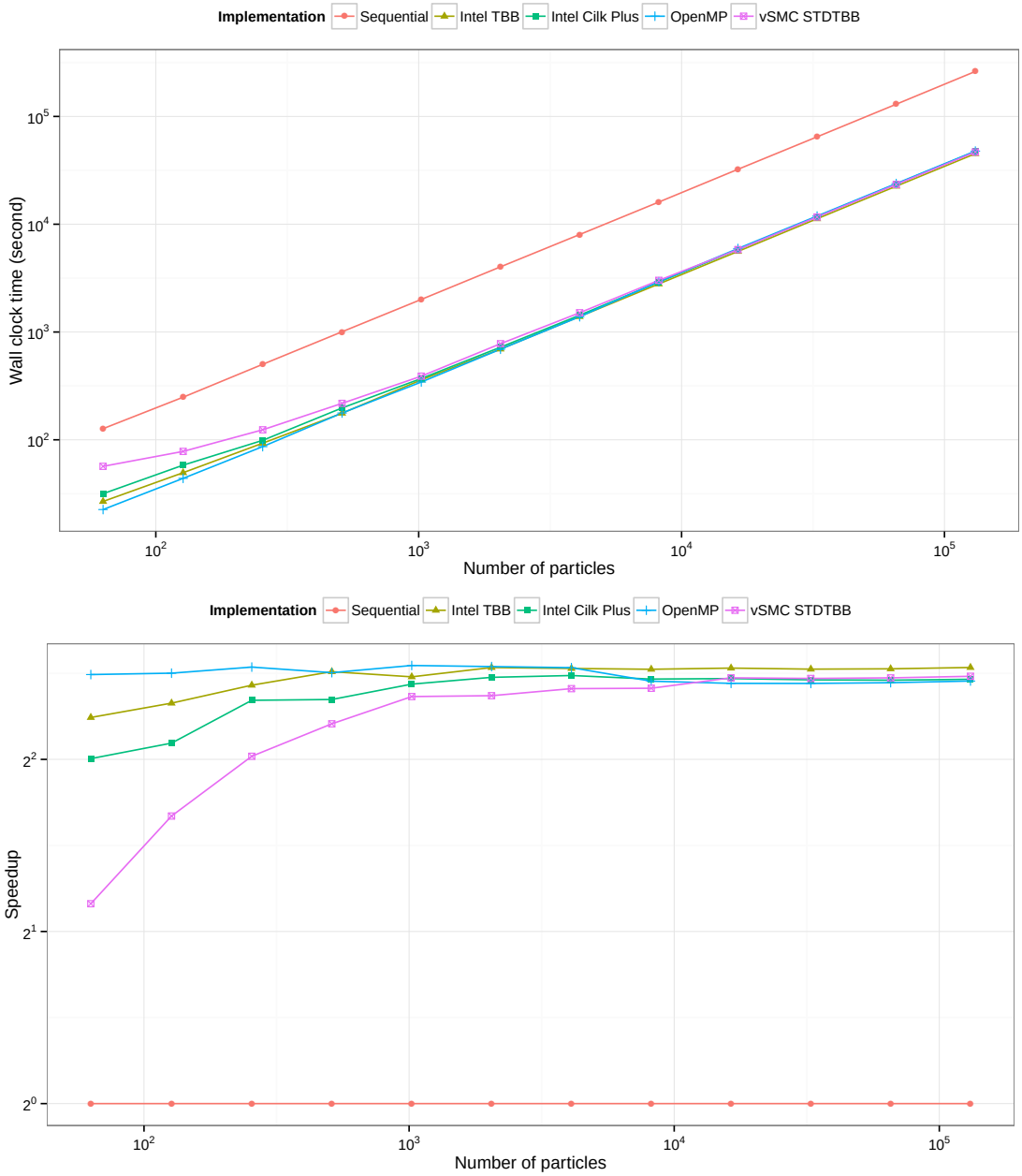

Figure 2: Performance of C++ implementations of Bayesian modeling for a GMM (Linux; Xeon W3550, 3.06GHz, 4 cores, 8 threads).

For different numbers of particles, the wall clock time and speedup are plotted in Figure 3. With smaller numbers of particles, the OpenCL implementations have a high overhead when compared to the Intel **TBB** implementation. With a large number of particles, AMD **APP** OpenCL has a similar performance as the Intel **TBB** implementation. Intel OpenCL is about 40% faster than the Intel **TBB** implementation. This is due to more efficient vectorization and compiler optimizations. The double precision performance of the NVIDIA GPU has a 220% speedup and the single precision performance has near 1600% speedup. As a rough

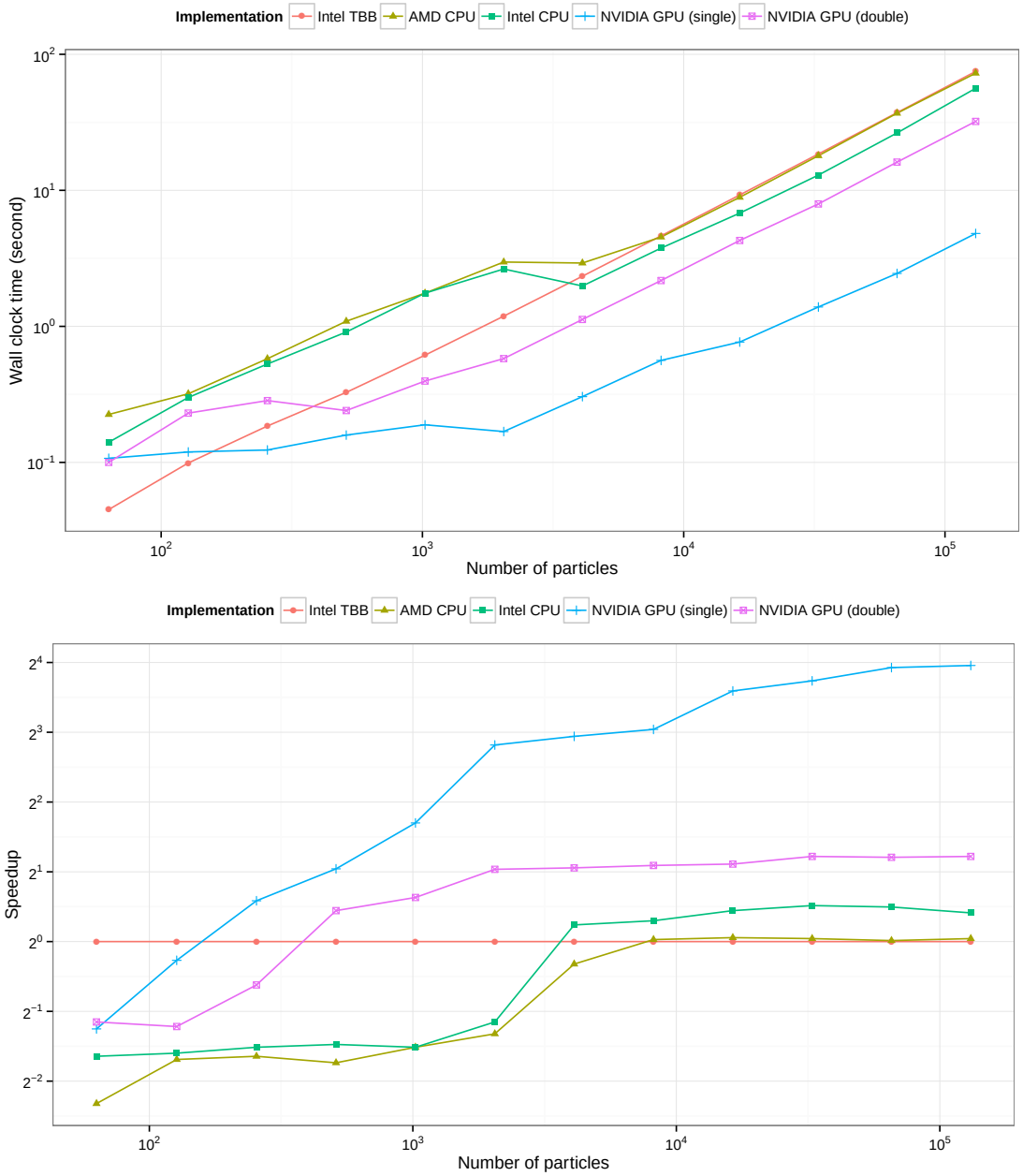

Figure 3: Performance of OpenCL implementations of Bayesian modeling for a GMM (Linux; Xeon W3550 GPU, 3.06GHz, 4 cores, 8 threads; NVIDIA Quadro 2000).

reference for the expected performance gain, the CPU has a theoretical peak performance of 24.48 GFLOPS. The GPU has a theoretical peak performance of 60 GFLOPS in double precision and 480 GFLOPS in single precision. This represents 245% and 1960% speedup compared to the CPU, respectively.

It is widely believed that OpenCL programming is tedious and hard. However, **vSMC** facilities to manage OpenCL platforms and devices as well as common operations. Limited by the scope of this paper, the OpenCL implementation (distributed with the **vSMC** source) is not

documented in this paper. Overall the OpenCL implementation has about 800 lines of code including both host and device code. It is not an enormous increase in effort when compared to the 500 lines of code for the SMP implementation. Less than doubling the code base but gaining more than 15 times performance speedup, we consider the programming effort is relatively small.

6. Discussion

This paper introduced a C++ template library intended for implementing generic SMC algorithms and constructing parallel samplers with different programming models. While it is possible to implement many realistic applications with the presented framework, some technical proficiency is still required to implement some problem specific parts of the algorithms. Some basic knowledge of C++ in general and how to use a template library is also required.

It is shown that with the presented framework it is possible to implement parallelized, scalable SMC samplers in an efficient and reusable way. The performance of some common parallel programming models are compared using an example.

Future work may consist of easing the implementation of SMC algorithms further. However, there is a balance between performance, flexibility and the ease of use. **vSMC** aims to be developer-friendly and to provide users as much control as possible for all performance related aspects. For a BUGS-like interface, users may be interested in other software such as **BiiPS** (Caron, Todeschini, Legrand, and Del Moral 2012). In addition **LibBi** (Murray 2013) provides a user friendly and high performance alternative with a focus on state-space models. Compared with these recent developments, **vSMC** is less accessible to those with little or no knowledge of C++. However, for researchers with expertise in C++ and template metaprogramming in particular, **vSMC** provides a framework within which potential superior performance can be obtained and greater flexibility and extensibility are possible.

References

Advanced Micro Devices Inc (2012). *AMD Accelerated Parallel Processing OpenCL Programming Guide*. Version 2.8, URL http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk.

Calderhead B, Girolami M (2009). "Estimating Bayes Factors via Thermodynamic Integration and Population MCMC." *Computational Statistics & Data Analysis*, **53**(12), 4028–4045.

Cappé O, Godsill SJ, Moulines E (2007). "An Overview of Existing Methods and Recent Advances in Sequential Monte Carlo." *Proceedings of the IEEE*, **95**(5), 899–924.

Caron F, Todeschini A, Legrand P, Del Moral P (2012). ***BiiPS**: Bayesian Inference with Interacting Particle Systems*. Version 0.7.2, URL https://alea.bordeaux.inria.fr/biips/doku.php.

Dawes B, others (2013). ***Boost** – C++ Libraries*. Version 1.53, URL http://www.boost.org/.

Del Moral P, Doucet A, Jasra A (2006a). "Sequential Monte Carlo Methods for Bayesian Computation." In *Bayesian Statistics 8*. Oxford University Press.

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

Douc R, Cappé O, Moulines E (2005). "Comparison of Resampling Schemes for Particle Filtering." In *Proceedings of the 4th International Symposium on Image and Signal Processing and Analysis*, pp. 64–69.

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

Gelman A, Meng XL (1998). "Simulating Normalizing Constants: From Importance Sampling to Bridge Sampling to Path Sampling." *Statistical Science*, **13**(2), 163–185.

GNU Project (2013). ***GCC*** *– GNU Compiler Collection*. Version 4.8.1, URL http://gcc.gnu.org/.

Intel Cooperation (2011). *Intel Cilk Plus Language Specification*. Version 1.1, URL http://cilkplus.org/.

Intel Cooperation (2013a). *Intel C++ Composer XE*. Version 13.1, URL http://software.intel.com/en-us/intel-compilers/.

Intel Cooperation (2013b). *Intel SDK for OpenCL Applications 2013*. URL http://software.intel.com/en-us/vcsource/tools/opencl-sdk/.

Intel Cooperation (2013c). *Intel Threading Building Blocks*. Version 4.1, URL http://threadingbuildingblocks.org/.

Jasra A, Stephens DA, Doucet A, Tsagaris T (2010). "Inference for Lévy-Driven Stochastic Volatility Models via Adaptive Sequential Monte Carlo." *Scandinavian Journal of Statistics*, **38**(1), 1–22.

Johansen AM (2009). "**SMCTC**: Sequential Monte Carlo in C++." *Journal of Statistical Software*, **30**(6), 1–41. URL http://www.jstatsoft.org/v30/i06/.

Kronos OpenCL Working Group (2012). *The OpenCL Specification*. Version 1.2, URL http://www.khronos.org/opencl/.

Lee A, Yau C, Giles MB, Doucet A, Holmes CC (2010). "On the Utility of Graphics Cards to Perform Massively Parallel Simulation of Advanced Monte Carlo Methods." *Journal of Computational and Graphical Statistics*, **19**(4), 769–789.

Liu JS, Chen R (1998). "Sequential Monte Carlo Methods for Dynamic Systems." *Journal of the American Statistical Association*, **93**(443), 1032–1044.

Lunn D, Spiegelhalter D, Thomas A, Best N (2009). "The BUGS Project: Evolution, Critique and Future Directions." *Statistics in Medicine*, **28**(25), 3049–3067.

Martin K, Hoffman B (2010). *Mastering **CMake***. Version 2.8, URL http://www.cmake.org/.

Message Passing Interface Forum (2012). *MPI: A Message-Passing Interface Standard*. Version 3.0, URL http://www.mpi-forum.org.

Microsoft Cooperation (2012). *Microsoft Visual C++ 2012*. URL http://msdn.microsoft.com/en-us/vstudio/hh386302/.

Murray LM (2013). "Bayesain State-Space Modelling on High-Performance Hardware Using **LibBi**." arXiv:1306.3277 [stat.CO], URL http://arxiv.org/abs/1306.3277.

Neal RM (2001). "Annealed Importance Sampling." *Statistics and Computing*, **11**(2), 125–139.

OpenMP Architecture Review Board (2011). *OpenMP Application Program Interface*. Version 3.1, URL http://www.openmp.org/.

Peters GW (2005). *Topics in Sequential Monte Carlo Samplers*. Master's thesis, Department of Engineering, University of Cambridge.

R Core Team (2014). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL http://www.R-project.org/.

Richardson S, Green PJ (1997). "On Bayesian Analysis of Mixtures with an Unknown Number of Components." *Journal of Royal Statistical Society B*, **59**(4), 731–792.

Robert CP (2007). *The Bayesian Choice: From Decision-Theoretic Foundations to Computational Implementation*. 2nd edition. Springer-Verlag, New York.

Salmon JK, Moraes MA, Dror RO, Shaw DE (2011). "Parallel Random Numbers: As Easy as 1, 2, 3." pp. 1–12.

Spiegelhalter D, Thomas A, Best N (1996). *BUGS – Bayesian Inference Using Gibbs Sampling*. Version 0.5, URL http://www.mrc-bsu.cam.ac.uk/bugs/.

The LLVM Developer Group (2013a). "**clang**: A C Language Family Frontend for LLVM." Version 3.2, URL http://clang.llvm.org/.

The LLVM Developer Group (2013b). *"**libc++**" C++ Standard Library*. Version 1.1, URL http://libcxx.llvm.org/.

Torvalds L, others (2013). ***git** – Distributed Source Code Management*. Version 1.8.3.1, URL http://git-scm.com/.

van Heesch D (2013). ***Doxygen** – Generating Documentation from Source Code*. Version 1.8.4, URL http://www.stack.nl/~dimitri/doxygen/.

Veldhuizen T (2006). "**Blitz++** User's Guide." Version 0.9, URL http://blitz.sourceforge.net/.

Zhou Y, Johansen AM, Aston JAD (2013). "Towards Automatic Model Comparison: Adaptive Sequential Monte Carlo Approach." arXiv:1303.3123 [stat.ME], URL http://arxiv.org/abs/1303.3123.

A. vSMC and SMCTC

As noted by one referee, it is of interest to compare the **vSMC** and **SMCTC** libraries. In particular, the differences and advantages of **vSMC** when parallel computing is not involved.

As mentioned in Section 1, the **vSMC** library has some similarity to **SMCTC**. This is by intention so users already familiar with **SMCTC** can learn the new library more easily. For example, they both use user defined types to abstract algorithm specific state space. They also both use user defined callbacks to perform algorithm specific operations such as updating particles. However there are also differences.

The most significant difference is perhaps how the particle system is abstracted. In **SMCTC**, users write custom classes to abstract $X^{(i)}$. In **vSMC**, users define classes to abstract $\{X^{(i)}\}_{i=1}^N$. In addition, in **SMCTC** user defined callbacks operate on each particle individually while in **vSMC** they operate on all particles as a whole. This design of **vSMC** allows different parallelization models to be implemented. For example, suppose that SIMD vectorization is desired, which is not currently directly supported by **vSMC**, one only needs to implement the value collection type such that the values are properly aligned in memory and in each callback, SIMD operations can be applied to the whole value collection since the **vSMC** interface allows users to access them all at once. This kind of constructs is not possible in **SMCTC** without changing internals of the library. See also Section 4.1 on the value collection type of **vSMC**.

Apart from parallelization, **vSMC** also has other advantages. Internally it has a very different design to that of **SMCTC**. It is more modular and almost all parts of the sampler can be replaced without changing the internal of the library. For example, consider that a new resampling algorithm is desired. In **SMCTC**, if it is not provided by the library, users have to implement all the operations ranging from generating replication numbers to copying each particle. In **vSMC**, one only needs to implement a function with the following signature,

```
template <typename SizeType>
void resample (std::size_t N, const double *weight, SizeType *copy_from);
```

which given the normalized weights generates the number of replicates of each particle. And in the `Sampler`'s constructor, one passes this object instead of the name of the built-in resampling scheme name. **vSMC**'s implementations of gathering the normalized weights into an array, and copying particles according to the replication numbers can be reused. This particular example is of interest when the sampler is parallelized using MPI, where copying particles across all nodes requires some effort that may not be trivial. On the other hand, if an existing resampling algorithm is sufficient, but the parallelization requires special memory management, all one needs to do is providing **vSMC** with a specialized `copy` member function inside their value collection class. And the implementation of the resampling can be reused. (This is exactly what **vSMC** does for the OpenCL and MPI module.)

In some situations, **SMCTC** might appear to be easier to use than **vSMC**. For example, it allows one to define a class that abstracts a single particle. And naturally such classes can easily define operations that manipulate a single particle. In contrast, in **vSMC** users can only interact with a single particle through the `SingleParticle<T>` object. However, as shown in Section 4.2, users can obtain similar effects by defining the `single_particle_type` class template inside the value collection type. Alternatively, one can also reuse classes defined for **SMCTC** by combining them with `StateMatrix` or `StateTuple`. The `single_particle_type` and the resampling example are only two of many possibilities to extend or replace the default

implementations of **vSMC**. Through the use of template metaprogramming, **vSMC** is much more flexible than **SMCTC**.

In summary, **vSMC** aims to be as flexible as possible. To quote Larry Wall: "Easy things should be easy, and hard things should be possible." In **vSMC**, this means when some standard algorithm design is used, for example a common resampling algorithm such as stratified resampling, it should be as easy as passing a parameter to a **vSMC** function. When the algorithm is not directly supported by the library, users can replace **vSMC**'s implementation without knowing the internal of the library while reusing as much as possible the rest of the library. For example, the library can be used to implement population MCMC algorithms, such as those used in Calderhead and Girolami (2009) for Bayesian model comparison. In addition, when multiple algorithms share common operations, such as MCMC kernels, they can be implemented once and shared between different algorithms easily. See the examples distributed with the source, where an SMC and a population MCMC implementation of the same GMM model share a large amount of code.

vSMC also enjoys higher performance even for sequential implementations. For instance, as noted before, a sequential implementation in **vSMC** of the particle filter in Section 5.1 is about twice faster than that in **SMCTC**.

In the author's opinion, **SMCTC** is suitable for implementing algorithms that can fit into its framework easily. In contrast, **vSMC** might be more suitable for developing new algorithms, especially those related to SMC. **vSMC** is also useful when different parallelization of the same algorithm is desired. For example, it is not uncommon that a new algorithm is developed on a personal computer, which only has access to multicore parallelization, and tested using a small sample or data size. And later the same algorithm is deployed to a distributed system for solving large problems. Using **vSMC**, much of the implementation can be reused during the production phase. It is also clear that **vSMC** has the potential to provide better performance. The last but not least, though no software can be said to be "future-proof", it is possible to enhance **vSMC** with new parallel programming models while reusing existing implementations of algorithms.

Affiliation:

Yan Zhou
National University of Singapore
21 Lower Kent Ridge Road
Singapore 119077
E-mail: zhouyan@me.com
URL: http://zhouyan.github.com/

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 62, Issue 9 *Submitted:* 2013-05-13
December 2014 *Accepted:* 2014-07-12